Numerical Renaissance

simulation, optimization, & control

by Thomas Bewley

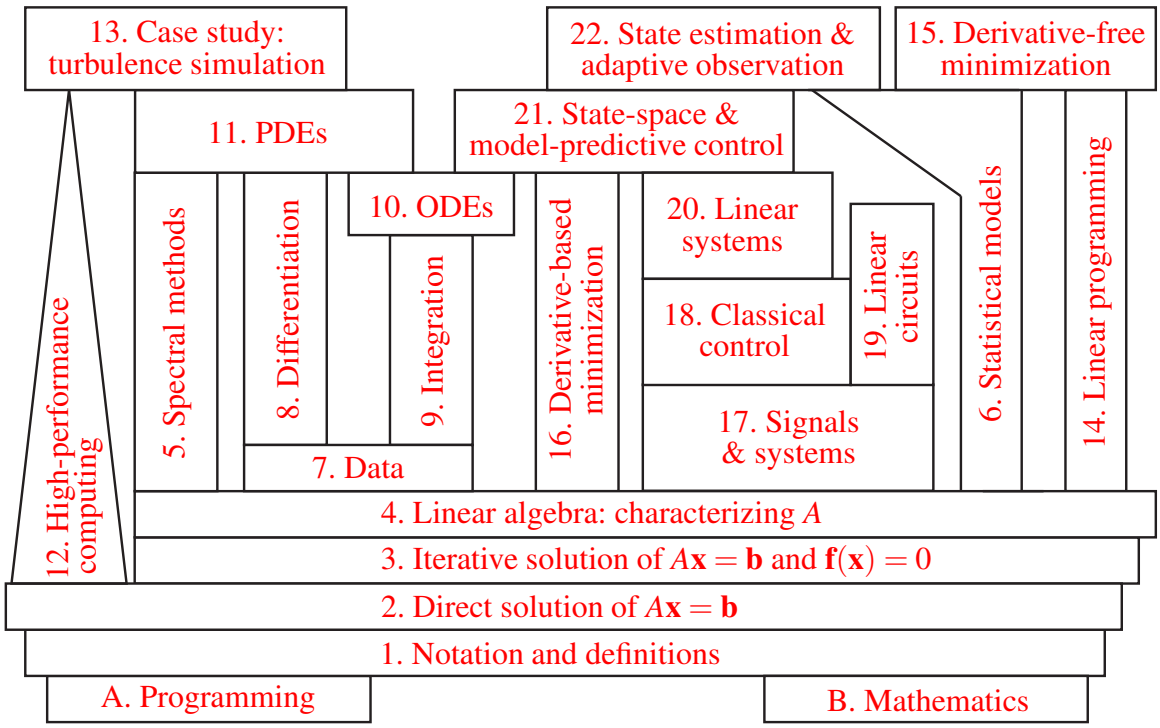


Figure 1: Organization of subjects presented in this book. The primary dependencies of each subject are those which support it in this stack, thereby helping the reader to develop a targeted initial plan for digesting any given subject without reading the entire book.





Numerical Renaissance: simulation, optimization, & control,
by Thomas R. Bewley, published by Renaissance Press.

First edition (hardbound, English language), 2014. ISBN # 978-0-9818359-0-7.
Library of Congress Control Number: 2008938554



Developed at the UCSD Flow Control & Coordinated Robotics Labs.
Online at: http://fccr.ucsd.edu



Published by Renaissance Press.
Online at: http://renaissance-press.com



This book is licensed by Thomas R. Bewley under the Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 Unported License, available online at: http://creativecommons.org/licenses/by-nc-nd/3.0



All numerical codes included in the Numerical Renaissance Codebase, including all codes listed in this book, are copyright (C) 2014 by Thomas R. Bewley, and are available online at: http://numerical-renaissance.com The Numerical Renaissance Codebase is free software: you can redistribute it and/or modify any and all codes included therein under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. The Numerical Renaissance Codebase is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details, available online at: http://www.gnu.org/licenses/gpl-3.0.html

Front cover: The full butterfly graph of the forward and inverse FFT (for $N = 8$), a cornerstone of many modern numerical simulation codes (see Figure 5.3).

Summary

Part I. Fundamentals 1

1 Notation and definitions 3

2 Direct solution of linear equations 27

3 Iterative solution methods 53

4 Linear algebra 67

5 Spectral methods, fast transforms, and the Dirac delta 141

6 Statistical representations 183

7 Data manipulation: sorting, interpolation, & compression 193

Part II. Simulation 235

8 Differentiation 237

9 Integration of functions 251

10 Ordinary differential equations 265

11 Partial differential equations 333

12 High performance computing 391

13 A case study in the simulation of turbulence 405

Part III. Optimization 429

14 Linear programming 431

15 Derivative-free minimization 433

16 Derivative-based minimization 453

Part IV. Control 469

17 Signals & systems: transform-based methods 471

18 Classical control design 527

19 Linear Circuits 587

20 Linear systems: state-space methods 615

21 State-space & model-predictive control design 669

22 State estimation & adaptive observation 695

Supplement 1

A Programming: a brief introduction 3

B Assorted mathematical foundations 15

Preface

The zeitgeist of science and engineering in the twenty-first century is the integration of disciplines—that is, the bridging of the gaps between the formerly fragmented and distinct scientific disciplines, and the grappling with the many remaining grand challenge problems that lie at their intersection. There is thus an emerging need for educational institutions to distill and relate these scientific disciplines for the new generation of scientists who will ultimately accomplish their seamless integration. Towards this end I have written this text, which aims to provide a systematic, integrated, succinct introduction of efficient techniques for solving a wide range of practical problems on modern digital computers. The text's unique treatment spans a number of essential subjects typically covered in several separate graduate-level and undergraduate-level textbooks and courses, as summarized in the dependency plot on page i.

Though succinct, the individual chapters of this text form suitable focused texts for introductory courses in several of these subjects, highlighting many of the key elements and outstanding algorithms of each of them. Significant care has been exercised to ensure that the entire presentation is both self contained, requiring no advanced concepts developed outside this text, and clearly organized, with a minimum of forward references[1] used[2]. At a higher level, by integrating the development of this range of subjects into a single common framework, it is my hope that the inherent relationships between these subjects may be better identified by the reader, and repetition minimized. In short, the dispositions of the subjects covered are presented in a manner for the reader to savor and appreciate over a period of time, not consume and forget in a single semester, and emphasize the many affinities between the several subjects that he or she is digesting.

The methods presented in this text are broadly used in many areas of science and engineering today, in academia, industry, and government, and are beginning to see significant applications in other fields as well, such as economics, finance, business, and politics. The focus of this text is on the streamlined development and presentation of the core elements of these methods and the central assumptions upon which they are based, while providing just enough analysis to facilitate a critical understanding of their efficiency, stability, and accuracy, and just enough examples to spur the reader to begin to apply them to his or her own applications of interest. *Numerical Renaissance* is not intended to be a cookbook illustrating how to brew numerical results from canned routines; while it describes and includes several efficient and well seasoned numerical algorithms, it generally encourages and helps the reader to develop substantial judgement to blend these ingredients together appropriately to suit his or her own particular appetite.

Part I of the text develops the fundamental tools upon which the rest of the text is based, addressing

- direct & iterative solution of linear (both full & sparse) and nonlinear systems of equations;
- the fundamental matrix decompositions: *LU*/*PLU*/Cholesky, *QR*, Schur, Eigen, Jordan, and the SVD;
- solution of the data fitting, representation, interpolation, sorting, and compression problems;
- spectral methods, the FFT, and the Dirac delta;
- statistical methods and random processes.

Part II focuses on approximating differentiation and integration numerically, thus presenting the core

[1] **Reasoning, Circular**: see explanation of **Circular Reasoning** in footnote on Page S-47.

[2] Apologies for our feeble attempts at humor interspersed throughout, which are included in an attempt to keep you on your toes...

algorithms for the efficient, stable, and accurate simulation of systems governed by ODEs and PDEs via finite difference and spectral methods. An introduction to how modern computers actually work is provided, along with some guidelines on how to use them effectively. Part II concludes with a case study which demonstrates how the numerical methods developed earlier may be extended to study a delicate physical phenomenon of importance in many science and engineering applications: turbulent flow. This provides both an appropriate capstone to the preceding chapters as well as a valuable springboard for the efficient numerical simulation of other complex PDE systems, and should be accessible to people from a broad range of backgrounds (that is, prior study of graduate-level fluid mechanics is not required).

Parts III and IV then extend this material to introduce what can be done once an accurate simulation tool to represent the system of interest has been developed: namely,

- identify the uncertain parameters in a system,
- estimate the current state of a system, and forecast its future evolution,
- optimize various parameters affecting the dynamics of a system, and, ultimately,
- control a system to achieve a desired objective by coördinating[3] actuators inputs with sensor measurements (usually for either the stabilization of a desired state or the tracking of a desired trajectory, but sometimes, less aggressively, simply for weakening, invigorating, or otherwise tuning of the chaotic motions of the system in order to alter its statistics in a favorable fashion).

This text does not intend to be all-inclusive of the broad range of subjects covered by its four parts. Rather, it intends to provide a solid, integrated foundation of a deliberate selection of essential material upon which more comprehensive graduate-level and undergraduate-level courses may build in each of these now fairly mature subject areas. Such courses should highlight additional examples of the instructor's choosing, many of which are readily available and thus substantial space for which has not been allotted in this volume. Some of the definitive texts in each constituent subject area are mentioned at the end of each chapter, and should ultimately be used to supplement this text for more in-depth discussions of each. Also, this text does not provide a detailed historical record of the development of the various algorithms it discusses. Several existing texts (including those mentioned at the end of each chapter, to which the reader is referred) do this far better than space allows for here. The perspective of the book is a practical one, focusing on what can be accomplished numerically and how, not simply on what can be proven theoretically, as the development of effective numerical methods inevitably involves a delicate blend elegant analysis and well-motivated heuristics.

I am indebted to innumerable sources for the material presented in this text. In addition to various algorithms developed in our Flow Control & Coordinated Robotics Labs at UCSD, the text builds on notes I have accumulated over the years from public lectures and private conversations with hundreds of individual scientists, as well as from a broad range of textbooks and research papers by my friends and colleagues working in related fields, some of which are cited at the end of each chapter; my apologies in advance for all such references which, for reasons of brevity, I have failed to acknowledge. I am especially beholden to the many who have provided corrections, comments, and suggestions on draft codes and draft copies of sections of this text, including Paul Belitz, Peter Blossey, Patricia Cathalifaud, Daniele Cavaglieri, Andrew Cavender, Laura Cerviño, Joe Cessna, Chris Colburn, Flavio Giannetti, Anish Karandikar, John Kim, Sharon Liu, Paolo Luchini, Haoxiang Luo, Fulvio Martinelli, Scott Miller, Nick Morozovsky, Bob Moser, Mauricio de Oliveira, Saam Ostovari, Costas Pozrikidis, Bartosz Protas, Chris Schmidt-Wetekam, John Taylor, David Zhang, and my many students over the years at UC San Diego in the courses mentioned previously. I would also like to give particular thanks to my BS/MS adviser, Prof. Anthony Leonard, and my PhD advisers, Profs. Parviz Moin, William Reynolds, and Roger Temam, for their advice and encouragement.

[3] The two dots over the second vowel (like in naïve and noël) is called a **diaeresis**, which may be placed over a vowel to indicate that it is sounded in a separate syllable in situations that might otherwise be ambiguous. For example, adding "'co" to "operative" gives a word which might easily be mispronounced if some form of **diacritical mark** isn't used. One could suggest using a hyphen, but then adding a second prefix (as is often done in scientific writing) becomes problematic: both nonco-operative and non-co-operative are downright silly, but noncoöperative works fine. This text, like the *New Yorker*, thus adopts a style that makes extensive use of diaereses.

How to use this text. A distinctive feature of this text is the degree to which it is designed to interconnect the various subjects it presents. As a reader or an instructor, you might well be tempted to skip forward in the text and read about a particular subject of interest without first reading the "prerequisite" material (see page i) upon which it is built. *The reader is encouraged to give in to this temptation*; a nonsequential exploration of this text motivates well a later sequential read in the organized order of presentation intended. For example, at UCSD, I use this text heavily in the following twelve one-quarter courses:

Undergraduate courses	chapters	**Graduate courses**	chapters
Introductory Numerical Methods	1,(2),3,(8),(9),(10)	Numerical Linear Algebra	[2],4,[8],[9],(10)
Aerospace Dynamics & Control	(10),(17),(18)	Num. Methods for ODEs & PDEs	[3],5,[10],11
Signals & Systems	17	Computational Fluid Dynamics	[10],[11],12,13
Classical Control Design	18	Linear Systems	[4],20
Linear Circuits	19	Linear Control & MPC	(21),(22)
Embedded Control & Robotics	[17],[18],[19]	Flow Control & Forecasting	[11],[13],21,22

The list above indicates chapters covered completely by each course, as well as chapters introduced only partially (in parentheses), and chapters reviewed and extended from prerequisite courses [in brackets].

Keywords are highlighted in **boldface**, and are generally defined via context; a quick web search on such keywords pulls up ample additional information. The available pdf is searchable, so an index is not provided.

Rather than using **pseudocode**, numerical algorithms are presented in the text directly in "Matlab syntax" (see Appendix A), which may be executed in Matlab, Octave, or LabVIEW's Mathscript module. Matlab, Octave, and LabVIEW all have a large number of prepackaged numerical algorithms included in convenient "toolboxes", many of which are based on calls to free software libraries like LAPACK. Presenting the algorithms in this text in executable Matlab syntax is done to facilitate easy experimentation with the essential ingredients of several key numerical algorithms directly on your own computer. This is *not* a text about how to apply the built-in toolboxes of Matlab, Octave, or LabVIEW to any specific class of problems; several existing texts accomplish this task quite adequately. Rather, this is a text about how the algorithms at the heart of these toolboxes actually work. In order to understand the algorithms developed herein at a fundamental level, the reader is asked to avoid *all* such advanced built-in numerical routines in this study, instead building up the core components of many of them from scratch (that is, based solely on `for` loops, `if` statements, `function` calls, and floating-point operations on vectors and matrices.

As illustrated by the multigrid algorithm provided in §11 (for comparison purposes, in Matlab, Fortran, and C), as well as the turbulence simulation code of §13 (in Fortran), transferring such algorithms to low-level, **compiler-based** languages such as Fortran and C, which are typically much more efficient than **interpreted languages** like Matlab (especially with memory/cache usage and parallelization), is a natural and straightforward step towards the efficient use of modern computers, from laptops to massively-parallel supercomputers, to solve a wide range of problems in science and engineering, as well as a host of other disciplines that may be studied with scientific methods. It is my pleasure to assist you in this exciting journey.

Thomas Bewley
La Jolla, California



In loving memory of Morticia and Pareese, sunpups extraördinaire,
and in immeasurable appreciation of Checkers and Morena,
and the renaissance in life which they bring.



Part I
Fundamentals

1	Notation and definitions	3
2	Direct solution of linear equations	27
3	Iterative solution methods	53
4	Linear algebra	67
5	Spectral methods, fast transforms, and the Dirac delta	141
6	Statistical representations	183
7	Data manipulation: sorting, interpolation, & compression	193

A review of the two appendices is generally the best starting point for this text. After this review, the "Fundamentals" section consists of seven chapters. In §1, we cover some basic linear algebraic concepts and set the precise mathematical notation to be used in the remainder of the text. Be advised that later chapters leverage heavily the material summarized in §1, so a careful read is advised.

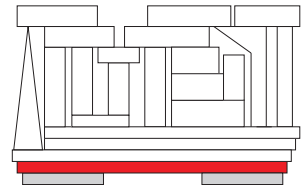
In §2, we then examine the direct solution of systems of linear equations. The idea of combining algebraic equations, ultimately reducing them to a single equation in a single unknown, then successively solving and substituting the results so obtained back into the other equations, is a familiar process that most students see in high school if not before. The process, described in §2, of systematizing this idea as a maximally efficient computer algorithm in various special cases, then implementing, debugging, and testing this algorithm in executable code, incorporates an introduction to the "art" of computer programming (as Donald Knuth so eloquently frames it), as much as it constitutes a study of the Gaussian elimination algorithm itself.

Some problems (including a few in §4) are too difficult to solve directly; we thus take a brief interlude in §3 to discuss some iterative solution methods for linear and nonlinear equations.

In §4, we pick up where we left off in §2. The constructions of the inverse and the echelon form, encountered when studying Gaussian elimination in §2, introduce the **four fundamental subspaces** defined by a matrix. However, §2 leaves open a number of important questions, including how orthogonal bases for these four fundamental subspaces may be constructed, how an optimal inverse mapping may be developed when the matrix inverse itself does not exist, and how certain matrix norms (defined in §1) may be computed. Such questions may be addressed by developing several **fundamental matrix decompositions**, the study of which forms the core of §4. We also identify and establish about fifty frequently useful **facts** (theorems, propositions, lemmata, . . .) in §4. The material presented in §4 is by far the most difficult and most important of the material presented in Part I of the text.

We conclude Part I with three relatively short chapters on

- spectral methods and the elegant **fast Fourier transform** algorithm in §5, which are presented alongside the development of the curious but useful **Dirac delta**,
- a practical discussion of statistical representations in §6, which are encountered in various data interpolation and state estimation strategies discussed later in the text, and
- a survey of some important related topics in data manipulation in §7.



Chapter 1

Notation and definitions

Contents

1.1 Vectors . . . 3
1.2 Matrices . . . 4
1.2.1 Matrix/vector multiplication . . . 5
1.2.2 Matrix/matrix multiplication . . . 7
1.2.3 The identity matrix . . . 9
1.2.4 The inverse of a square matrix . . . 10
1.2.5 Permutation matrices . . . 11
1.2.6 Diagonal groupings of matrix elements . . . 12
1.2.7 Sparse matrices . . . 12
1.2.8 Block matrices . . . 14
1.2.9 The Householder reflector matrix . . . 15
1.2.10 The Givens rotation matrix . . . 17
1.2.11 The fast Givens transformation matrix† . . . 18
1.3 Vector spaces, norms, independence, and orthogonality . . . 20
1.3.1 Vector norms and related concepts . . . 21
1.3.2 Matrix norms . . . 23
1.3.3 Signal norms . . . 24
Exercises . . . 25

1.1 Vectors

A **vector** is defined as an ordered collection of elements (numbers or algebraic expressions):

$$\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}.$$

The number of elements of a vector is called its **order**. All vectors in the present text are assumed to be arranged in columns (and thus may also be referred to as **column vectors**) unless indicated otherwise (which is rare). Vectors are represented with lower-case letters and are denoted in print (as in this text) with boldface ($\mathbf{c}$), whereas they are often denoted in writing (*i.e.*, on the blackboard) with an arrow above the letter ($\vec{c}$). The i'th element of the vector $\mathbf{c}$ is referred to (in **index notation**) as c_i. When several vectors are being considered, they will often be distinguished by their superscript[1] e.g., $\mathbf{c}^j$ for $j = 1, 2, 3, \ldots$ Note also that, for convenience, some chapters will enumerate the elements of vectors and matrices from zero rather than from one.

Numerical methods have generally been developed for the analysis of physical systems. The elements of the vectors and matrices that arise in the numerical approximation of such systems are thus usually, but not always, real. Two important analysis tools will motivate us to consider complex representations even though our interest is derived from physical systems. The first such tool is the analysis of eigenvalues and eigenvectors, as introduced in §4.3. The second such tool is the spectral (Fourier) representation of physical systems, as introduced in §5. Both tools necessitate the use of complex arithmetic, as reviewed briefly in Appendix B, even when considering physical systems.

Thus, for the purpose of generality, *the elements of all vectors and matrices in this text are assumed to be complex unless indicated otherwise*. In order to better visualize the algorithms developed herein, thereby obtaining a more geometric, intuitive understanding, it is generally advisable to simply *ignore the complex components of these algorithms upon first read*, wherever possible, restricting your attention at first to real systems. The complex components of the algorithms usually (but not always, as discussed, e.g., in §1.2.9) follow in a straightforward manner.

Two vectors of the same order are added by adding their individual elements. Likewise, in order to multiply a vector with a scalar, operations are performed on each element. Thus,

$$\mathbf{c} + \mathbf{d} = \begin{pmatrix} c_1 + d_1 \\ c_2 + d_2 \\ \vdots \\ c_n + d_n \end{pmatrix} \quad \text{and} \quad \alpha\,\mathbf{c} = \begin{pmatrix} \alpha\, c_1 \\ \alpha\, c_2 \\ \vdots \\ \alpha\, c_n \end{pmatrix}. \tag{1.1}$$

Vector addition (as well as many other operations yet to be presented) is defined only if the corresponding vectors are of a compatible order such that the corresponding definitions make sense; e.g., if $\mathbf{c}$ and $\mathbf{d}$ are of different orders, $\mathbf{c} + \mathbf{d}$ is undefined.

1.2 Matrices

A **matrix** is defined as a two-dimensional ordered array of elements:

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} = \begin{bmatrix} | & | & & | \\ \mathbf{a}^1 & \mathbf{a}^2 & \ldots & \mathbf{a}^n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} -A_1- \\ -A_2- \\ \vdots \\ -A_m- \end{bmatrix}, \quad \mathbf{a}^j = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{pmatrix}, \quad \mathbf{a} \triangleq \operatorname{vec}(A) = \begin{bmatrix} \mathbf{a}^1 \\ \mathbf{a}^2 \\ \vdots \\ \mathbf{a}^n \end{bmatrix},$$

and $A_j = \begin{pmatrix} a_{j1} & a_{j2} & \ldots & a_{jn} \end{pmatrix}$. The matrix shown above has m rows and n columns; the **order** of this matrix is thus said to be $m \times n$. Matrices are represented with uppercase letters, with their elements represented with

[1] Note: many vector calculus and dynamics texts use superscripts and subscripts to distinguish between:

- **contravariant** vector components, which transform the *opposite* way as the corresponding reference axes upon a change of coördinates (rotation/dilation), and which describe physical vector quantities such as velocities and forces, and
- **covariant** vector components, which transform the *same* way as the corresponding reference axes upon a change of coördinates, and which describe **dual** vector quantities such as **adjoints** and **gradients** (see §21).

The present text enumerates vectors in a more elementary fashion, and does *not* use superscripts and subscripts to indicate this distinction.

lowercase letters; the element of the matrix A in the i'th row and the j'th column is referred to as a_{ij}. Note that, though it is commonly distinguished with a different notation, a vector is just a special case of a matrix with a single column. A matrix with a single row is sometimes called a **row vector**. The (column) vector in the j'th column of A is often referred to as $\mathbf{a}^j$; the row vector in the j'th row of A is occasionally referred to as A_j. The notation $\text{vec}(A)$ is sometimes useful to align all elements of A in vector form. When several matrices are being considered, they will be distinguished by a subscript (e.g., A_k); the notation A^k is reserved exclusively for the k'th power of a matrix. In many cases, the order of a matrix may be determined by context. When ambiguity might otherwise arise, the order of a matrix is denoted in its subscript, e.g., $A_{m\times n}$. A **square** matrix is one for which $m = n$; we refer to a matrix with $m > n$ as **tall**, and a matrix with $m < n$ as **fat**.

The **transpose** of a matrix A, denoted A^T, is found by replacing the rows by the columns. If $B = A^T$, we may write in index notation that $b_{ij} = a_{ji}$ (for all i and j). A **symmetric** matrix is a real matrix for which $A = A^T$; a **skew-symmetric** matrix is a real matrix for which $A = -A^T$.

The **conjugate transpose** (a.k.a. **Hermitian transpose** or **adjoint**) of a matrix A is denoted in this text as A^H. If $B = A^H$, we may write in index notation (denoting with the overbar the complex conjugate) that $b_{ij} = \overline{a}_{ji}$ (for all i and j). For example, defining $\mathrm{i} = \sqrt{-1}$ (see Appendix A),

$$A = \begin{pmatrix} 1 & 2\mathrm{i} \\ 1+3\mathrm{i} & 0 \\ 3 & -4\mathrm{i} \end{pmatrix} \qquad \Rightarrow \qquad A^H = \begin{pmatrix} 1 & 1-3\mathrm{i} & 3 \\ -2\mathrm{i} & 0 & 4\mathrm{i} \end{pmatrix}.$$

The conjugate transpose is often (but not always!) the appropriate operation to use (rather than the transpose) for matrices that are (or might be) complex. A **Hermitian** (a.k.a. **self-adjoint**) matrix is a matrix for which $A = A^H$; a **skew-Hermitian** (a.k.a. **skew-adjoint**) matrix is a matrix for which $A = -A^H$. To recap:

symmetric: $A = A^T$	**Hermitian:** $A = A^H$
skew-symmetric: $A = -A^T$	**skew-Hermitian:** $A = -A^H$

Two matrices of the same order are added by adding their elements. Thus, if $C = A + B$, then $c_{ij} = a_{ij} + b_{ij}$.

Submatrices (a.k.a. **minors**) of the matrix $A = A_{m\times n}$ are created by removing (or retaining) some number of rows and columns from A, and are denoted

$$A(\mathbf{i},\mathbf{j}) = \begin{pmatrix} a_{i_1 j_1} & a_{i_1 j_2} & \dots & a_{i_1 j_b} \\ a_{i_2 j_1} & a_{i_2 j_2} & \dots & a_{i_2 j_b} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_a j_1} & a_{i_a j_2} & \dots & a_{i_a j_b} \end{pmatrix} \quad \text{where} \quad \mathbf{i} = \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_a \end{pmatrix} \quad \text{and} \quad \mathbf{j} = \begin{pmatrix} j_1 \\ j_2 \\ \vdots \\ j_b \end{pmatrix},$$

where $i_k \in \{1,2,\dots m\}$ and $j_k \in \{1,2,\dots,n\}$ with $i_1 < i_2 < \dots < i_a$ and $j_1 < j_2 < \dots < j_b$. Contiguous blocks may be denoted with the Matlab "colon" notation; that is, $A(a\!:\!b, c\!:\!d) = A(\mathbf{i},\mathbf{j})$, where $\mathbf{i} = \begin{pmatrix} a & a+1 & \dots & b \end{pmatrix}^T$ and $\mathbf{j} = \begin{pmatrix} c & c+1 & \dots & d \end{pmatrix}^T$. A **principle submatrix** of A is formed by retaining the same rows as columns of the original matrix. The k'th **leading principle submatrix** of $A_{n\times n}$ is given by $A(1\!:\!k, 1\!:\!k)$, whereas the k'th **trailing principle submatrix** of $A_{n\times n}$ is given by $A(n-k+1\!:\!n, n-k+1\!:\!n)$.

1.2.1 Matrix/vector multiplication

The product $A\mathbf{x} = \mathbf{b}$ (more precisely, $A_{m\times n}\mathbf{x}_{n\times 1} = \mathbf{b}_{m\times 1}$) is defined in **index notation**, for each valid value of the subscript i, as:

$$\text{index notation:} \quad b_i = \sum_{j=1}^{n} a_{ij} x_j. \tag{1.2a}$$

In **summation notation** (a.k.a. **Einstein notation**), any term in an equation with lower-case Roman letters $(i, j, k, \ldots)$ as indices (subscripts or superscripts) repeated exactly twice implies summation over all values of that index, without explicitly writing the summation sign. In summation notation, matrix/vector multiplication is thus written simply as

$$\text{summation notation:} \quad b_i = a_{ij} x_j. \tag{1.2b}$$

Summation notation is often convenient for manipulation of involved algebraic equations involving vectors and matrices, but is rarely used when outlining the individual steps of a numerical code; when summation notation is implied is usually obvious by context. To explicitly suppress summation notation in the present text, Greek indices $(\iota, \kappa, \ldots)$ may be used. Thus, the term $\lambda_\kappa \mathbf{s}^\kappa$ does *not* imply summation over κ. Also, if a summation sign appears somewhere in an equation, then summation signs will be inserted everywhere summation is applied (that is, summation notation is suppressed for the entire equation).

As defined above, the first few elements of the vector $\mathbf{b}$ are given by:

$$\begin{aligned} b_1 &= a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n, \\ b_2 &= a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n, \end{aligned}$$

etc. The vector $\mathbf{b}$ may therefore be written:

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} = \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} x_1 + \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} x_2 + \ldots + \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} x_n \quad \Leftrightarrow \quad \mathbf{b} = \mathbf{a}^1 x_1 + \mathbf{a}^2 x_2 + \ldots + \mathbf{a}^n x_n.$$

Thus, *the vector* $\mathbf{b}$ *is a linear combination of the columns of A with the elements of* $\mathbf{x}$ *as weights*; this is called the **column-wise interpretation of matrix/vector multiplication**. Alternatively, *the element* b_i *is seen to be the scalar formed by taking the product of the row vector in row i of A with the column vector* $\mathbf{x}$:

$$\underbrace{\begin{pmatrix} b_1 \\ \boxed{b_2} \\ \vdots \\ b_m \end{pmatrix}}_{\mathbf{b}} = \underbrace{\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1p} \\ \boxed{a_{21} & a_{22} & \ldots & a_{2p}} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mp} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \boxed{x_1} \\ x_2 \\ \vdots \\ x_p \end{pmatrix}}_{\mathbf{x}}; \tag{1.3}$$

this is called the **row-wise interpretation of matrix/vector multiplication**. These two interpretations of matrix/vector multiplication are associated with two different orderings of the loops defining the computations necessary to compute $A\mathbf{x}$. In Matlab syntax,

Column-wise approach

```
b=zeros(m,1)
for j=1:n
   for i=1:m
      b(i)=b(i)+A(i,j)*x(j);
   end
end
```

Row-wise approach

```
for i=1:m
   b(i)=0;
   for j=1:n
      b(i)=b(i)+A(i,j)*x(j);
   end
end
```

[Note that Matlab syntax refers to the elements of A as `A(i,j)`, though the (fairly standard) convention used in this text is to use lowercase letters for the elements of matrices.] Both approaches can be written in a more compact form in Matlab syntax:

Column-wise approach

```
b=zeros(m,1);
for j=1:n
   b=b+A(:,j)*x(j);
end
```

Row-wise approach

```
for i=1:m
   b(i)=A(i,:)*x;
end
```

If $m = n$ and all the elements are nonzero (that is, if the matrix A is **full**), the number of additions, subtractions, multiplications, and divisions [referred to as **flops**[2]] required in either case is $2n^2$. For brevity, most of this text (except §5) does not distinguish the difference in computational cost between real flops and complex flops, as precise flops counts are useful primarily as a comparative measure.

In modern computers, memory is typically accessed (loaded or returned) to/from the main memory and to/from the various levels of higher-speed **cache** memory as contiguous finite-size vectors of data (that is, memory is not accessed just one element at a time). If A is a large, full matrix, then, depending on the ordering of the storage of the elements of A in the computer memory, one of the above approaches will be significantly faster to compute than the other, because the elements involved in successive calculations (many of which can actually be performed simultaneously, that is, in **parallel**) will more often be adjacent to each other in memory (referred to as **unit stride** calculations), and therefore can be retrieved from the main memory more efficiently. There are many subtle considerations involved in order to use

- the main memory and the various levels of high-speed cache memory,
- the presence of two or more **floating-point units** (**FPUs**) on a **central processing unit** (**CPU**) **core**,
- the presence of two or more CPU cores in the computer, and, sometimes,
- the presence of multiple independent computers on an interconnected network

as efficiently as possible to perform the necessary flops in a given numerical algorithm. Fortunately, a good **self-optimizing compiler** (such as those available for C and Fortran) can often analyze loops like those shown above and restructure them as necessary in order to use a single CPU core fairly efficiently. However, in order to use the entire computing system effectively, the programmer often needs to be aware of several additional considerations; such **high-performance computing** (**HPC**) issues are discussed further in §12.

It is important to recognize the amount of work a good compiler can do to make something as simple as matrix/vector multiplication maximally efficient, and the very significant effect this can have on the execution time of a numerical code. In this regard, not all compilers are created equal (far from it, in fact—it is definitely worthwhile to shop around when selecting a compiler). Further, simple data structures (e.g., statically-allocated arrays whose type, size, and layout in the main memory are known at compile time) are easier for a good compiler to optimize effectively than more flexible and complicated data structures. Thus, even as relatively high-level languages like Matlab and Python (which are generally interpreted at runtime) continue to grow in both complexity and popularity, lower-level languages such as Fortran and C (which must be compiled before executing), programmed with simple data structures, will continue to play a vital role in the efficient numerical solution of large-scale problems.

1.2.2 Matrix/matrix multiplication

The product $A_{m\times p}X_{p\times n} = B_{m\times n}$ is defined in summation notation, for the $\{i, j\}$'th element of the matrix B, as

$$b_{ij} = a_{ik}x_{kj}.$$

As the index k appears twice in the term on the right-hand side (rhs), summation is implied over the index k. Note the following important facts which follow from this definition:

[2]The abbreviation **flops** means either **floating point operations**, when measuring the complexity of a computer algorithm, or **floating point operations per second**, when measuring the speed of a computer. Which definition is implied is usually clear by context; the former definition is used for the remainder of this text.

Fact 1.1 *In general, $AB \neq BA$, that is, matrix multiplication usually does not commute.*

Fact 1.2 *The transpose of a product equals the product of the transposes in reverse order: $(AB)^H = B^H A^H$.*

Fact 1.3 *The product of two Hermitian matrices is, in general, not Hermitian.*

Defining $C = A^T$, $D = B^T$, and $e_{ij} = a_{ki} b_{jk}$, note that, since $a_{ki} = c_{ik}$ and $b_{jk} = d_{kj}$, it follows that $e_{ij} = c_{ik} d_{kj}$, and thus $E = CD = A^T B^T$ (note in particular that, in general, $E \neq AB$).

As defined above, the matrix product $B = AX$ may be written:

$$B = \underbrace{\begin{bmatrix} \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} & \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} & \cdots & \begin{pmatrix} a_{1p} \\ a_{2p} \\ \vdots \\ a_{mp} \end{pmatrix} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \begin{pmatrix} x_{11} & x_{12} & \ldots & x_{1n} \end{pmatrix} \\ \begin{pmatrix} x_{21} & x_{22} & \ldots & x_{2n} \end{pmatrix} \\ \vdots \\ \begin{pmatrix} x_{p1} & x_{p2} & \ldots & x_{pn} \end{pmatrix} \end{bmatrix}}_{X} = \sum_{k=1}^{p} \begin{pmatrix} a_{1k} \\ a_{2k} \\ \vdots \\ a_{mk} \end{pmatrix} \begin{pmatrix} x_{k1} & x_{k2} & \ldots & x_{kn} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} & \ldots & x_{1n} \end{pmatrix} + \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} \begin{pmatrix} x_{21} & x_{22} & \ldots & x_{2n} \end{pmatrix} + \ldots + \begin{pmatrix} a_{1p} \\ a_{2p} \\ \vdots \\ a_{mp} \end{pmatrix} \begin{pmatrix} x_{p1} & x_{p2} & \ldots & x_{pn} \end{pmatrix}.$$

Thus, *the matrix B is a linear combination of the products of the column vectors of A and the corresponding row vectors of X*; this is called the **column-wise interpretation of matrix/matrix multiplication**. Alternatively, *the element b_{ij} is seen to be the scalar formed by taking the product of the row vector in row i of A with the column vector in column j of X*:

$$\underbrace{\begin{pmatrix} b_{11} & \boxed{b_{12}} & \ldots & b_{1n} \\ b_{21} & b_{22} & \ldots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \ldots & b_{mn} \end{pmatrix}}_{B} = \underbrace{\begin{pmatrix} \boxed{a_{11} & a_{12} & \ldots & a_{1p}} \\ a_{21} & a_{22} & \ldots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mp} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x_{11} & \boxed{x_{12}} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{p1} & x_{p2} & \ldots & x_{pn} \end{pmatrix}}_{X}, \tag{1.4}$$

this is called the **row-wise interpretation of matrix/matrix multiplication**. These two interpretations of matrix/matrix multiplication are associated with two different orderings of the loops defining the computations necessary to compute AX. In Matlab syntax[3],

Column-wise approach

```
B=zeros(m,n)
for k=1:p
   for j=1:n
      x=X(k,j);
      for i=1:m
         B(i,j)=B(i,j)+A(i,k)*x;
      end
   end
end
```

Row-wise approach

```
for j=1:n
  for i=1:m
     b=0;
     for k=1:p
        b=b+A(i,k)*X(k,j);
     end
     B(i,j)=b;
  end
end
```

[3]Note that, by using the temporary variables `x` and `b` in these algorithms, the number of array references in the innermost loop is reduced. Using temporary variables in such a manner can often significantly accelerate the execution of a numerical code.

If $m = n = p$ and both matrices are full, the number of flops required is $2n^3$. Thus, for large n, relative to other problems addressed in this text, *the multiplication of full matrices is expensive and thus should be avoided wherever possible*. The art of designing efficient numerical methods involves, among other things, using a variety of clever tricks to avoid such expensive calculations in the algorithms developed.

Note that there are primarily two ways to store matrices in the computer memory: **column-major format** (storing in the computer memory all elements of the first column, followed by all elements of the second column, etc.—the default in Matlab and Fortran) and **row-major format** (storing all elements of the first row, followed by all elements of the second row, etc.—the default in C). In Matlab and Fortran, the column-wise approach shown above is unit stride on the inner loop for both A and B (that is, it steps through the first index of both A and B one at a time), and references just a single element of X (which we may alert to the compiler by use of the temporary variable x). Thus, this configuration uses cache efficiently and will therefore execute quickly. In C, neither of the above orderings of the calculations is unit stride on the innermost loop in all three matrices (stepping through the second index of the matrices one at a time rather than the first), thereby rendering both forms less than maximally efficient in terms of accessing cache, which can result in a significant increase in execution time. This situation is easily circumvented by rearranging the data structure such that the transpose of these matrices is actually what is stored in memory, in which case the column-wise approach to matrix/matrix multiplication is again unit stride.

Each step of the outer loop (for k) in the column-wise approach is often referred to as an **outer-product update**, whereas each step of the intermediate loop (for i) in the row-wise approach is often referred to as an **inner-product** calculation. Note that both names are used somewhat loosely in this context, because the actual definitions of inner and outer products (to be presented in §1.3) involve (for complex systems) complex conjugation. Another common operation in numerical algorithms is called a **gaxpy** (generalized $A\mathbf{x}$ plus $\mathbf{y}$) calculation, which can be performed using the approaches discussed in §1.2.1. As mentioned previously, good compilers go to great lengths to restructure matrix storage and command execution orders so that such basic linear algebra operations, for various problem sizes, are handled with maximum efficiency on a given CPU with a given size hierarchy of high-speed caches. To ensure your numerical codes use the most highly optimized routines possible for such basic operations, your codes (in Fortran or C) can call a set of routines called the **basic linear algebra subroutines** (**blas**), which are provided, in a standardized format, with every modern CPU/operating-system combination as well as with many high-performance compilers.

The element-wise product of two matrices or vectors, known as the **Hadamard product** (a.k.a. **Schur product**), is denoted $X = A \bullet B$ and defined such that, in index notation, $x_{\imath\kappa} = a_{\imath\kappa} * b_{\imath\kappa}$.

Another matrix product which is occasionally encountered is the **Kronecker product**:

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix}.$$

1.2.3 The identity matrix

The **identity** matrix is a square matrix with ones on the diagonal and zeros off the diagonal.

$$I = \begin{pmatrix} 1 & & & 0 \\ & 1 & & \\ & & \ddots & \\ 0 & & & 1 \end{pmatrix} = \begin{bmatrix} | & | & & | \\ \mathbf{e}^1 & \mathbf{e}^2 & \dots & \mathbf{e}^n \\ | & | & & | \end{bmatrix} \quad \Rightarrow \quad I\mathbf{x} = \mathbf{x}, \quad IA = AI = A.$$

In the notation used for I at the left, in which there are several blank spots in the matrix, all blank elements are assumed to be zero. Note that a matrix or a vector is not changed when multiplied by I. For historical reasons, the naming convention used for the columns and elements of the identity matrix is different than that

used for other matrices (as described in §1.2). Specifically, the columns of the identity matrix are referred to as the **Cartesian unit vectors** and denoted by $\mathbf{e}^i$, and the elements of the identity matrix are referred to as the **Kronecker delta** and denoted by δ_{ij}:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{1.5}$$

1.2.4 The inverse of a square matrix

If A is square and $BA = I$, we may refer to B as the **inverse** of A, denoted $B = A^{-1}$. Note, however, that for a given square matrix A, an inverse does not necessarily exist. When it does (a simple test for which is given in §4.2.1), A is said to be **nonsingular** or **invertible**; When it does not, A is said to be **singular** or **noninvertible**. Premultiplying the equation $A\mathbf{x} = \mathbf{b}$ by A^{-1} (if it exists) results in

$$A^{-1}\Big[A\mathbf{x} = \mathbf{b}\Big] \qquad \Rightarrow \qquad \mathbf{x} = A^{-1}\mathbf{b}.$$

Computation of the inverse of a square matrix (an algorithm for which is illustrated in §2.1) thus leads to one method for determining $\mathbf{x}$ given a (nonsingular) A and $\mathbf{b}$; however, for large n, this method is computationally quite expensive as the order of the problem is increased, and thus approaches based on the direct or iterative methods discussed in §2 and §3 are preferred. Note that, since linear algebra (§4) is, in general, **noncommutative** (e.g., $AB \neq BA$), one always has to be careful when multiplying an equation by a matrix to multiply out all terms consistently (either from the left, as illustrated above, or from the right). *Never "divide" by a matrix or vector, as doing so is nonsense.* Instead, work with equations in matrix form by, e.g., premultiplying (or postmultiplying, as appropriate) the entire equation by an inverse, as illustrated above.

Fact 1.4 *The left and right inverses of a square matrix, if they exist, are identical.*

Proof: Assume $AB = I$ and $CA = I$. Then $C[AB = I] \;\Rightarrow\; [CA]B = C \;\Rightarrow\; B = C$. □

Fact 1.5 *The inverse of a square matrix, if it exists, is unique.*

Proof: Assume $AX = I$ and $AY = I$. By Fact 1.4, $YA = I$. Thus, $Y[AX = I] \;\Rightarrow\; [YA]X = Y \;\Rightarrow\; X = Y$. □

Fact 1.6 $(A^H)^{-1} = (A^{-1})^H \triangleq A^{-H}$.

Proof: Assuming $(A^H B) = I$, it follows that $(A^H B)^H = B^H A = I$. By the first relation, $B = (A^H)^{-1}$. By the second, $B^H = A^{-1} \;\Rightarrow\; B = (A^{-1})^H$. □

The following useful facts are easily verified by multiplying the original square matrices by the formulae given for their inverse.

Fact 1.7 *The inverse of a product equals the product of the inverses in reverse order:* $(AB)^{-1} = B^{-1}A^{-1}$.

Fact 1.8 *If* $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ *and* A *is nonsingular, then* $A^{-1} = \dfrac{1}{bc - ad}\begin{pmatrix} -d & b \\ c & -a \end{pmatrix}$.

Fact 1.9 (The Matrix Inversion Lemma, part 1) *If* A, C, *and* $(A + BCD)$ *are nonsingular, then*

$$(A + BCD)^{-1} = A^{-1} - A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1}.$$

Note that the Matrix Inversion Lemma is extended in §4.2.1. Note also that discussion of the (nonunique) left inverses of tall matrices and right inverses of fat matrices is deferred to the footnote in §4.7.1.

A complex square matrix A with orthonormal columns is called a **unitary** matrix and satisfies the condition $A^{-1} = A^H$; by Fact 1.4, a unitary matrix must also have orthonormal rows. A real square matrix A with orthonormal columns is commonly called an **orthogonal** matrix and satisfies the condition $A^{-1} = A^T$. Since $(AB)^H = B^H A^H$, it follows from Fact 1.7 that, if A and B are unitary, (AB) is also unitary. To recap:

$$\textbf{orthogonal: } A^{-1} = A^T \quad \Big| \quad \textbf{unitary: } A^{-1} = A^H$$

1.2.5 Permutation matrices

An identity matrix with some of its rows (or, some of its columns) reordered is called a **permutation** matrix, often denoted P. As easily verified,

- the product $P^T A$ reorders the rows of A, whereas
- the product AP reorders the columns of A.

The columns of P are orthonormal, and thus $P^T P = I$ (that is, $P^{-1} = P^T$).

In the special case that P is formed via the reordering of just two rows (or columns) of I, it follows that $P^T = P$ and thus $PP = I$ (that is, $P^{-1} = P$). Note that a matrix such as P in this case, satisfying $A^k = I$ for $k \geq 2$, is said to be a **unipotent matrix of degree** k; a unipotent matrix of degree 2 is also said to be **involutory**.

A matrix A is said to be **reducible** if a permutation matrix P exists such that

$$P^T A P = \begin{bmatrix} B & C \\ 0 & D \end{bmatrix}$$

where B and D are square. A matrix that is not reducible is said to be **irreducible**.

A permutation matrix may be used to separate the odd and even components of the vector or matrix that it multiplies. In the (usual) case that matrix elements are enumerated from one, the matrix that accomplishes this reordering is referred to as the **odd-even permutation matrix** P_{oe}, and is given by the odd columns of the identity matrix followed by the even columns of the identity matrix. In this case, $P_{oe}^T \mathbf{x}$ reorders the vector $\mathbf{x}$ to put all of the odd components of $\mathbf{x}$ first, followed by the even components of $\mathbf{x}$. For $n = 8$, the odd-even permutation matrix may be written

$$P_{oe} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \Rightarrow \quad P_{oe}^T \mathbf{x} = \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ x_7 \\ x_2 \\ x_4 \\ x_6 \\ x_8 \end{pmatrix} \triangleq \begin{bmatrix} \mathbf{x}_o \\ \mathbf{x}_e \end{bmatrix}.$$

In the (occasional) case that matrix elements are enumerated from zero, the matrix that accomplishes this reordering is referred to as the **even-odd permutation matrix** P_{eo} (and, for $n = 8$, looks identical to the value of P_{oe} shown above). In this case, $P_{eo}^T \mathbf{x}$ reorders the vector $\mathbf{x}$ to put the even components of $\mathbf{x}$ first (that is, starting from zero), followed by the odd components of $\mathbf{x}$ (starting from one).

1.2.6 Diagonal groupings of matrix elements

The **main diagonal** of an $m \times n$ matrix A is the collection of elements along the line from a_{11} to a_{pp}, where $p = \min(m,n)$. The **first subdiagonal** is the collection of elements along the diagonal immediately below the main diagonal, the **second subdiagonal** is immediately below the first subdiagonal, etc.; the **first superdiagonal** is immediately above the main diagonal, the **second superdiagonal** is immediately above the first superdiagonal, etc. The **antidiagonals** of a matrix are collections of elements along lines perpendicular to the main diagonal. A matrix whose elements are constant along each of its diagonals is called **Toeplitz**; a matrix whose elements are constant along each of its antidiagonals is called **Hankel**. An $n \times n$ matrix is called **circulant** if the j'th superdiagonal and $(n-j)$'th subdiagonal (for all j) naturally group into vectors of length n, which we will call **extended diagonals**. Circulant matrices commonly arise in the spatial discretization of periodic systems, in which the first few and last few lines of the matrix relate the state of the discretized system at its two ends. A typical example is given by the matrix C below with $\mathbf{c} = \mathbf{d} = 0$. If the elements along each extended diagonal of a circulant matrix are constant, the matrix is said to be **circulant Toeplitz**.

$$Z = \underbrace{\begin{pmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{pmatrix}}_{\text{Toeplitz}}, \quad C = \underbrace{\begin{pmatrix} a_1 & b_1 & c_1 & d_1 & e_1 \\ e_2 & a_2 & b_2 & c_2 & d_2 \\ d_3 & e_3 & a_3 & b_3 & c_3 \\ c_4 & d_4 & e_4 & a_4 & b_4 \\ b_5 & c_5 & d_5 & e_5 & a_5 \end{pmatrix}}_{\text{circulant}}, \quad H = \underbrace{\begin{pmatrix} a & b & c & d & e \\ b & c & d & e & f \\ c & d & e & f & g \\ d & e & f & g & h \\ e & f & g & h & i \end{pmatrix}}_{\text{Hankel}}.$$

1.2.7 Sparse matrices

A **sparse** matrix has many elements that are known to be zero (in contrast with a **full** matrix, with no known sparsity structure). Typical examples are shown below, denoting by $*$ the nonzero elements.

$$T = \underbrace{\begin{pmatrix} * & * & & & 0 \\ * & * & * & & \\ & * & * & * & \\ & & * & * & * \\ 0 & & & * & * \end{pmatrix}}_{\text{tridiagonal}}, \quad B = \underbrace{\begin{pmatrix} * & * & & & 0 \\ & * & * & & \\ & & * & * & \\ & & & * & * \\ 0 & & & & * \end{pmatrix}}_{\text{upper bidiagonal}}, \quad S = \underbrace{\begin{pmatrix} * & & 0 & & * \\ & * & & & * \\ 0 & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{pmatrix}}_{\text{arrow (sparse but not banded)}},$$

$$L = \underbrace{\begin{pmatrix} 1 & & & & 0 \\ * & 1 & & & \\ * & * & 1 & & \\ * & * & * & 1 & \\ * & * & * & * & 1 \end{pmatrix}}_{\text{unit lower triangular}}, \quad U = \underbrace{\begin{pmatrix} * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \\ 0 & & & & * \end{pmatrix}}_{\text{upper triangular}}, \quad T_0 = \underbrace{\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ 0 & & & * & * \end{pmatrix}}_{\text{upper Hessenberg}}.$$

A **banded** matrix is a special type of sparse matrix with nonzero elements only near the main diagonal. Such matrices arise in, e.g., the discretization of differential equations. As we will show, the tighter (more narrow) the band of nonzero elements, the easier it is to solve the problem $A\mathbf{x} = \mathbf{b}$. A **diagonal** matrix has nonzero elements only on the main diagonal; a **tridiagonal** matrix (see Algorithm 1.1) has nonzero elements only on the main diagonal, the first subdiagonal, and the first superdiagonal; a **pentadiagonal** matrix has nonzero elements only on the main diagonal, the first two subdiagonals, and the first two superdiagonals.

Algorithm 1.1: Construction of a (sparse) tridiagonal matrix from three vectors.

```
function [A]=TriDiag(a,b,c)
% Construct a tridiagonal or tridiagonal circulant matrix from the vectors {a,b,c}.
n=length(b); A=diag(a(2:n),-1)+diag(b,0)+diag(c(1:n-1),1); A(1,n)=a(1); A(n,1)=c(n);
end % function TriDiag
```

View
Test

A square diagonal matrix is denoted $\mathrm{diag}(\mathbf{d})$ or $\mathrm{diag}(d_1, d_2, \ldots, d_n)$, where $\mathbf{d}$ is the vector of diagonal elements; similarly, a square tridiagonal matrix is denoted $\mathrm{tridiag}(\mathbf{a}, \mathbf{b}, \mathbf{c})$, where $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are vectors of the extended subdiagonal, diagonal, and extended superdiagonal elements.

An **upper bidiagonal** matrix has nonzero elements only on the main diagonal and first superdiagonal; an **upper tridiagonal** matrix has nonzero elements only on the main diagonal and first two superdiagonals. An **arrow** matrix is a banded matrix with additional nonzero elements in the row(s) at the bottom and the column(s) on the right. An **upper triangular** matrix has nonzero elements only on the main diagonal and the superdiagonals. A **strictly upper triangular** matrix has nonzero elements only on the superdiagonals. A **unit upper triangular matrix** is an upper triangular matrix with 1's on the main diagonal. An **upper Hessenberg** matrix has nonzero elements only on the main diagonal, the superdiagonals, and the first subdiagonal. **Lower** bidiagonal, triangular, and Hessenberg forms may be defined as the transpose of their "upper" counterparts.

The natural combinations of the above names are also common. For instance, the form of Z given earlier is a **tridiagonal Toeplitz** matrix if only a, b, and f are nonzero. The form of C given earlier is often called a **tridiagonal circulant** matrix if only $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{e}$ contain nonzero elements (even though, strictly speaking, such a matrix is not tridiagonal); such a matrix may further be distinguished as a **tridiagonal circulant Toeplitz** matrix if the elements are constant along each of the extended diagonals.

The following facts may now be verified by inspection:

Fact 1.10 *The product of upper triangular matrices is upper triangular. The product of unit upper triangular matrices is unit upper triangular. The product of an upper Hessenberg and an upper triangular matrix is upper Hessenberg. Analogous statements hold for the corresponding lower and block forms (see §1.2.8).*

Fact 1.11 *If Λ is diagonal, then the product $S\Lambda$ scales the columns of S by the elements on the main diagonal of Λ, whereas the product ΛB scales the rows of B by the diagonal elements of Λ.*

Note that the inverse of a sparse matrix (banded or otherwise) is, in general, full. That is, *the process of taking the inverse destroys the sparsity structure* upon which efficient numerical methods may be based. For example, as easily verified (by multiplying A by the formula given for its inverse),

$$A = \begin{pmatrix} -2 & 1 & & & 0 \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & 1 & -2 & 1 \\ 0 & & & 1 & -2 \end{pmatrix} \quad \Rightarrow \quad A^{-1} = -\frac{1}{6} \begin{pmatrix} 5 & 4 & 3 & 2 & 1 \\ 4 & 8 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 8 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}.$$

A strictly upper triangular (or strictly lower triangular) matrix $A_{n\times n}$ satisfies $A^k = 0$ where $k = n$; such a matrix is said to be a **nilpotent matrix of degree** k. Note that a nilpotent matrix need not be strictly upper or lower triangular; an example of a full 3×3 nilpotent matrix of degree 2 is

$$A = \begin{pmatrix} -4 & -2 & -6 \\ -10 & -5 & -15 \\ 6 & 3 & 9 \end{pmatrix}.$$

1.2.8 Block matrices

A **block** (a.k.a. **partitioned**) matrix has elements that naturally group into smaller submatrices, and is distinguished in this text with square brackets (instead of round brackets). Block versions of the sparse matrice forms described in §1.2.7 are common. In particular, block banded matrices often arise when discretizing systems of partial differential equations in more than one direction. For example, we will see that the following block tridiagonal Toeplitz matrix M arises when discretizing the Laplacian operator in two dimensions on a uniform grid using a second-order central finite-difference approximation of the second derivatives:

$$M = \underbrace{\begin{bmatrix} B & C & & & 0 \\ A & B & C & & \\ & \ddots & \ddots & \ddots & \\ & & A & B & C \\ 0 & & & A & B \end{bmatrix}}_{\text{block tridiagonal Toeplitz}} \quad \text{with} \quad B = \underbrace{\begin{pmatrix} -4 & 1 & & & 0 \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ 0 & & & 1 & -4 \end{pmatrix}}_{\text{tridiagonal Toeplitz}}, \quad A = C = I. \tag{1.6}$$

Working with block matrices is similar to working with regular matrices, noting that each operation performed on the blocks must be consistent with the rules described previously. Note in particular that:

- the block sizes must match appropriately so that the operations performed are well defined,
- matrix multiplication is noncommutative, so operations must be performed in a consistent order, and
- matrix "division" is not defined (thus, instead, premultiply or postmultiply by the inverse).

Example:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \end{bmatrix}.$$

A $n \times n$ **companion** matrix is a matrix in a simple block form with an $(n-1) \times (n-1)$ identity matrix and a single row or column of nonzero coefficients. This type of matrix comes in the following four forms:

$$\underbrace{\begin{pmatrix} -a_{n-1} & \dots & -a_1 & -a_0 \\ 1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & 1 & 0 \end{pmatrix}}_{\textbf{top companion form}}, \quad \underbrace{\begin{pmatrix} 0 & \dots & 0 & -a_0 \\ 1 & & 0 & -a_1 \\ & \ddots & & \vdots \\ 0 & & 1 & -a_{n-1} \end{pmatrix}}_{\textbf{right companion form}}, \quad \underbrace{\begin{pmatrix} 0 & 1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & 1 \\ -a_0 & -a_1 & \dots & -a_{n-1} \end{pmatrix}}_{\textbf{bottom companion form}}, \quad \underbrace{\begin{pmatrix} -a_{n-1} & 1 & & 0 \\ \vdots & & \ddots & \\ -a_1 & & & 1 \\ -a_0 & 0 & \dots & 0 \end{pmatrix}}_{\textbf{left companion form}}.$$

As easily verified by multiplication (try it!), the inverses of these four matrices (which exist iff $a_0 \neq 0$, and are also in companion form) are, respectively,

$$\begin{pmatrix} 0 & 1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & 1 \\ \frac{-1}{a_0} & \frac{-a_{n-1}}{a_0} & \dots & \frac{-a_1}{a_0} \end{pmatrix}, \quad \begin{pmatrix} \frac{-a_1}{a_0} & 1 & & 0 \\ \vdots & & \ddots & \\ \frac{-a_{n-1}}{a_0} & & & 1 \\ \frac{-1}{a_0} & 0 & \dots & 0 \end{pmatrix}, \quad \begin{pmatrix} \frac{-a_1}{a_0} & \dots & \frac{-a_{n-1}}{a_0} & \frac{-1}{a_0} \\ 1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & 1 & 0 \end{pmatrix}, \quad \begin{pmatrix} 0 & \dots & 0 & \frac{-1}{a_0} \\ 1 & & 0 & \frac{-a_{n-1}}{a_0} \\ & \ddots & & \vdots \\ 0 & & 1 & \frac{-a_1}{a_0} \end{pmatrix}.$$

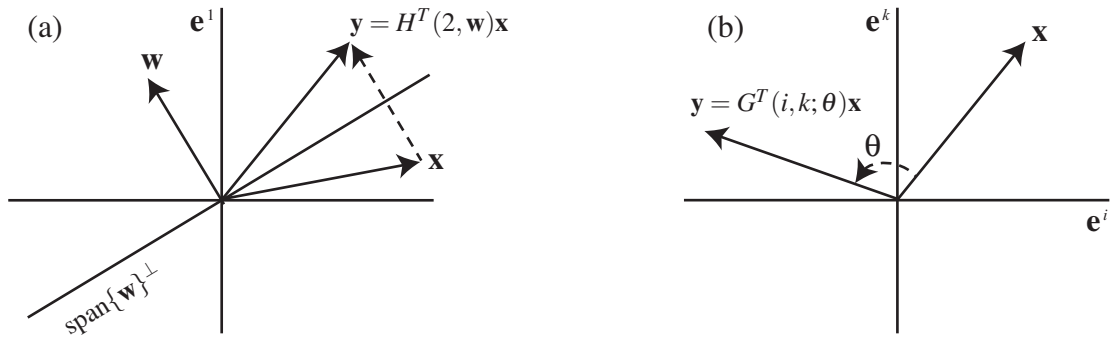


Figure 1.1: The result $\mathbf{y}$ given by multiplying a real vector $\mathbf{x}$ by (a) a real Householder reflector matrix, $\mathbf{y} = H^T(2,\mathbf{w})\mathbf{x}$ [see §1.2.9], and (b) a real Givens rotation matrix, $\mathbf{y} = G^T(i,k;\theta)\mathbf{x}$ [see §1.2.10].

1.2.9 The Householder reflector matrix

The **Householder reflector** matrix[4] $H = H(\sigma,\mathbf{w})$ is an important elementary building block for certain numerical algorithms to come. It is defined as

$$H = I - \sigma \mathbf{w}\mathbf{w}^H \quad \text{with } \sigma \text{ selected and } \mathbf{w} \text{ scaled such that} \quad |\sigma|^2\|\mathbf{w}\|^2 = \sigma + \overline{\sigma} = 2\Re(\sigma). \tag{1.7}$$

The Householder reflector matrix so defined is unitary:

$$H^H H = (I - \overline{\sigma}\mathbf{w}\mathbf{w}^H)(I - \sigma\mathbf{w}\mathbf{w}^H) = I - (\sigma + \overline{\sigma})\mathbf{w}\mathbf{w}^H + (|\sigma|^2\|\mathbf{w}\|^2)\mathbf{w}\mathbf{w}^H = I. \tag{1.8}$$

For real systems, a "simple and logical" scaling is $\sigma = 2$ and $\|\mathbf{w}\| = 1$. With this scaling, H happens to be symmetric, and thus $H = H^T = H^{-1}$, and thus H is involutory. Geometrically, the effect of multiplying a real matrix H so defined times any real vector $\mathbf{x}$ is to subtract from $\mathbf{x}$ twice the projection of $\mathbf{x}$ onto the vector $\mathbf{w}$, thereby reflecting $\mathbf{x}$ across the line (for $n = 2$), plane (for $n = 3$), or hyperplane (for $n > 3$) denoted by span$\{\mathbf{w}\}^\perp$, as illustrated in Figure 1.1a. Multiplying the result by H again reflects the vector back to its original location. Note that, in the real case, assuming $\|\mathbf{x}\| \neq 0$ and defining

$$\sigma = 2, \quad \mathbf{w} = \frac{\mathbf{x} + \nu\mathbf{e}^1}{\left\|\mathbf{x} + \nu\mathbf{e}^1\right\|}, \quad \text{where} \quad \nu = \text{Sign}(x_1)\|\mathbf{x}\| \quad \text{with} \quad \text{Sign}(a) = \begin{cases} 1 & a \geq 0, \\ -1 & a < 0, \end{cases} \tag{1.9a}$$

and taking $\mathbf{e}^1$ as the first Cartesian unit vector (see §1.2.3), it follows that

$$H^T\mathbf{x} = \mathbf{x} - 2\mathbf{w}\mathbf{w}^T\mathbf{x} = \mathbf{x} - \left(\mathbf{x} + \nu\mathbf{e}^1\right)\underbrace{\frac{2\mathbf{x}^T\mathbf{x} + 2\nu x_1}{\left\|\mathbf{x} + \nu\mathbf{e}^1\right\|^2}}_{=1} = \mathbf{x} - \left(\mathbf{x} + \nu\mathbf{e}^1\right) = -\nu\mathbf{e}^1.$$

Geometrically (see Figure 1.1a), this amounts to selecting a vector $\mathbf{w}$ of unit length such that, when $\mathbf{x}$ is reflected through the (hyper-)plane normal to $\mathbf{w}$, the reflected vector aligns with the positive or negative Cartesian unit vector, $\pm\mathbf{e}^1$. The careful choice of the sign of ν in (1.9a) is made in order to maximize the length of the vector $\mathbf{x} + \nu\mathbf{e}^1$ upon which H is constructed; if the opposite choice is made, this calculation can sometimes lead to the subtraction of two vectors that are almost equal, thereby leading to an inaccurate result if finite-precision arithmetic is used (which is always the case in a numerical calculation!).

[4] An expression like $H = H(\sigma,\mathbf{w})$ indicates that, in this case, H has a functional dependence on $\{\sigma,\mathbf{w}\}$.

Algorithm 1.2: Computation of a Householder reflection onto the $\mathbf{e}^1$ direction.

View

```
function [sig,w] = ReflectCompute(x)
% Compute the parameters {sig,w} of a Householder reflection matrix designed to reflect
% the vector x to (*;0;0;...;0).
if real(x(1))<0, s=-1; else, s=1; end, nu=s*norm(x);    % (1.8b)
if nu==0, sig=0; w=0;   else, sig=(x(1)+nu)/nu; w=[x(1)+nu; x(2:end)]/(x(1)+nu); end
end % function ReflectCompute
```

Algorithm 1.3: Application of a Householder reflection to a set of vectors.

View
Test

```
function [X] = Reflect(X,sig,w,i,k,p,q,which)
% Apply a Householder reflection H(sig,w), embedded in rows i:k and columns i:k of an
% identiy matrix, to the matrix X, with {sig,w} as given by ReflectCompute.m
% Use which='L' to premultiply by H^H, 'R' to postmultiply by H, 'B' to do both.
% Note: the elements outside the range p:q in the columns of X (if premultiplying) and/or
% the rows of X (if postmultiplying) are assumed to be zero, and thus left untouched.
if or(which=='L',which=='B')
  X(i:k,p:q)=X(i:k,p:q)-(conj(sig)*w)*(w'*X(i:k,p:q));   % (1.10a)
end, if or(which=='R',which=='B')
  X(p:q,i:k)=X(p:q,i:k)-(X(p:q,i:k)*w)*(sig*w');         % (1.10b)
end
end % function Reflect
```

If σ, $\mathbf{w}$, and $\mathbf{x}$ are complex, the situation is a bit more involved. In this case, it turns out that H is *not* Hermitian; fortunately, we don't need it to be. The Householder reflector is useful in numerical methods primarily because, as noted above, σ and $\mathbf{w}$ can be selected such that H is unitary and H transforms (or "reflects") an arbitrary vector $\mathbf{x}$ into a vector aligned in the direction of $\pm\mathbf{e}^1$. To accomplish this in complex systems, one definition that may be used (among several—see Lehoucq 1996 for a comparison with other definitions), assuming $\|\mathbf{x}\| \neq 0$, is

$$\sigma = \frac{x_1 + \nu}{\nu}, \quad \mathbf{w} = \frac{\mathbf{x} + \nu\mathbf{e}^1}{x_1 + \nu}, \quad \text{where} \quad \nu = \text{Sign}(\Re(x_1))\|\mathbf{x}\| \quad \text{with} \quad \text{Sign}(a) = \begin{cases} 1 & a \geq 0, \\ -1 & a < 0. \end{cases} \tag{1.9b}$$

With this definition, if $\mathbf{x}$ is real, then σ is real and somewhere in the range $1 \leq \sigma \leq 2$; that is, the definition of σ and $\mathbf{w}$ for complex systems, as given in (1.9b), does not necessarily reduce to the "simple and logical" scaling of σ and $\mathbf{w}$ for real systems, as given in (1.9a). However, H is still unitary [by (1.8)], and the product $H^H\mathbf{x}$ gives the following, which is sufficient for this matrix to be useful in the algorithms to come:

$$H^H\mathbf{x} = \mathbf{x} - \overline{\sigma}\mathbf{w}\mathbf{w}^H\mathbf{x} = \mathbf{x} - \frac{(\overline{x}_1 + \overline{\nu})(\mathbf{x} + \nu\mathbf{e}^1)(\mathbf{x} + \nu\mathbf{e}^1)^H\mathbf{x}}{\overline{\nu}(x_1 + \nu)(\overline{x}_1 + \overline{\nu})} = \ldots = -\nu\mathbf{e}^1. \tag{1.10}$$

Note that the product H^HX, where $H = H(\sigma, \mathbf{w})$ is a Householder reflector matrix, may be written as

$$H^HX = (I - \overline{\sigma}\mathbf{w}\mathbf{w}^H)X = X - (\overline{\sigma}\mathbf{w})(\mathbf{w}^HX). \tag{1.11a}$$

Calculating $\overline{\sigma}\mathbf{w}$ and $\mathbf{w}^HX$ separately, then multiplying and subtracting from X, is *much* less expensive than calculating H then multiplying H^H times X, as illustrated in Algorithms 1.2-1.3 (see also Exercise 1.5). Similarly,

$$XH = X(I - \sigma\mathbf{w}\mathbf{w}^H) = X - (X\mathbf{w})(\sigma\mathbf{w}^H). \tag{1.11b}$$

1.2.10 The Givens rotation matrix

The **Givens rotation** matrix $G = G(i,k;c,s)$ is another important elementary building block for certain numerical algorithms to come. It may be defined as the $n \times n$ matrix built from the identity matrix with only four elements changed such that $g_{ii} = c$, $g_{kk} = \overline{c}$, $g_{ik} = \overline{s}$, and $g_{ki} = -s$, where $|c|^2 + |s|^2 = 1$. For example, if $n = 5$, $i = 1$, and $k = 3$, then G is defined (again, denoting with the overbar the complex conjugate) as

$$G = \begin{pmatrix} c & & \overline{s} & & 0 \\ & 1 & & & \\ -s & & \overline{c} & & \\ & & & 1 & \\ 0 & & & & 1 \end{pmatrix}, \quad G^H = \begin{pmatrix} \overline{c} & & -\overline{s} & & 0 \\ & 1 & & & \\ s & & c & & \\ & & & 1 & \\ 0 & & & & 1 \end{pmatrix}. \tag{1.12}$$

Note that the Givens rotation matrix is unitary, and thus $G^H = G^{-1}$.

Note that the product $Y = G^H X$, where $G = G(i,k;c,s)$ is a Givens rotation matrix, is simple to compute: denoting by X_i the i'th row of the matrix X, the i'th and k'th rows of X are transformed by G^H according to

$$Y_i = \overline{c}X_i - \overline{s}X_k, \quad Y_k = sX_i + cX_k, \tag{1.13a}$$

with the remaining rows unchanged (i.e., $Y_j = X_j$ for $j \notin \{i,k\}$), as illustrated in Algorithms 1.4-1.5 (see also Exercise 1.6). Similarly, denoting by $\mathbf{x}^i$ the i'th column of X, $Z = XG$ may be computed according to

$$\mathbf{z}^i = c\mathbf{x}^i - s\mathbf{x}^k, \quad \mathbf{z}^k = \overline{s}\mathbf{x}^i + \overline{c}\mathbf{x}^k, \tag{1.13b}$$

with the remaining columns unchanged (i.e., $\mathbf{z}^j = \mathbf{x}^j$ for $j \notin \{i,k\}$).

In real systems, $c = \cos(\theta)$ and $s = \sin(\theta)$ for some angle θ, and the Givens rotation matrix is denoted $G(i,k;\theta)$. Geometrically, the effect of multiplying $G^T(i,k;\theta)$ times any real vector $\mathbf{x}$ is to rotate $\mathbf{x}$ counterclockwise through an angle θ in the plane span$\{\mathbf{e}^i, \mathbf{e}^k\}$, as illustrated in Figure 1.1b. Multiplying the result by $G(i,k;\theta) = G^T(i,k;-\theta)$ rotates the vector back to its original location.

For any i and k and any corresponding (and possibly complex) x_i and x_k, we may define the Givens rotation matrix $G(i,k;c,s)$ with c and s and a convenient "storage" variable γ selected such that

$$\begin{cases} c = \gamma = 1, \quad s = 0, \quad \text{(no rotation)} & & & & & \text{if } |x_k| = 0 \\ \tau = -x_k/x_i, \quad d = 1 + |\tau|^2, \quad c = 1/\sqrt{d}, \quad s = \tau/\sqrt{d}, & & & & \gamma = 1/s & \text{if } |x_i| \geq |x_k| \neq 0 \\ \tau = -x_i/x_k, \quad d = 1 + |\tau|^2, \quad s = 1/\sqrt{d}, & c = \tau/\sqrt{d}, & & & \gamma = c & \text{otherwise.} \end{cases} \tag{1.14}$$

By construction, $|c|^2 + |s|^2 = 1$. The above formulae defining the parameters of the Givens rotation matrix G avoid the (potentially expensive) computation of the trigonometric functions associated with θ in the real case. With this definition of G, as easily verified, the product $\mathbf{y} = G^H\mathbf{x}$ results in $y_k = 0$. Geometrically (see Figure 1.1b), this amounts (in the real case) to selecting an angle θ such that, when $\mathbf{x}$ is rotated counterclockwise by the angle θ in the plane span$\{\mathbf{e}^i, \mathbf{e}^k\}$, the component of the rotated vector $\mathbf{y}$ in the direction $\mathbf{e}^k$ vanishes.

The storage variable γ defined in (1.14) provides a convenient means to store the information necessary to reconstruct c and s using a single number (real or complex, depending on A), leveraging the fact that $|c|^2 + |s|^2 = 1$. In the second line of (1.14), note that c is real with $1 > c > 1/\sqrt{2}$ and $1/\sqrt{2} > |s| > 0$. In the third line of (1.14), note that s is real with $1 \geq s > 1/\sqrt{2}$ and $1/\sqrt{2} > |c| \geq 0$. Thus, given γ, we can recover c and s as follows:

$$\begin{cases} c = 1, & s = 0 \quad \text{(no rotation)} & \text{if } |\gamma|^2 = 1 \\ s = 1/\gamma, & c = \sqrt{1 - 1/|\gamma|^2} & \text{if } |\gamma|^2 > 1 \\ c = \gamma, & s = \sqrt{1 - |\gamma|^2} & \text{otherwise.} \end{cases} \tag{1.15}$$

Algorithm 1.4: Computation of a Givens rotation onto the $\mathbf{e}^1$ direction.

View

```
function [c,s] = RotateCompute(f,g)
% Compute the parameters {c,s} of a Givens rotation matrix designed to rotate
% the vector (f;g) to (*;0).
gs=real(g)^2+imag(g)^2;
if gs==0, c=1; s=0; else, fs=real(f)^2+imag(f)^2;
  if fs>=gs, c=1/sqrt(1+gs/fs); s=-c*g/f;
  else,      s=1/sqrt(1+fs/gs); c=-s*f/g;  end
end
end % function RotateCompute
```

Algorithm 1.5: Application of a Givens rotation to a set of vectors.

View
Test

```
function [X] = Rotate(X,c,s,i,k,p,q,which)
% Apply a Givens rotation G(c,s), embedded in elements (i,i) (i,k) (k,i) and (k,k)
% of an identiy matrix, to the matrix X, with {c,s} as given by RotateCompute.
% Use which='L' to premultiply by G^H, 'R' to postmultiply by G, 'B' to do both.
% Note: the elements outside the range p:q in the columns of X (if premultiplying) and/or
% the rows of X (if postmultiplying) are assumed to be zero, and thus left untouched.
if or(which=='L',which=='B')
  X([i k],p:q)=[conj(c)*X(i,p:q)-conj(s)*X(k,p:q); s*X(i,p:q)+c*X(k,p:q)];
end, if or(which=='R',which=='B')
  X(p:q,[i k])=[c*X(p:q,i)-s*X(p:q,k), conj(s)*X(p:q,i)+conj(c)*X(p:q,k)];
end
end % function Rotate
```

1.2.11 The fast Givens transformation matrix†

The fact that c and s may be selected in the definition of an $n \times n$ Givens rotation matrix $G(i,k;c,s)$ such that the product $\mathbf{y} = G^H\mathbf{x}$ results in $y_k = 0$ renders it useful in certain numerical algorithms to come (in particular, see §4.4.2). Recall that, by its definition [see (1.12)], the Givens rotation matrix is unitary. If we now relax this condition, we may define a related transformation matrix that is even faster to apply to a vector or set of vectors, as it has even fewer entries that are neither 0 or 1. The resulting matrix, called a **fast Givens transformation** matrix due to its structural similarity to the Givens rotation matrix discussed in §1.2.10, comes in two types. The first type, $F(1;i,k;\alpha,\beta)$, is built from the identity matrix with only four elements changed such that $f_{ii} = \beta$, $f_{kk} = \overline{\alpha}$, $f_{ik} = 1$, and $f_{ki} = 1$. The second type, $F(2;i,k;\alpha,\beta)$, is built from the identity matrix with only two elements changed such that $f_{ki} = \beta$ and $f_{ik} = \overline{\alpha}$. For example, if $n = 5$, $i = 1$, and $k = 3$, then the $n \times n$ matrices $F(1;i,k;\alpha,\beta)$ and $F(2;i,k;\alpha,\beta)$ are defined [cf. (1.12)] as

$$F(1;i,k;\alpha,\beta) = \begin{pmatrix} \beta & & 1 & & 0 \\ & 1 & & & \\ 1 & & \overline{\alpha} & & \\ & & & 1 & \\ 0 & & & & 1 \end{pmatrix}, \quad F(2;i,k;\alpha,\beta) = \begin{pmatrix} 1 & & \overline{\alpha} & & 0 \\ & 1 & & & \\ \beta & & 1 & & \\ & & & 1 & \\ 0 & & & & 1 \end{pmatrix}. \tag{1.16}$$

Fast Givens transformation matrices are not unitary. As in (1.13), the products $Y = F^H X$ and $Z = XF$ are simple to compute; in fact, as two of the four nontrivial elements of the Givens rotation matrix are set to 1 in the fast Givens transformation matrix, each of the expressions corresponding to (1.13) requires 33% fewer flops to compute.

†Note: this section amounts essentially to a subtle refinement of the Givens rotation matrix and may, like other sections of this text marked with a dagger (†), be skipped on first read.

Algorithm 1.6: Computation of a fast Givens transform onto the $\mathbf{e}^1$ direction.

View

```
function [a,b,gamma,donothing,dnew]=FastGivensCompute(f,g,di,dk)
% Compute the parameters {a,b,gamma,donothing} of a Fast Givens transformation matrix
% designed to transform the vector (f;g) to (*;0).
if g==0, donothing=1; else                                  % see sentence before (1.16)
  a=-f/g; b=-a*dk/di; gamma=-(real(a)*real(b)+imag(a)*imag(b)); donothing=0;
  if gamma<=1, dnew=(1+gamma)*[dk di]; else dnew=(1+1/gamma)*[di dk]; a=1/a; b=1/b; end
end
end % function FastGivensCompute
```

Algorithm 1.7: Application of a fast Givens transform to a set of vectors.

View
Test

```
function [X]=FastGivens(X,a,b,gamma,donothing,i,k,p,q,which)
% Apply a fast Givens transformation F(gamma;a,b), embedded in elements (i,k) and (k,i) of
% an identiy matrix, to the matrix X, with {gamma;a,b} as given by FastGivensCompute.m
% Use which='L' to premultiply by F^H, 'R' to postmultiply by F, 'B' to do both.
% Note: the elements outside the range p:q in the columns of X (if premultiplying) and/or
% the rows of X (if postmultiplying) are assumed to be zero, and thus left untouched.
if ~donothing, if or(which=='L',which=='B')
    if gamma<=1
      X([i k],p:q)=[conj(b)*X(i,p:q)+X(k,p:q); X(i,p:q)+a*X(k,p:q)]; % (1.12a), modified
    else
      X([i k],p:q)=[X(i,p:q)+conj(b)*X(k,p:q); a*X(i,p:q)+X(k,p:q)]; % (1.12a), modified
    end
  end, if or(which=='R',which=='B')
    if gamma<=1
      X(p:q,[i k])=[b*X(p:q,i)+X(p:q,k), X(p:q,i)+conj(a)*X(p:q,k)]; % (1.12b), modified
    else
      X(p:q,[i k])=[X(p:q,i)+b*X(p:q,k), conj(a)*X(p:q,i)+X(p:q,k)]; % (1.12b), modified
    end
end, end
end % function FastGivens
```

For any i and k, any corresponding x_i and $x_k \neq 0$, and any real $d_i > 0$ and $d_k > 0$, we may define the fast Givens transformation matrix $F(type;i,k;\alpha,\beta)$ by taking $\alpha = -x_i/x_k$, $\beta = -\alpha d_k/d_i$, $\gamma = -\alpha\overline{\beta} > 0$, and

$$\begin{cases} type = 1, & d_i^{new} = (1+\gamma)d_k, \quad d_k^{new} = (1+\gamma)d_i & \text{if } \gamma \leq 1, \\ type = 2, & d_i^{new} = (1+\gamma^{-1})d_i, \quad d_k^{new} = (1+\gamma^{-1})d_k, \quad \alpha \leftarrow \alpha^{-1}, \quad \beta \leftarrow \beta^{-1} & \text{otherwise.} \end{cases} \tag{1.17}$$

If $x_k = 0$, we simply take $F = I$. Note that the above formulae defining the fast Givens transformation matrix $F(type;i,k;\alpha,\beta)$ avoid the (potentially expensive) computation of a square root, as required by (1.14). With this definition of F, as easily verified, the product $\mathbf{y} = F^H\mathbf{x}$ results in $y_k = 0$. Also, as easily verified, if D is a diagonal matrix with diagonal elements d_j, then the product $F^H DF = D_{new}$ is cheap to compute, as D_{new} itself is just a diagonal matrix with diagonal elements d_j^{new}, with the elements d_i^{new} and d_k^{new} defined as in (1.17) and the remaining elements of D unchanged (i.e., $d_j^{new} = d_j$ for $j \notin \{i,k\}$). These properties render the fast Givens transformation matrix useful in certain numerical algorithms to come (in particular, see §4.4.2). Efficient implementation of these formulae is given in Algorithms 1.6-1.7.

Example 1.1 Robot path following.
Con

1.3 Vector spaces, norms, independence, and orthogonality

Regardless of its order (be it a scalar, a column vector, a row vector, or an $m \times n$ matrix), a matrix with all zero elements, known as the **zero matrix** or **zero vector**, is denoted 0 in this text. The **empty set** (a set with no elements) is denoted $\emptyset$. In contrast, a vector with all elements unity, known as **one vector**, is denoted $\mathbf{1}$.

A **vector space** V is defined as a set of complex (or real) vectors $\mathbf{v}$ of the same order such that any **linear combination** of any two vectors in V is itself also in V, that is,

$$c_1\mathbf{v}^1 + c_2\mathbf{v}^2 \in \mathsf{V} \;\text{ for any }\; \mathbf{v}^1, \mathbf{v}^2 \in \mathsf{V} \;\text{ and any }\; c_1, c_2 \in \mathsf{F},$$

where F is the set of all complex (or real, depending on V) numbers. Thus, a vector space V always contains the zero element. The vector space of all complex vectors of order n is denoted $\mathbb{C}^n$, the vector space of all real vectors of order n is denoted $\mathbb{R}^n$, and the vector space containing only the zero element is denoted $\{0\}$. A **subspace** is a **subset** of a vector space (denoted, e.g., $\mathsf{X} \subseteq \mathsf{V}$) which itself is also a vector space.

The **inner product** of two vectors is defined in this work as the scalar

$$(\mathbf{u}, \mathbf{v}) = \mathbf{u}^H\mathbf{v} = \sum_{i=1}^{n} \overline{u}_i\, v_i = \overline{u}_1\, v_1 + \overline{u}_2\, v_2 + \ldots + \overline{u}_n\, v_n.$$

Two vectors are said to be **orthogonal** if[5] their inner product equals zero. The **outer product** of two vectors $\mathbf{u}$ and $\mathbf{v}$ is the matrix $\mathbf{u}\mathbf{v}^H$.

A **norm** of some quantity e, denoted $\|e\|$, is a real measure that satisfies the following four properties:

- **non-negativity:** $\|e\| \geq 0$,
- **positivity:** $e = 0 \;\Leftrightarrow\; \|e\| = 0$,
- **homogeneity:** $\|\alpha e\| = |\alpha| \cdot \|e\|$ for any complex scalar α,
- **triangle inequality:** $\|e_1 + e_2\| \leq \|e_1\| + \|e_2\|$.

A **semi-norm** is defined the same way, except with the second property relaxed to: $e = 0 \;\Rightarrow\; \|e\| = 0$.

We consider norms of the following types of quantities in this work:

- **vectors** [see §1.3.1],
- **matrices** [see §1.3.2],
- **signals**, i.e., scalars or vectors that are either continuous or discrete functions of time [see §1.3.3], and
- **linear systems**, i.e., linear relationships between **input** signals and **output** signals[6] [see §20.2.2].

[5] In mathematical definitions, the word "if" is commonly used as a matter of convenience, though, strictly speaking, the phrase "if and only if" is implied. The phrase "if and only if" (commonly abbreviated as "**iff**" and written symbolically as $\Leftrightarrow$) is generally reserved for **biconditionals**, that is, statements/facts/theorems/lemmas requiring mathematical proof.

[6] Which signal is taken as the "input" and which is taken as the "output" is, of course, a matter of perspective, and needs to be defined in the description of the corresponding physical problem. Also, as discussed in §17, such input/output relationships are specified in **continuous time** (**CT**) via **forced differential equations**, the **impulse response** $e(t)$ of such forced differential equations, or the Laplace transform of such impulse responses, known as the **transfer function** $E(s)$ of the CT system (see §17.3.3). Such relationships are specified in **discrete time** (**DT**) via **forced difference equations**, the **impulse response** e_k of such forced difference equations, or the Z transform of such impulse responses, known as the **transfer function** $E(z)$ of the DT system (§17.4.3). Note that, for any t, s, k, or z, respectively, these quantities are simply scalars, vectors, or matrices.

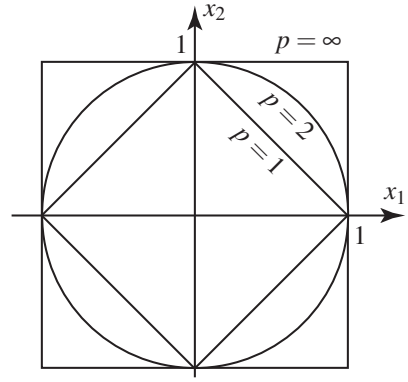


Figure 1.2: Contours of $\|\mathbf{x}\|_p = 1$ for $p = 1$, 2, and ∞.

1.3.1 Vector norms and related concepts

The **vector p-norm**, denoted $\|\mathbf{v}\|_p$, is a norm of a vector defined, for $1 \le p \le \infty$, by

$$\|\mathbf{v}\|_p = (|v_1|^p + |v_2|^p + \ldots + |v_n|^p)^{1/p} \quad \Rightarrow \quad \begin{cases} \|\mathbf{v}\|_1 = |v_1| + |v_2| + \ldots + |v_n|, \\ \|\mathbf{v}\|_2 = \sqrt{|v_1|^2 + |v_2|^2 + \ldots + |v_n|^2} = \sqrt{(\mathbf{v},\mathbf{v})}, \\ \|\mathbf{v}\|_\infty = \max_{1\le i\le n} |v_i|. \end{cases} \tag{1.18}$$

The most common vector p-norms are the **vector 1-norm**, a.k.a. the **Manhattan norm** (as it measures "distance" assuming movement is confined to a cartesian grid of "streets"), the **vector 2-norm**, a.k.a. the **Euclidean norm** or the **vector length**, and the **vector ∞-norm**, a.k.a. the **max norm**; the geometric interpretation of these three cases is illustrated in Figure 1.2. The vector 2-norm is used the most in this text, and is therefore often denoted $\|\mathbf{v}\| = \|\mathbf{v}\|_2$. We will also have occasion to use the **weighted inner product**, $(\mathbf{u},\mathbf{v})_Q = \mathbf{u}^H Q\mathbf{v}$, and the **weighted vector 2-norm**, $\|\mathbf{v}\|_Q = \sqrt{(\mathbf{v},\mathbf{v})_Q}$ (for some positive definite Q, as defined in §4.4.4).

The **angle between two real vectors**, $\measuredangle(\mathbf{u},\mathbf{v})$, may be defined using the inner product such that

$$\cos\measuredangle(\mathbf{u},\mathbf{v}) = \frac{(\mathbf{u},\mathbf{v})}{\|\mathbf{u}\|\,\|\mathbf{v}\|} \quad \text{where} \quad -1 \le \cos\measuredangle(\mathbf{u},\mathbf{v}) \le 1. \tag{1.19}$$

Note in particular[7] the **Cauchy-Schwartz inequality** $|(\mathbf{u},\mathbf{v})| \le \|\mathbf{u}\|\,\|\mathbf{v}\|$, and its generalization, Hölder's inequality $|(\mathbf{u},\mathbf{v})| \le \|\mathbf{u}\|_p \|\mathbf{v}\|_q$ where $1/p + 1/q = 1$. The **projection** of $\mathbf{u}$ on the vector $\mathbf{v}$ is defined by $\mathbf{u}_\mathbf{v} = (\mathbf{u},\mathbf{v})\cdot\mathbf{v}/\|\mathbf{v}\|^2 = \|\mathbf{u}\|\cdot\cos\measuredangle(\mathbf{u},\mathbf{v})\cdot\mathbf{v}/\|\mathbf{v}\|$; its length is $\|\mathbf{u}_\mathbf{v}\| = \|\mathbf{u}\|\cdot|\cos\measuredangle(\mathbf{u},\mathbf{v})|$.

Figure 1.2 also indicates a geometric relationship between the vector p norms, from which it follows that

$$\begin{aligned} \|x\|_1/\sqrt{n} \le \|x\|_2 \le \|x\|_1 &\le \sqrt{n}\,\|x\|_2 \\ \|x\|_2/\sqrt{n} \le \|x\|_\infty \le \|x\|_2 &\le \sqrt{n}\,\|x\|_\infty \\ \|x\|_1/n \;\le \|x\|_\infty \le \|x\|_1 &\le\; n\,\|x\|_\infty; \end{aligned}$$

these three norms are thus said to be **equivalent**. Remarkably, *all* norms on finite-dimensional vector spaces are equivalent; however, as the order, n, of the problem increases, these bounds become increasingly loose.

A set of vectors $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^r\}$ is called **linearly independent** if none of the vectors in the set may be formed by linear combination of the others (equivalently, if $c_1\mathbf{x}^1 + c_2\mathbf{x}^2 + \ldots + c_r\mathbf{x}^r = 0 \;\Rightarrow\; \mathbf{c} = 0$). Such a set is called **orthogonal** if $(\mathbf{x}^i,\mathbf{x}^j) = 0$ for $i \ne j$, and such a set is called **orthonormal** if $(\mathbf{x}^i,\mathbf{x}^j) = \delta_{ij}$. A set of

[7] The Cauchy-Schwartz inequality follows immediately by squaring both sides of the triangle inequality $\|\mathbf{u}+\mathbf{v}\| \le \|\mathbf{u}\| + \|\mathbf{v}\|$.

vectors $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^r\}$ is said to **span** a vector space V if every $\mathbf{v} \in \mathsf{V}$ can be expressed as a linear combination of the vectors in the set; that is, in summation notation, $\mathbf{v} = c_i \mathbf{x}^i$ for any $\mathbf{v} \in \mathsf{V}$ for some $c_i \in \mathsf{F}$ where, again, F is the set of all complex (or real, depending on V) numbers; in such a case, we write $\mathsf{V} = \text{span}\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^r\}$. A linearly independent set of r vectors that spans a vector space V is referred to as a **basis**; in this case, no set containing fewer than r vectors also spans V, and it is said that the **dimension** of V is r. If the r vectors of a basis are all mutually orthogonal, the basis is called an **orthogonal basis** [an example in $\mathbb{R}^n$ is $\{\mathbf{e}^1, \mathbf{e}^2, \ldots, \mathbf{e}^n\}$]. The dimension of the vector space with only the zero element, $\{0\}$, is defined as zero.

A **positive linear combination** of a set of vectors in a real vector space is a linear combination with nonnegative coefficients. A set of vectors in a real vector space is called **positively linearly independent** if none of the vectors in the set may be formed by positive linear combination of the others. A set of vectors $\{\mathbf{x}^1,\mathbf{x}^2,\ldots,\mathbf{x}^r\}$ is said to **positively span** a real vector space V if every $\mathbf{v}\in\mathsf{V}$ can be expressed as a positive linear combination of the vectors in the set. A positively linearly independent set of r vectors that positively spans a real vector space V is referred to as a **positive basis**; in this case, no subset of these vectors containing fewer than r vectors also positively spans V. A positive basis in $\mathbb{R}^n$ has between $n+1$ and $2n$ vectors in the basis. A positive basis in $\mathbb{R}^n$ with $n+1$ vectors is called a **minimum positive basis** [an example is $\{\mathbf{e}^1,\mathbf{e}^2,\ldots,\mathbf{e}^n,\mathbf{f}\}$, where $\mathbf{f}=\begin{pmatrix}-1 & -1 & \ldots & -1\end{pmatrix}^T$]. A positive basis in $\mathbb{R}^n$ with $2n$ vectors is called a **maximum positive basis** [an example is $\{\mathbf{e}^1,\mathbf{e}^2,\ldots,\mathbf{e}^n,-\mathbf{e}^1,-\mathbf{e}^2,\ldots,-\mathbf{e}^n\}$].

Let X and Y be subspaces of a vector space V. We say that X and Y **span** V, and write $\mathsf{V}=\mathsf{X}+\mathsf{Y}$, if every $\mathbf{v}\in\mathsf{V}$ can be expressed as a sum $\mathbf{v}=\mathbf{x}+\mathbf{y}$ for some $\mathbf{x}\in\mathsf{X}$ and $\mathbf{y}\in\mathsf{Y}$. If, in addition, $(\mathbf{x},\mathbf{y})=0$ for all $\mathbf{x}\in\mathsf{X}$ and all $\mathbf{y}\in\mathsf{Y}$, we say that X is the **orthogonal complement** of Y in V. This is denoted $\mathsf{X}=\mathsf{Y}^\perp$ (and, thus, $\mathsf{Y}=\mathsf{X}^\perp$). In such a situation, the decomposition $\mathbf{v}=\mathbf{x}+\mathbf{y}$ is unique for all $\mathbf{v}\in\mathsf{V}$.

1.3.2 Matrix norms

A **matrix norm** is a norm of a (possibly, complex) matrix A [i.e., a real measure of A which satisfies the four properties outlined in §1.3] which additionally satisfies the **submultiplicative property** $\|AB\|\leq\|A\|\|B\|$.

The **induced p-norm** of some (possibly nonsquare) matrix A, denoted $\|A\|_{ip}$, is defined by:

$$\|A\|_{ip}\triangleq\max_{\mathbf{x}\neq0}\frac{\|A\mathbf{x}\|_p}{\|\mathbf{x}\|_p}=\max_{\|\mathbf{x}\|_p=1}\|A\mathbf{x}\|_p, \tag{1.20}$$

where $\|\mathbf{x}\|_p$ denotes the vector p-norm defined in §1.3. The induced p-norm $\|A\|_{ip}$ is an upper bound on the amount the matrix A can "amplify" the vector p-norm of the vector $\mathbf{x}$ when determining the vector $\mathbf{b}=A\mathbf{x}$:

$$\mathbf{b}=A\mathbf{x}\quad\Rightarrow\quad\|\mathbf{b}\|_p=\|A\mathbf{x}\|_p\leq\|A\|_{ip}\|\mathbf{x}\|_p. \tag{1.21}$$

The most common induced p-norms are the **induced 1-norm** $\|A\|_{i1}$, the **induced 2-norm** $\|A\|_{i2}$ [a.k.a. the **spectral radius** $\rho(A)$], and the **induced ∞-norm** $\|A\|_{i\infty}$. Among these measures of a matrix, the induced 2-norm is used most frequently in this text, and is therefore often denoted by $\|A\|=\|A\|_{i2}$. By (1.18), the last formula in (1.20), and straightforward geometric arguments (see Figure 1.1), it follows that

- the induced 1-norm is given by the **maximum column sum** [i.e., $\|A\|_{i1}=\max_j(\sum_{i=1}^m|a_{ij}|)$], and
- the induced ∞-norm is given by the **maximum row sum** [i.e., $\|A\|_{i\infty}=\max_i(\sum_{j=1}^n|a_{ij}|)$];

these formulae are straightforward to calculate. An efficient method for determining the induced 2-norm is deferred to Fact 4.38. Note also that, when characterizing the combined effect of several matrix multiplications, the following fact, effectively extending the Cauchy-Schwarz inequality to matrices, is often useful.

Fact 1.12 *The induced p-norm is a matrix norm, thus satisfying* $\|AB\|_{ip}\leq\|A\|_{ip}\|B\|_{ip}$.

Proof: Defining $\mathbf{y}=B\mathbf{x}$, we have

$$\|AB\|_{ip}=\max_{\mathbf{x}\neq0}\frac{\|AB\mathbf{x}\|_p}{\|\mathbf{x}\|_p}=\max_{\mathbf{x}\neq0}\frac{\|A\mathbf{y}\|_p}{\|\mathbf{y}\|_p}\cdot\frac{\|\mathbf{y}\|_p}{\|\mathbf{x}\|_p}\leq\max_{\mathbf{y}\neq0}\frac{\|A\mathbf{y}\|_p}{\|\mathbf{y}\|_p}\cdot\max_{\mathbf{x}\neq0}\frac{\|B\mathbf{x}\|_p}{\|\mathbf{x}\|_p}=\|A\|_{ip}\|B\|_{ip}. \qquad \square$$

Another common matrix measure is the **Frobenius norm**, defined by the following equivalent formulae

$$\|A\|_F=\sqrt{\sum_{i=1}^m\sum_{j=1}^n|a_{ij}|^2}=\sqrt{\sum_{j=1}^n\sum_{i=1}^m|(\mathbf{e}^i)^HA\mathbf{e}^j|^2}=\sqrt{\sum_{j=1}^n(\mathbf{e}^j)^HA^HIA\mathbf{e}^j}=\sqrt{\text{trace}(A^HA)}=\sqrt{\text{trace}(AA^H)}. \tag{1.22}$$

Note that the Frobenius norm is straightforward to calculate using the first form given above.

Fact 1.13 *The Frobenius norm is also a matrix norm, thus satisfying* $\|AB\|_F \leq \|A\|_F \|B\|_F$.

Proof: Note first that

$$\|A\|_F^2 = \sum_{j=1}^{n} (\mathbf{e}^j)^H A^H A \mathbf{e}^j = \sum_{j=1}^{n} \|A\mathbf{e}^j\|^2,$$

where $\|A\mathbf{e}^j\|$ denotes the 2-norm of the vector $(A\mathbf{e}^j)$. Note also that

$$\|AB\|_F^2 = \sum_{j=1}^{n}\sum_{i=1}^{m} |(\mathbf{e}^i)^H AB\mathbf{e}^j|^2 = \sum_{j=1}^{n}\sum_{i=1}^{m} |(A^H\mathbf{e}^i, B\mathbf{e}^j)|^2,$$

where $(\mathbf{x},\mathbf{y})$ denotes the inner product of $\mathbf{x}$ and $\mathbf{y}$. By the Cauchy-Schwartz inequality

$$|(A^H\mathbf{e}^i, B\mathbf{e}^j)|^2 \leq \|A^H\mathbf{e}^i\|^2 \|B\mathbf{e}^j\|^2.$$

Thus,

$$\|AB\|_F^2 \leq \sum_{j=1}^{n}\sum_{i=1}^{m} \|A^H\mathbf{e}^i\|^2 \|B\mathbf{e}^j\|^2 = \sum_{i=1}^{m} \|A^H\mathbf{e}^i\|^2 \sum_{j=1}^{n} \|B\mathbf{e}^j\|^2 = \|A^H\|_F^2 \|B\|_F^2 = \|A\|_F^2 \|B\|_F^2. \qquad \square$$

Note that *not all norms of matrices are matrix norms*. A common example is the max norm applied to a matrix, $\|A\|_M \triangleq \max_{i,j} |a_{ij}|$. This norm satisfies the four required properties of norms, as reviewed in §1.3; however, taking

$$A = B = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix},$$

it is easily seen that $\|AB\|_M \not\leq \|A\|_M \|B\|_M$; that is, this norm does *not* satisfy the submultiplicative property.

1.3.3 Signal norms

In the remainder of this text, we make use of the following norms of **continuous-time** (**CT**) signals $\mathbf{e}(t)$:

- **1-norm** (the integral of the absolute value of the CT signal) $\|\mathbf{e}(t)\|_1 = \int_{-\infty}^{\infty} \|\mathbf{e}(\tau)\|_1 \, d\tau$,
- **2-norm** (the square root of the **total energy** of the CT signal) $\|\mathbf{e}(t)\|_2^2 = \int_{-\infty}^{\infty} \|\mathbf{e}(\tau)\|_2^2 \, d\tau$,
- **rms-norm** (the square root of the **mean energy** of the CT signal) $\|\mathbf{e}(t)\|_{\text{rms}}^2 = \lim_{T\to\infty} \frac{1}{2T} \int_{-T}^{T} \|\mathbf{e}(\tau)\|_2^2 \, d\tau$,
- **∞-norm** (the peak of the absolute value of the CT signal) $\|\mathbf{e}(t)\|_\infty = \max_\tau \|\mathbf{e}(\tau)\|_\infty$;

likewise, we make use of the following norms of **discrete-time** (**DT**) signals $\mathbf{e}_k$:

- **1-norm** (the integral of the absolute value of the DT signal) $\|\mathbf{e}_k\|_1 = \sum_{\kappa=-\infty}^{\infty} \|\mathbf{e}_\kappa\|_1$,
- **2-norm** (the square root of the **total energy** of the DT signal) $\|\mathbf{e}_k\|_2^2 = \sum_{\kappa=-\infty}^{\infty} \|\mathbf{e}_\kappa\|_2^2$,
- **rms-norm** (the square root of the **mean energy** of the DT signal) $\|\mathbf{e}_k\|_{\text{rms}}^2 = \lim_{N\to\infty} \frac{1}{2N} \sum_{\kappa=-N}^{N} \|\mathbf{e}_\kappa\|_2^2$,
- **∞-norm** (the peak of the absolute value of the DT signal) $\|\mathbf{e}_k\|_\infty = \max_\kappa \|\mathbf{e}_\kappa\|_\infty$.

Note the slight **abuse of notation**[8] used here: on the RHS of the "=" sign in all eight of these definitions, the norms indicated are to be interpreted as *vector* norms, as defined in §1.3.1. In both CT and DT, signals with a finite 2-norm have a zero rms-norm, and signals with a finite rms-norm have an infinite 2-norm. Note also that $\|\mathbf{e}(t)\|_{\text{rms}}$ and $\|\mathbf{e}_k\|_{\text{rms}}$ are actually semi-norms, not norms.

[8] An **abuse of notation** is a mathematical expression that is not precisely correct or strictly unambiguous notationally but, once properly explained, clearly and compactly expresses the mathematical concept being stated.

Algorithm 1.8: Code for assembling a Hankel matrix from its top row and right column.

```
function R=Hankel(top,right)
% Build an nxn Hankel matrix with the specified top row and right column.
n=length(top); for row=1:n; R(row,:)=[top(row:n) right(2:row)]; end
end % function Hankel
```

View
Test

Exercises

Exercise 1.1 Let $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ and $B = \begin{pmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$. Calculate AB and BA, by hand and with Matlab/Octave.

Exercise 1.2 A simple code to build a Hankel matrix from its top row and right column is given in Algorithm 1.8. Write similar codes to build a Toeplitz matrix from its top row and left column, and two additional codes to build circulant Hankel and circulant Toeplitz matrices from their top rows only.

Exercise 1.3 Consider a matrix A that satisfies $A^2 = A$ (and is thus said to be **idempotent**). **(a)** Show that $B = I - A$ is also idempotent. **(b)** Show that, if A is also invertible, then $A = I$.

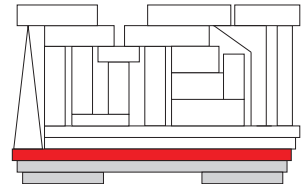
Exercise 1.4 Verify the correctness of (1.10).

Exercise 1.5 **(a)** Compute the Householder reflector matrix $H(2,\mathbf{w})$ in order to reflect an arbitrary vector $\mathbf{x} \in \mathbb{R}^3$ through the plane normal to $\mathbf{w} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^T$. Compute $H^T\mathbf{x}$, where $\mathbf{x} = \begin{pmatrix} 3 & 6 & 2 \end{pmatrix}^T$. Discuss. **(b)** Determine the vector $\mathbf{w}$ and the corresponding Householder reflector matrix $H(2,\mathbf{w})$ in order to reflect the vector $\mathbf{x} = \begin{pmatrix} 3 & 6 & 2 \end{pmatrix}^T$ to a direction parallel to $\mathbf{e}^1$. Compute $H^T\mathbf{x}$ to confirm that the reflector matrix so constructed has the desired effect. Discuss.

Exercise 1.6 **(a)** Compute the (real) Givens rotation matrix $G(1,2,\pi/2)$ in order to rotate an arbitrary vector $\mathbf{x} \in \mathbb{R}^3$ by $\pi/2$ radians counterclockwise in the x_1–x_2 plane. Compute $G^T\mathbf{x}$, where $\mathbf{x} = \begin{pmatrix} 3 & 4 & 1 \end{pmatrix}^T$. Discuss. **(b)** Determine the angle θ and the corresponding Givens rotation matrix $G(1,2,\theta)$ in order to rotate the vector $\mathbf{x} = \begin{pmatrix} 3 & 4 & 1 \end{pmatrix}^T$ in the x_1–x_2 plane such that the resulting vector is zero in its x_2 component. Compute $G^T\mathbf{x}$ to confirm that the rotation matrix so constructed has the desired effect. Discuss.

References

Lehoucq, RB (1996) The Computation of Elementary Unitary Matrices, *ACM Trans. Math. Software* **22**, 393-400.



Chapter 2

Direct solution of linear equations

Contents

2.1 An introduction to the direct solution of $A\mathbf{x} = \mathbf{b}$ . . . 28

2.2 Gaussian elimination for square nonsingular systems . . . 29

2.2.1 Gaussian elimination without pivoting and the LU decomposition . . . 30

2.2.2 Gaussian elimination with partial pivoting and the PLU decomposition . . . 34

2.2.3 Gaussian elimination with complete pivoting and the $PLUQ^T$ decomposition . . . 37

2.2.4 Diagonal dominance: justification for Gaussian elimination without pivoting . . . 38

2.3 Exploiting structured sparsity . . . 39

2.4 Parallelization . . . 43

2.5 Condition number . . . 44

2.6 Singular and nonsquare systems, echelon form, and rank . . . 46

2.7 Chapter summary . . . 49

Exercises . . . 50

Any set of linear algebraic equations may be represented in the standard form $A\mathbf{x} = \mathbf{b}$. For example:

$$\begin{aligned} 2u + 3v - 4w &= 0 \\ u \quad\quad - 2w &= 7 \\ u + v + w &= 12 \end{aligned} \quad \Rightarrow \quad \underbrace{\begin{pmatrix} 2 & 3 & -4 \\ 1 & 0 & -2 \\ 1 & 1 & 1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} u \\ v \\ w \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} 0 \\ 7 \\ 12 \end{pmatrix}}_{\mathbf{b}}.$$

Given an A and $\mathbf{b}$, one often needs to solve such a system for $\mathbf{x}$. In fact, efficient algorithms for the solution of problems in this class, for various different sparsity structures of A, form an important cornerstone for many efficient numerical methods. Thus, this chapter devotes substantial attention to this problem.

2.1 An introduction to the direct solution of $A\mathbf{x} = \mathbf{b}$

If A is square and diagonal, the solution may be found by inspection. For example,

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 7 \end{pmatrix} \quad \Rightarrow \quad \begin{cases} x_1 = 5/2 \\ x_2 = 2 \\ x_3 = 7/4. \end{cases}$$

If A is square and upper triangular, the problem (working from the bottom row up) is also easy. For example,

$$\begin{pmatrix} 3 & 4 & 5 \\ 0 & 6 & 7 \\ 0 & 0 & 8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 19 \\ 8 \end{pmatrix} \quad \Rightarrow \quad \begin{cases} 8x_3 = 8 & \Rightarrow \ x_3 = 1 \\ 6x_2 + 7x_3 = 19 & \Rightarrow \ x_2 = 2 \\ 3x_1 + 4x_2 + 5x_3 = 1 & \Rightarrow \ x_1 = -4. \end{cases}$$

The procedure illustrated above is known as **back substitution**. Note that square, lower triangular systems are just as easy to solve, applying an analogous algorithm working from the top row down (this is sometimes called **forward substitution**). The elements on the main diagonal of a triangular system, referred to as the **pivots**, must be nonzero for such algorithms to succeed in determining a unique solution $\mathbf{x}$. If the i'th pivot is zero, then the matrix is **singular** or **noninvertible**, and the system either has

- *zero solutions* (if, when solving the system via back substitution and reaching the i'th row, one reaches an equation like $0 = 1$, which cannot be made true for any value of x_i), or
- *infinitely many solutions* (if, when solving the system via back substitution and reaching the i'th row, one reaches the equation $0 = 0$, in which case the corresponding element x_i can take any value).

An effective method for treating such singular systems is deferred to §2.6.

To solve a square, full, nonsingular matrix problem $A\mathbf{x} = \mathbf{b}$, we simply reduce the linear system to an equivalent triangular system, from which the solution may readily be found by the back substitution procedure illustrated above. For example, consider the following system:

$$\begin{pmatrix} 0 & 4 & -1 \\ 1 & 1 & 1 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ 1 \end{pmatrix}. \tag{2.1}$$

Interpreting this matrix equation as a collection of rows, each representing a separate equation, we can perform row exchanges and add linear combinations of some of the rows to others and still have the same linear system represented. For example:

1. Exchange first two rows: $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & -1 \\ 2 & -2 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ 1 \end{pmatrix}.$
2. Multiply first row by 2 and subtract from last row: $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & -1 \\ 0 & -4 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ -11 \end{pmatrix}.$
3. Add second row to third: $\begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & -1 \\ 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 5 \\ -6 \end{pmatrix}.$

The matrix on the LHS is now an upper triangular matrix, which we will call U. Reducing the problem $A\mathbf{x} = \mathbf{b}$ to an equivalent problem $U\mathbf{x} = \mathbf{y}$ in this manner is called **Gaussian elimination**. Once we have reduced the problem to a triangular system of this form, the solution may be calculated directly using back substitution, as shown previously. Alternatively (and equivalently), we can continue by scaling and recombining the rows until the matrix becomes the identity; this is referred to as the **Gauss-Jordan** process:

4. Divide last row by -2, then add result to second row: $\begin{pmatrix}1&1&1\\0&4&0\\0&0&1\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix}=\begin{pmatrix}6\\8\\3\end{pmatrix}$.

5. Divide second row by 4: $\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix}=\begin{pmatrix}6\\2\\3\end{pmatrix}$.

6. Subtract second and third rows from first row: $\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix}=\begin{pmatrix}1\\2\\3\end{pmatrix}\;\Rightarrow\;\mathbf{x}=\begin{pmatrix}1\\2\\3\end{pmatrix}$.

The letters x_1, x_2, and x_3 clutter this process, so we may devise a shorthand **augmented matrix** in which we can conduct the same series of operations without the extraneous symbols:

$$\underbrace{\left(\begin{array}{ccc|c}0&4&-1&5\\1&1&1&6\\2&-2&1&1\end{array}\right)}_{A\quad\mathbf{b}}\Rightarrow\ldots\Rightarrow\underbrace{\left(\begin{array}{ccc|c}1&1&1&6\\0&4&-1&5\\0&0&-2&-6\end{array}\right)}_{U\quad\mathbf{y}}\Rightarrow\ldots\Rightarrow\underbrace{\left(\begin{array}{ccc|c}1&0&0&1\\0&1&0&2\\0&0&1&3\end{array}\right)}_{I\quad\mathbf{x}}.$$

Throughout the entire procedure illustrated above, the determination of what operations to perform on each row were based on the matrix A, not the vector $\mathbf{b}$. Thus, we may easily carry along several RHS vectors $\mathbf{b}^i$ and solve the several problems $A\mathbf{x}^i=\mathbf{b}^i$ simultaneously, as illustrated in the following subsection.

Computation of the inverse

The computation of the inverse of a square matrix A may be accomplished using the Gaussian elimination process illustrated above, with the several RHS vectors taken to be the Cartesian unit vectors. In the case of a 3×3 matrix A, we solve the problems $A\mathbf{x}^1=\mathbf{e}^1$, $A\mathbf{x}^2=\mathbf{e}^2$, and $A\mathbf{x}^3=\mathbf{e}^3$ simultaneously, as illustrated by the following example (written here as the full Gauss Jordan process in augmented matrix notation):

$$\underbrace{\left(\begin{array}{ccc|ccc}1&0&2&1&0&0\\1&1&1&0&1&0\\0&1&1&0&0&1\end{array}\right)}_{A\qquad B=I}\Rightarrow\left(\begin{array}{ccc|ccc}1&0&2&1&0&0\\0&1&-1&-1&1&0\\0&1&1&0&0&1\end{array}\right)\Rightarrow\underbrace{\left(\begin{array}{ccc|ccc}1&0&2&1&0&0\\0&1&-1&-1&1&0\\0&0&2&1&-1&1\end{array}\right)}_{U\qquad Y}\Rightarrow$$

$$\left(\begin{array}{ccc|ccc}1&0&2&1&0&0\\0&1&-1&-1&1&0\\0&0&1&\frac12&-\frac12&\frac12\end{array}\right)\Rightarrow\underbrace{\left(\begin{array}{ccc|ccc}1&0&0&0&1&-1\\0&1&0&-\frac12&\frac12&\frac12\\0&0&1&\frac12&-\frac12&\frac12\end{array}\right)}_{I\qquad X}$$

We have thus determined an X such that $AX=I\Leftrightarrow X=A^{-1}$. Following such a procedure, it is seen that

Fact 2.1 *The inverse of a unit lower triangular matrix is unit lower triangular. The inverse of a lower triangular matrix T is lower triangular, with the diagonal elements of T^{-1} being the inverse of the diagonal elements of T. Analogous statements hold for the corresponding upper and block forms.*

2.2 Gaussian elimination for square nonsingular systems

Gaussian elimination may easily be automated, using the following augmented matrix notation:

$$(A\,|\,\mathbf{b})=\left(\begin{array}{cccc|c}a_{11}&a_{12}&\ldots&a_{1n}&b_1\\a_{21}&a_{22}&\cdots&a_{2n}&b_2\\\vdots&\vdots&\ddots&\vdots&\vdots\\a_{n1}&a_{n2}&\cdots&a_{nn}&b_n\end{array}\right).$$

2.2.1 Gaussian elimination without pivoting and the LU decomposition

We begin our study of Gaussian elimination by presenting a simple implementation that does not perform any row exchanges. In §2.2.4 and §4.4.4.1, we will identify important classes of matrices that frequently arise in numerical algorithms for which such an approach is guaranteed to work. In general, however, note that this routine will fail on problems that encounter a zero pivot if no row exchanges are performed, such as the system illustrated in (2.1). Note that the Gaussian elimination part of the problem (that is, reducing the system to an upper triangular form) is referred to as the **forward sweep** in the following description.

Forward sweep, step 1. Eliminate the elements below a_{11} (the first "pivot") in the first column:
Let $m_{21} = -a_{21}/a_{11}$. Multiply the first row by m_{21} and add to the second row.
Let $m_{31} = -a_{31}/a_{11}$. Multiply the first row by m_{31} and add to the third row.
... etc. The modified augmented matrix soon has the form

$$\left(\begin{array}{c|ccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right), \tag{2.2}$$

where all elements except those in the first row have been changed; that is, the operation is performed **in place** in the computer memory.

Forward sweep, step 2. Repeat step 1 for the **reduced augmented matrix** [highlighted by the box in (2.2)]. The pivot for the second column is a_{22}.

... etc. After $n-1$ steps, the modified augmented matrix takes the form

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & a_{nn} & b_n \end{array}\right).$$

Note that at each stage we need to divide by the pivot (the diagonal element above the column of subdiagonal elements being set to zero at that step), so it is pivotal that the pivot be nonzero. If it is not, the present algorithm breaks down, and the algorithm described in §2.2.2 must be used instead.

Back substitution. The process of back substitution is straightforward. Simply scale b_n to determine x_n (and store the result in b_n):

$$b_n \leftarrow b_n/a_{nn}; \tag{2.3a}$$

then, working from $i = n-1$ back to $i = 1$, subtract from b_i the contributions from x_{i+1} to x_n, which have already been determined (and stored in b_{i+1} to b_n), and scale to determine x_i (and store the result in b_i):

$$b_i \leftarrow \left(b_i - \sum_{k=i+1}^{n} a_{ik} b_k\right)/a_{ii}. \tag{2.3b}$$

Once finished, the vector $\mathbf{b}$ contains the solution $\mathbf{x}$ of the original system $A\mathbf{x} = \mathbf{b}$. Generalization of this algorithm to account for multiple RHS vectors assembled into the matrix B is straightforward (see §2.1).

Efficient implementation of the above procedure is given in Algorithm 2.1. Note that the computations in this algorithm are performed **in place** using the existing A and B matrices only.

Operation count. The **leading-order computational cost** of an algorithm [written $\sim (cn^d)$ flops for specified values of c and d] is determined by finding an expression for the total number of flops required to complete the algorithm as a function of the order of the system, n, and retaining the term of this expression with the highest power of n, as this term dominates when n is large. Note that some texts only report the exponent of the leading-order computational cost [denoted $O(n^d)$]; it is often useful, however, to know when one algorithm is more expensive than another by an overall multiplicative constant (e.g., c) for a given large value of n.

We now calculate the leading-order computational cost of the Gaussian elimination algorithm[1] to solve the problem $A\mathbf{x} = \mathbf{b}$. The operations required for the forward sweep may be summarized as follows:

	divisions	multiplications	additions
To eliminate a_{21}:	1	n	n
To eliminate the entire first column:	$(n-1)$	$n(n-1)$	$n(n-1)$
To eliminate a_{32}:	1	$(n-1)$	$(n-1)$
To eliminate the entire second column:	$(n-2)$	$(n-1)(n-2)$	$(n-1)(n-2)$

... etc. Thus, summing over all of the $n-1$ columns over which the elimination is performed and applying the identities (B.63) and (B.64),

- The total number of divisions is $\sum_{k=1}^{n-1}(n-k) = n(n-1)/2$.
- The total number of multiplications is $\sum_{k=1}^{n-1}(n-k+1)(n-k) = (n^3-n)/3$.
- The total number of additions is $\sum_{k=1}^{n-1}(n-k+1)(n-k) = (n^3-n)/3$.

The leading-order computational cost for the forward sweep is thus $\sim (\frac{2}{3}n^3)$ flops.

The operation count for the back substitution process is straightforward:

- The total number of divisions is n.
- The total number of multiplications is $\sum_{k=1}^{n-1}(n-k) = n(n-1)/2$.
- The total number of additions is $\sum_{k=1}^{n-1}(n-k) = n(n-1)/2$.

The leading-order computational cost for the back substitution is thus $\sim (n^2)$ flops (much cheaper than the forward sweep!); the leading-order computational cost of the entire algorithm is therefore $\sim (\frac{2}{3}n^3)$ flops.

The LU decomposition. We now show that the forward sweep of the algorithm for Gaussian elimination without pivoting, as described above, inherently constructs the LU decomposition of A. Through several row operations, the matrix A is transformed by the Gaussian elimination procedure into an upper triangular form, which we call U. Furthermore, each row operation (which is simply the multiplication of one row by a number and adding the result to another row) may also be denoted by the premultiplication of A by a simple transformation matrix M_{ij}. It turns out that the transformation matrix which does the job at each step is simply an identity matrix with the (i,j)'th component replaced by m_{ij}. For example, if we define

$$M_{21} = \begin{pmatrix} 1 & & & & 0 \\ m_{21} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ 0 & & & & 1 \end{pmatrix}, \quad M_1 = \begin{pmatrix} 1 & & & & 0 \\ m_{21} & 1 & & & \\ m_{31} & & 1 & & \\ \vdots & & & \ddots & \\ m_{n1} & & & & 1 \end{pmatrix} = I + \mathbf{m}^1 (\mathbf{e}^1)^T \quad \text{with} \quad \mathbf{m}^1 = \begin{pmatrix} 0 \\ m_{21} \\ m_{31} \\ \vdots \\ m_{n1} \end{pmatrix},$$

then $M_{21}A$ means simply to multiply the first row of A by m_{21} and add it to the second row, which is exactly the first row operation performed in the process of Gaussian elimination without pivoting; the first $n-1$ such row operations in this algorithm are given simply by $M_1 A$, called a **Gauss transformation**. The entire

[1] Note that a careful implementation of the algorithm described above needs, for example, only n multiplications and n additions during the step that eliminates a_{21}. This is because we know (by construction of m_{21}) that $m_{21}a_{11} + a_{21} = 0$, so this multiplication and addition don't actually need to be performed when multiplying the first row by m_{21} and adding to the second row. A careful programmer should always be on the lookout to exploit such opportunities for improved efficiency.

Algorithm 2.1: Gaussian elimination without pivoting.

View Test

```
function [B,A] = Gauss(A,B,n)
% Solve AX=B for X using Gaussian elimination without pivoting.  The matrix B is replaced
% by the solution X on exit, and (if requested) the matrix A is replaced by the m_ij and U.
for j = 1:n-1,                                                    % FORWARD SWEEP
   % Looping through each column j<n, compute the m_ij=-a_ij/a_jj for j+1<=i<=n.
   % For efficiency, store these m_ij in the column of A below the pivot, where the a_ij
   % used to sit. This is done without disrupting the rest of the algorithm, as these a_ij
   % are set to zero by construction during this iteration and not referenced later.
   A(j+1:n,j)     = - A(j+1:n,j) / A(j,j);
   % Add the m_ij, for j+1<=i<=n, times the upper triangular part of the j'th row of the
   % augmented matrix to the rows j+1:n (below the pivot) in the augmented matrix.
   A(j+1:n,j+1:n) = A(j+1:n,j+1:n) + A(j+1:n,j) * A(j,j+1:n);    % (Outer product update)
   B(j+1:n,:)     = B(j+1:n,:)     + A(j+1:n,j) * B(j,:);
end
for i = n:-1:1,                                                   % BACK SUBSTITUTION
   B(i,:) = ( B(i,:) - A(i,i+1:n) * B(i+1:n,:) ) / A(i,i);        % (Inner product update)
end
end % function Gauss
```

forward sweep involves the premultiplication of A by $n-1$ such Gauss transformation matrices,

$$\underbrace{M_{n-1}\cdots M_2 M_1}_{M} A = U, \quad \text{where} \quad M_k = I + \mathbf{m}^k (\mathbf{e}^k)^T \quad \text{with} \quad \mathbf{m}^k = \begin{pmatrix} 0_{k\times 1} \\ m_{k+1,k} \\ \vdots \\ m_{n,k} \end{pmatrix}. \tag{2.4}$$

Note that

$$M_1^{-1} = \begin{pmatrix} 1 & & & 0 \\ -m_{21} & 1 & & \\ -m_{31} & & 1 & \\ \vdots & & & \ddots \\ -m_{n1} & & & 1 \end{pmatrix}$$

and, in general, $M_k^{-1} = I - \mathbf{m}^k (\mathbf{e}^k)^T$. As easily verified [via premultiplication by $M = M_{n-1}\cdots M_2 M_1$, where M_k is defined in (2.4)], we may write

$$M^{-1} = M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1} = \begin{pmatrix} 1 & & & & 0 \\ -m_{21} & 1 & & & \\ -m_{31} & -m_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ -m_{n1} & -m_{n2} & \cdots & -m_{n,n-1} & 1 \end{pmatrix}. \tag{2.5}$$

Defining $L = M^{-1}$ and noting that $MA = U$, it follows that $A = LU$, where U is upper triangular and L is unit lower triangular. We thus see that both L and U may be determined from the matrix that has replaced A after the forward sweep of the Gaussian elimination algorithm without pivoting. This algorithm thus demonstrates a curious **conservation of information** property: the information necessary to describe the LU decomposition of A takes precisely the same number of real or complex numbers as it takes to describe A itself. The following simple code segment, included in the test code provided with Algorithms 2.1 and 2.2, extracts L and U from the modified value of A returned by Algorithm 2.1:

Algorithm 2.2: Gaussian elimination without pivoting, leveraging a previously-computed LU decomposition.

View

```
function [B] = GaussLU(Amod,B,n)
% This function uses the LU decomposition returned (in the modified A matrix) by a prior
% call to Gauss.m to solve the system AX=B using forward and back substitution.
for j = 2:n,
   B(j,:) = B(j,:) + Amod(j,1:j-1) * B(1:j-1,:);                    % FORWARD SUBSTITUTION
end
for i = n:-1:1,
   B(i,:) = ( B(i,:) - Amod(i,i+1:n) * B(i+1:n,:) ) / Amod(i,i);    % BACK SUBSTITUTION
end
end % function GaussLU
```

```
L=eye(n); for i=2:n, for j=1:i-1, L(i,j)=-A(i,j); end, end
U=zeros(n); for i=1:n, for j=i:n, U(i,j)= A(i,j); end, end
```

As mentioned at the beginning of §2.2.1, the Gaussian elimination algorithm without pivoting will fail if it encounters a zero pivot. This is consistent with the fact that *a matrix A does not necessarily have an LU decomposition*. Consider, for example, the attempt to construct the following LU decomposition:

$$\underbrace{\begin{pmatrix}1 & 0\\ a & 1\end{pmatrix}}_{L}\underbrace{\begin{pmatrix}b & c\\ 0 & d\end{pmatrix}}_{U} = \begin{pmatrix}b & c\\ ab & ac+d\end{pmatrix} \stackrel{?}{=} \underbrace{\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}}_{A}.$$

For the equality in question to hold, $0 = b$ and $1 = ab$ must be satisfied simultaneously, which is impossible.

Further, *a matrix A may have infinitely many LU decompositions*. For example, the following LU decomposition is valid for any a:

$$\underbrace{\begin{pmatrix}1 & 0\\ a & 1\end{pmatrix}}_{L}\underbrace{\begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}}_{U} \stackrel{?}{=} \underbrace{\begin{pmatrix}0 & 0\\ 0 & 1\end{pmatrix}}_{A}.$$

Leveraging the LU decomposition to solve $A\mathbf{x} = \mathbf{b}$. Once we have determined the LU decomposition of A [e.g., after we run the full Gaussian elimination procedure once, which costs $\sim (\frac{2}{3}n^3)$ flops], we can solve a system with a new RHS $\mathbf{b}$ with a much less expensive algorithm by splitting the problem into two parts:

$$\left.\begin{aligned} L\mathbf{y} &= \mathbf{b}\\ U\mathbf{x} &= \mathbf{y}\end{aligned}\right\} \quad \Leftrightarrow \quad \underbrace{L(U\mathbf{x})}_{A\mathbf{x}} = \mathbf{b}. \tag{2.6}$$

As L is (unit lower) triangular, $L\mathbf{y} = \mathbf{b}$ can be solved for $\mathbf{y}$ in $\sim (n^2)$ flops via *forward* substitution. As U is (upper) triangular, once $\mathbf{y}$ is found, $U\mathbf{x} = \mathbf{y}$ can be solved for $\mathbf{x}$ in $\sim (n^2)$ flops via back substitution. Substituting $U\mathbf{x} = \mathbf{y}$ into $L\mathbf{y} = \mathbf{b}$ and noting that $A = LU$, we see that the value of $\mathbf{x}$ determined by this two-step process is identical to the value of $\mathbf{x}$ determined by solving $A\mathbf{x} = \mathbf{b}$ using Gaussian elimination, but at a significantly reduced computational cost [$\sim (2n^2)$ flops instead of $\sim (\frac{2}{3}n^3)$ flops]. Thus, if a numerical algorithm produces several problems of the form $A\mathbf{x} = \mathbf{b}$ in turn, for several vectors $\mathbf{b}$ but with A remaining fixed, it is quite beneficial to reuse the LU decomposition of A rather than repeatedly running the Gaussian elimination routine from scratch—for $n = 1000$, the LU-based approach is faster by a factor of over 300.

Note that the nontrivial components of L and U are collected in the modified A matrix returned by the Gaussian elimination algorithm. As illustrated in Algorithm 2.2, the code that solves the two back substitution problems in (2.6) may, in order to be efficient with memory, reference these elements directly in the modified A matrix, avoiding the unnecessary construction of separate, sparse L and U matrices.

Algorithm 2.3: Gaussian elimination with partial pivoting, and a convenient code for computing the inverse.

View Test

```
function [B,A,p] = GaussPP(A,B,n)
% This function solves AX=B for X using Gaussian elimination with partial pivoting.
% The matrix B is replaced by the solution X on exit, and (if requested) the matrix A
% is replaced by m_ij and U on exit, with the vector of pivots returned in p.
p=[1:n]';                                   % initialize permutation vector
for j = 1:n-1,                              % FORWARD SWEEP
   [amax,imax]=max(abs(A(j:n,j)));          % Find the largest element in the column.
   if amax > abs(A(j,j))                    % If necessary, exchange the rows of A along
      A([j j-1+imax],:)=A([j-1+imax j],:);  % with the rows of the previously-determined
      B([j j-1+imax],:)=B([j-1+imax j],:);  % m_ij (stored in the subdiagonal elements
      p([j j-1+imax])  =p([j-1+imax j]);    % of A), the rows of the RHS matrix B, and
   end                                      % the rows of the permutation vector p.
% --- THE REMAINDER OF THIS FUNCTION IS IDENTICAL TO Gauss.m ---
```

View Test

```
function [A]=Inv(A)
% Compute the inverse of a nonsingular matrix.
n=length(A); [A]=GaussPP(A,eye(n),n);
% end function Inv
```

2.2.2 Gaussian elimination with partial pivoting and the *PLU* decomposition

As you test Algorithm 2.1 on several matrices created with Matlab's random-number generator, you might be lulled into a false sense of security that pivoting isn't very important. However, a random-number generator rarely (if ever) produces a matrix with a zero pivot, which often appear in practice (that is, unless the system is **diagonally dominant**, as discussed in §2.2.4, or **Hermitian positive definite** or **Hermitian negative definite**, as discussed in §4.4.4.1). Beware that just because a routine works well on several randomly-generated matrices does not mean it will work well in general. For example, testing Algorithm 2.1 on a randomly-generated matrix with a_{11} set to zero (try it!) illustrates how this routine can easily fail.

Thus, the approach described in §2.2.1 is not always adequate, and the fix, as introduced in §2.1, is to exchange rows when necessary to ensure a nonzero pivot for each column. To maximize accuracy of the algorithm when using finite-precision arithmetic, we may in fact exchange rows before *each* Gauss transformation in order to maximize the magnitude of each pivot. This approach generally reduces the magnitude of the updates to the rows during each Gauss transformation, thereby reducing round-off error.

The result, Algorithm 2.3, is a straightforward modification of Algorithm 2.1. Before the k'th Gauss transformation, it checks the magnitudes of the lower triangular elements in the pivot column, $|a_{kk}|$ through $|a_{nk}|$, and exchanges rows (if necessary) in order to maximize the magnitude of the pivot. It then performs a Gauss transformation for that column as before, proceeds to the next column, and repeats. Such a procedure is referred to as **partial pivoting**, and ensures success of the Gaussian elimination algorithm if A is nonsingular.

Gaussian elimination with partial pivoting only requires a modest number of additional flops in order to calculate the magnitudes of the lower triangular elements in each pivot column, and thus has the same leading-order computational cost as Gaussian elimination without pivoting. However, the comparisons themselves (to find the maximum) and the subsequent row exchanges in memory can significantly slow the execution of the numerical code, and thus an algorithm that avoids pivoting (like Algorithm 2.1), if it can be justified (see, e.g., §2.2.4 and §4.4.4.1), is often preferred for reasons of efficiency.

The *PLU* decomposition[2]. Each row exchange in the partial pivoting algorithm described above may be represented in matrix form via premultiplication by a (symmetric) permutation matrix P_k^T. For example,

[2] Some authors and numerical codes, including those in the built-in Matlab toolboxes, choose to define this decomposition as $A = P^T LU$ (that is, as $PA = LU$). The notational convention used in this text, $A = PLU$ (that is, $P^T A = LU$), was chosen in order to be consistent with the several other matrix decompositions presented in §4.

Algorithm 2.4: Gaussian elimination with partial pivoting, leveraging a *PLU* decomposition.

View

```
function [C] = GaussPLU(Amod,B,p,n)
% This function uses the PLU decomposition returned (in the modified A and p) by a prior
% call to GaussPP to solve the system AX=B using forward / back substitution.
for j=1:n, C(j,:)=B(p(j),:); end, [C] = GaussLU(Amod,C,n);
end % function GaussPLU
```

if the k'th and j'th rows are to be exchanged before the k'th Gauss transformation, then P_k^T is defined as the permutation matrix formed by exchanging the k'th and j'th columns of the identity matrix. [As the P_k matrices are involutory (see §1.2.5), we refer to P_k^T as P_k in the discussion that follows.] This row exchange is followed immediately by premultiplication by a Gauss transformation matrix M_k, which, as discussed above, has the effect of adding the correct multiple of the k'th row to the rows below the k'th in order to eliminate the lower-triangular elements in the column below the pivot. The sequence of operations applied may thus be represented in matrix notation in a straightforward manner. By appropriate definition of modified Gauss transformation matrices $M_{k,j}$ for $j > k$, all of the $n-1$ permutation matrices P_k may then be "shifted" to the right in order to group them into overall M and P matrices, as follows:

$$M_{n-1}P_{n-1}M_{n-2}P_{n-2}\cdots M_3P_3M_2\underbrace{P_2M_1}_{M_{1,2}P_2}P_1A = U, \quad \text{where} \quad P_2M_1 = M_{1,2}P_2 \quad \Leftrightarrow \quad M_{1,2} = P_2M_1P_2$$

$$\Rightarrow \quad M_{n-1}P_{n-1}M_{n-2}P_{n-2}\cdots M_3\underbrace{P_3M_2}_{M_{2,3}P_3}M_{1,2}P_2P_1A = U, \quad \text{where} \quad P_3M_2 = M_{2,3}P_3 \quad \Leftrightarrow \quad M_{2,3} = P_3M_2P_3$$

$$\Rightarrow \quad M_{n-1}P_{n-1}M_{n-2}P_{n-2}\cdots M_3M_{2,3}\underbrace{P_3M_{1,2}}_{M_{1,3}P_3}P_2P_1A = U, \quad \text{where} \quad P_3M_{1,2} = M_{1,3}P_3 \quad \Leftrightarrow \quad M_{1,3} = P_3M_{1,2}P_3$$

$$\Rightarrow \quad M_{n-1}P_{n-1}M_{n-2}P_{n-2}\cdots M_3M_{2,3}M_{1,3}P_3P_2P_1A = U, \quad \text{etc.}$$

Thus, if we take $M_{k,k} = M_k$ and define $M_{k,j} = P_jM_{k,j-1}P_j$ for $j > k$, we ultimately arrive at

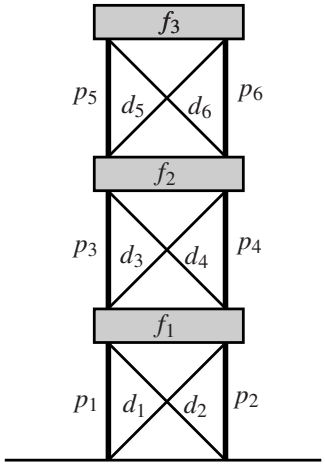
$$\underbrace{M_{n-1}M_{n-2,n-1}\cdots M_{3,n-1}M_{2,n-1}M_{1,n-1}}_{M}\underbrace{P_{n-1}P_{n-2}\cdots P_3P_2P_1}_{P^T}A = U \quad \Rightarrow \quad P^TA = LU \quad \text{where} \quad L = M^{-1}.$$

The matrix P^T that results from this procedure is itself just an (asymmetric) permutation matrix that may be constructed by starting from the identity matrix and applying, in order, all of the row exchanges that are also applied to A. Remarkably, as may be verified by its recursive definition, the matrix $M_{k,n-1}$ (for any k) may also be constructed in a simple fashion, simply by applying all of the subsequent row exchanges (that is, those preceding the $k+1$'th to n'th Gauss transformations) to the vector $\mathbf{m}^k$ used to build the Gauss transformation matrix M_k. Note that $L = M^{-1}$ may then be computed by the simple formula given in (2.5) and, as it is a permutation matrix, $P^{-1} = P^T$. We may thus write $MP^TA = U$ as $P^TA = LU$ or $A = PLU$.

Leveraging the *PLU* decomposition to solve $A\mathbf{x} = \mathbf{b}$. Since $A\mathbf{x} = \mathbf{b}$ and $A = PLU$, we may write $L(U\mathbf{x}) = P^T\mathbf{b} = \mathbf{c}$. Thus, the solution procedure leveraging a *PLU* decomposition is analogous to the algorithm leveraging an unpivoted *LU* decomposition: first, calculate $\mathbf{c} = P^T\mathbf{b}$ and solve $L\mathbf{y} = \mathbf{c}$ for $\mathbf{y}$, then solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. The leading-order computational cost is the same as without the permutation, $\sim (2n^2)$ flops; the only modification required is a reordering of the $\mathbf{b}$ vector by premultiplying it by the matrix P^T.

We don't need the entire permutation matrix P to reorder the $\mathbf{b}$ vector appropriately. The information we need may be extracted from a **permutation vector**, $\mathbf{p}$, that may be determined by initializing $p_k = k$, then applying, in order, all of the row exchanges that are also applied to A. The permutation vector so constructed contains, as its k'th element, the row of the nonzero entry in the k'th column of P (that is, p_k is the column of the nonzero entry in the k'th row of P^T). Based on this permutation vector, the necessary reordering of the $\mathbf{b}$ vector that is equivalent to calculating $\mathbf{c} = P^T\mathbf{b}$ is easy to obtain using $\mathbf{p}$, as shown in Algorithm 2.4.

Example 2.1 Static forces in a pretensioned three-story structure



vertical forces:

$$\begin{cases} p_5 - \alpha d_6 = mg/2 \\ p_6 - \alpha d_5 = mg/2 \\ p_3 - p_5 - \alpha(d_4 - d_5) = mg/2 \\ p_4 - p_6 - \alpha(d_3 - d_6) = mg/2 \\ p_1 - p_3 - \alpha(d_2 - d_3) = mg/2 \\ p_2 - p_4 - \alpha(d_1 - d_4) = mg/2 \end{cases}$$

horizontal forces:

$$\begin{cases} \alpha d_6 - f_3 = 0 \\ -\alpha d_5 + f_3 = 0 \\ \alpha(d_4 + d_5) - f_2 = 0 \\ -\alpha(d_3 + d_6) + f_2 = 0 \\ \alpha(d_2 + d_3) - f_1 = 0 \\ -\alpha(d_1 + d_4) + f_1 = 0 \end{cases}$$

Figure 2.1: (a) A simple three-story building, and (b) the equations governing the static equilibrium of this structure, where $\alpha = \sin\pi/4 = \cos\pi/4$, $m = 1000$ kg (the mass of each floor), and $g = 9.8$ m/sec^2.

A typical application of Gaussian elimination is to compute the static forces in a structure, such as the pillars p_i, diagonals d_i, and floors f_i of the simple three-story building illustrated in Figure 2.1a. We will set a nominal pretension load of 1000 N in the diagonals $\{d_1, d_3, d_5\}$. Setting the net vertical (positive up) and horizontal (positive right) forces to zero at the pins at each end of floors f_3, f_2, and f_1 then results in 12 equations, which may be solved for the 9 unknown compressive forces $\{p_1, p_2, p_3, p_4, p_5, p_6, f_1, f_2, f_3\}$ and the 3 unknown tension forces $\{d_2, d_4, d_6\}$, as listed in Figure 2.1b.

The 12 linear equations listed in Figure 2.1b may be written in matrix form as

$$\underbrace{\begin{pmatrix} & & & 1 & & & & & & & & -\alpha \\ & & & & 1 & & & & & & & \\ & & 1 & & -1 & & & & & -\alpha & & \\ & & & 1 & & -1 & & & & & \alpha & \\ 1 & & -1 & & & & & & -\alpha & & & \\ & 1 & & -1 & & & & & & \alpha & & \\ & & & & & & & -1 & & & \alpha & \\ & & & & & & & 1 & & & & \\ & & & & & & -1 & & & \alpha & & \\ & & & & & & 1 & & & & & -\alpha \\ & & & & & -1 & & & \alpha & & & \\ & & & & & 1 & & & & -\alpha & & \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ f_1 \\ f_2 \\ f_3 \\ d_2 \\ d_4 \\ d_6 \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} mg/2 \\ mg/2 + \alpha d_5 \\ mg/2 - \alpha d_5 \\ mg/2 + \alpha d_3 \\ mg/2 - \alpha d_3 \\ mg/2 + \alpha d_1 \\ 0 \\ \alpha d_5 \\ -\alpha d_5 \\ \alpha d_3 \\ -\alpha d_3 \\ \alpha d_1 \end{pmatrix}}_{\mathbf{b}}$$

Solving (for Matlab code, see here) gives:

$$\begin{gathered} p_1 = p_2 = 15407 \text{ N}, \quad p_3 = p_4 = 10507 \text{ N}, \quad p_5 = p_6 = 5607 \text{ N}, \\ f_1 = f_2 = 1414 \text{ N}, \quad f_3 = 707 \text{ N}, \\ d_2 = d_4 = d_6 = 1000 \text{ N}; \end{gathered} \tag{2.7}$$

as expected, the nominal forces in this case are seen to be symmetric from left to right. The dynamics of this structure are considered in Example 17.9 and Exercise 17.9.

Algorithm 2.5: Gaussian elimination with complete pivoting.

View
Test

```
function [X,A,p,q] = GaussCP(A,B,n)
% This function solves AX=B for X using Gaussian elimination with complete pivoting.
% The solution X is returned on exit, and (if requested) the matrix A
% is replaced by m_ij and U on exit, with the vectors of pivots returned in p and q.
p=[1:n]';  q=[1:n]';                                % initialize permutation vectors
for j = 1:n-1,                                      % FORWARD SWEEP
   [treal,tint]=max(abs(A(j:n,j:n)));               % Find the largest element in A(j:n,j:n)
   [amax,jmax]=max(treal); imax=tint(jmax); acurrent=abs(A(j,j));
   if imax > j & amax > acurrent                    % Exchange the rows of A along with the
      A([j j-1+imax],:)=A([j-1+imax j],:);          % rows of the previously-determined m_ij
      B([j j-1+imax],:)=B([j-1+imax j],:);          % (stored in the lower triangle of A),
      p([j j-1+imax])  =p([j-1+imax j]);            % the rows of the RHS matrix B, and
   end                                              % the rows of the permutation vector p.
   if jmax > j & amax > acurrent
      A(:,[j j-1+jmax])=A(:,[j-1+jmax j]);          % Then, exchange the columns of A and
      q([j j-1+jmax])  =q([j-1+jmax j]);            % the rows of the permutation vector q.
   end
% --- THIS LINE IS FOLLOWED BY THE REMAINDER OF Gauss.m ---
% ...
% --- AFTER THE REMAINDER OF Gauss.m, ONE MORE LINE OF CODE FOLLOWS TO DESCRAMBLE B ---
for j=1:n, X(q(j),:)=B(j,:); end
end % function GaussCP
```

Algorithm 2.6: Gaussian elimination with complete pivoting, leveraging a $PLUQ^T$ decomposition.

View

```
function [B] = GaussPLUQT(Amod,B,p,q,n)
% This function uses the PLUQ'T decomposition returned (in the modified A, p, and q) by a
% prior call to GaussCP to solve the system AX=B using forward / back substitution.
for j=1:n, C(j,:)=B(p(j),:); end, [C]=GaussLU(Amod,C,n);  for j=1:n, B(q(j),:)=C(j,:); end
end % function GaussPLUQT
```

2.2.3 Gaussian elimination with complete pivoting and the $PLUQ^T$ decomposition

Gaussian elimination with partial pivoting is a very effective algorithm. Recall that partial pivoting involves exchanging the top row of the reduced augmented matrix [illustrated by the box in (2.2)] with any of the rows below it in order to maximize the magnitude of the pivot (the element in the upper-left corner of the reduced augmented matrix). Partial pivoting can sometimes improve the numerical accuracy of the Gaussian elimination algorithm when using finite-precision arithmetic even if pivoting is not required to prevent division by zero, as pivoting tends to reduce the occurrence of the subtraction of two numbers that are almost identical in the Gaussian elimination procedure, thereby reducing the error introduced by the use of finite-precision arithmetic. This idea of reordering the problem to improve numerical accuracy may be extended by exchanging the *columns*, in addition to the rows, in order to maximize each pivot. This is known as **complete pivoting**.

The $PLUQ^T$ decomposition. Each column exchange in the complete pivoting algorithm described above may be represented in matrix form via postmultiplication by a permutation matrix Q_k. For example, if the k'th and j'th columns are to be exchanged before the k'th Gauss transformation, then Q_k is defined as the symmetric permutation matrix formed by exchanging the k'th and j'th columns of the identity matrix. The process of Gaussian elimination with complete pivoting is then easily represented in matrix form as

$$M_{n-1}P_{n-1}\cdots M_2P_2M_1P_1AQ_1Q_2\cdots Q_{n-1} = U.$$

As with the PLU decomposition, the permutation matrices may be shifted to the right using the modified Gauss transformation matrices $M_{k,j}$ introduced previously, thereby leading to a result that may be written in

matrix form as $MP^TAQ = U$, and thus $P^TAQ = LU$ or $A = PLUQ^T$, where $L = M^{-1}$ and $Q = Q_1Q_2\cdots Q_{n-1}$. Implementation in numerical code is again straightforward (see Algorithm 2.5).

Leveraging the $PLUQ^T$ decomposition to solve $A\mathbf{x} = \mathbf{b}$. Since $A\mathbf{x} = \mathbf{b}$ and $A = PLUQ^T$, we may write $LU(Q^T\mathbf{x}) = P^T\mathbf{b}$. Thus, the solution procedure leveraging a $PLUQ^T$ decomposition is analogous to the algorithm leveraging a PLU decomposition, as discussed previously: first, calculate $\mathbf{c} = P^T\mathbf{b}$ and solve $L\mathbf{y} = \mathbf{c}$ for $\mathbf{y}$, then solve $U\mathbf{z} = \mathbf{y}$ for $\mathbf{z}$; the desired answer is then given by a simple reordering of the result, as $Q^T\mathbf{x} = \mathbf{z}$. The leading-order computational expense is the same as without both permutations, $\sim (2n^2)$ flops.

As with P, we don't need the entire permutation matrix Q in order to reorder the $\mathbf{z}$ vector appropriately. In fact, the information we need may be extracted from a permutation vector, denoted $\mathbf{q}$, initialized as $q_k = k$ then permuted in the same manner (and order) as the permutations that are also applied to the columns of A. The permutation vector so constructed contains, as its k'th element, the row of the nonzero entry in k'th column of Q. Based on this permutation vector, the necessary reordering of the $\mathbf{z}$ vector that is equivalent to solving $Q^T\mathbf{x} = \mathbf{z}$ for $\mathbf{x}$ (i.e., calculating $\mathbf{x} = Q\mathbf{z}$) is easy to obtain using $\mathbf{q}$, as shown in Algorithm 2.6.

Note that Algorithm 2.5 suffers from the considerable computational expense involved with identifying the element of an $(n-j+1)\times(n-j+1)$ matrix with the maximum absolute value during the determination of the j'th pivot. Except for particularly poorly scaled problems, complete pivoting does not usually improve the solution that much beyond that obtained with partial pivoting. In practice, Gaussian elimination with partial pivoting is found to be quite adequate for most problems typically encountered (except certain contrived test cases that might be labelled as **pathological**). Thus, the computational expense involved with complete pivoting is rarely justified in large-scale systems.

Pivoting summary. Though several algebraic manipulations of sparse matrices were performed in order to *derive* both the partial pivoting algorithm (§2.2.2) and the complete pivoting algorithm (§2.2.3), their ultimate *implementation* in numerical code (see Algorithms 2.3 and 2.5) are only minor modifications of the `Gauss.m` code developed in §2.2.1. This is typical in the derivation of efficient numerical methods in general: though the derivation and analysis of such methods is sometimes algebraically involved, various "tricks" may usually be identified in order to simplify the resulting algorithms and minimize their computational expense (in terms of both flops and memory usage) when implemented in numerical code.

2.2.4 Diagonal dominance: justification for Gaussian elimination without pivoting

A **diagonally dominant** matrix is defined as a square matrix A for which the magnitude of each diagonal element is greater than or equal to the sum of the magnitude of the other elements on that row, that is,

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \quad \text{for } i = 1,2,\ldots,n. \tag{2.8}$$

A **strictly diagonally dominant** matrix is defined by the same relation with a strict inequality sign ($>$).

Fact 2.2 *If A^T is diagonally dominant, then Gaussian elimination with partial pivoting (Algorithm 2.3) applied to the matrix $A_{n\times n}$ results in no row exchanges, thereby determining $A = LU$. This decomposition may thus be determined using Gaussian elimination* without *pivoting (Algorithm 2.1). Further, if A^T is strictly diagonally dominant, then A is nonsingular with nonzero diagonal elements.*

Proof (by induction[3]): The statement holds trivially for the case of order $n = 1$. Assume it holds for the case of order $n-1$, and now consider the case of order n. Partition the matrix A into blocks such that

[3]This proof uses **mathematical induction**, a procedure which proceeds as follows: we first establish that the statement holds for a "base case" (in this problem, of order $n = 1$). Next, we assume the statement holds for a case of order $n-1$, then establish that it follows directly that the statement must also hold for a case of order n, thereby establishing that the statement holds for all orders $n \geq 1$.

$$A = \begin{bmatrix} \alpha & \mathbf{w}^T \\ \mathbf{v} & C \end{bmatrix},$$

where $C = C_{(n-1)\times(n-1)}$. As A^T is assumed to be diagonally dominant, it follows that

$$|\alpha| \geq \sum_{i=1}^{n-1} |v_i| \quad \text{and} \quad |c_{jj}| \geq |w_j| + \sum_{\substack{i=1 \\ i\neq j}}^{n-1} |c_{ij}|. \tag{2.9}$$

Thus, $|\alpha| \geq |v_i|$ for all i, and therefore the first step of the Gaussian elimination algorithm with partial pivoting will, in fact, not pivot. If $\alpha = 0$, then $v_i = 0$ for all i, and the problem reduces immediately to the case of order $n-1$ (assumed true via the induction hypothesis); if $\alpha \neq 0$, then applying the first Gauss transformation of the Gaussian elimination algorithm to A results in

$$\underbrace{\begin{bmatrix} 1 & 0 \\ -\mathbf{v}/\alpha & I \end{bmatrix}}_{M_1} A = \begin{bmatrix} \alpha & \mathbf{w}^T \\ 0 & B \end{bmatrix} \quad \text{where} \quad B = C - \frac{\mathbf{v}\mathbf{w}^T}{\alpha}. \tag{2.10}$$

We now determine whether or not B^T is diagonally dominant. For any j, we have

$$\sum_{\substack{i=1 \\ i\neq j}}^{n-1} |b_{ij}| = \sum_{\substack{i=1 \\ i\neq j}}^{n-1} |c_{ij} - v_i w_j/\alpha| \leq \sum_{\substack{i=1 \\ i\neq j}}^{n-1} |c_{ij}| + \frac{|w_j|}{|\alpha|} \sum_{\substack{i=1 \\ i\neq j}}^{n-1} |v_i| \leq (|c_{jj}| - |w_j|) + \frac{|w_j|}{|\alpha|}(|\alpha| - |v_j|) \leq \left| c_{jj} - \frac{w_j v_j}{\alpha} \right| = |b_{jj}|.$$

Thus, B^T is diagonally dominant. It follows by the induction hypothesis that all subsequent steps of the Gaussian elimination algorithm with partial pivoting will not pivot either, and that an LU decomposition of B may be constructed such that $B = L_1 U_1$. By (2.10), we conclude that

$$A = \underbrace{\begin{bmatrix} 1 & 0 \\ \mathbf{v}/\alpha & I \end{bmatrix}}_{M_1^{-1}} \begin{bmatrix} \alpha & \mathbf{w}^T \\ 0 & L_1 U_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \mathbf{v}/\alpha & L_1 \end{bmatrix} \begin{bmatrix} \alpha & \mathbf{w}^T \\ 0 & U_1 \end{bmatrix} = LU.$$

If A is strictly diagonally dominant, then strict inequalities apply in (2.9), and thus $\alpha \neq 0$; by the induction hypothesis, it follows that all of the diagonal elements of U are nonzero, and thus A is nonsingular. If A is strictly diagonally dominant, it follows trivially that the diagonal elements of A are nonzero. □

2.3 Exploiting structured sparsity

We now consider the problem of solving $A\mathbf{x} = \mathbf{g}$ for $\mathbf{x}$ when A is sparse, tightly banded, and diagonally dominant (so row exchanges are not required). The algorithm to be presented is exactly the Gaussian elimination algorithm without pivoting, as presented in §2.2, capitalizing as much as possible on the sparsity structure of A to eliminate unnecessary calculations. We will demonstrate this idea for tridiagonal A, in which case the algorithm is referred to as the **Thomas algorithm**. This algorithm, which costs $\sim (8n)$ flops [or, $\sim (5n)$ flops if the LU decomposition is used], is remarkably efficient. Straightforward generalizations of this idea produce similarly efficient simplifications of Gaussian elimination without pivoting for other sparsity patterns, including: pentadiagonal (Exercise 2.3), tridiagonal circulant (Algorithm 2.10), arrow (Exercise 2.4), and n coupled tridiagonal systems (Exercise 2.6).

The following notation is used for the augmented matrix in this problem:

$$(A\,|\,\mathbf{g}) = \left(\begin{array}{ccccccc|c} b_1 & c_1 & & & & & 0 & g_1 \\ a_2 & b_2 & c_2 & & & & & g_2 \\ & a_3 & b_3 & c_3 & & & & g_3 \\ & & \ddots & \ddots & \ddots & & & \vdots \\ & & & a_{n-1} & b_{n-1} & c_{n-1} & & g_{n-1} \\ 0 & & & & a_n & b_n & & g_n \end{array}\right).$$

Forward sweep, step 1. Eliminate the element below b_1 (the first pivot) in the first column:
Let $m_2 = -a_2/b_1$. Multiply the first row by m_2 and add to the second row.

Forward sweep, step 2. Repeat step 1 for the new (smaller) augmented matrix, as in the original Gaussian elimination procedure. The pivot for the second column is b_2.

... etc. After $n-1$ steps, the modified augmented matrix takes the form

$$\left(\begin{array}{ccccccc|c} b_1 & c_1 & & & & & 0 & g_1 \\ 0 & b_2 & c_2 & & & & & g_2 \\ & 0 & b_3 & c_3 & & & & g_3 \\ & & \ddots & \ddots & \ddots & & & \vdots \\ & & & 0 & b_{n-1} & c_{n-1} & & g_{n-1} \\ 0 & & & & 0 & b_n & & g_n \end{array}\right),$$

where all of the elements of $\mathbf{b}$ and $\mathbf{g}$ have changed except those in the first row.

Back substitution. The process of back substitution is again straightforward. As before, initiate the back substitution with:

$$g_n \leftarrow g_n/b_n; \tag{2.11a}$$

then, working from $i = n-1$ back to $i = 1$, subtract from g_i the contribution from x_{i+1}, which has already been determined (and stored in g_{i+1}), and scale to determine x_i:

$$g_i \leftarrow \Big(g_i - c_i\, g_{i+1}\Big)/b_i. \tag{2.11b}$$

Once finished, the vector $\mathbf{g}$ contains the solution $\mathbf{x}$ of the original system $A\mathbf{x} = \mathbf{g}$.

Efficient implementation of this procedure, generalized to multiple RHS vectors assembled in G, is given in Algorithm 2.7.

Operation count. We now calculate the total flops required by the Thomas algorithm to solve the problem $A\mathbf{x} = \mathbf{b}$ in the case of tridiagonal A, being careful to perform floating point operations only where necessary.
The operations required for the forward sweep may be summarized as follows:

	divisions	multiplications	additions
To eliminate a_2:	1	2	2
To eliminate entire subdiagonal:	$(n-1)$	$2(n-1)$	$2(n-1)$

The leading-order computational cost for the forward sweep is thus $\sim (5n)$ flops. For the back substitution:

- The total number of divisions is n.
- The total number of multiplications is $n-1$.
- The total number of additions is $n-1$.

Algorithm 2.7: The Thomas algorithm.

View
Test

```
function [G,a,b,c] = Thomas(a,b,c,G,n)
% This function solves AX=G for X using the Thomas algorithm, where A = tridiag(a,b,c).
% The solution X is returned on exit, and (if requested), the three diagonals of A are
% replaced by the m_ij and U.
for j = 1:n-1,                                  % FORWARD SWEEP
   a(j+1)   = - a(j+1) / b(j);                  % This code is just a simplified version of
   b(j+1)   = b(j+1)   + a(j+1)*c(j);           % Gauss.m (using a different notation for the
   G(j+1,:) = G(j+1,:) + a(j+1)*G(j,:);         % matrix and RHS vector), to which the reader
end                                             % is referred for explanatory comments.
G(n,:) = G(n,:) / b(n);                         % BACK SUBSTITUTION
for i = n-1:-1:1,
   G(i,:) = ( G(i,:) - c(i) * G(i+1,:) ) / b(i);
end
end % function Thomas
```

Algorithm 2.8: The Thomas algorithm, leveraging an *LU* decomposition.

View

```
function [G] = ThomasLU(a,b,c,G,n)
% This function uses the LU decomposition returned [in the modified (a,b,c) vectors] by a
% prior call to Thomas.m to solve the system AX=G using forward / back substitution.
for j = 1:n-1,
   G(j+1,:) = G(j+1,:) + a(j+1)*G(j,:);                 % FORWARD SUBSTITUTION
end
G(n,:) = G(n,:) / b(n);
for i = n-1:-1:1,
   G(i,:) = ( G(i,:) - c(i) * G(i+1,:) ) / b(i);        % BACK SUBSTITUTION
end
end % function ThomasLU
```

Algorithm 2.9: Thomas for tridiagonal Toeplitz matrices (cf. Algorithm 2.7).

View
Test

```
function [G] = ThomasTT(a,b,c,G,n)
% Solves the system AX=G for X using the Thomas algorithm, assuming A is tridiagonal,
% Toeplitz, and diagonally dominant, with (a,b,c) the scalars on the subdiagonal, main
% diagonal, and superdiagonal of A.  On exit, the matrix G is replaced by the solution X.
bt(1)=b;
for j = 1:n-1,                          % FORWARD SWEEP
   m = - a / bt(j);                     % A temporary vector bt (of length n) is created by
   bt(j+1)   = b + m * c;               % this algorithm, but it doesn't store enough to
   G(j+1,:) = G(j+1,:) + m * G(j,:);    % later reconstruct the LU decomposition of A.
end
G(n,:) = G(n,:) / bt(n);                % BACK SUBSTITUTION
for i = n-1:-1:1,
   G(i,:) = ( G(i,:) - c * G(i+1,:) ) / bt(i);
end
end % function ThomasTT
```

The leading-order computational cost for the back substitution is thus $\sim (3n)$ flops. The leading-order computational cost of the entire algorithm is $\sim (8n)$ flops—*much* less expensive than full Gaussian elimination.

Leveraging the *LU* decomposition to solve $A\mathbf{x} = \mathbf{g}$ for tridiagonal A. As in the Gaussian elimination procedure from which it was derived, the forward sweep of the Thomas algorithm inherently constructs an *LU* decomposition of the tridiagonal matrix A. Note that L and U inherit the banded structure of A; specifically, L is unit lower bidiagonal and U is upper bidiagonal.

Once we have the *LU* decomposition of A, we can solve a system with a new RHS $A\mathbf{x} = \mathbf{g}$ with a two-step

Algorithm 2.10: Reduced Gaussian elimination for circulant matrices.

View Test

```
function [G,a,b,c,d,e] = Circulant(a,b,c,G,n)
% This function solves the tridiagonal circulant system AX=G for X.  On exit, the matrix G
% is replaced by the solution X and (if requested), {a,b,c,d,e} contain the m_ij and U.
d(1) = a(1);    e(1) = c(n);                % Initialize d and e vectors
for j = 1:n-2,                              % FORWARD SWEEP
   a(j+1)   = - a(j+1) / b(j);
   b(j+1)   = b(j+1)   + a(j+1)*c(j);
   d(j+1)   = a(j+1) * d(j);
   G(j+1,:) = G(j+1,:) + a(j+1)*G(j,:);
   e(j)     = - e(j) / b(j);
   e(j+1)   = e(j) * c(j);
   b(n)     = b(n)  + e(j) * d(j);
   G(n,:)   = G(n,:) + e(j)*G(j,:);
end
d(n-1) = d(n-1) + c(n-1);                   % Fix d and e vectors in their n-1 components.
e(n-1) = e(n-1) + a(n);
a(n)   = - e(n-1) / b(n-1);                 % Now handle j=n-1 case of the loop seperately,
b(n)   = b(n) + a(n) * d(n-1);              % as variables have different names in the corner.
G(n,:) = G(n,:) + a(n) * G(n-1,:);
G(n,:)   = G(n,:) / b(n);                   % BACK SUBSTITUTION
G(n-1,:) = ( G(n-1,:) - d(n-1) * G(n,:) ) / b(n-1);
for i = n-2:-1:1,
   G(i,:) = ( G(i,:) - c(i) * G(i+1,:) - d(i) * G(n,:) ) / b(i);
end
end % function Circulant
```

Algorithm 2.11: Reduced Gaussian elimination for circulant matrices, leveraging an LU decomposition.

View

```
function [G] = CirculantLU(a,b,c,d,e,G,n)
% This function uses the LU decomposition returned [in the (a,b,c,d,e) vectors] by a
% prior call to Circulant.m to solve the system AX=G using forward / back substitution.
for j = 1:n-1,
   G(j+1,:) = G(j+1,:) + a(j+1)*G(j,:);                    % FORWARD SUBSTITUTION
   G(n,:)   = G(n,:) + e(j)*G(j,:);
end
G(n,:) = G(n,:) - e(n-1) * G(n-1,:);
G(n,:) = G(n,:) / b(n);                                    % BACK SUBSTITUTION
G(n-1,:) = ( G(n-1,:) - d(n-1) * G(n,:) ) / b(n-1);
for i = n-2:-1:1,
   G(i,:) = ( G(i,:) - c(i) * G(i+1,:) - d(i) * G(n,:) ) / b(i);
end
end % function CirculantLU
```

procedure as before. The cost of efficiently solving $L\mathbf{y} = \mathbf{g}$ for the vector $\mathbf{y}$ is $\sim (2n)$ flops (similar to the cost of the back substitution in the Thomas algorithm, but noting that the divisions are not required because the diagonal elements are unity). The cost of efficiently solving $U\mathbf{x} = \mathbf{y}$ for the vector $\mathbf{x}$ is $\sim (3n)$ flops (the same as the cost of back substitution in the Thomas algorithm). Thus, solving $A\mathbf{x} = \mathbf{g}$ by leveraging the LU decomposition of A costs $\sim (5n)$ flops, whereas solving it by the Thomas algorithm costs $\sim (8n)$ flops. Efficient implementation of this procedure is given in Algorithm 2.8.

Thomas for tridiagonal Toeplitz. As shown in Algorithm 2.9, if A is tridiagonal Toeplitz, and one does not aspire to later reconstruct the LU decomposition of A, then one can rewrite the Thomas algorithm using just a single vector for the main diagonal of A. Reducing storage requirements in this manner can often decrease execution time significantly by fitting the algorithm into a (smaller) higher speed cache on the CPU.

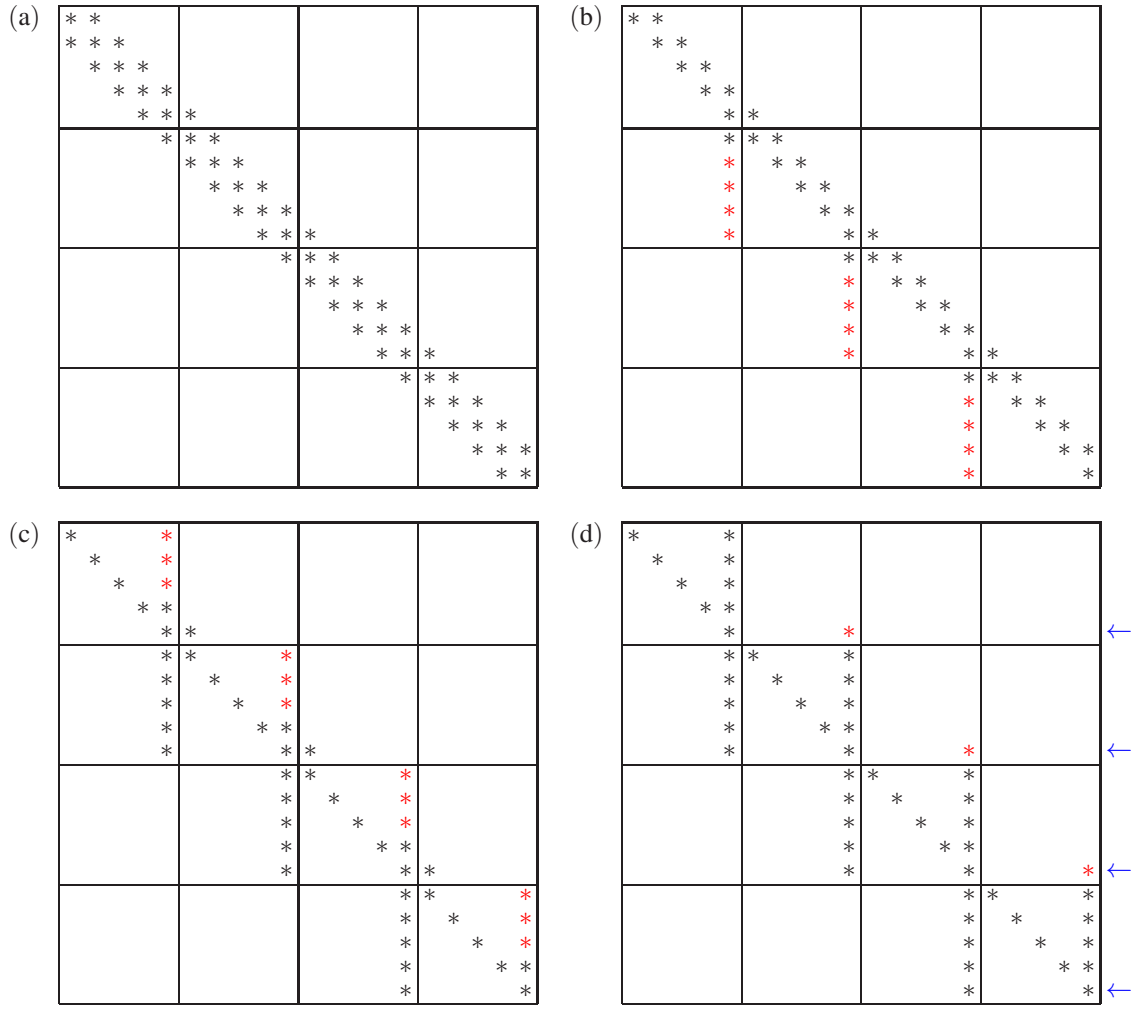


Figure 2.2: The essential steps of the parallel Thomas algorithm.

Reducing Gaussian elimination for matrices with other sparsity patterns. The idea of exploiting banded sparsity in order to streamline the full Gaussian elimination algorithm extends easily to other banded matrices (for the pentadiagonal case, see Exercise 2.3). This idea can also be extended to matrices of arrow structure. Note in particular Algorithm 2.10, in which a tridiagonal circulant matrix essentially fills out into an arrow matrix as it is solved with a specialized form of reduced Gaussian elimination. This algorithm requires $\sim (17n)$ flops, and enough information may be saved such that the LU decomposition may be reconstructed and used, as illustrated in Algorithm 2.11, thus reducing the leading-order computational cost to $\sim (9n)$ flops.

2.4 Parallelization

As discussed further in §12, modern computers achieve their speed by **parallelization** (that is, by the simultaneous calculation of many of the floating-point operations in the algorithm). Each step of the Thomas algorithm depends upon the result of the previous step, and thus the algorithm as described thus far does not

readily parallelize[4]. However, with some additional ingenuity (and flops), a parallel version of the Thomas algorithm may indeed be crafted. The essential steps of the **parallel Thomas algorithm**[5] (Wang 1981) are indicated in Figure 2.2. Starting from a tridiagonal matrix (see Figure 2.2a) of length n (in the example illustrated, $n = 20$), the problem is split into p blocks of length $m = n/p$ and distributed on p processors (in the example illustrated, $p = 4$ and $m = 5$). As the $p \times p$ block matrix so created is not quite block diagonal, these p problems are coupled; however, this coupling can be accounted for fairly inexpensively, as shown below. In a typical application of this algorithm, n is huge, and p is the number of processors available to do the computational work. Also, n is not necessarily an integer multiple of p, so some care is needed to account for the fact that some processors need to do more computations than others.

On each processor, a Thomas-like forward sweep is first performed in order to zero out the first subdiagonal in each diagonal block (see Figure 2.2b). These forward sweeps are done in parallel on the p processors; on $p-1$ of the processors, an extra column fills up during these forward sweeps. A *back* sweep, analogous to the forward sweep, is also introduced to zero out elements in the first superdiagonal of each block, from the element above the $(m-1)$'th diagonal element back to the element above the second diagonal element in each block (see Figure 2.2c). These back sweeps are also done in parallel on the p processors; on all of the processors, an extra column also fills up during these back sweeps. Finally, on the last $p-1$ of the processors, each back sweep is extended one extra step into the corresponding preceding block (see Figure 2.2d). The m'th equation on each processor (see arrows in Figure 2.2d) is now be lumped together into a single, tridiagonal, fairly small (that is, $p \times p$) problem and solved (on the "master" processor) with the ordinary Thomas algorithm. Once the m'th variable in each block is determined in this fashion, the remaining variables in each block are determined via straightforward back substitution. The leading-order computational cost of this algorithm is $\sim (17n)$ [about twice that of the standard Thomas algorithm].

An implementation of the parallel Thomas algorithm described above is given in Algorithm 2.12; as opposed to most other codes presented in this text [which are based solely on `for` loops, `if` statements, `function` calls, and floating-point operations on vectors and matrices], Algorithm 2.12 makes use of a few of the advanced parallel programming features of Matlab. Unfortunately, the parallel performance of this implementation (using the 2011a release of Matlab) is poor, and the parallelized code is actually significantly slower than the standard (serial) Thomas algorithm, even when using four processors and large values of n. This is apparently due to the fact that, as of 2011, parallelization is a fairly new addition to Matlab; it is hoped that the performance and flexibility of the parallel Matlab tools will improve significantly in the near future. In contrast, the parallel capabilities of lower-level languages like Fortran and C [as discussed further in §§12-13] are much more mature and can provide a significant speedup using the algorithm described above.

Another strategy for parallel solution of tridiagonal systems is **cyclic reduction** (see Exercise 4.5).

2.5 Condition number

Let $A\mathbf{x} = \mathbf{b}$ for nonsingular A and consider a small perturbation to the RHS. The perturbed system is written

$$A(\mathbf{x} + \delta\mathbf{x}) = (\mathbf{b} + \delta\mathbf{b}) \qquad \Rightarrow \qquad A\,\delta\mathbf{x} = \delta\mathbf{b}.$$

We now determine a bound on the change $\delta\mathbf{x}$ in the solution $\mathbf{x}$ that results from a change $\delta\mathbf{b}$ to $\mathbf{b}$, using the induced p-norms defined in §1.3.2. Applying (1.21) to the equations $\mathbf{b} = A\mathbf{x}$ and $\delta\mathbf{x} = A^{-1}\,\delta\mathbf{b}$ gives

$$\|\mathbf{b}\|_p \le \|A\|_{ip}\,\|\mathbf{x}\|_p \;\;\Rightarrow\;\; \|\mathbf{x}\|_p \ge \|\mathbf{b}\|_p / \|A\|_{ip} \qquad \text{and} \qquad \|\delta\mathbf{x}\|_p \le \|A^{-1}\|_{ip}\,\|\delta\mathbf{b}\|_p.$$

[4] In many problems, one encounters several different tridiagonal systems $A_\kappa \mathbf{x}^\kappa = \mathbf{g}^\kappa$ that may all be worked on simultaneously, thereby easily achieving the desired **load balancing** (that is, the distribution of tasks over several processors) of a well-parallelized code. A problem of this sort is sometimes said to be **embarassingly parallel**.

[5] Note that, as opposed to the Thomas and circulant algorithms described previously, the parallel Thomas algorithm is not simply Gaussian elimination exploiting structured sparsity, as there are a couple of additional clever steps involved.

Algorithm 2.12: The parallel Thomas algorithm.

View
Test

```
function [g] = ThomasParallel(a,b,c,g,n,p)
% This function solves AX=g for X using the parallel Thomas algorithm on p proessors.
a=distributed(a); b=distributed(b); c=distributed(c); g=distributed(g); % Move data to labs

spmd % ---------------------------------- THIS BLOCK DONE IN PARALLEL ----------------------------------
  aa=getLocalPart(a); bb=getLocalPart(b); cc=getLocalPart(c); gg=getLocalPart(g);
  jm=length(aa);
  for j=1:jm-1                                % PARALLEL FORWARD SWEEPS
    mult     = -aa(j+1)/bb(j);
    bb(j+1)  = bb(j+1) + mult*cc(j);
    gg(j+1)  = gg(j+1) + mult*gg(j);
    if labindex>1, aa(j+1) = mult*aa(j); else, aa(j+1) = 0; end
  end
  for j=jm-1:-1:2                             % PARALLEL BACKWARD SWEEPS
    mult     = -cc(j-1)/bb(j);
    aa(j-1)  = aa(j-1) + mult*aa(j);
    cc(j-1)  =           mult*cc(j);
    gg(j-1)  = gg(j-1) + mult*gg(j);
  end
  afirst=aa(1); bfirst=bb(1); cfirst=cc(1); gfirst=gg(1); % Make select data available
  alast=aa(jm); blast=bb(jm); clast=cc(jm); glast=gg(jm); % to the client
end % --------------------------------------------------------------------------------------------------

% Now set up and solve the tridiagonal problem that relates the p labs.
% Note: the notation alast{k} denotes data that is actually still resident on the labs,
% whereas the notation aaa(k) sets up a regular vector on the client, to be used by Thomas.
for k=2:p                                     % One extra step of the last p-1 backward sweeps.
  mult     = -clast{k-1}/bfirst{k};
  aaa(k-1,1)= alast{k-1};
  bbb(k-1,1)= blast{k-1} + mult*afirst{k};
  ccc(k-1,1)=              mult*cfirst{k};
  ggg(k-1,1)= glast{k-1} + mult*gfirst{k};
end
aaa(p,1)=alast{p}; bbb(p,1)=blast{p}; ccc(p,1)=0; ggg(p,1)=glast{p};
ggg=Thomas(aaa,bbb,ccc,ggg,p);

spmd % ---------------------------------- THIS BLOCK DONE IN PARALLEL ----------------------------------
  gg(jm)=ggg(labindex);
  for j = jm-1:-1:1                           % PARALLEL BACK SUBSTITUTIONS
    if labindex>1, gg(j) = (gg(j)-aa(j)*ggg(labindex-1)-cc(j)*gg(jm))/bb(j);
    else,          gg(j) = (gg(j)                       -cc(j)*gg(jm))/bb(j); end
  end
end % --------------------------------------------------------------------------------------------------

g=[]; for k=1:p, g=[g; gg{k}]; end            % Accumulate result to return from function.
end % function ThomasParallel
```

Dividing the inequality on the right by the inequality in the center (and noting that, if $a \leq b$ and $c \geq d$, then $a/c \leq b/d$), we see that the relative change in the p-norm of the vector $\mathbf{x}$ is bounded by a constant κ_p times the relative change in the p-norm of vector $\mathbf{b}$, that is,

$$\frac{\|\delta\mathbf{x}\|_p}{\|\mathbf{x}\|_p} \leq \kappa_p \frac{\|\delta\mathbf{b}\|_p}{\|\mathbf{b}\|_p},$$

where $\kappa_p(A) = \|A\|_{ip}\|A^{-1}\|_{ip}$ is called the ***p*-norm condition number** of the matrix A. The 2-norm condition number is used most often in this text, and is thus sometimes referred to simply as the **condition number**,

$\kappa(A) = \kappa_2(A)$; an efficient method of computing the 2-norm condition number is deferred to Fact 4.40.

Now, let $A\mathbf{x} = \mathbf{b}$ and consider a small perturbation to the matrix A itself, and perform a similar analysis. The perturbed system may be written as

$$(A - \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} \qquad \Rightarrow \qquad A\,\delta\mathbf{x} = \delta A\,(\mathbf{x} + \delta\mathbf{x}).$$

Applying (1.21) to the equations $\delta\mathbf{x} = (A^{-1})\,[\delta A\,(\mathbf{x} + \delta\mathbf{x})]$ and $[\delta A\,(\mathbf{x} + \delta\mathbf{x})] = (\delta A)(\mathbf{x} + \delta\mathbf{x})$ in turn gives

$$\|\delta\mathbf{x}\|_p \le \|A^{-1}\|_{ip}\,\|\delta A\,(\mathbf{x}+\delta\mathbf{x})\|_p \le \|A^{-1}\|_{ip}\,\|\delta A\|_{ip}\,\|\mathbf{x}+\delta\mathbf{x}\|_p \approx \|A^{-1}\|_{ip}\,\|\delta A\|_{ip}\,\|\mathbf{x}\|_p, \qquad \frac{\|\delta\mathbf{x}\|_p}{\|\mathbf{x}\|_p} \lesssim \kappa_p \frac{\|\delta A\|_{ip}}{\|A\|_{ip}}.$$

Thus, we see that the relative change in the p-norm of the vector $\mathbf{x}$ is bounded by κ_p times the relative change in the induced p-norm of the matrix A.

If the condition number is small [say, $\kappa \lesssim O(10^4)$], then the matrix is referred to as **well conditioned**, meaning that small errors in either A or $\mathbf{b}$ (or, in their numerical representations using finite-precision arithmetic) will result in appropriately bounded errors in the resulting value of $\mathbf{x}$. However, if the condition number is large [say, $\kappa \gg O(10^4)$], then the matrix is **poorly conditioned**, and the accuracy of the solution $\mathbf{x}$ computed for the problem $A\mathbf{x} = \mathbf{b}$ is a matter of significant concern.

2.6 Singular and nonsquare systems, echelon form, and rank

Gaussian elimination with partial pivoting may easily be extended to singular and nonsquare systems by carrying the process of "upper triangularization" as far as possible at each step. For example,

$$\underbrace{\left(\begin{array}{ccccc|c} 0 & 0 & 3 & 9 & 3 & b_1 \\ -1 & -2 & 2 & 0 & 5 & b_2 \\ 2 & 4 & -1 & -3 & 5 & b_3 \\ 1 & 2 & 1 & -3 & 10 & b_4 \end{array}\right)}_{\begin{array}{cc} A & \mathbf{b} \end{array}} \Rightarrow \cdots \Rightarrow \underbrace{\left(\begin{array}{ccccc|c} 2 & 4 & -1 & -3 & 5 & b_3 \\ 0 & 0 & 3 & 9 & 3 & b_1 \\ 0 & 0 & 0 & -6 & 6 & b_2 + b_3/2 - b_1/2 \\ 0 & 0 & 0 & 0 & 0 & b_4 - b_3 - b_2 \end{array}\right)}_{\begin{array}{cc} U & \mathbf{y} \end{array}}. \tag{2.12}$$

The procedure of Gaussian elimination with partial pivoting is applied in an almost identical manner as for nonsingular systems, with the following exception: if at some intermediate step of the procedure the first column of the reduced augmented matrix [illustrated by the box in (2.2)] is the zero vector, then that column is skipped, the zero column in question is eliminated from the reduced augmented matrix, and the algorithm searches for a nonzero pivot (of maximum magnitude) in the next column to the right. Eventually, the procedure builds what is called an **echelon matrix** U, which looks something like an upper triangular matrix, but with additional zero elements preceding the nonzero elements on certain rows, as illustrated above. The first nonzero element in each row of the echelon matrix is called the **pivot**. The number of such nonzero pivots of the echelon matrix U is called the **rank** of the corresponding matrix A, denoted $r = \mathrm{rank}(A)$.

Note that the Gauss-Jordan elimination procedure may also be extended to singular and nonsquare systems. In the example shown above, this may be done to further reduce the echelon form to

$$\underbrace{\left(\begin{array}{ccccc|c} 1 & 2 & 0 & 0 & 2.5 & * \\ 0 & 0 & 1 & 0 & 1 & * \\ 0 & 0 & 0 & 1 & -1 & * \\ 0 & 0 & 0 & 0 & 0 & * \end{array}\right)}_{R}.$$

The procedure of Gauss-Jordan elimination is also applied in an almost identical manner as for nonsingular systems. The algorithm works from the r'th row of the echelon matrix back up, scaling the rows such that

the pivots are unity and adding linear combinations of the lower rows to the upper rows in order to zero the elements in the columns above the pivots. For a square nonsingular A, this procedure reduces A all the way to the identity matrix I; for singular and nonsquare systems, it reduces A to a **reduced echelon matrix** R, which looks something like an identity matrix with some extra columns added that extend it to an echelon form, as illustrated above. In the remainder of this section, for computational efficiency, we work primarily with the echelon matrix U as illustrated in (2.12), not the reduced echelon matrix R illustrated above.

Interpreting the echelon matrix. Recall that, in the case of nonsingular square systems, there exists a unique solution $\mathbf{x}$ to the problem $A\mathbf{x} = \mathbf{b}$, which may always be found with the procedure of Gaussian elimination with partial pivoting. In the case of singular or nonsquare systems, it thus comes as no surprise that the natural extension of this procedure determines both necessary and sufficient conditions for the existence of solution(s) to $A\mathbf{x} = \mathbf{b}$, as well as a complete parameterization of these solutions, if any exist.

Existence of solutions to $A\mathbf{x} = \mathbf{b}$. As illustrated in (2.12), assuming $A = A_{m\times n}$, the last $m - r \geq 0$ rows of the echelon matrix U are zero. Define $\overline{\mathbf{y}}$ as the vector containing the last $m - r$ elements of $\mathbf{y}$. It is seen by inspection that it is possible to find a solution to the problem $U\mathbf{x} = \mathbf{y}$ iff $\overline{\mathbf{y}} = 0$. As the problem $U\mathbf{x} = \mathbf{y}$ is completely equivalent to the original problem $A\mathbf{x} = \mathbf{b}$, this condition is also necessary and sufficient for the existence of solution(s) to the original problem $A\mathbf{x} = \mathbf{b}$. In the example given above, solutions to $A\mathbf{x} = \mathbf{b}$ exist iff $b_4 - b_3 - b_2 = 0$.

Parameterization of all solutions to $A\mathbf{x} = \mathbf{b}$. All solutions to the system $U\mathbf{x} = \mathbf{y}$ (and, therefore, to $A\mathbf{x} = \mathbf{b}$) may now be identified. Denote the r columns of U with the pivots as the **pivot columns**, $\underline{\mathbf{u}}^1$ to $\underline{\mathbf{u}}^r$, and the corresponding components of $\mathbf{x}$ as the **fixed variables** $\underline{x}_1$ to $\underline{x}_r$. Denote the remaining $n - r \geq 0$ columns of U as the **nonpivot columns**, $\overline{\mathbf{u}}^1$ to $\overline{\mathbf{u}}^{n-r}$, and the corresponding components of $\mathbf{x}$ as the **free variables** $\overline{x}_1$ to $\overline{x}_{n-r}$. In the solution of $U\mathbf{x} = \mathbf{y}$, if any such solutions exist (i.e., if $\overline{\mathbf{y}} = 0$), the vector of free variables, $\overline{\mathbf{x}}$, may be selected arbitrarily. Note that the matrix $\underline{U} = \underline{U}_{m\times r}$ (i.e., the matrix with the pivot columns of U as columns) is upper triangular with the (nonzero) pivots as diagonal components. To ensure $U\mathbf{x} = \mathbf{y}$ is satisfied, the vector of fixed variables, $\underline{\mathbf{x}}$, may then be determined as the unique solution to the problem $\underline{U}\underline{\mathbf{x}} = \mathbf{y} - \overline{U}\overline{\mathbf{x}}$. This problem may be solved efficiently using back substitution, working from the r'th row of $\underline{U}$ back up, as the rows below the r'th are satisfied trivially when $\overline{\mathbf{y}} = 0$. To summarize,

Fact 2.3 *Upon reducing the system $A\mathbf{x} = \mathbf{b}$ to echelon form $U\mathbf{x} = \mathbf{y}$, denote the matrix of pivot columns of U as $\underline{U}$, the matrix of nonpivot columns of U as $\overline{U}$, the corresponding fixed and free components of $\mathbf{x}$ as $\underline{\mathbf{x}}$ and $\overline{\mathbf{x}}$, respectively, and the vector containing the last $m - r$ elements of $\mathbf{y}$ as $\overline{\mathbf{y}}$. Then:*

- *At least one solution to $A\mathbf{x} = \mathbf{b}$ exists iff $\overline{\mathbf{y}} = 0$.*
- *All solutions $\mathbf{x}$ to $A\mathbf{x} = \mathbf{b}$, if any exist, are given by taking $\overline{\mathbf{x}}$ arbitrarily and solving $\underline{U}\underline{\mathbf{x}} = \mathbf{y} - \overline{U}\overline{\mathbf{x}}$ for $\underline{\mathbf{x}}$.*

The row exchanges and Gauss transformations performed by the extension of Gaussian elimination with partial pivoting to singular and nonsquare systems (as described above and implemented in Algorithm 2.13) may again be written in matrix form as $MP^TA = U$ or $A = PLU$, where $L = L_{m\times m} = M^{-1}$ is unit lower triangular, $P = P_{m\times m}$ is a permutation matrix, and $U = U_{m\times n}$ is an echelon matrix. All three of these matrices may be determined by straightforward extension of Gaussian elimination with partial pivoting (see Algorithm 2.13). As this algorithm constructs a PLU decomposition of any potentially singular or nonsquare A, with U in echelon form, the problem $A\mathbf{x} = \mathbf{b}$ may thus always be written in matrix form as $U\mathbf{x} = L^{-1}P^T\mathbf{b} = \mathbf{y}$.

Characterizing the column space and the row space. Any vector $\mathbf{b}$ that may be formed by a linear combination of the columns of A (that is, any vector $\mathbf{b}$ in the **column space** of A) may be written in the form $A\mathbf{x} = \mathbf{b}$ for some $\mathbf{x}$. As noted above, one solution of this problem is given by taking $\overline{\mathbf{x}} = 0$, then solving

$$\underline{U}\underline{\mathbf{x}} = \mathbf{y} \tag{2.13}$$

Algorithm 2.13: Extension of Gaussian elimination to singular and nonsquare systems.

View Test

```
function [A,p,r,v] = GaussEchelon(A)
% This function computes the PLU decomposition of a singular or nonsquare matrix A, where
% P is a permutation matrix, L is unit lower triangular, and U is in echelon form, using
% an extension of Gaussian elimination with partial pivoting. The matrix A is replaced by
% the m_ij and U on exit, p is the permutation vector, r is the rank, and v is a vector
% containing the column of each pivot.
[m,n]=size(A);   p=[1:m]';   r=0; v=[];              % Initialize rank = 0, no pivots
for j = 1:n,
   [amax,imax]=max(abs(A(r+1:m,j)));                 % Find the max element in left column
   if amax > 0                                       % If it is nonzero, increment the rank,
      r=r+1; v(r)=j;                                 % store this pivot location, and,
      if r<m                                         % if not in the very last row, then
         if amax > abs(A(r,j))                       % exchange rows if necessary ...
            A([r r-1+imax],:)=A([r-1+imax r],:);
            p([r r-1+imax])  =p([r-1+imax r]);
         end                                         % then perform the Gauss transformation.
         A(r+1:m,j)     = - A(r+1:m,j) / A(r,j);
         A(r+1:m,j+1:n) = A(r+1:m,j+1:n) + A(r+1:m,j) * A(r,j+1:n);
      end
   end
end
end % function GaussEchelon
```

for $\underline{\mathbf{x}}$. Following this approach, $\mathbf{y}$ is seen to be a linear combination of the columns of $\underline{U}$ (that is, the pivot columns of U). Multiplying (2.13) by PL and noting that $PL\mathbf{y} = \mathbf{b}$, it is seen that

$$PL\underline{U}\underline{\mathbf{x}} = PL\mathbf{y} \quad \Rightarrow \quad \underline{A}\underline{\mathbf{x}} = \mathbf{b} \quad \text{where } \underline{A} = PL\underline{U}. \tag{2.14}$$

As $\underline{U}$ is the matrix containing the pivot columns of U, $\underline{A}$ contains the corresponding columns of A. Thus, any vector $\mathbf{b}$ in the column space of A may be formed by linear combination of the columns of $\underline{A}$. Note also that, as $\underline{U}$ is upper triangular with nonzero diagonal elements, (2.13) [or, equivalently, (2.14)] has a unique solution $\underline{\mathbf{x}}$, given by back substitution applied to (2.13). Thus the columns of $\underline{A}$ must be linearly independent.

Similarly, any row vector $\mathbf{c}^T$ that may be formed by linear combination of the rows of A (that is, any row vector $\mathbf{c}^T$ in the **row space** of A) may be written in the form $\mathbf{z}^T A = \mathbf{c}^T$, or $A^T\mathbf{z} = \mathbf{c}$, for some $\mathbf{z}$. Defining $\mathbf{w} = L^T P^T \mathbf{z}$ and noting that $A = PLU$, we may thus write $U^T\mathbf{w} = \mathbf{c}$, or $\mathbf{w}^T U = \mathbf{c}^T$. Thus, any row vector $\mathbf{c}^T$ in the row space of A may be formed by linear combination of the nonzero rows of U. Note also that the r nonzero rows of U are linearly independent (that is, due to their echelon form, it is clear by inspection that none of the rows can be formed as a linear combination of the others). To summarize,

Fact 2.4 *The columns of A corresponding to the pivot columns of U form a basis for the column space of A, whereas the nonzero rows of U form a basis for the row space of A.*

Note that, by a similar argument, the nonzero rows of R also form a (somewhat simpler) basis for the row space of A. The following statements follow directly:

Fact 2.5 *The number of linearly independent columns of A (the* **column rank** *of A) is identical to the number of linearly independent rows of A (the* **row rank** *of A). Put another way,* **column rank** $=$ **row rank** $=$ **rank**.

Fact 2.6 *If $C = AB$, then the columns of C are linear combinations of the columns of A, and the rows of C are linear combinations of the rows of B. Thus, by Fact 2.5,* $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$.

Fact 2.7 *If $r = m$ (that is, if $A = A_{m\times n}$ has* **full row rank**), *there are no rows with all zeros in U, and at least one solution to $A\mathbf{x} = \mathbf{b}$ exists.*

Fact 2.8 *If $r = n$ (that is, if $A = A_{m\times n}$ has* **full column rank***), then there are no free variables in the system, and the solution to $A\mathbf{x} = \mathbf{b}$, if it exists, is unique.*

Fact 2.9 $\text{rank}(A_{m\times n}) \leq \min\{m,n\}$. *If equality holds, A is said to be* **full rank***, otherwise, it is* **rank deficient**.

Note also that:

- A square matrix that has full rank is nonsingular/invertible, whereas
- a square matrix that is rank deficient is singular/noninvertible.
- If A is a fat matrix with full row rank (that is, $r = m$), then at least one solution exists to $A\mathbf{x} = \mathbf{b}$, whereas
- if A is a tall matrix with full column rank (that is, $r = n$), then, if a solution to $A\mathbf{x} = \mathbf{b}$ exists, it is unique.

2.7 Chapter summary

As seen in §1.2.4, the unique solution to a nonsingular system $A\mathbf{x} = \mathbf{b}$ is given by $\mathbf{x} = A^{-1}\mathbf{b}$, where A^{-1} is the inverse matrix. More generally, we will show in §4.7 that the "best" solution to the system $A\mathbf{x} = \mathbf{b}$ in the inconsistent and/or overdetermined case is given by $\mathbf{x} = A^{+}\mathbf{b}$, where A^{+} is an appropriately defined **pseudoïnverse** matrix. Coupling these observations with the fact that there are stable, prepackaged numerical algorithms available in most computer languages to calculate both A^{-1} (see §2.1) and A^{+} (see §4.7) might lead one to the false impression that the problem of solving $A\mathbf{x} = \mathbf{b}$ requires no further of our attention. In fact, this impression couldn't be further from the truth. The calculation of A^{-1} and A^{+} when the order of the system is large is *extraördinarily expensive* and *destroys the sparsity structure of A*. Thus, all hopes of solving $A\mathbf{x} = \mathbf{b}$ efficiently are lost if we use A^{-1} or A^{+}, and techniques to determine $\mathbf{x}$ which do not require the computation of A^{-1} or A^{+} are essential.

The efficient direct method for determining the solution to the problem $A\mathbf{x} = \mathbf{b}$ (for square, nonsingular A) was illustrated by example in §2.1. We then took a careful look at how to automate and accelerate this algorithm, leveraging any exploitable (e.g., banded) sparsity structure that A might possess and, when several problems of the form $A\mathbf{x} = \mathbf{b}$ (for fixed A but different $\mathbf{b}$) are to be encountered in succession, the LU, PLU, or $PLUQ^T$ decomposition of A that emerges from the first execution of the Gaussian elimination procedure. We saw that partial pivoting (exchanging rows) often improves the accuracy of the Gaussian elimination algorithm and is sometimes required to make it actually work; complete pivoting may also be applied, but its extra computational expense is rarely justified. For the important class of diagonally dominant matrices, which appear often in the framing of numerical problems, pivoting is unnecessary.

Most of the systems we will encounter in our numerical algorithms will be sparse. When the equations and unknowns of the system may be enumerated in such a manner that the nonzero elements lie only near the main diagonal, resulting in a tightly banded matrix, the Gaussian elimination procedure may be streamlined and made quite efficient. For example, diagonally dominant tridiagonal systems may be solved via the Thomas algorithm. For most large sparse systems that are not banded (arrow and circulant matrices being notable exceptions), iterative solution methods are generally more efficient than direct methods; a few such methods will be developed in §3.2.

With the important exception of calculating eigenvectors (see §4.3), it is usually preferable to ensure, by construction, that the system $A\mathbf{x} = \mathbf{b}$ that you are setting out to solve is nonsingular before attempting to solve it numerically. Thus, methods for the efficient solution of nonsingular problems have been the primary focus of attention in this chapter. However, as seen in §2.6, it is easy to generalize the procedure of Gaussian elimination algorithm with partial pivoting to singular and nonsquare systems. This leads to the transformation of the matrix A to a matrix U in echelon form (instead of to a matrix in upper triangular form with nonzero elements on the main diagonal). This procedure reveals

- the necessary and sufficient conditions on $\mathbf{b}$ such that a solution to $A\mathbf{x} = \mathbf{b}$ exists,
- a parameterization of *all* solutions to the problem $A\mathbf{x} = \mathbf{b}$ (if any exist),
- the rank of A (that is, the number of independent columns, and rows, of A),
- a basis for the column space of A (that is, all vectors that may be formed by linear combination of the columns of A), and
- a basis for the row space of A (that is, all row vectors that may be formed by linear combination of the rows of A).

In the framing and analysis of efficient numerical methods (e.g., in the derivation of the *PLU* decomposition in §2.2.2), many important intermediate steps involve the nontrivial manipulation of matrices, several of which are sparse. Note that this is true even though efficient numerical implementations of these methods only use the nontrivial components of these matrices in a maximally compact representation. In order to derive more advanced numerical algorithms, it is essential to have a solid understanding of how the matrices at the heart of such algorithms may be characterized and decomposed. This is the subject of §4.

Exercises

Exercise 2.1 Determine the leading-order computational cost of each of the following operations (avoiding, as much as possible, writing out the detailed algorithms). Assume all matrices are $n \times n$ and diagonally dominant, and that it does not cost anything to fill one matrix element with the contents of another (thus, e.g., addition of a full matrix to a diagonal matrix costs n flops). Recall that $\sum_{k=1}^{n} k \sim (n^2/2)$ and $\sum_{k=1}^{n} k^2 \sim (n^3/3)$.
(a) Computing MC, where M and C are full.
(b) Computing MC, where M is full and C is diagonal.
(c) Computing $A\mathbf{x}$, where A is full.
(d) Computing $A\mathbf{x}$, where A is diagonal.
(e) Solving $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$, where A is tridiagonal.
(f) Solving $A\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$, where A is full.
(g) Solving $BM = A$ for M, where B is tridiagonal and A is diagonal. Show your work.
(h) Solving $BM = A$ for M, where B is full and A is diagonal. Show your work.

Exercise 2.2 Generalizing slightly the first few lines of Algorithm 2.2, note that, if L is lower triangular, $LX = B$ may be solved via forward substitution as follows:

```
for j = 1:n
   B(j,:) = (B(j,:) - L(j,1:j-1) * B(1:j-1,:))/L(j,j);
end
```

Initializing with $B = I$, as in §2.1, thus provides an algorithm to compute the inverse of L. In this special case with $B = I$, this algorithm may be streamlined such that the computation may be done *in place* in the lower-triangular part of the L matrix (that is, without creating a separate B matrix). Write this streamlined algorithm as a Matlab function `InvertL.m`, as well as a test script `InvertLTest.m`, in a style similar to the other codes presented in this chapter. Run this algorithm on a randomly-generated lower triangular matrix L; does Fact 2.1 hold in your numerical experiment?

Exercise 2.3 **(a)** Following closely the code `Thomas.m` in Algorithm 2.7, and its derivation in §2.3, write a streamlined Matlab function `Penta.m`, and test script `PentaTest.m`, implementing and testing a reduced form of Gaussian elimination to solve $A\mathbf{x} = \mathbf{b}$ for diagonally-dominant pentadiagonal matrices A. As with the implementation of the Thomas algorithm, all computations should be performed in place, and the vectors returned should contain the nontrivial elements of the LU decomposition of A. What is the leading-order computational cost of this algorithm?

(b) Following closely the code `ThomasLU.m` in Algorithm 2.8, write a streamlined Matlab function `PentaLU.m` to solve $A\mathbf{x} = \mathbf{b}$ for diagonally-dominant pentadiagonal matrices A, leveraging the previously-determined LU decomposition of A as returned by `Penta.m`. Extend the `PentaTest.m` script to verify the correctness of your `PentaLU.m` code. What is the leading-order computational cost of this algorithm?

Exercise 2.4 **(a)** Following closely the code `Thomas.m` in Algorithm 2.7, and its derivation in §2.3, write a streamlined Matlab function `Arrow.m`, and test script `ArrowTest.m`, implementing a reduced form of Gaussian elimination to solve $A\mathbf{x} = \mathbf{b}$ for diagonally-dominant arrow matrices A that are nonzero only on their main diagonal, last column, and last row. As with the implementation of the Thomas algorithm, all computations should be performed in place, and the vectors returned should contain the nontrivial elements of the LU decomposition of A. What is the leading-order computational cost of this algorithm?

(b) Following closely the code `ThomasLU.m` in Algorithm 2.8, write a streamlined Matlab function `ArrowLU.m` to solve $A\mathbf{x} = \mathbf{b}$ for diagonally-dominant arrow matrices A that are nonzero only on their main diagonal, last column, and last row, leveraging a previously-determined LU decomposition of A as returned by `Arrow.m`. Extend the `ArrowTest.m` script to verify the correctness of your `ArrowLU.m` code. What is the leading-order computational cost of this algorithm?

Exercise 2.5 Following closely the Thomas algorithm derived in §2.3, consider now an algorithm that leverages *block* triangular structure to solve, with maximum efficiency, a diagonally-dominant $(nm) \times (nm)$ system of the form

$$\begin{pmatrix} B_1 & C_1 & & & & \\ A_2 & B_2 & C_2 & & & \\ & A_3 & B_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & A_{m-1} & B_{m-1} & C_{m-1} \\ & & & & A_m & B_m \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \vdots \\ \mathbf{x}_{m-1} \\ \mathbf{x}_m \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \mathbf{f}_3 \\ \vdots \\ \mathbf{f}_{m-1} \\ \mathbf{f}_m \end{pmatrix},$$

where the blocks A_i, B_i, and C_i are $n \times n$, the blocks A_i and C_i are diagonal, the blocks B_i are tridiagonal, and the vectors $\mathbf{x}_i$ and $\mathbf{f}_i$ each have n components. [Note that this type of matrix arises in the second-order finite-difference approximation of a partial differential equation, such as that encountered in §1.2.8.] The resulting **block Thomas** algorithm follows the same essential steps as the now-familiar Thomas algorithm, with each step along the way now involving one of the matrix operations mentioned in Exercise 2.1.
(a) Keeping track of which blocks are diagonal, tridiagonal, and full, record carefully the number of matrix operations of each type, as summarized in Exercise 2.1, and compute the leading-order computational cost of the entire algorithm. Given a choice of having $m = 10$ and $n = 1000$, or $m = 1000$ and $n = 10$, which is cheaper? By what factor is one choice cheaper than the other?
(b) Write a streamlined Matlab function `BlockThomas.m`, and test script `BlockThomasTest.m`, that implements and tests this algorithm. Write `BlockThomas.m` in a manner that the LU decomposition of A may be recovered from the (modified) blocks returned by the block Thomas algorithm.
(c) Keeping track of which blocks are diagonal, tridiagonal, and full, compute the leading-order computational cost of the block Thomas algorithm when leveraging the LU decomposition of A, which may be recovered from the (modified) blocks returned by a previous call to the block Thomas algorithm.
(d) Write a streamlined Matlab function `BlockThomasLU.m` to solve a system of this form while leveraging the LU decomposition of A, which may be recovered from the (modified) blocks returned by a previous call to `BlockThomasTest.m`. Extend the `BlockThomasTest.m` script to verify the correctness of your `BlockThomasLU.m` code.

Exercise 2.6 Write a streamlined Matlab function `NLeggedThomas.m` to efficiently solve n simultaneous matrix equations of the form

$$\underbrace{\begin{pmatrix} b_1^{(i)} & c_1^{(i)} & & & & 0 \\ a_2^{(i)} & b_2^{(i)} & c_2^{(i)} & & & \\ & \ddots & \ddots & \ddots & & \\ & & a_{p-1}^{(i)} & b_{p-1}^{(i)} & c_{p-1}^{(i)} & \\ 0 & & & a_p^{(i)} & b_p^{(i)} & c_p^{(i)} \end{pmatrix}}_{A^{(i)}} \begin{pmatrix} x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_p^{(i)} \\ y \end{pmatrix} = \begin{pmatrix} g_1^{(i)} \\ \vdots \\ g_p^{(i)} \end{pmatrix} \quad \text{for} \quad i = 1,2,\ldots,n,$$

together with a single scalar equation of the form

$$\begin{pmatrix} d_1 & \ldots & d_n & d_{n+1} \end{pmatrix} \begin{pmatrix} x_p^{(1)} \\ \vdots \\ x_p^{(n)} \\ y \end{pmatrix} = e.$$

In total, this represents $pn+1$ equations in the $pn+1$ unknowns $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}, y\}$, where each of the $\mathbf{x}^{(i)}$ are of dimension $p \times 1$. Note that the (fat) tridiagonal matrices $A^{(i)}$ are each of dimension $p \times (p+1)$, and that both y and e are scalars. This system might represent, for example, the discretization of the heat equation along n thin rods all joined at a common central point, with the temperature along each rod discretized on p points (excluding the common central point y). Assume that n is small (say, between 2 and 20) and p is large [$O(100)$ or more].

An efficient code may be written to solve this problem by first performing n forward sweeps to reduce each of the matrix equations defined above to upper bidiagonal form. Then, assemble the last line of each of these modified matrix problems together with the scalar equation given above to construct a new matrix equation with $n+1$ equations and $n+1$ unknowns; this problem is easily solved using `Gauss.m`, thereby determining the values of y and of $x_p^{(1)}$ through $x_p^{(n)}$. Once these values are determined, it is straightforward to back substitute [from the $(p-1)$'th row back to the first] in each of the (now, upper bidiagonal) matrix equations in order to determine the remaining unknowns.

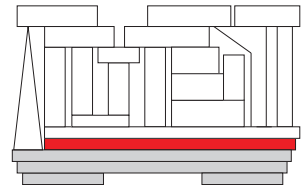
Write a test script `NLeggedThomasTest.m` which demonstrates that your solver works, defining the variables setting up the problem, $\{a_k^{(i)}, b_k^{(i)}, c_k^{(i)}, g_k^{(i)}, d_i, e\}$, both **(a)** randomly, and **(b)** in a manner which models the equilibrium temperature distribution in a collection of thin rods joined at one end, as described above, with random Dirichlet boundary conditions applied to the free ends of the rods. Describe how this solver may be parallelized efficiently on a system with n processors. [**Extra credit**: using Algorithm 2.12 as an example, implement this algorithm in parallel on n processors, and test its efficiency.] Also, describe in detail how this solver may be used to parallelize a regular tridiagonal set of equations (as studied in §2.3) on a two-processor computer. For a two-processor system, why would this approach be superior to Algorithm 2.12?

References

Fox, L (1964) *An introduction to numerical linear algebra*, Oxford.

Stewart, G (1998) *Matrix Algorithms, Volume I: Basic Decompositions*, SIAM.

Wang, HH (1981) A Parallel Method for Tridiagonal Equations. *ACM Transactions on Mathematical Software* **7**, 170-183.



Chapter 3

Iterative solution methods

Contents

3.1 Nonlinear equations . . . 53
3.1.1 The Newton-Raphson method . . . 54
3.1.1.1 Scalar case . . . 54
3.1.1.2 Quadratic convergence . . . 54
3.1.1.3 Multivariable case—systems of nonlinear equations . . . 55
3.1.2 Bracketing approaches for scalar root finding . . . 57
3.2 High-dimensional linear equations . . . 60
3.2.1 Splitting methods . . . 60
3.2.1.1 Jacobi . . . 60
3.2.1.2 Gauss-Seidel . . . 61
3.2.1.3 Successive overrelaxation . . . 61
3.2.1.4 Red/Black Gauss-Seidel . . . 61
3.2.2 Multigrid: a preview . . . 64
3.2.3 Framing $A\mathbf{x} = \mathbf{b}$ as a quadratic minimization problem: a preview . . . 64
Exercises . . . 65

Nonlinear equations and high-dimensional linear equations without exploitable sparsity structure are often not solvable directly, and thus numerical approximation via iterative solution methods are often required. The fundamental idea of all such iterative methods is to start from one or more initial guess(es) for the solution and then to seek successive refinements of this guess until a desired tolerance of the solution is reached.

3.1 Nonlinear equations

Consider first the nonlinear equation $f(x) = 0$; the problem considered here is to find the value(s) of x for which $f(x)$ is zero. Geometrically, we seek the crossing point(s) where the function $f(x)$ crosses the x-axis. Unfortunately, there are no systematic methods for nonlinear equations to determine how many such crossing points exist, so such searches are always something of a matter of trial and error.

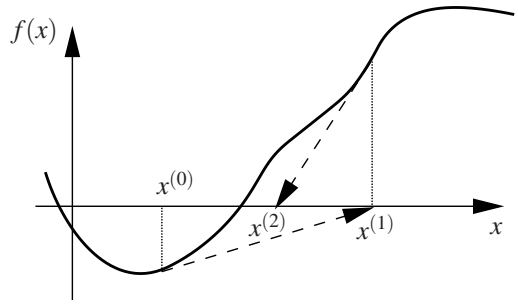


Figure 3.1: Geometrical interpretation of the Newton-Raphson method.

3.1.1 The Newton-Raphson method

3.1.1.1 Scalar case

Suppose an initial guess of the solution is $x = x^{(0)}$. Consider the Taylor series expansion of $f(x)$ about $x^{(0)}$:

$$f(x) = f(x^{(0)}) + (x - x^{(0)}) f'(x^{(0)}) + \ldots \tag{3.1}$$

To proceed in a tractable fashion, consider the truncation of this expansion after the first two terms on the RHS, take $f(x) = 0$ in this approximation, and solve for x, denoting the result $x^{(1)}$:

$$x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})}.$$

Successive iterates are obtained in an analogous fashion:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \qquad \text{for} \quad k = 0, 1, 2, \ldots \tag{3.2}$$

Geometrically, as illustrated in Figure 3.1, the function f at point $x^{(0)}$ is approximated by a tangent line given by the truncation of the Taylor series expansion (3.1), and the intersection of this line with the x-axis gives the refined value $x^{(1)}$. At the next iteration, the function at $x^{(1)}$ is approximated by a tangent line whose intersection with the x-axis gives $x^{(2)}$, etc.

3.1.1.2 Quadratic convergence

We now show that, once the iterative Newton-Raphson method approaches an exact solution x^{opt}, it converges quadratically. Let $x^{(k)}$ denote the iterate at the k'th iteration. Consider the truncated Taylor series expansion

$$f(x^{\text{opt}}) = 0 \approx f(x^{(k)}) + (x^{\text{opt}} - x^{(k)}) f'(x^{(k)}) + \frac{(x^{\text{opt}} - x^{(k)})^2}{2} f''(x^{(k)}).$$

If $f'(x^{(k)}) \neq 0$, dividing by $f'(x^{(k)})$ gives

$$x^{(k)} - x^{\text{opt}} \approx \frac{f(x^{(k)})}{f'(x^{(k)})} + \frac{(x^{\text{opt}} - x^{(k)})^2}{2} \frac{f''(x^{(k)})}{f'(x^{(k)})}.$$

Combining this with the Newton-Raphson formula (3.2) leads to

$$x^{(k+1)} - x^{\text{opt}} \approx \frac{(x^{(k)} - x^{\text{opt}})^2}{2} \frac{f''(x^{(k)})}{f'(x^{(k)})}.$$

Defining the error at iteration k as $\varepsilon^{(k)} = \left|x^{(k)} - x^{\text{opt}}\right|$, the error at the iteration $k+1$ is thus given by

$$\varepsilon^{(k+1)} \approx \frac{1}{2}\left|\frac{f''(x^{(k)})}{f'(x^{(k)})}\right| \left(\varepsilon^{(k)}\right)^2; \tag{3.3}$$

that is, **convergence is quadratic**. Convergence is guaranteed only if the initial guess is fairly "close" to the exact root; otherwise, the neglected higher-order terms in (3.1) dominate, and divergence is likely.

3.1.1.3 Multivariable case—systems of nonlinear equations

A system of n nonlinear equations in n unknowns is written

$$f_i(x_1, x_2, \ldots, x_n) = 0 \qquad \text{for} \qquad i = 1, 2, \ldots, n,$$

or, more compactly, as $\mathbf{f}(\mathbf{x}) = 0$. Generalization of the Newton-Raphson method to such systems is achieved via a multi-dimensional Taylor-series expansion, written here for the k'th iteration:

$$\begin{aligned} f_i(x_1, x_2, \ldots, x_n) &= f_i(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}) + (x_1 - x_1^{(k)}) \frac{\partial f_i}{\partial x_1}\bigg|_{\mathbf{x}=\mathbf{x}^{(k)}} + (x_2 - x_2^{(k)}) \frac{\partial f_i}{\partial x_2}\bigg|_{\mathbf{x}=\mathbf{x}^{(k)}} + \ldots \\ &= f_i(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}) + \sum_{j=1}^{n} (x_j - x_j^{(k)}) \frac{\partial f_i}{\partial x_j}\bigg|_{\mathbf{x}=\mathbf{x}^{(k)}} + \ldots \end{aligned}$$

where, again, superscripts denote iteration number [that is, the components of the vector $\mathbf{x}^{(k)}$ are $x_i^{(k)}$ for $i = 1, 2, \ldots, n$]. As in the scalar case, consider the truncation of this expansion after the linear terms, take $\mathbf{f}(\mathbf{x}) = 0$ in this approximation, and solve for $\mathbf{x}$, denoting the result $\mathbf{x}^{(k+1)}$. This results in the linear system of equations

$$\sum_{j=1}^{n} \left[\frac{\partial f_i}{\partial x_j}\right]_{\mathbf{x}=\mathbf{x}^{(k)}} \left(x_j^{(k+1)} - x_j^{(k)}\right) = -f_i(\mathbf{x}^{(k)}) \quad \Rightarrow \quad \sum_{j=1}^{n} \left[a_{ij}^{(k)}\right] \left(h_j^{(k+1)}\right) = -f_i(\mathbf{x}^{(k)}) \quad \text{for} \quad i = 1, 2, \ldots, n.$$

The above system constitutes n linear equations for the n components of $\mathbf{h}^{(k+1)} \triangleq (\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})$, which is the desired update to $\mathbf{x}^{(k)}$. In matrix notation, we have

$$A^{(k)}\mathbf{h}^{(k+1)} = -\mathbf{f}(\mathbf{x}^{(k)}) \quad \text{where} \quad a_{ij}^{(k)} = \left[\frac{\partial f_i}{\partial x_j}\right]_{\mathbf{x}=\mathbf{x}^{(k)}}. \tag{3.4a}$$

Note that the elements of $A^{(k)}$, called the **Jacobian matrix** of $\mathbf{f}(\mathbf{x})$, are evaluated at $\mathbf{x} = \mathbf{x}^{(k)}$. After solving the above system of linear equations for $\mathbf{h}^{(k+1)}$ using Gaussian elimination, the next estimate for $\mathbf{x}$ is given by

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{h}^{(k+1)}, \tag{3.4b}$$

and the process repeated. Efficient implementation of the Newton-Raphson method described above is given in Algorithms 3.1-3.2. Note in particular the use of **function handles** to pass problem-specific function names to the main code, and a handy **verbose flag** to activate or suppress progress reporting to the screen.

Example 3.1 Newton-Raphson applied to a nonlinear system of equations

Typical results when applying the Newton-Raphson approach to solve a nonlinear system of equations

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^2 + 3\cos x_2 - 1 \\ x_2 + 2\sin x_1 - 2 \end{pmatrix} \quad \Rightarrow \quad A^{(k)} = \begin{pmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} \end{pmatrix}_{\mathbf{x}=\mathbf{x}^{(k)}} = \begin{pmatrix} 2x_1^{(k)} & -3\sin x_2^{(k)} \\ 2\cos x_1^{(k)} & 1 \end{pmatrix}$$

are shown in Table 3.1. A "good" (lucky?) choice of initial conditions converges rapidly to one of possibly many solutions to $\mathbf{f}(\mathbf{x}) = 0$. A poor choice will not converge smoothly, and may not converge at all.

Algorithm 3.1: The Newton-Raphson method (3.4).

View

```
function [x] = NewtonRaphson(x,n,Compute_f,Compute_A,tol,verbose)
% This function solves f(x)=0 using the Newton Raphson method given an initial guess for
% x, where the function f(x) and its Jacobian are defined in Compute_f and Compute_A.
% Take verbose=1 for printing progress reports to screen, or verbose=0 to suppress.
if nargin<5, tol=1e-10; end, residual=2*tol;
if nargin<6, verbose=1; end, if verbose, disp('Convergence:'), end
while (residual>tol)
   f=Compute_f(x);  A=Compute_A(x);  residual=norm(f);  x=x+GaussPP(A,-f,n);
   if verbose, disp(sprintf('%20.13f ',x(1:n),residual)); end
end
end % function NewtonRaphson
```

Algorithm 3.2: The Newton-Raphson method tested on the system of Example 3.1.

View

```
function NewtonRaphsonTest
% Note the use of function handles to pass the names of the problem-specific functions
% Example_3_1_Compute_f & Example_3_1_Compute_A into the general NewtonRaphson code.
disp('Now testing NewtonRaphson on the function in Example 3.2.')
x=NewtonRaphson([0; 1],2,@Example_3_1_Compute_f,@Example_3_1_Compute_A)  % smooth
x=NewtonRaphson([0;-1],2,@Example_3_1_Compute_f,@Example_3_1_Compute_A)  % erratic
end % function NewtonRaphsonTest
%%%
```

3.1.2 Bracketing approaches for scalar root finding

It was shown in the previous section that the Newton-Raphson technique is efficient for finding the solution of both scalar nonlinear equations, $f(x) = 0$, and multivariable nonlinear systems of equations, $\mathbf{f}(\mathbf{x}) = 0$, when good initial guesses are available. Unfortunately, good initial guesses are not always available. When they are not, it is desirable to seek the roots of such equations by simpler means which, though somewhat pedestrian, guarantee success. The techniques we present in this section, though they do not achieve quadratic convergence, are guaranteed to converge to a solution of a scalar nonlinear equation so long as:

- the function is continuous and bounded, and
- an initial bracketing pair of the minimum can be identified.

They also have the added benefit that they are based on function evaluations alone (i.e., you don't need to write a function to compute the Jacobian), which makes them simple to implement. Unfortunately, these techniques are based on a bracketing principle which does not extend readily to multi-dimensional functions.

Bracketing a root

Our first task is to find a pair of values for x which bracket the minimum, i.e., we want to find an x_1 and an x_2 (with $x_2 > x_1$) such that $f(x_1)$ and $f(x_2)$ have opposite signs. This may often be done by hand with a minor amount of trial and error. At times, however, it is convenient to have an automatic procedure to find such a bracketing pair. For example, for functions which have opposite sign for sufficiently large and small arguments, a simple approach is to start with an initial guess for the bracket and then geometrically increase the distance between these points until a bracketing pair is found. This may be implemented with Algorithm 3.3 (or simple variants thereof).

Refining the bracket—bisection

Once a bracketing pair is found, the task that remains is simply to refine this bracket until a desired degree of precision is obtained. The most straightforward approach, as implemented in Algorithm 3.4, is to chop the interval in half repeatedly, keeping after each chop those two values of x that bracket the root. The convergence of such an algorithm is linear; at each iteration, the bounds on the solution are reduced by a factor of 2.

Refining the bracket—false position

A technique that is sometimes faster to converge than the bisection technique, and (unlike the Newton-Raphson method) still retains the safety of maintaining and refining a bracketing pair, is to compute each new point by what may be thought of as a *numerical approximation* of the Newton-Raphson formula (3.2) such that

$$x = x_1 - \frac{f(x_1)}{\delta f/\delta x}, \tag{3.5}$$

where the quantity $\delta f/\delta x$ is a simple difference approximation to the slope of the function f

$$\frac{\delta f}{\delta x} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}.$$

Efficient implementation of this algorithm is given in Algorithm 3.5. This approach sometimes stalls, so it is useful to put in an ad hoc check to keep the progress moving: specifically, the fall-back scheme implemented in Algorithm 3.5 is to revert to a bisection step if the update that would otherwise be performed by the false position technique is deemed to be too small.

Algorithm 3.3: A simple code to bracket a root of a function.

View Test

```
function [x1,x2] = FindRootBracket(x1,x2,Compute_f,p)
% Assuming the scalar function defined in Compute_f is smooth, bounded, and has opposite
% signs for sufficiently large and small arguments, find x1 and x2 that bracket a root.
while Compute_f(x1,0,p)*Compute_f(x2,0,p)>=0, int=x2-x1; x1=x1-0.5*int; x2=x2+0.5*int; end
end % function FindRootBracket
```

Algorithm 3.4: The bisection approach to refining the bracket of a root of a function.

View Test

```
function [x,evals]=Bisection(x1,x2,Compute_f,tol,verbose,p)
% This function refines the bracket of a root with the bisection algorithm.
f1=Compute_f(x1,verbose,p); f2=Compute_f(x2,verbose,p); evals=2;
while x2-x1>tol
   x = (x2 + x1)/2; f=Compute_f(x,verbose,p); evals=evals+1;
   if f1*f<0, x2=x; f2=f;
   else,      x1=x; f1=f; end
end
x=(x2+x1)/2;
end % function Bisection
```

Algorithm 3.5: The false position approach to refining the bracket of a root of a function.

View Test

```
function [x,evals]=FalsePosition(x1,x2,Compute_f,tol,verbose,p)
% This function refines the bracket of a root with the false position algorithm.
f1=Compute_f(x1,1,p); f2=Compute_f(x2,1,p); evals=2;
while x2-x1>tol
   if verbose, plot([x1 x2],[f1 f2],'r-'); end
   interval=x2-x1;  fprime=(f2-f1)/interval;  x=x1 - f1/fprime;
   % Ad hoc check:  reset to bisection technique if update by false position is too small.
   tol1 = interval/8;  if ((x-x1) < tol1 | (x2-x) < tol1), x = (x1+x2)/2;  end
   % Now perform the function evaluation and update the bracket.
   f = Compute_f(x,1,p); evals=evals+1;
   if f1*f < 0,  x2=x;  f2=f;
   else,         x1=x;  f1=f;  end
end
x=(x2+x1)/2;
end % function FalsePosition
```

Algorithm 3.6: A simple function for testing Algorithms 3.3 through 3.5.

View

```
function [f] = Example_3_2_Compute_f(x,verbose,p)
f=PolyVal(p,x);  if verbose, plot([x x],[0,f],'b-',x,f,'bx'), pause(0.2), end
end % function Example_3_2_Compute_f
```

Example 3.2 Root bracket finding, bisection, and false position applied to a scalar nonlinear equation

The simple root bracket finding algorithm described above to determine an initial bracket, in addition the bisection and false position algorithms to refine this bracket, are applied to the scalar nonlinear equation

$$f(x) = x^3 + x^2 - 20x + 50$$

in the test codes provided with Algorithms 3.3, 3.4, and 3.5 which, in turn, evaluate the above function by calling Algorithm 3.6. Under the assumptions stated (that is, that an initial bracketing pair can be found, and that the function under consideration is continuous and bounded), convergence of these algorithms is *guaranteed*[1], as they simply refine the bracket of the root at each iteration. It is seen that the false position

[1] That is, assuming that the fall-back scheme mentioned previously is implemented in the false position method.

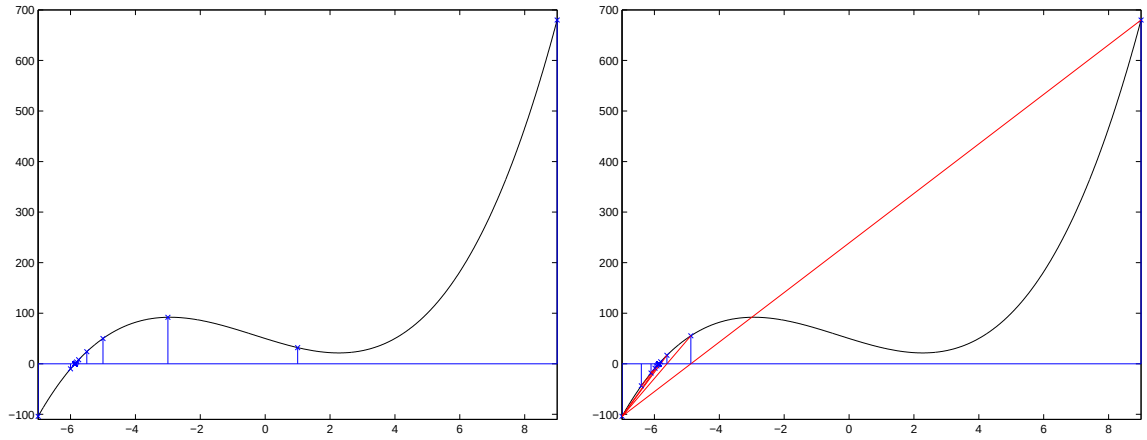


Figure 3.2: Convergence of (a) the bisection method, and (b) the false-position method on the system considered in Example 3.2. Convergence to a tolerance of $x_2 - x_1 = 10^{-6}$ is achieved in 26 function evaluations using the bisection method, and in 28 function evaluations using the false-position method.

approach, though it initially seems like a creative and useful idea, doesn't turn out to be worth the effort in this example; the bisection approach actually converges to the desired tolerance in this example using a smaller number of function evaluations (and significantly simpler logic).

Framing root finding as a nonquadratic minimization problem

The Newton-Raphson method is an effective technique to find the root (when one exists) of a nonlinear system of equations $\mathbf{f}(\mathbf{x}) = 0$ when a sufficiently-accurate initial guess is available. When such a guess is not available, an alternative technique is to examine the square of the norm of the vector $\mathbf{f}$:

$$J(\mathbf{x}) = \|\mathbf{f}(x)\|^2 = [\mathbf{f}(\mathbf{x})]^T \mathbf{f}(\mathbf{x}).$$

Note that this quantity is never negative, so any point $\mathbf{x}$ for which $\mathbf{f}(\mathbf{x}) = 0$ minimizes $J(\mathbf{x})$. Thus, seeking a minimum of this $J(\mathbf{x})$ with respect to $\mathbf{x}$ *might* result in an $\mathbf{x}$ such that $\mathbf{f}(\mathbf{x}) = 0$. However, there are quite likely many minimum points of $J(\mathbf{x})$ (at which the gradient of J is zero), only some of which (if any!) will correspond to $\mathbf{f}(\mathbf{x}) = 0$. Root finding in systems of nonlinear equations is very difficult—though this method has significant drawbacks, variants of this method are really about the best one can do when one does not have a good initial guess for the solution. Two efficient techniques to solve this problem are the nonquadratic conjugate gradient algorithm and the BFGS algorithm, both of which are deferred to §16.

Framing nonquadratic minimization problem as a root finding problem

If the original problem is a minimization problem rather than a root finding problem, straightforward application of the Newton-Raphson method may be used to find a minimum simply by looking for a root of the gradient of J, as discussed further in §16.

3.2 High-dimensional linear equations

We now turn our attention to the sparse $n \times n$ linear problem $A\mathbf{x} = \mathbf{b}$, where n is so large that Gaussian elimination, a **direct** solution method which completes in a finite number [$\sim (\frac{2}{3}n^3)$] of flops, is **numerically intractable**, and the unknowns in the problem can not be ordered in such a manner that A has a sparsity structure that can be leveraged by a reduced form of Gaussian elimination. In such cases, it is often necessary to use an **iterative** solution method to approximate the solution to $A\mathbf{x} = \mathbf{b}$.

3.2.1 Splitting methods

In the first class of schemes we consider, called **splitting methods**, the matrix A is first split into two parts such that $A = M + N$. Rewriting $A\mathbf{x} = \mathbf{b}$ as $M\mathbf{x} = -N\mathbf{x} + \mathbf{b}$ and taking the LHS **implicitly** (that is, at the new step) and the RHS **explicitly** (that is, at the old step), the following iterative algorithm is proposed:

$$M\mathbf{x}^{(k+1)} = -N\mathbf{x}^{(k)} + \mathbf{b}. \tag{3.6}$$

This sequence will converge to the true solution $\mathbf{x}$ as $k \to \infty$ if the error $\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}$ converges to zero. Subtracting $M\mathbf{x} = -N\mathbf{x} + \mathbf{b}$ from (3.6), defining $P = -M^{-1}N$, and noting (1.21), it follows that[2]

$$M\mathbf{e}^{(k+1)} = -N\mathbf{e}^{(k)} \quad \Rightarrow \quad \mathbf{e}^{(k)} = P^k\mathbf{e}^{(0)} \underset{k\to\infty}{\longrightarrow} 0 \quad \text{if} \quad \|P\|_{i2} < 1 \quad \text{(that is, if } P \text{ is } \textbf{convergent}\text{)}. \tag{3.7}$$

It will also be useful in this chapter to think of the iteration in (3.6) in terms of a **correction** $\mathbf{v}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ based on the **defect**

$$\mathbf{d}^{(k)} = A\mathbf{x}^{(k)} - \mathbf{b}. \tag{3.8a}$$

Subtracting $M\mathbf{x}^{(k)}$ from both sides of (3.6) yields

$$M\mathbf{v}^{(k)} = -\mathbf{d}^{(k)}. \tag{3.8b}$$

Rearranging the above definition for the correction $\mathbf{v}^{(k)}$ allows us to identify our new iterate as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k)}. \tag{3.8c}$$

The task at hand is thus to select the splitting $A = M + N$ such that (3.6) [or, equivalently, (3.8)] converges to the desired tolerance in a relatively small number of iterations k (that is, that $\|P\|_{i2}$ is as small as possible) while the matrix M is such that (3.6) [or, equivalently, (3.8b)] is relatively easy to solve (e.g., by selecting M to be diagonal or triangular). Various choices for this splitting are discussed below, based on the simple partitioning of A defined (in §3.2 only) such that $A = L + D + U$, where L is strictly lower triangular, D is diagonal, and U is strictly upper triangular.

3.2.1.1 Jacobi

The **Jacobi** method takes $M_\text{J} = D$ and $N_\text{J} = L + U$, resulting in $\mathbf{x}^{(k+1)} = D^{-1}[-(L+U)\mathbf{x}^{(k)} + \mathbf{b}]$; note that D in this case is diagonal, so D^{-1} is trivial to compute. If A is diagonally dominant, then it can be shown (see Exercise 4.6a-b) that $\|P_\text{J}\|_{i2} < 1$ where $P_\text{J} = -M_\text{J}^{-1}N_\text{J}$, and thus convergence is guaranteed. However, for the typical high-dimensional linear equations of interest for splitting methods, convergence of the Jacobi method is unacceptably slow (as compared with the methods described below) for it to be of practical use.

[2] To interpret this method in terms of the eigenvalues λ_i [as introduced in §4.3] and corresponding eigenvectors $\mathbf{s}^i$ of P, expand $\mathbf{e}^{(0)}$ in terms of the eigenvectors of P such that $\mathbf{e}^0 = \chi_1\mathbf{s}^1 + \chi_2\mathbf{s}^2 + \ldots + \chi_n\mathbf{s}^n$. It follows that $\mathbf{e}^{(k)} = \chi_1\lambda_1^k\mathbf{s}^1 + \chi_2\lambda_2^k\mathbf{s}^2 + \ldots + \chi_n\lambda_n^k\mathbf{s}^n$. For large k, the error is eventually dominated by the component(s) corresponding to the eigenvalue(s) of P of largest magnitude, the magnitude of which is referred to as the **induced 2-norm** (see §1.3.2) or **spectral radius** of P, that is, $\|P\|_{i2} = \rho(P)$. If $\|P\|_{i2} < 1$, the iteration (3.6) will thus eventually converge to the desired solution. Note that, to *apply* a splitting method, we don't actually need to solve any eigenvalue problems, or for that matter even to compute $P = -M^{-1}N$. We only consider these quantities here for the purpose of *analysis* of the method. Note also that this explanation assumes that P has a complete set of eigenvectors; however, this assumption is easily relaxed, following a Schur-based convergence analysis akin to that illustrated in §4.4.5.

3.2.1.2 Gauss-Seidel

The **Gauss-Seidel** method takes $M_{\text{GS}} = D + L$ and $N_{\text{GS}} = U$, resulting in $(D+L)\mathbf{x}^{(k+1)} = -U\mathbf{x}^{(k)} + \mathbf{b}$, which may be solved at each iteration by back substitution (see §2.1). [Note that, alternatively, the Gauss-Seidel method may be defined by taking $M_{\text{GS}} = D + U$ and $N_{\text{GS}} = L$, or by alternating between these two definitions at even and odd iteration steps.] Again, if A is diagonally dominant, then it can be shown (see Exercise 4.6c) that $\|P_{\text{GS}}\|_{i2} < 1$ where $P_{\text{GS}} = -M_{\text{GS}}^{-1} N_{\text{GS}}$, and thus convergence is guaranteed. Heuristically, since we are effectively including "more of the problem" into the implicit part (that is, into M_{GS}), convergence is generally a bit faster following the Gauss-Seidel approach than following the Jacobi approach. For the important example considered in Example 3.3 below, it turns out that $\|P_{\text{J}}\|_{i2} < 1$ and $\|P_{\text{GS}}\|_{i2} = \|P_{\text{J}}\|_{i2}^2$, and thus $\|P_{\text{GS}}\|_{i2}$ is indeed a bit smaller in the Gauss-Seidel approach. Convergence of the plain Gauss-Seidel method described here is still, in general, unacceptably slow for it to be of practical use as is; nonetheless, when implemented properly (for example, in the red/black fashion described in §3.2.1.4) and coupled with the multigrid acceleration technique introduced in §3.2.2 (and described in detail in §11.4.1), the Gauss-Seidel approach forms the foundation for some the most efficient techniques available for solving large linear systems derived from elliptic PDEs.

3.2.1.3 Successive overrelaxation

The **successive overrelaxation** (**SOR**) method is an iterative method based on the Gauss-Seidel approach, with $M_{\text{SOR}} = D + L$ and $N_{\text{SOR}} = U$, initially written in the form (3.8) with (3.8c) replaced by

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega \mathbf{v}^{(k)} \tag{3.8c'}$$

for some **relaxation parameter** $\omega \in (0,2)$; note that $\omega = 1$ reduces the SOR method to the standard Gauss-Seidel method. The motivation for the SOR method is that the correction $\mathbf{v}^{(k)}$ as calculated by the Gauss-Seidel method is often aligned in a fairly good *direction* for an update to $\mathbf{x}^{(k)}$, but is not *scaled* optimally to maximize the rate of convergence of the iterative scheme; a factor of $\omega \neq 1$ can thus sometimes accelerate convergence significantly. Unfortunately, the optimal value for ω for any given problem is usually not known a priori, and numerically tractable methods to estimate it are generally fairly involved. Applying $M_{\text{SOR}} = D + L$ and $N_{\text{SOR}} = U$ to (3.8c'), with (3.8a) and (3.8b), leads to

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \omega(D+L)^{-1}[(U+D+L)\mathbf{x}^{(k)} - \mathbf{b}] \quad \Rightarrow \quad (D+L)\mathbf{x}^{(k+1)} = [(1-\omega)D + (1-\omega)L - \omega U]\mathbf{x}^{(k)} + \omega\mathbf{b}.$$

Note that many implementations of the SOR method swing the $(1-\omega)L\mathbf{x}^{(k)}$ term to the LHS and apply it implicitly, resulting in a system which is still lower triangular:

$$(D + \omega L)\mathbf{x}^{(k+1)} = [(1-\omega)D - \omega U]\mathbf{x}^{(k)} + \omega\mathbf{b}.$$

This form is not equivalent to (3.8c'), though it is found in practice to be similarly effective. Unfortunately, convergence of this approach is usually unacceptably slow, as compared with the multigrid technique introduced in §3.2.2 (and described in detail in §11.4.1), for it to be of much practical use.

3.2.1.4 Red/Black Gauss-Seidel

An important class of matrices, called **checkerboard matrices** (a.k.a. **red/black matrices**), have a special sparsity structure such that $a_{ij} = 0$ when $i + j =$ even and $i \neq j$. If A is checkerboard, then a suitable reordering of the problem $A\mathbf{x} = \mathbf{b}$, leveraging the **odd-even permutation matrix** P_{oe} introduced in §1.2.5, renders it particularly attractive for the application of Gauss-Seidel iterations. For example, if A is 6×6 and

checkerboard, then

$$A = \begin{pmatrix} a_{11} & a_{12} & & a_{14} & & a_{16} \\ a_{21} & a_{22} & a_{23} & & a_{25} & \\ & a_{32} & a_{33} & a_{34} & & a_{36} \\ a_{41} & & a_{43} & a_{44} & a_{45} & \\ & a_{52} & & a_{54} & a_{55} & a_{56} \\ a_{61} & & a_{63} & & a_{65} & a_{66} \end{pmatrix} \quad \text{and} \quad P_{oe} = \begin{pmatrix} 1 & & & & & \\ & & & 1 & & \\ & 1 & & & & \\ & & & & 1 & \\ & & 1 & & & \\ & & & & & 1 \end{pmatrix}. \tag{3.9a}$$

Via premultiplication by P_{oe}^T, the problem $A\mathbf{x} = \mathbf{b}$ is seen to be equivalent to

$$P_{oe}^T A P_{oe}(P_{oe}^T \mathbf{x}) = P_{oe}^T \mathbf{b} \quad \Rightarrow \quad \underbrace{\begin{bmatrix} D_o & F_e \\ F_o & D_e \end{bmatrix}}_{P_{oe}^T A P_{oe}} \underbrace{\begin{bmatrix} \mathbf{x}_o \\ \mathbf{x}_e \end{bmatrix}}_{P_{oe}^T \mathbf{x}} = \underbrace{\begin{bmatrix} \mathbf{b}_o \\ \mathbf{b}_e \end{bmatrix}}_{P_{oe}^T \mathbf{b}},$$

where $\mathbf{x}_o$ denotes the vector containing the odd elements of $\mathbf{x}$ and $\mathbf{x}_e$ denotes the vector containing the even elements of $\mathbf{x}$ (ditto for $\mathbf{b}$), and D_o and D_e are diagonal. In our 6×6 example,

$$P_{oe}^T A P_{oe} = \begin{pmatrix} a_{11} & & & a_{12} & a_{14} & a_{16} \\ & a_{33} & & a_{32} & a_{34} & a_{36} \\ & & a_{55} & a_{52} & a_{54} & a_{56} \\ a_{21} & a_{23} & a_{25} & a_{22} & & \\ a_{41} & a_{43} & a_{45} & & a_{44} & \\ a_{61} & a_{63} & a_{65} & & & a_{66} \end{pmatrix}, \quad P_{oe}^T \mathbf{x} = \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ x_2 \\ x_4 \\ x_6 \end{pmatrix}, \quad P_{oe}^T \mathbf{b} = \begin{pmatrix} b_1 \\ b_3 \\ b_5 \\ b_2 \\ b_4 \\ b_6 \end{pmatrix}. \tag{3.9b}$$

Application of the Gauss-Seidel method to this permuted checkerboard system, taking

$$M_{\text{GS}} = \begin{bmatrix} D_o & 0 \\ F_o & D_e \end{bmatrix} \quad \text{and} \quad N_{\text{GS}} = \begin{bmatrix} 0 & F_e \\ 0 & 0 \end{bmatrix},$$

leads to an iteration method that can be written as the repeated application of two distinct substeps that are symmetric in the even and odd variables:

$$D_o \mathbf{x}_o^{(k+1)} = -F_e \mathbf{x}_e^{(k)} + \mathbf{b}_o \quad \Rightarrow \quad \text{Substep 1:} \quad D_o \mathbf{x}_o = -F_e \mathbf{x}_e + \mathbf{b}_o, \tag{3.10a}$$

$$D_e \mathbf{x}_e^{(k+1)} + F_o \mathbf{x}_o^{(k+1)} = \mathbf{b}_e \quad \Rightarrow \quad \text{Substep 2:} \quad D_e \mathbf{x}_e = -F_o \mathbf{x}_o + \mathbf{b}_e. \tag{3.10b}$$

It is thus seen that the Gauss-Seidel method applied to this reordering of the checkerboard system, referred to as **red/black Gauss-Seidel**, does not have any preferred directions[3] (that is, the defect $\mathbf{d}^{(k)}$ [see (3.8a)] is uniformly distributed over the domain). It turns out that the equation $A\mathbf{x} = \mathbf{b}$ is solved exactly on the odd gridpoints after the first substep of each iteration, and on the even gridpoints after the second substep of each iteration. Also, though the method applied is in fact a Gauss-Seidel scheme (that is, not a Jacobi scheme), the matrix on the LHS at each substep is in fact *diagonal*. Thus, unlike the Gauss-Seidel method applied to general systems (as discussed in §3.2.1.2), this realization is immediately parallelizable, as during each of its two substeps the calculations are decoupled and may be performed in any order. Note finally that, though the system was considered in a reordered form in our conceptualization of this scheme, the system does not actually need to be reordered to apply the two step procedure (3.10) that implements it.

Algorithms of this level of complexity must generally be implemented on a case-by-case bases in order to be coded with maximum efficiency. Thus, rather than present a code which implements the red/black Gauss-Seidel method for arbitrary A matrices of checkerboard structure, we now consider a representative example problem, and present a code which efficiently implements the red/black Gauss-Seidel method for this illustrative example.

[3] This is in contrast with the Gauss-Seidel method for general systems as introduced in §3.2.1.2, which must be swept from either top to bottom or bottom to top, and thus error accumulates more at one end of the domain than the other.

Algorithm 3.7: Red/black Gauss-Seidel smoothing applied to the 2D Poisson problem of Example 3.3.

View

```
function PoissonRBGSTest
% Apply 50 steps of red/black Gauss-Seidel "smoothing" with a (checkerboard) A matrix
% from a SOFD approximation of the 2D Poisson equation on a square grid (Example 3.3).
% The set of points updated first, which we label as "red", includes the corners.
disp('Now applying 50 steps of red/black Gauss-Seidel smoothing to a checkerboard system.')
n=32; L=1; h=L/n; z=[2:2:n]*h; b=zeros(n+1); x=zeros(n+1); b(2:n,2:n)=randn(n-1); close all
for i=1:50
  [x]=PoissonRBGSsmooth(x,b,n,h);
  r=(x(2:2:n,3:2:n+1)+x(2:2:n,1:2:n-1)+x(3:2:n+1,2:2:n)+x(1:2:n-1,2:2:n) ...
     -4*x(2:2:n,2:2:n))/h^2-b(2:2:n,2:2:n);  surf(z,z,r);  pause;
end, disp(' ')
end % function PoissonRBGSTest
%%%
```

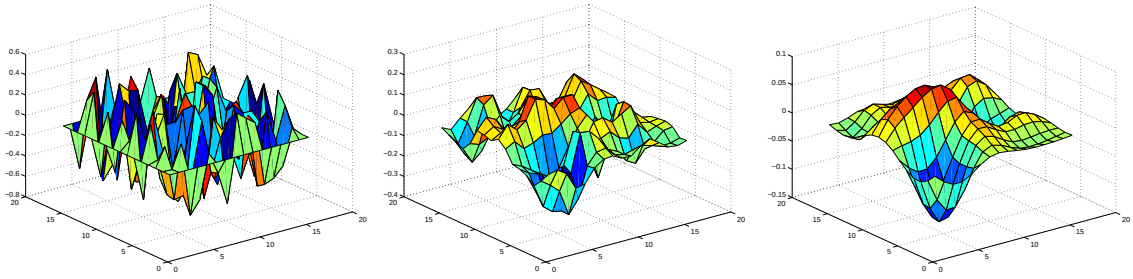


Figure 3.3: Defect [see (3.8a)] on half of the red gridpoints after application of (left) zero, (middle) one, and (right) ten applications of the red/black Gauss-Seidel method; note the rapid *smoothing* of the defect, in addition to the gradual reduction of its peak magnitude.

3.2.2 Multigrid: a preview

Though they are relatively easy to apply, none of the simple splitting techniques presented in §3.2.1, on their own, are particularly efficient at solving systems of the form $A\mathbf{x} = \mathbf{b}$ when the dimension of A is large. As seen by example in Figure 3.3, when applied to a numerical discretization of the Poisson equation, the red/black Gauss-Seidel method is generally best thought of as an unbiased and numerically efficient *smoother* of the defect $\mathbf{d}$ at the highest spatial frequencies represented on the numerical grid. Leveraging this smoothing behavior in a clever way, the **multigrid** method is now one of the fastest methods available for solving systems of the form $A\mathbf{x} = \mathbf{b}$ derived from the discretization of PDEs like the Poisson equation on simple domains. The key steps of the multigrid algorithm are as follows:

1. Apply one or two iterations of the red/black Gauss-Seidel smoother to the original problem in the form (3.6) [discretized on a fine grid] to **smooth** the error.
2. **Restrict** (that is, approximate) the defect of the result on a grid coarsened by a factor of 2 in each direction, and apply one or two more iterations of the red/black Gauss-Seidel smoother to the problem of determining the correction $\mathbf{v}$ from the defect $\mathbf{d}$ in the form (3.8) [discretized on the coarse grid].

- Continue this restrict/smooth/restrict/smooth process until the grid is so coarse that the correction problem can be solved directly, and solve it.

3. **Prolongate** (that is, interpolate) the correction $\mathbf{v}$ to the previous (finer) grid, update the problem being solved there, and apply one or two more iterations of the red/black Gauss-Seidel smoother.

- Continue this prolongate/smooth/prolongate/smooth process until returned to the original (finest) level.

4. Repeat from step 2 until convergence.

The full presentation of the multigrid method is deferred to §11.4.1, after further characterization (in §11.1) of the **elliptic** class of partial differential equations, such as the 2D Poisson equation (3.11a), which it is designed to solve.

3.2.3 Framing $A\mathbf{x} = \mathbf{b}$ as a quadratic minimization problem: a preview

An alternative approach to solving high-dimensional linear equations derived from PDE systems, which obviates the need to define "coarsened" discretizations (which is difficult in complex geometries), is to pose and solve a corresponding **minimization problem**. With this approach, a suitable **cost function** $J(\mathbf{x})$ is defined such that, once (approximately) minimized via an iterative technique (in a potentially high-dimensional space $\mathbf{x}$), the desired problem $A\mathbf{x} = \mathbf{b}$ is (approximately) solved. In the case that A is such that $\mathbf{x}^T A\mathbf{x} > 0$ for all $\mathbf{x}$

(that is, in the case that A is **positive definite**, as defined in §4.4.4.1), we can accomplish this by defining

$$J(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} = \frac{1}{2}x_i a_{ij} x_j - b_i x_i.$$

Requiring that A be positive definite ensures that $J \to \infty$ for $|\mathbf{x}| \to \infty$ in all directions, and thus that a minimum point indeed exists. Differentiating J with respect to an arbitrary component of $\mathbf{x}$, we find that

$$\frac{\partial J}{\partial x_k} = \frac{1}{2}\left(\delta_{ik} a_{ij} x_j + x_i a_{ij}\,\delta_{jk}\right) - b_i\,\delta_{ik} = a_{kj}x_j - b_k.$$

The unique minimum of $J(\mathbf{x})$ is characterized by

$$\frac{\partial J}{\partial x_k} = a_{kj}x_j - b_k = 0 \quad \text{or} \quad \nabla J = A\mathbf{x} - \mathbf{b} = 0.$$

Thus, solution of large linear systems of the form $A\mathbf{x} = \mathbf{b}$ may be found by minimization of a quadratic function $J(\mathbf{x})$. Efficient techniques to solve problems of this type are presented in §16.

Exercises

Exercise 3.1 Plot the Chebyshev polynomial $T_9(x) = 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x$ for $x \in [-1,1]$ (this function will be studied in greater detail in §5.13). Then, using the Newton-Raphson, bisection, and false position methods described in §3.1, find *all* real values of $x \in [-1,1]$, to *eight* digits of precision (!), such that $f(x) = 0$. For each method, plot the convergence to each root (that is, plot the error of the best estimate of the root thus far) as a function of iteration. Discuss.

Exercise 3.2 A new technique is proposed to both accelerate the rate of convergence and increase the domain of convergence of the Newton-Raphson algorithm (3.2) for solving nonlinear equations. The new technique is based on the first *three* terms of the Taylor-series expansion for $f(x_{k+1})$ near x_k, which may be written

$$f_{k+1} = f_k + (x_{k+1} - x_k)f'_k + \frac{(x_{k+1} - x_k)^2}{2!}f''_k + \ldots,$$

where $f_k \triangleq f(x_k)$, etc. Neglecting the terms cubic and higher in $(x_{k+1} - x_k)$, setting $f_{k+1} = 0$, and solving the resulting quadratic equation for x_{k+1}, assuming that $f''_k \neq 0$, gives

$$x_{k+1} = x_k - \frac{f'_k}{f''_k} \pm \sqrt{\frac{f'^2_k}{f''^2_k} - 2\frac{f_k}{f''_k}}.$$

To ensure that the algorithm converges (that is, to ensure that the update to x_k is small when f_k is small), the positive root needs to be taken in this expression, resulting in

$$x_{k+1} = x_k - \frac{f'_k}{f''_k} + \sqrt{\frac{f'^2_k}{f''^2_k} - 2\frac{f_k}{f''_k}} \tag{3.12a}$$

$$= x_k - \frac{f'_k}{f''_k}\left(1 - \sqrt{1 - 2\frac{f_k f''_k}{(f'_k)^2}}\right). \tag{3.12b}$$

Note that, unlike the Newton-Raphson algorithm, an algorithm based on (3.12a) may be well behaved even when $f'_k \approx 0$; that is, this algorithm will generally be successful when the nonlinear function is dominated by

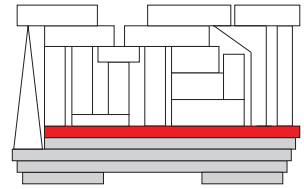
a "locally quadratic" character, rather than the "locally linear" character necessary for the Newton-Raphson convergence to be well behaved. Thus, we might expect this new algorithm to have a significantly improved domain of convergence as compared with the Newton-Raphson algorithm.

(**a**) Unfortunately, both (3.12a) and (3.12b) take the difference of two numbers which are almost equal when f_k is small (that is, as convergence is approached), thus resulting in a significant loss of accuracy on a machine with finite-precision arithmetic. To correct for this, noting the expansion (B.81) and retaining the first three terms on the RHS, develop an expression for x_{k+1} which does not exhibit such a problem when f_k is small. Compare this modified update formula to the Newton-Raphson update formula (3.2). Discuss.

(**b**) Write an iterative code to apply the new algorithm to determine numerically a root of a scalar equation $f(x) = 0$. Defining $\varepsilon = -2f_k f_k''/(f_k')^2$, use the update formula (3.12a) when $|\varepsilon| > 0.01$ [as (3.12a) does not divide by f_k'], and use the update formula determined in part (a) when $|\varepsilon| \leq 0.01$. Test this code on the function $T_9(x)$ given in Exercise 3.1, and compare the *domain* of convergence to the root near $x = 0.85$, as well as the *rate* of convergence to this root as convergence is approached, with the standard Newton-Raphson method (Algorithm 3.1) applied to the same problem. Quantify the domain of convergence by establishing how large the region is around the root in question that converges reliably to the root near $x = 0.85$. Quantify the rate of convergence by plotting both the log of the **error** $\varepsilon^{(k)} = \left|x^{(k)} - x^{\text{opt}}\right|$ as a function of k as well as the log of the **residual** $|f_k|$ as a function of k. Recall from (3.3) that the rate of convergence of the Newton Raphson method, for sufficiently small $\varepsilon^{(k)}$, is predicted to be quadratic; is this evident in your error plots? Discuss.

References

Press, WH, Teukolsky, SA, Vetterling, WT, & Flannery, BP (2007). *Numerical Recipes, the Art of Scientific Computing*. Cambridge.



Chapter 4

Linear algebra

Contents

4.1 The four fundamental subspaces of a matrix . . . 68
4.2 The determinant . . . 69
4.2.1 Some important properties of the determinant, and their consequences . . . 69
4.2.2 Computing the determinant . . . 73
4.3 An introduction to eigenvalues and eigenvectors . . . 73
4.3.1 Computing the eigenvalues and eigenvectors of small matrices . . . 73
4.3.2 Eigenmode analysis of an oscillating string . . . 75
4.3.3 Eigenmode analysis of surface waves in a shallow rectangular pool . . . 78
4.4 Fundamental matrix decompositions . . . 80
4.4.1 The Hessenberg decomposition . . . 81
4.4.2 The *QR* decomposition . . . 83
4.4.3 Characterizing the Schur decomposition . . . 89
4.4.4 Characterizing the eigen decomposition . . . 92
4.4.4.1 Hermitian positive definite and Hermitian positive semidefinite matrices . . . 94
4.4.4.2 Hamiltonian and symplectic matrices . . . 96
4.4.5 Computing the eigen and Schur decompositions† . . . 97
4.4.6 The Jordan decomposition† . . . 108
4.4.7 The singular value decomposition (SVD) . . . 110
4.4.8 Decompositions related to Gaussian elimination: LDM^H, LDL^H, and Cholesky . . . 114
4.5 Efficient solution of some important matrix equations . . . 116
4.5.1 Solving the DLE and CALE . . . 117
4.5.1.1 Solving the Sylvester equation . . . 118
4.5.2 Solving the DRE and CARE . . . 119
4.5.3 Solving the LDE and DALE . . . 121
4.5.4 Solving the RDE and DARE . . . 122
4.5.5 Reordering the Schur decomposition . . . 125
4.6 The trace . . . 127
4.6.1 Identities involving the derivative of the trace of a product . . . 127

4.6.2 Computing the sensitivity of an eigenvalue to matrix perturbations . . . 127
4.6.3 Rewriting the characteristic polynomial using the trace . . . 128
4.7 The Moore-Penrose pseudoïnverse . . . 130
4.7.1 Inconsistent and/or underdetermined systems . . . 131
4.7.1.1 The *QR* approach to inconsistent systems . . . 131
4.7.2 The least-squares solution to the data fitting problem . . . 132
4.8 Chapter summary . . . 133
Exercises . . . 134

To recap, §1 reviewed the notation to be used throughout this text to frame efficient numerical methods for solving a variety of practical problems. Using this notation, §2 presented several algorithms for the efficient direct solution of linear equations of the form $A\mathbf{x} = \mathbf{b}$ for the unknown vector $\mathbf{x}$. For both linear and nonlinear equations which are too difficult to solve directly, §3 introduced a selection of iterative algorithms for approximating the solution; we will resort to such iterative algorithms in many of the problems to come. We now focus on several additional concepts and algorithms that clarify the **linear algebra** describing how the matrices at the heart of more involved problems may be characterized and decomposed.

4.1 The four fundamental subspaces of a matrix

An appropriate starting point for this chapter is to describe the transformation $\mathbf{y} = A\mathbf{x}$ related to an $m \times n$ matrix A in terms of the **four fundamental subspaces** of the **domain** X and the **codomain** Y, as introduced in §2.6, defined precisely below, summarized in Figures 4.1-4.3, and illustrated by example in Table 4.1.

- The **column space** of A (a.k.a. the **image** or, sometimes, the **range** of the corresponding linear transformation) is the subspace of all complex (or real) vectors $\mathbf{y}$ of order m (that is, all $\mathbf{y} \in \mathsf{Y}$) such that $\mathbf{y} = A\mathbf{x}$ for at least one value of $\mathbf{x}$ (that is, it is the set of all vectors $\mathbf{y}$ spanned by the columns of A). It is denoted $\mathsf{C} = \text{im}(A) = \text{span}\{\mathbf{a}^1, \mathbf{a}^2, \ldots, \mathbf{a}^n\}$, and has dimension r.
- The **row space** of A is the subspace of all complex (or real) vectors $\mathbf{x}$ of order n (that is, all $\mathbf{x} \in \mathsf{X}$) such that $\mathbf{x}^H = \mathbf{y}^H A$ for at least one value of $\mathbf{y}$. In other words, it is the set of all $\mathbf{x}$ such that $\mathbf{x} = A^H\mathbf{y}$ for at least one value of $\mathbf{y}$. It is thus denoted $\mathsf{R} = \text{im}(A^H)$, and has dimension r.
- The **nullspace** of A (a.k.a. the **kernel** of the corresponding linear transformation) is the subspace of all $\mathbf{x} \in \mathsf{X}$ such that $A\mathbf{x} = 0$. The nullspace is the orthogonal complement of the row space. It is denoted $\mathsf{N} = \ker(A) = \mathsf{R}^\perp$, and has dimension $n - r$.
- The **left nullspace** of A is the subspace of all $\mathbf{y} \in \mathsf{Y}$ such that $\mathbf{y}^H A = 0$, and is the orthogonal complement of the column space. In other words, it is the set of all $\mathbf{y}$ such that $A^H\mathbf{y} = 0$. It is thus denoted $\mathsf{L} = \ker(A^H) = \mathsf{C}^\perp$, and has dimension $m - r$.

If a square matrix A is nonsingular, then $r = \text{rank}(A) = m = n$ (see Fact 2.5), and thus the dimension of both the nullspace and the left nullspace of A, as depicted in Figure 4.1, are zero (that is, they both contain only the zero element). In this case, the mapping between X and Y in Figure 4.1 is one-to-one. That is, given any $\mathbf{y} \in \mathsf{Y}$, one can uniquely determine the corresponding $\mathbf{x} \in \mathsf{X}$ such that $\mathbf{y} = A\mathbf{x}$. This mapping is given by the inverse such that, for any $\mathbf{x} \in \mathsf{X}$, $A^{-1}\mathbf{y} = A^{-1}(A\mathbf{x}) = \mathbf{x}$.

If a matrix A is singular or nonsquare, then the mapping between X and Y in Figure 4.1 is not one-to-one. If $r < n$, then the component $\mathbf{x}_\mathsf{N}$ of a vector $\mathbf{x}$ lying in the nullspace of A is mapped to zero by the transformation $A\mathbf{x}$. If $r < m$, then the component $\mathbf{y}_\mathsf{L}$ of a vector $\mathbf{y}$ lying in the left nullspace of A may not be reached by the transformation $\mathbf{y} = A\mathbf{x}$ for any $\mathbf{x}$. The "best" mapping possible from $\mathbf{y}$ back to $\mathbf{x}$ under such conditions is given by the **Moore-Penrose pseudoinverse** A^+ (see Figure 4.3 and §4.7); the **singular value decomposition** developed in §4.4.7 renders the computation and analysis of A^+ straightforward.

4.2 The determinant

There are several measures with which the properties of a matrix may be quantified. Perhaps the most fundamental of these measures is the **determinant**, denoted $|A|$. The determinant of a square matrix A may be defined iteratively as follows:

- 1×1 **case:** The determinant of a 1×1 matrix $A = \begin{pmatrix} a_{11} \end{pmatrix}$ is just $|A| = a_{11}$.
- 2×2 **case:** The determinant of a 2×2 matrix $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ is $|A| = a_{11}a_{22} - a_{12}a_{21}$.
- $n \times n$ **case:** The determinant of an $n \times n$ matrix is defined as a function of the determinant of several $(n-1) \times (n-1)$ matrices as follows: the determinant of A is a linear combination of the elements of row α (for any α such that $1 \leq \alpha \leq n$) and their corresponding **cofactors** $A_{\alpha\beta}$:

$$|A| = a_{\alpha 1}A_{\alpha 1} + a_{\alpha 2}A_{\alpha 2} + \cdots + a_{\alpha n}A_{\alpha n} = \sum_{\beta=1}^{n} (-1)^{\alpha+\beta} a_{\alpha\beta}|M_{\alpha\beta}| \quad \text{for some } \alpha \in [1,2,\ldots n], \quad (4.1a)$$

where the **cofactor** $A_{\alpha\beta}$ is defined as the determinant of $M_{\alpha\beta}$ with alternating sign: $A_{\alpha\beta} = (-1)^{\alpha+\beta}|M_{\alpha\beta}|$, where the **minor** $M_{\alpha\beta}$ is the matrix formed by deleting row α and column β of the matrix A. Alternatively, the determinant of an $n \times n$ matrix may be defined as a linear combination of the elements of column β (for any β such that $1 \leq \beta \leq n$) and their corresponding cofactors:

$$|A| = a_{1\beta}A_{1\beta} + a_{2\beta}A_{2\beta} + \cdots + a_{n\beta}A_{n\beta} = \sum_{\alpha=1}^{n} (-1)^{\alpha+\beta} a_{\alpha\beta}|M_{\alpha\beta}| \quad \text{for some } \beta \in [1,2,\ldots n]. \quad (4.1b)$$

4.2.1 Some important properties of the determinant, and their consequences

The determinant has five important properties that may, with some effort, be verified by its definition:

Property 1 *Adding a multiple of one row (or block row) of a matrix to another row (or block row) leaves the determinant unchanged. In particular,*

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} a & b \\ 0 & d - \frac{c}{a}b \end{vmatrix} = \begin{vmatrix} a - \frac{b}{d}c & 0 \\ c & d \end{vmatrix} \qquad \textit{and} \qquad \begin{vmatrix} A & B \\ C & D \end{vmatrix} = \begin{vmatrix} A & B \\ 0 & D - CA^{-1}B \end{vmatrix} = \begin{vmatrix} A - BD^{-1}C & 0 \\ C & D \end{vmatrix},$$

provided that each of the operations performed is valid (that is, in the examples shown above, when $a \neq 0$, $d \neq 0$, A is nonsingular, or D is nonsingular, respectively).

Property 2 *Exchanging two rows of the matrix flips the sign of the determinant, e.g.,*

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = -\begin{vmatrix} c & d \\ a & b \end{vmatrix}.$$

Property 3a *If A is (upper or lower) triangular (or diagonal), then $|A|$ is the product $a_{11}a_{22}\cdots a_{nn}$ of the elements on the main diagonal. In particular, the determinant of the identity matrix is $|I| = 1$.*
Property 3b *If A is (upper or lower) block triangular (or block diagonal), then $|A|$ is the product of the determinants of the blocks on the main diagonal.*

Property 4a *If $|A| \neq 0$, then the rows of A are linearly independent, and A is nonsingular / invertible / full rank (i.e., $A\mathbf{x} = \mathbf{b}$ has a unique solution $\mathbf{x}$).*
Property 4b *If $|A| = 0$, then the rows of A are linearly dependent, and A is singular / non-invertible / rank deficient (i.e., $A\mathbf{x} = \mathbf{b}$ either has an infinite number of solutions or zero solutions, depending on $\mathbf{b}$).*

Property 5 $|AB| = |A| \cdot |B|$. *In particular, $|AB| = 0$ iff $|A| = 0$ or $|B| = 0$, and $|A^{-1}| = 1/|A|$.*

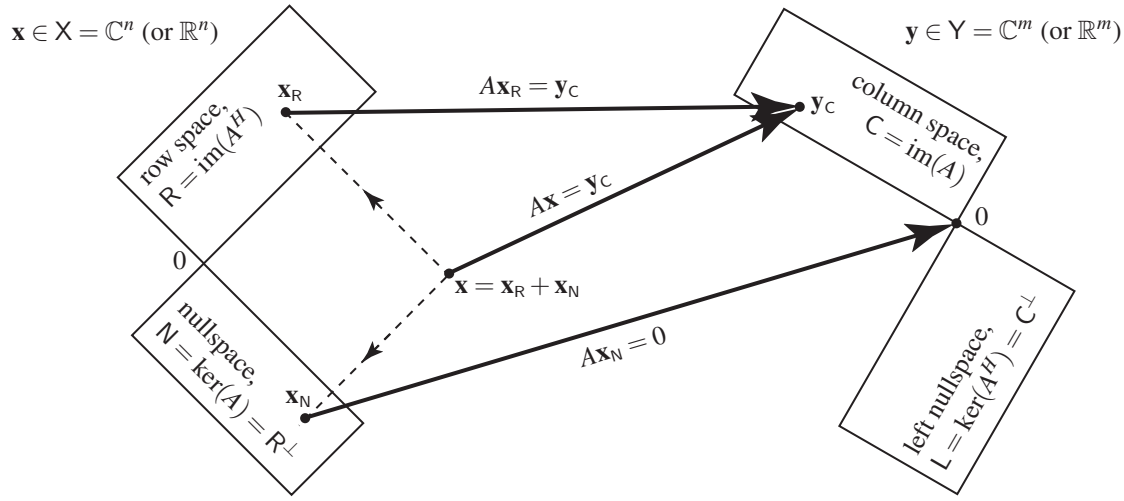


Figure 4.1: Cartoon depicting the four fundamental subspaces of the matrix A, adapted from Strang (1988).

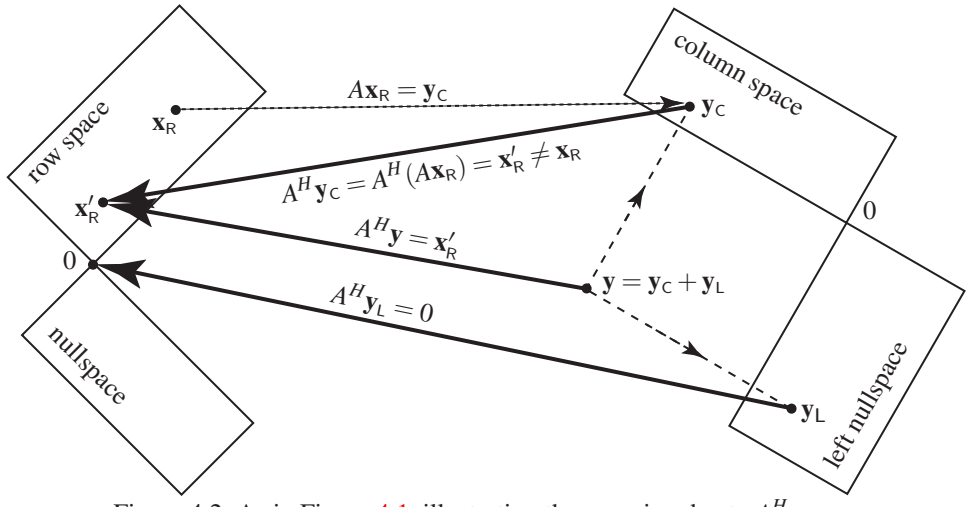


Figure 4.2: As in Figure 4.1, illustrating the mapping due to A^H.

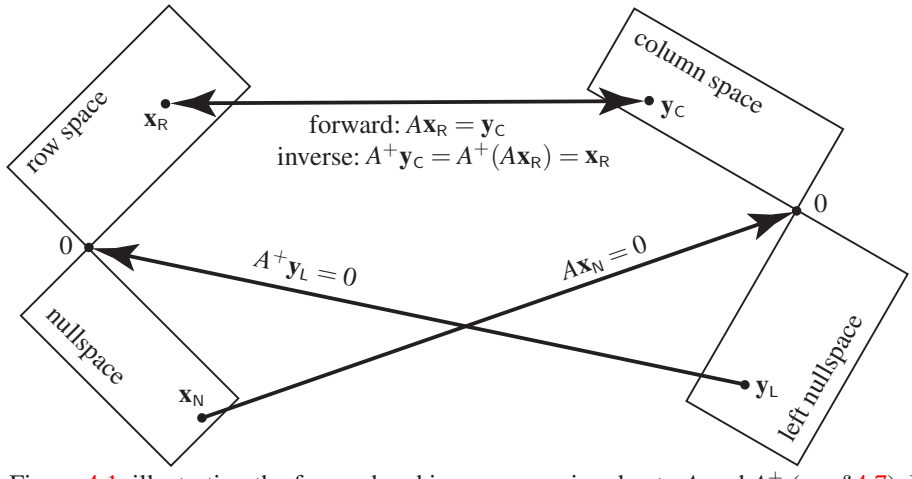


Figure 4.3: As in Figure 4.1, illustrating the forward and inverse mapping due to A and A^+ (see §4.7). If A is nonsingular (i.e., if $r = m = n$), then the nullspace and left nullspace are given by $\{0\}$, and thus $A^+ = A^{-1}$.

matrix A	**column space** $\mathsf{C} = \mathrm{im}(A)$	**left nullspace** $\mathsf{L} = \ker(A^H)$	**row space** $\mathsf{R} = \mathrm{im}(A^H)$	**nullspace** $\mathsf{N} = \ker(A)$	A is ...	$A\mathbf{x} = \mathbf{b}$ is ...
$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	$\left\{ \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \begin{pmatrix} 2 \\ 4 \end{pmatrix} \right\}$	$\{0\}$	$\left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 4 \end{pmatrix} \right\}$	$\{0\}$	**square** ($m = n$), **invertible / nonsingular / full rank** ($r = m$, $r = n$)	**uniquely determined** (as $\mathsf{N} = \mathsf{L} = \{0\}$) $\Rightarrow$ 1 solution
$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$	$\left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$	$\{0\}$	**tall** ($m > n$), **full (column) rank** ($r = n$)	**potentially inconsistent** (since $\mathsf{L} \neq \{0\}$) $\Rightarrow$ 1 sol. if $\mathbf{b} \in \mathsf{C}$, otherwise none
$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 2 & 3 \end{pmatrix}$	$\left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right\}$	$\{0\}$	$\left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 0 \\ 3 \\ -2 \end{pmatrix} \right\}$	**fat** ($m < n$), **full (row) rank** ($r = m$)	**underdetermined** (since $\mathsf{N} \neq \{0\}$) $\Rightarrow \infty$ solutions
$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}$	**fat** ($m < n$), **rank deficient** ($r < m$, $r < n$)	**underdetermined** (since $\mathsf{N} \neq \{0\}$) and **potentially inconsistent** (since $\mathsf{L} \neq \{0\}$) $\Rightarrow \infty$ sol. if $\mathbf{b} \in \mathsf{C}$, otherwise none
$\begin{pmatrix} 1 & 3 \\ 2 & 6 \end{pmatrix}$	$\left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 2 \\ -1 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 1 \\ 3 \end{pmatrix} \right\}$	$\left\{ \begin{pmatrix} 3 \\ -1 \end{pmatrix} \right\}$	**square** ($m = n$), **noninvertible / singular / rank deficient** ($r < m$, $r < n$)	**underdetermined** (since $\mathsf{N} \neq \{0\}$) and **potentially inconsistent** (since $\mathsf{L} \neq \{0\}$) $\Rightarrow \infty$ sol. if $\mathbf{b} \in \mathsf{C}$, otherwise none

Table 4.1: Five examples indicating simple (but nonorthogonal) bases of the column space, row space, nullspace, and left nullspace of some representative matrices A, applicable names for these matrices, and descriptions of the corresponding systems of linear equations $A\mathbf{x} = \mathbf{b}$.

Taking Properties 4a and 4b together, it is seen that $|A| = 0$ iff A is singular. Properties 1 and 3 establish (by taking $A = I_{m\times m}$ and $D = I_{n\times n}$) that $|I - CB| = |I - BC|$. Properties 1 and 3 also establish the following.

Fact 4.1 *Let* $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$. *Then:* $\begin{cases} \text{If } |M| \neq 0 \text{ and } |A| \neq 0, \text{ then } |D - CA^{-1}B| \neq 0. \\ \text{If } |M| \neq 0 \text{ and } |D| \neq 0, \text{ then } |A - BD^{-1}C| \neq 0. \end{cases}$

This fact allows us to establish a block form of the Matrix Inversion Lemma (again, easily verified simply by multiplying the original matrix by the formulae given for its inverse).

Fact 4.2 (The Matrix Inversion Lemma, part 2) *Let* $\tilde{A} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ *and assume that* $|\tilde{A}| \neq 0$. *Then:*

a) If $|A| \neq 0$, *then define a* **Schur complement** *of* $\tilde{A}$ *as* $G = D - CA^{-1}B$. *By Fact 4.1,* $|G| \neq 0$, *and thus*

$$\tilde{A}^{-1} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}BG^{-1}CA^{-1} & -A^{-1}BG^{-1} \\ -G^{-1}CA^{-1} & G^{-1} \end{bmatrix}.$$

b) If $|D| \neq 0$, *define the other* **Schur complement** *of* $\tilde{A}$ *as* $H = A - BD^{-1}C$. *By Fact 4.1,* $|H| \neq 0$, *and thus*

$$\tilde{A}^{-1} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} H^{-1} & -H^{-1}BD^{-1} \\ -D^{-1}CH^{-1} & D^{-1} + D^{-1}CH^{-1}BD^{-1} \end{bmatrix}.$$

c) If both $|A| \neq 0$ *and* $|D| \neq 0$, *then by (a) and (b) above and the uniqueness of the matrix inverse (Fact 1.5), using both Schur complements* $G = D - CA^{-1}B$ *and* $H = A - BD^{-1}C$, *we may write*

$$\tilde{A}^{-1} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} H^{-1} & -H^{-1}BD^{-1} \\ -G^{-1}CA^{-1} & G^{-1} \end{bmatrix}.$$

Parts (a) and (b) of Fact 4.2 demonstrate how a given matrix inverse (A^{-1} or D^{-1}, respectively) may be updated via a computationally inexpensive algorithm when a (block) row and column are appended to the original matrix (A or D). It follows from part (c) of Fact 4.2 that

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} \mathbf{x}_o \\ \mathbf{x}_e \end{bmatrix} = \begin{bmatrix} \mathbf{b}_o \\ \mathbf{b}_e \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} \mathbf{x}_o \\ \mathbf{x}_e \end{bmatrix} = \begin{bmatrix} H^{-1} & -H^{-1}BD^{-1} \\ -G^{-1}CA^{-1} & G^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{b}_o \\ \mathbf{b}_e \end{bmatrix} \tag{4.2a}$$

$$\Rightarrow \quad \begin{bmatrix} A - BD^{-1}C & 0 \\ 0 & D - CA^{-1}B \end{bmatrix} \begin{bmatrix} \mathbf{x}_o \\ \mathbf{x}_e \end{bmatrix} = \begin{bmatrix} I & -BD^{-1} \\ -CA^{-1} & I \end{bmatrix} \begin{bmatrix} \mathbf{b}_o \\ \mathbf{b}_e \end{bmatrix}; \tag{4.2b}$$

this relation forms the basis for the **cyclic reduction** algorithm for the parallel solution of checkerboard systems described in Exercise 4.5.

Note also that, if $\alpha \neq \beta$, we may write

$$a_{\alpha 1}A_{\beta 1} + a_{\alpha 2}A_{\beta 2} + \cdots + a_{\alpha n}A_{\beta n} = 0. \tag{4.3}$$

The above expression is valid because, as easily verified, it expresses the determinant of a new matrix B which is identical to matrix A except in row $\beta \neq \alpha$, which has its former elements replaced by a copy of row α; note that this modification to row β leaves the cofactors $A_{\beta j}$ unchanged for all j, but (by Property 4b above) makes the determinant of the new matrix zero. We may thus assemble (4.1a) (for all α) together with (4.3) (for all $\{\alpha, \beta\}$ such that $\alpha \neq \beta$) in the following convenient matrix form

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} A_{11} & A_{21} & \dots & A_{n1} \\ A_{12} & A_{22} & \dots & A_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{1n} & A_{2n} & \dots & A_{nn} \end{pmatrix}}_{A_{\text{cof}}} = \begin{pmatrix} |A| & & & 0 \\ & |A| & & \\ & & \ddots & \\ 0 & & & |A| \end{pmatrix}.$$

Note that the $\{i,j\}$ element of the **cofactor matrix** A_{cof} is the cofactor A_{ji}; in other words, the cofactors of A are assembled into the cofactor matrix A_{cof} in a transposed fashion. The following fact follows immediately:

Fact 4.3 (Cramer's rule) $A^{-1} = A_{\text{cof}}/|A|$.

Fact 4.3 is an extraördinarily expensive formula for the inverse when A is large; it is thus used only once (in §20.1.5) in the remainder of this entire text. Note that Fact 1.8 is a special case of Fact 4.3 for $n = 2$.

4.2.2 Computing the determinant

For a large matrix A, the determinant is most easily computed by performing the row operations mentioned in Properties 1 and 2 of §4.2.1 to reduce A to an upper triangular matrix U. In fact, this is the heart of Gaussian elimination procedure, described in §2. Taking Properties 1, 2, and 3 together, it follows that

$$|A| = (-1)^r |U| = (-1)^r u_{11} u_{22} \cdots u_{nn},$$

where r is the number of row exchanges performed during Gaussian elimination, and the $u_{\kappa\kappa}$ are the elements on the main diagonal of the upper-triangular matrix U that results from the Gaussian elimination procedure.

4.3 An introduction to eigenvalues and eigenvectors

Consider the equation

$$A\mathbf{s} = \lambda \mathbf{s}. \tag{4.4}$$

For most values of λ, the only solution of this problem is the **trivial** solution $\mathbf{s} = 0$. However, for certain special values of λ, this equation admits other **nontrivial** solutions $\mathbf{s} \neq 0$. These special values of λ are called the **eigenvalues**, and the corresponding vectors $\mathbf{s}$ are called the **right eigenvectors**, or more commonly simply as the **eigenvectors**. For these special values of λ, premultiplying $\mathbf{s}$ by the matrix A is equivalent to simply scaling $\mathbf{s}$ by the factor λ. Such a situation has the important physical interpretation as a natural **mode** of a system when A represents the system matrix for a given dynamic system, as discussed further in §4.3.2. Note also that those vectors $\mathbf{r}$ that satisfy the equation $\mathbf{r}^H A = \lambda \mathbf{r}^H$ (equivalently, $A^H \mathbf{r} = \lambda^H \mathbf{r}$) are referred to as the **left eigenvectors**; note that the alternative definition $\mathbf{q}^T A = \lambda \mathbf{q}^T$ (equivalently, $A^T \mathbf{q} = \lambda^T \mathbf{q}$) of the left eigenvector is sometimes more convenient both algebraically and computationally, so which definition is being used must always be specified when considering complex systems.

4.3.1 Computing the eigenvalues and eigenvectors of small matrices

The most direct way to determine for which λ it is possible to solve the equation $A\mathbf{s} = \lambda \mathbf{s}$ for $\mathbf{s} \neq 0$ is to rewrite this equation as

$$(\lambda I - A)\mathbf{s} = 0.$$

If $(\lambda I - A)$ is a nonsingular matrix, then this equation has a unique solution, and since the RHS is zero, that solution must be $\mathbf{s} = 0$. However, for those values of λ for which $(\lambda I - A)$ is singular, this equation admits other solutions with $\mathbf{s} \neq 0$. The values of λ for which $(\lambda I - A)$ is singular are the eigenvalues of the matrix A, and the corresponding vectors $\mathbf{s}$ are the eigenvectors. Making use of Property 4 of the determinant (see §4.2.1), we see that the eigenvalues must therefore be exactly those values of λ for which

$$|\lambda I - A| = 0 \qquad \Rightarrow \qquad p(\lambda) \triangleq \lambda^n + a_{n-1}\lambda^{n-1} + \ldots + a_1 \lambda + a_0 = 0. \tag{4.5a}$$

The determinant on the LHS of the first equation in (4.5a), when multiplied out, is seen to be a polynomial in λ of degree n, and is known as the **characteristic polynomial** of A and is usually denoted $p(\lambda)$; the equation $p(\lambda) = 0$ is referred to as the **characteristic equation** of A.

Fact 4.4 (The Fundamental Theorem of Algebra) *Any n'th-order polynomial of the form (4.5a) has exactly n complex roots, λ_1 through λ_n, and may thus be written in the form*

$$(\lambda-\lambda_1)(\lambda-\lambda_2)\cdots(\lambda-\lambda_n)=0. \tag{4.5b}$$

Proof of the Fundamental Theorem of Algebra is given in Appendix B. Note that the roots λ_1 through λ_n are not necessarily **distinct** (that is, they are not necessarily all different); the number of times a particular eigenvalue is repeated is referred to as its **multiplicity** (a.k.a. **algebraic multiplicity**; cf. geometric multiplicity as defined in footnote 21 on page 108). An eigenvalue with multiplicity 1 is referred to as a **simple eigenvalue**.

The set of all eigenvalues of the matrix A is denoted $\lambda(A)$. To calculate the eigenvalues and eigenvectors of a small matrix by hand, simply calculate the roots of its characteristic polynomial. Once the eigenvalues λ are known, the corresponding eigenvectors $\mathbf{s}$ may be found by determining a nontrivial solution to the singular system of equations $(\lambda I-A)\mathbf{s}=0$. Efficient techniques to solve a singular system of this sort are discussed in §2.6. Note that the eigenvectors $\mathbf{s}$ are defined only to within a multiplicative constant, which cannot be specified uniquely because $(\lambda I-A)$ is singular. In other words, if $\mathbf{s}$ is an eigenvector corresponding to a particular eigenvalue λ, then $c\mathbf{s}$ is also an eigenvector for any complex scalar c. It follows that

Fact 4.5 *If* $A=A_{2\times2}=\begin{pmatrix}a & b\\ c & d\end{pmatrix}$, *then* $\lambda_\pm=\frac{1}{2}\left[(a+d)\pm\sqrt{4bc+(a-d)^2}\right]$.
If $b\neq0$, *the eigenvectors are* $\mathbf{s}_\pm=\begin{pmatrix}b\\ \lambda_\pm-a\end{pmatrix}$; *if* $c\neq0$, *the eigenvectors are* $\mathbf{s}_\pm=\begin{pmatrix}\lambda_\pm-d\\ c\end{pmatrix}$.
If $b=c=0$, *the eigenvalues are* $\lambda_+=a$ *and* $\lambda_-=d$ *and the eigenvectors are* $\mathbf{s}_1=\begin{pmatrix}1\\0\end{pmatrix}$ *and* $\mathbf{s}_2=\begin{pmatrix}0\\1\end{pmatrix}$.
If $4bc+(a-d)^2=0$ *and either* $b\neq0$ *or* $c\neq0$, *or both, then there is only a single eigenvector (and a single corresponding eigenvalue with multiplicity two); otherwise, there are two linearly independent eigenvectors [that is, the angle between the eigenvectors is nonzero; see (1.19)].*
If $A=A_{2\times2}$ *is Hermitian, then a and d are real and* $b=\overline{c}$, *and thus the eigenvalues of A are real.*
If $A=A_{2\times2}$ *is real, then the eigenvalues, if they are complex, are a complex conjugate pair, and the eigenvectors are orthogonal to each other (that is,* $\mathbf{s}_+\cdot\mathbf{s}_-=0$*).*

In general, for a matrix $A_{n\times n}$ with $n>4$, it is impossible to compute the roots of the characteristic polynomial, and thus determine the eigenvalues of A, with a finite sequence of calculations (see Facts 4.8 and 4.9). However, any of several efficient iterative algorithms (akin to those introduced in §3) may be used for approximating eigenvalues and eigenvectors numerically. An introduction to such iterative techniques is presented in §4.4, along with the matrix decompositions upon which these techniques are based. Note that robust and efficient eigenvalue solvers have already been developed in most computer languages used today for scientific computing. For the purpose of many typical applications, these prepackaged **black box** algorithms may often be called effectively without knowing exactly how they work. In order to use such algorithms reliably, however, it is valuable to understand the fundamental algorithms upon which they are built.

4.3.2 Eigenmode analysis of an oscillating string

To demonstrate the significance of eigenvalues and eigenvectors for characterizing physical systems, as discussed further in later chapters, it is enlightening to diverge for a bit from the present line of development and discuss two physical problems motivating eigenvalue/eigenvector analyses. Consider first the time evolution of a taut string which has just been struck or plucked. Neglecting damping, the deflection of the string, $f(x,t)$, obeys the linear **partial differential equation** (**PDE**) known as the **1D wave equation**:

$$\frac{\partial^2 f}{\partial t^2} = c^2 \frac{\partial^2 f}{\partial x^2}, \tag{4.6a}$$

where c is the speed of wave propogation in the string, subject to

$$\textbf{boundary conditions (BCs)}: \quad \left\{ f = 0 \quad \text{at } x = 0 \text{ and } x = L, \right. \tag{4.6b}$$

$$\text{and } \textbf{initial conditions (ICs)}: \quad \left\{ f = a(x) \text{ and } \frac{\partial f}{\partial t} = b(x) \quad \text{at } t = 0. \right. \tag{4.6c}$$

For a gently plucked guitar string, $b(x) \approx 0$ and $a(x)$ describes the initial displacement of the string. For an impulsively struck piano string, $a(x) \approx 0$ and $b(x)$ describes the initial velocity of the string.

We will solve this system using the **separation of variables** (**SOV**) approach. With this approach, an assumed **separable** (that is, decoupled) form is imposed on the individual **modes** of the solution (this is often called the **SOV ansatz**). We then attempt to construct a general solution to the full system (4.6) built from such modes. We thus first seek modes, f^m, which satisfy the PDE (4.6a) *and* the BCs (4.6b) of the separable form

$$f^m(x,t) = X(x)\,T(t). \tag{4.7}$$

[Note that $X = X(x)$ and $T = T(t)$ in this section are simply scalar functions of their arguments, not matrices.] If we can find enough nontrivial (that is, nonzero) solutions of (4.6a) which fit this form, we will be able to reconstruct a solution which also satisfies the ICs (4.6c) as a superposition of these modes. Inserting (4.7) into (4.6a) and assuming c =constant, we identify two associated **ordinary differential equations** (**ODEs**)

$$XT'' = c^2 X''T \quad \Rightarrow \quad \frac{T''}{T} = c^2 \frac{X''}{X} \triangleq -\omega^2 \quad \Rightarrow \quad \begin{array}{l} T'' = -\omega^2 T, \\ X'' = -{k_x}^2 X, \end{array}$$

where $\omega = ck_x$, and where the constant ω (and, thus, k_x) must be independent of both x and t due to the middle form above combined with the facts that $X = X(x)$ and $T = T(t)$. The two ODEs at right are solved with:

$$T = A\cos(\omega t) + B\sin(\omega t), \tag{4.8a}$$

$$X = C\cos(k_x x) + D\sin(k_x x). \tag{4.8b}$$

Due to the BCs at $x = 0$, it follows that $C = 0$. Due to the BCs at $x = L$, it follows for most k_x that $D = 0$ as well, and thus $f^m(x,t) = 0$ for all $\{x,t\}$. However, for certain values of k_x (specifically, for $k_{x_i}L = i\pi$ for integer values of i), X satisfies the homogeneous BCs at $x = L$ even for nonzero values of D. These special values of k_{x_i} are the **eigenvalues of the PDE system** in SOV form[1] Defining (for convenience) the combined

[1] The problem of determining the admissible values k_{x_i}, and corresponding functions $X_i(x)$, such that each nontrivial function $X_i(x)$ satisfies both the ODE $X'' = -{k_x}^2 X$ and homogeneous BCs at both $x = 0$ and $x = L$ is a special case of the general problem of determining the **eigenvalues** λ_i, and corresponding **eigenfunctions** $u_i(x)$, of the homogeneous **Sturm-Liouville eigenvalue problem**

$$\left[\frac{d}{dx}p(x)\frac{d}{dx} + q(x) + \lambda r(x)\right]u(x) = 0 \quad \text{with} \quad \begin{cases} c_1 u(a) + c_2 u'(a) = 0, \\ c_3 u(b) + c_4 u'(b) = 0, \end{cases}$$

where $p(x) > 0$, $p'(x)$, $q(x)$, and $r(x) > 0$ are continuous on $x \in (a,b)$, and $\{c_1, c_2, c_3, c_4\}$ are constants. Another example of a problem of this class, related to **Bessel functions**, is given in (11.36); see Amrein *et al.* (2005) for further discussion.

constants $\hat{a}^s = AD$ and $\hat{b}^s = \omega BD$ and forming a **linear superposition** of the nontrivial modes f_i^m [each of which satisfying the SOV Ansatz (4.7) for each value of k_{x_i} and corresponding value of ω_i] in an attempt to additionally satisfy the ICs (4.6c), we now write

$$f = \sum_{i=1}^{\infty} f_i^m = \sum_{i=1}^{\infty} \left[\hat{a}_i^s \cos(\omega_i t) + \frac{\hat{b}_i^s}{\omega_i} \sin(\omega_i t) \right] \sin(k_{x_i} x). \tag{4.9}$$

The coefficients $\hat{a}_i^s$ and $\hat{b}_i^s$ are now determined by enforcing the ICs:

$$f(x,t=0) = a(x) = \sum_{i=1}^{\infty} \hat{a}_i^s \sin(k_{x_i} x), \qquad \frac{\partial f}{\partial t}(x,t=0) = b(x) = \sum_{i=1}^{\infty} \hat{b}_i^s \sin(k_{x_i} x). \tag{4.10a}$$

Noting the **orthogonality of the sine and cosine functions**[2], we multiply both of the above equations by $\sin(k_{x_p} x) = \sin(p\pi x/L)$ and integrate over the domain $x \in [0,L]$, which results in:

$$\left.\begin{aligned} \int_0^L a(x) \sin(k_{x_p} x)\,dx = \hat{a}_p^s \frac{L}{2} \quad &\Rightarrow \quad \hat{a}_p^s = \frac{2}{L} \int_0^L a(x) \sin(k_{x_p} x)\,dx \\ \int_0^L b(x) \sin(k_{x_p} x)\,dx = \hat{b}_p^s \frac{L}{2} \quad &\Rightarrow \quad \hat{b}_p^s = \frac{2}{L} \int_0^L b(x) \sin(k_{x_p} x)\,dx \end{aligned}\right\} \quad \text{for } p = 1,2,3,\ldots \tag{4.10b}$$

Thus, the $\hat{a}_p^s$ and $\hat{b}_p^s$ may be calculated directly. As discussed further in §5.11.1, these representations are referred to as the **infinite sine transforms** of $a(x)$ and $b(x)$ on the interval $x \in [0,L]$.

An analytic solution of the PDE (4.6a) satisfying both the BCs (4.6b) and the ICs (4.6c) may thus be constructed as a linear combination of separable modes, as shown in (4.9), the coefficients of which may easily be determined, as shown in (4.10b).

Now consider the PDE[3] in (4.6a) in the case in which c is a function of x. We can no longer represent the mode shapes in x analytically with sines and cosines. In this case, we can still seek decoupled modes of the form $f = X(x)T(t)$, but in general we must now determine the $X(x)$ numerically. The ODE for $X(x)$ is

$$X'' = -\frac{\omega^2}{c^2} X \qquad \text{for } x \in (0,L), \quad \text{with} \tag{4.11a}$$

$$X = 0 \qquad \text{at } x = 0 \text{ and } x = L. \tag{4.11b}$$

Consider now the values of X at $N+1$ discrete locations (a.k.a. **gridpoints**) located at $x = x_j = j\Delta x$ for $j = 0 \ldots N$, where $\Delta x = L/N$. Note that, at these gridpoints, derivatives may be approximated as follows:

$$\left.\frac{\partial X}{\partial x}\right|_{x_j} \approx \frac{X_{j+1} - X_{j-1}}{2\Delta x}, \qquad \left.\frac{\partial^2 X}{\partial x^2}\right|_{x_j} \approx \left(\frac{X_{j+1} - X_j}{\Delta x} - \frac{X_j - X_{j-1}}{\Delta x} \right) / \Delta x = \frac{X_{j+1} - 2X_j + X_{j-1}}{(\Delta x)^2},$$

where we denote $X_j \triangleq X(x_j)$. By the BCs (4.11b), $X_0 = X_N = 0$. The ODE (4.11a) at each of the $N-1$ grid points on the interior may be approximated by the relation

$$c_j^2 \frac{X_{j+1} - 2X_j + X_{j-1}}{(\Delta x)^2} = -\omega^2 X_j.$$

---

[2] This orthogonality principle states that, for i, k integers, $\displaystyle\int_0^L \sin\left(\frac{i\pi x}{L}\right) \cos\left(\frac{k\pi x}{L}\right) dx = 0,$

$$\int_0^L \sin\left(\frac{i\pi x}{L}\right) \sin\left(\frac{k\pi x}{L}\right) dx = \begin{cases} L/2 & i = k \neq 0 \\ 0 & \text{otherwise,} \end{cases} \quad \text{and} \quad \int_0^L \cos\left(\frac{i\pi x}{L}\right) \cos\left(\frac{k\pi x}{L}\right) dx = \begin{cases} L & i = k = 0 \\ L/2 & i = k \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

[3] Note: this is *not* the PDE governing waves in a nonuniform bar; treatment of this physical problem is considered in Exercise 4.4.

Algorithm 4.1: Compute and animate the leading natural modes of vibration of a string.

View

```
function WireTest
% This function computes, plots, & animates the leading modes of vibration of a wire.
disp('Now computing, ploting, & animating the leading modes of vibration of a wire.')
clear; close all; c=1; L=1; n=128; numplots=3;  % Initialize the simulation parameters
DeltaX=L/n;  X=[0:DeltaX:L];                    % Set up grid and the A matrix
A=(c^2/DeltaX^2)*(diag(ones(n-2,1),-1) - 2*diag(ones(n-1,1),0) + diag(ones(n-2,1),1));
[lam,S]=Eig(A);   [scratch,index]=MergeSort(abs(lam),0,n-1); S=S(:,index); lam=lam(index);
omega_exact=[1:numplots]*pi*c/L, n, omega_numerical=sqrt(-lam(1:numplots))'
for m=1:numplots, disp(sprintf('This is a plot of mode %d',m))
   amp=1/max(abs(S(:,m))); clf; plot(X,amp*[0 S(:,m)' 0],'*'), axis([0 1 -1 1]), pause
end, WireAnimate(lam,S,X,n,DeltaX), disp(' ')
end % function WireTest
%%%
```

mode shapes $X_i(x) = \sin(\omega_i x/c)$. Note that this numerical code is easily modified to handle the situation in which c varies with x, though this case cannot be solved analytically.

Thus, the eigenvalues and eigenvectors that satisfy (4.12) may be used to approximate the ω_i and $X_i(x)$ of the SOV modes satisfying the PDE (4.6a) and the BCs (4.6b) even in the case that the eigenvalues of the PDE system cannot be determined analytically. The equation for $T_i(t)$ is the same as before; its solution is given by (4.8a). Forming a superposition of the nontrivial modes $\mathbf{f}^i$ in this spatially discretized setting in an attempt to additionally satisfy the ICs (4.6c), we express [cf. (4.9)]

$$\mathbf{f} = \sum_{i=1}^{N} \mathbf{f}^i = \sum_{i=1}^{N} \left[\hat{a}_i^s \cos(\omega_i t) + \frac{\hat{b}_i^s}{\omega_i} \sin(\omega_i t) \right] \mathbf{s}^i. \tag{4.13}$$

Enforcing the ICs at the grid points, denoting $a_i = a(x_i)$ and $b_i = b(x_i)$, we have [cf. (4.10a)]

$$\mathbf{f}(t=0) = \mathbf{a} = S\hat{\mathbf{a}}^s, \quad \frac{d\mathbf{f}}{dt}(t=0) = \mathbf{b} = S\hat{\mathbf{b}}^s \quad \text{where} \quad S = \begin{bmatrix} | & | & & | \\ \mathbf{s}^1 & \mathbf{s}^2 & \dots & \mathbf{s}^N \\ | & | & & | \end{bmatrix} \tag{4.14a}$$

Assuming S is invertible, we may calculate the appropriate values of $\hat{\mathbf{a}}^s$ and $\hat{\mathbf{b}}^s$ directly [cf. (4.10b)]

$$\hat{\mathbf{a}}^s = S^{-1}\mathbf{a}, \qquad \hat{\mathbf{b}}^s = S^{-1}\mathbf{b}. \tag{4.14b}$$

As discussed further in §5.11, $\hat{\mathbf{a}}^s$ and $\hat{\mathbf{b}}^s$ are referred to as the **discrete sine transforms**[4] of $\mathbf{a}$ and $\mathbf{b}$.

Thus, a numerical approximation of the solution of the PDE (4.6a) satisfying both the BCs (4.6b) and the ICs (4.6c) may be constructed as a linear combination of separable modes, as shown in (4.13), the coefficients of which may easily be determined, as shown in (4.14).

The eigenvalue problem formulated above is implemented in Algorithm 4.1. Note that the function `WireTest.m` calls a built-in auxiliary function `WireAnimate.m`, which is isolated as a subfunction for clarity, as it is ancillary to the purpose of the main function `Wire.m`.

4.3.3 Eigenmode analysis of surface waves in a shallow rectangular pool

The analysis considered in §4.3.2 extends directly to multidimensional systems. To illustrate, consider the evolution of a low-amplitude wave in a shallow pool which is $L_x = 5$m wide $\times$ $L_y = 10$m long $\times$ $b = 1$m deep. Neglecting surface tension and viscosity, the height of the water, $h(x,y,t)$, obeys the **2D wave equation**:

$$\frac{\partial^2 h}{\partial t^2} = c^2 \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) \tag{4.15a}$$

where $g = 9.8\text{m/s}^2$ and $c = \sqrt{bg}$ is the speed of wave propogation in the system, subject to

$$\text{boundary conditions (BCs):} \quad \begin{cases} \dfrac{\partial h}{\partial x} = 0 & \text{at } x = 0 \text{ and } x = L_x \\ \dfrac{\partial h}{\partial y} = 0 & \text{at } y = 0 \text{ and } y = L_y, \end{cases} \tag{4.15b}$$

$$\text{and initial conditions (ICs):} \quad \left\{ h = a(x,y) \text{ and } \frac{\partial h}{\partial t} = b(x,y) \quad \text{at } t = 0. \right. \tag{4.15c}$$

[4] As S and S^{-1} are full, it appears at first glance that the transformations from $\mathbf{a}$ to $\hat{\mathbf{a}}^s$ and from $\hat{\mathbf{a}}^s$ back to $\mathbf{a}$ in (4.14) involve full matrix/vector multiplication, and thus each requires $\sim 2N^2$ real flops if $\mathbf{a}$ is real. We will see in §5.11.1 that we can in fact explot the several symmetries within S via a remarkable algorithm called the **fast fourier transform** (**FFT**; see §5.4.1) to solve this problem in only $\sim 5M \log_2 M$ real flops, where $M = N/2$. If $N = 256$, the latter algorithm is almost 30 times cheaper than the former. (!)

[The waves on a rectangular membrane may be analyzed in an analogous manner, simply by replacing the **homogeneous Neumann** BCs given above with **homogeneous Dirichlet** BCs (that is, $h = 0$ on the boundaries), by taking $c = \sqrt{T/\rho}$ where T is the tension per unit length (in N/m) of the membrane and ρ is the density per unit area (in kg/m^2) of the membrane, and by retaining the sine expansions in x and y instead of the cosine expansions in the derivation below.]

Inspired by (4.7), we seek the eigenmodes of the solution, f^m, which separate into the form

$$f^m(x,y,t) = X(x)\,Y(y)\,T(t). \tag{4.16}$$

Following the analysis given previously, we identify three associated ODEs

$$\frac{T''}{T} = c^2\left(\frac{X''}{X} + \frac{Y''}{Y}\right) \triangleq -\omega^2, \qquad \frac{X''}{X} = \frac{1}{c^2}\frac{T''}{T} - \frac{Y''}{Y} \triangleq -k_x^2, \qquad \frac{Y''}{Y} = \frac{1}{c^2}\frac{T''}{T} - \frac{X''}{X} \triangleq -k_y^2,$$

from which it follows that the constants ω, k_x, and k_y, are independent of x, y, and t, and are related such that $\omega^2 = c^2(k_x^2 + k_y^2)$. It follows that

$$T(t) = A\cos(\omega t) + B\sin(\omega t), \quad X(x) = C\cos(k_x x) + D\sin(k_x x), \quad Y(y) = E\cos(k_y y) + F\sin(k_y y).$$

Applying the BCs at $x = 0$ and $y = 0$ for all t implies that $D = 0$ and $F = 0$; applying the BCs at $x = L_x$ and $y = L_y$ for all t implies that, for most values of k_x and k_y, the coefficients C and E must equal zero as well, and thus the mode f^m is trivial (that is, zero). However, for certain values of k_x and k_y (specifically, for $k_{x_i}L_x = i\pi$ for $i = 0,1,2,\ldots$ and $k_{y_j}L_y = j\pi$ for $j = 0,1,2,\ldots$), $X(x)$ and $Y(y)$ satisfy the homogeneous Neumann BCs at $x = L_x$ and $y = L_y$ even for nonzero values of C and E. Defining $\hat{a}^c = ACE$ and $\hat{b}^c = \omega BCE$ and assembling a superposition of these nontrivial modes, defining $\omega_{ij} = c\sqrt{k_{x_i}^2 + k_{y_j}^2}$, we may write

$$f = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} f_{ij}^m = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\left[\hat{a}_{ij}^c\cos(\omega_{ij}t) + \frac{\hat{b}_{ij}^c}{\omega_{ij}}\sin(\omega_{ij}t)\right]\cos(k_{x_i}x)\cos(k_{y_j}y), \tag{4.17a}$$

where the coefficients $\hat{a}_{ij}^c$ and $\hat{b}_{ij}^c$ are determined by enforcing the ICs:

$$f(x,y,0) = a(x,y) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\hat{a}_{ij}^c\cos(k_{x_i}x)\cos(k_{y_j}y), \qquad \frac{\partial f}{\partial t}(x,y,0) = b(x,y) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\hat{b}_{ij}^c\cos(k_{x_i}x)\cos(k_{y_j}y).$$

Noting the orthogonality of the sine and cosine functions mentioned previously results in

$$\left.\begin{aligned}\hat{a}_{pq}^c &= \frac{4}{L_xL_y}\int_0^{L_x}\int_0^{L_y} a(x,y)\cos(k_{x_p}x)\cos(k_{y_q}x)\,dx\,dy \\ \hat{b}_{pq}^c &= \frac{4}{L_xL_y}\int_0^{L_x}\int_0^{L_y} b(x,y)\cos(k_{x_p}x)\cos(k_{y_q}x)\,dx\,dy\end{aligned}\right\} \quad \text{for } p = 0,1,2,\ldots \text{ and } q = 0,1,2,\ldots \tag{4.17b}$$

An analytic solution of the PDE (4.15a) satisfying both the BCs (4.15b) and the ICs (4.15c) may thus be constructed as a linear combination of separable modes, as shown in (4.17a), the coefficients of which may easily be determined, as shown in (4.17b).

The problem of the simulation of wave evolution in a shallow pool is considered further in §11.3.3 (see in particular Figure 11.6), where we extend our analysis to handle both variable depth of the pool and finite amplitude of the waves (and the associated nonlinear interactions). In the limiting case that the depth is constant and the waves are small in amplitude, the analytic solution given above may be used to quantify the accuracy of this more involved numerical simulation.

4.4 Fundamental matrix decompositions

This section presents a variety of useful ways to **decompose** or **factor** a given matrix A as the product of other matrices with special properties. The resulting **decompositions** (a.k.a. **factorizations**) form the foundation for many powerful and efficient numerical tools that are useful for a variety of tasks, including

- solving $A\mathbf{x} = \mathbf{b}$ in the case of square, nonsingular A,
- solving $A\mathbf{x} = \mathbf{b}$ in the inconsistent and/or underdetermined case,
- computing eigenvalues and eigenvectors,
- understanding linear dynamic systems, and
- solving linear feedback control problems.

Five of the decompositions we will consider,

- the **Hessenberg decomposition** $A = VT_0V^H$ where V is unitary and T_0 is upper Hessenberg (§4.4.1),
- the **Schur decomposition** $A = UTU^H$ where U is unitary and T is upper triangular (§4.4.3),
- the **real Schur decomposition** $A = U\hat{T}U^T$, for real A, where U is real and orthogonal and $\hat{T}$ is real and *block* upper triangular with 1×1 and 2×2 blocks on the main diagonal (§4.4.3),
- the **eigen decomposition** $A = S\Lambda S^{-1}$ where Λ is diagonal (§4.4.4), and
- the **Jordan decomposition** $A = MJM^{-1}$ where J is in so-called Jordan form (§4.4.6),

are referred to as **similarity transformations**. Such decompositions play a particularly important role in the analysis of a variety of problems in numerical linear algebra because of the following two facts.

Fact 4.6 *If $A = CBC^{-1}$ for some C (that is, if A and B are* **similar***), then A and B have the same eigenvalues, and if $\mathbf{s}$ is an eigenvector of A, then $C^{-1}\mathbf{s}$ is an eigenvector of B.*

Proof: Follows immediately from: $A\mathbf{s} = \lambda\mathbf{s} \quad \Rightarrow \quad CBC^{-1}\mathbf{s} = \lambda\mathbf{s} \quad \Rightarrow \quad B(C^{-1}\mathbf{s}) = \lambda(C^{-1}\mathbf{s})$. □

Fact 4.7 *If $A = CBC^H$ for some unitary C (that is, if A and B are* **unitarily similar** *or* **congruent***) and A is Hermitian (or symmetric), then B is also Hermitian (or symmetric).*

Proof: Follows immediately from: $B = C^HAC = C^HA^HC = (C^HAC)^H = B^H$. □

The subsections that follow are intended to be read sequentially: the **Hessenberg decomposition** forms a valuable preparatory step for applying the **QR method** (based on repeated **QR decompositions**) to compute the useful **Schur decomposition** (or the related **real Schur decomposition**), from which the immensely useful **eigen decomposition** may readily be determined (if it exists). The **Jordan decomposition** represents the matrix that is, in a sense, as close as you can get to a diagonalization of A via a similarity transformation when an eigen decomposition does not exist; unfortunately, as we will show, it is numerically ill-behaved. Rather, the final decomposition we introduce,

- the **singular value decomposition** $A = U\Sigma V^H$ where U and V are unitary and Σ is diagonal (§4.4.7),

though it is *not* a similarity transformations (as, in general, $U \neq V$), is the most useful generalization of the eigen decomposition that may be used even if an eigen decomposition of A does not exist or, indeed, even if A is nonsquare; it is shown in §4.7 that the singular value decomposition leads directly to an efficient construction of the **Moore-Penrose pseudoïnverse** A^+, as introduced in Figure 4.3. We also review the **decompositions related to Gaussian elimination**, as introduced in §2, as they may be understood a bit more deeply once the properties of positive definite and positive semidefinite matrices are established (in §4.4.4.1).

Recalling the caution at the beginning of §4 related to its difficulty and importance, it is no exaggeration to say that §*4.4 is the most difficult and important section of* §*4*. Definitive references on this material include Wilkenson (1965) and Golub & Van Loan (1996), which significantly extend the present discussion.

4.4.1 The Hessenberg decomposition

Every $n \times n$ matrix A has a **Hessenberg decomposition** $A = VT_0V^H$, where V is unitary and T_0 is upper Hessenberg. The Hessenberg decomposition, which is a unitary similarity transformation (see Fact 4.7), is useful primarily as a preparatory step in the computation of the Schur and eigen decompositions of a square matrix A. Recall from §1.2.7 that a Hessenberg form T_0 is an upper triangular matrix with extra nonzero elements populating its first subdiagonal. The Hessenberg decomposition is useful because it introduces many zero elements while preserving the eigenvalues (Fact 4.6) and, if it exists, the Hermitian structure of the matrix A (Fact 4.7). Thus, if A happens to be Hermitian, then T_0 is both Hessenberg and Hermitian—that is, T_0 is *tridiagonal*! Further, the Hessenberg decomposition can be completed with a finite sequence of calculations, as described below. Note that, by presenting this construction, we also effectively establish that the Hessenberg decomposition itself exists. As the Hessenberg decomposition reduces a general square matrix A to Hessenberg form while preserving its eigenvalues, and it reduces a Hermitian matrix A to a tridiagonal form, this decomposition makes the problem of determining eigenvalues significantly easier.

The algorithm to construct a Hessenberg decomposition of an $n \times n$ matrix A is comprised of $n-2$ steps, each of which is based on an appropriately configured Householder reflector matrix (as described in §1.2.9), embedded within an appropriately-sized identity matrix, to introduce zeros into the transformed matrix. Specifically, at the first step, define

$$\mathbf{x}^1 = \begin{pmatrix} a_{21} \\ a_{31} \\ \vdots \\ a_{n1} \end{pmatrix}, \quad \{\sigma_1, \mathbf{w}^1, \nu_1\} \text{ according to (1.9b) and } H(\sigma_1, \mathbf{w}^1) \text{ according to (1.7)},$$

$$V_1 = \begin{bmatrix} 1 & 0 \\ 0 & H(\sigma_1, \mathbf{w}^1)_{(n-1)\times(n-1)} \end{bmatrix} \quad \Rightarrow \quad V_1^H A V_1 = \begin{bmatrix} a_{11} & * \\ H^H(\sigma_1, \mathbf{w}^1)\mathbf{x}^1 & * \end{bmatrix} = \begin{pmatrix} a_{11} & * & * & * & \dots \\ -\nu_1 & * & * & * & \dots \\ 0 & * & * & * & \dots \\ 0 & * & * & * & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

Thus, via a unitary similarity transformation, we have reduced the first column of A to upper Hessenberg form. Referring to the elements of the transformed matrix as a_{ij} (that is, performing this transformation *in place* in the computer memory), we proceed by working next on the second column: define

$$\mathbf{x}^2 = \begin{pmatrix} a_{32} \\ a_{42} \\ \vdots \\ a_{n2} \end{pmatrix}, \quad \{\sigma_2, \mathbf{w}^2, \nu_2\} \text{ according to (1.9b) and } H(\sigma_2, \mathbf{w}^2) \text{ according to (1.7)},$$

$$V_2 = \begin{bmatrix} I_{2\times 2} & 0 \\ 0 & H(\sigma_2, \mathbf{w}^2)_{(n-2)\times(n-2)} \end{bmatrix} \quad \Rightarrow \quad V_2^H (V_1^H A V_1) V_2 = \begin{pmatrix} a_{11} & a_{12} & * & * & \dots \\ -\nu_1 & a_{22} & * & * & \dots \\ 0 & -\nu_2 & * & * & \dots \\ 0 & 0 & * & * & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

We continue this way for $n-2$ steps, after which the transformed matrix, which we denote by T_0, is in Hessenberg form; efficient implementation is given in Algorithm 4.2. The sequence of operations that are performed in the process may be summarized in matrix notation as

$$T_0 = V^H A V \quad \Leftrightarrow \quad A = V T_0 V^H, \quad \text{where} \quad V = V_1 V_2 \dots V_{n-2}.$$

Algorithm 4.2: Computation of the Hessenberg decomposition.

View
Test

```
function [A,V] = Hessenberg(A)
% Pre and post-multiply an nxn matrix A by a sequence of Householder reflections to reduce
% it to upper Hessenberg form T_0, thus computing the unitary similarity transformation
% A=V T_0 V^H.
[m,n]=size(A);   if nargout>1, V=eye(n,n); end
for i=1:n-2
   [sig,w] = ReflectCompute(A(i+1:n,i));   A=Reflect(A,sig,w,i+1,n,1,n,'B');
   if nargout>1, V=Reflect(V,sig,w,i+1,n,1,n,'R'); end
end
end % function Hessenberg
```

In the algorithm above, we never actually even need to compute the matrices V_i. In order to compute both T_0 and, if required, V, it is sufficient to compute the σ_k and $\mathbf{w}^k$, noting (1.11).

Significantly, *this is the best we can do in terms of introducing zeros into A exactly*[5] *via a similarity transformation using a finite sequence of simple reflections and rotations*. The difficulty that prevents us from taking this approach any further is that any transformation matrix, such as the Householder, Givens, and fast Givens matrices introduced in §1, must be applied from both the left *and* the right when building a similarity transformation. Thus, if two or more rows are combined via premultiplication by some convenient transformation matrix, then the corresponding columns must immediately also be combined in the subsequent postmultiplication by the conjugate transpose of that transformation matrix. Therefore, if we attempt, e.g., to apply a series of premultiplications by Givens rotation matricies in order to zero out the first subdiagonal of a Hessenberg matrix T_0 in order to reduce it to a triangular form via a similarity transformation, the required postmultiplications by the conjugate transpose of these rotation matrices immediately destroys certain subdiagonal zeros which we have worked so hard to create. This is true no matter how much rotation we apply at each step or what sequence these steps are applied (see Exercise 4.8). Thus,

Fact 4.8 *In general, the Schur and eigen decompositions must be computed iteratively.*

This statement, which makes concrete the fundamental difficulty of the problem of determining eigenvalues, is in fact quite deep, and follows as a direct consequence of the following classical result:

Fact 4.9 (The Abel-Ruffini Theorem) *A polynomial equation of order higher than four is incapable of general algebraic solution by radicals (that is, in terms of a finite number of additions, subtractions, multiplications, divisions, and root extractions).*

Note, of course, that there are some polynomial equations of order higher than four that in fact are solvable by radicals ($x^5-x^4-x+1=0$ is an example, with roots $\{1,1,-1,\mathrm{i},-\mathrm{i}\}$). However, there are other polynomial equations of order higher than four that are *not* solvable by radicals ($x^5-x+1=0$ is an example). For a description of Abel's proof of Fact 4.9, the reader is referred to Pesic (2004). The criterion that ultimately distinguishes between those polynomial equations that can be solved by radicals and those that cannot is made precise by **Galois theory** (see, e.g., Postnikov 2004).

The connection between the Abel-Ruffini Theorem and Fact 4.8 may be seen immediately by, for example, considering the eigenvalues of the following matrix in top companion form (see §1.2.8):

$$A=\begin{pmatrix} -a_{n-1} & \cdots & -a_1 & -a_0 \\ 1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & 1 & 0 \end{pmatrix}, \quad |\lambda I - A| = 0 \quad \Leftrightarrow \quad p(\lambda)=\lambda^n + a_{n-1}\lambda^{n-1} + \ldots + a_1\lambda + a_0 = 0.$$

[5] That is, assuming infinite-precision arithmetic.

Algorithm 4.3: Code to convert the polynomial root finding problem into an equivalent eigenvalue problem.

```
function x = Roots(a)
% Compute the roots of a polynomial a(1)*x^n+a(2)*x^(n-1)+...+a(n+1)=0 with |a(1)|>0.
n=size(a,2); A=[-a(2:n)/a(1); eye(n-2,n-1)]; x=Eig(A);
end % function Roots
```

View
Test

It follows that any polynomial equation $p(\lambda) = 0$ is the characteristic equation of a matrix A in companion form, thus allowing one to convert the problem of finding the roots of a polynomial into the problem of computing the eigenvalues of A, as illustrated in Algorithm 4.3. This relation also establishes that, if there are some polynomial equations that are not solvable via a finite number of additions, subtractions, multiplications, divisions, and root extractions, then there are some eigenvalue problems that are not solvable via a finite sequence of such operations as well.

In the following section, we thus step back from eigenvalue-preserving similarity transformations for a moment, and investigate how to triangularize a matrix when (only) premultiplying by a finite sequence of unitary transformation matrices, which we assemble into a unitary matrix Q, thereby constructing the decomposition $A = QR$. It is then shown how the resulting QR decomposition algorithms may be used as the core of efficient *iterative* algorithms for determining the Schur and eigen decompositions.

4.4.2 The QR decomposition

Every $m \times n$ matrix A has a QR **decomposition** $A = QR$, where $Q = Q_{m\times m}$ is unitary and $R = R_{m\times n}$ is upper triangular. The QR decomposition is a tool of fundamental importance in linear algebra. We thus present *five* different algorithms to construct various forms of this decomposition in the subsections that follow. Note that, by presenting these constructions, we also establish that the QR decomposition itself exists.

Further, if $r = \text{rank}(A) = n$ (which is often the case when using the QR decomposition), or $r < n$ but column pivoting (via a permutation matrix Π) is applied to ensure the QR decomposition is ordered correctly (such that, e.g., $|r_{ii}|$ decreases with i), then the QR decomposition can be partitioned such that:

$$A_{m\times n}\Pi_{n\times n} = Q_{m\times m}R_{m\times n} = \begin{bmatrix} \underline{Q}_{m\times r} & \overline{Q}_{m\times(m-r)} \end{bmatrix} \begin{bmatrix} \underline{R}_{r\times n} \\ 0 \end{bmatrix} = \underline{Q}_{m\times r}\underline{R}_{r\times n}, \tag{4.18}$$

where $\underline{R}$ is upper triangular (and possibly fat), $\underline{Q}$ is an orthogonal basis for the column space of A, $\overline{Q}$ is an orthogonal basis for the left nullspace of A (see §4.1), and Π is a permutation matrix. We thus refer to

- a **complete form**, $A_{m\times n} = Q_{m\times m}R_{m\times n}$ (taking $\Pi = I$), as a QR **decomposition** of A,
- a **pivoted complete form**, $A_{m\times n}\Pi_{n\times n} = Q_{m\times m}R_{m\times n}$ with a permutation matrix Π selected so that $|r_{ii}|$ decreases with i [thus revealing the block partitioning of (4.18)], as a **pivoted** QR **decomposition** of A,
- a **reduced form**, $A_{m\times n} = \underline{Q}_{m\times r}\underline{R}_{r\times n}$ (with[6] $\Pi = I$), as a $\underline{QR}$ **decomposition** of A, and
- a **pivoted reduced form**, $A_{m\times n}\Pi_{n\times n} = \underline{Q}_{m\times r}\underline{R}_{r\times n}$, as a **pivoted** $\underline{QR}$ **decomposition** of A.

Beware that other texts are not as strict notationally, so you sometimes need to look carefully to see whether the pivoted or nonpivoted and the complete or reduced form of this decomposition is being used. Note also that a QL decomposition may also be developed via procedures similar to those outlined below, where Q is unitary and L is *lower* triangular.

[6] In the important special case in which $r = \text{rank}(A) = n$, all n elements on the main diagonal of R are nonzero; in this special case, we may always determine a $\underline{QR}$ decomposition of A without pivoting

Algorithm 4.4: Compute a $\underline{QR}$ decomposition of a matrix A of full column rank via Classical Gram-Schmidt.

View Test

```
function [A,R] = QRcgs(A,s)
% Compute a reduced QR decomposition A=Q*R of an mxn matrix A via Classical Gram-Schmidt.
% Pivoting is NOT implemented; redundant columns of A are simply set to zero.

[m,n]=size(A); R=eye(n,n); if nargin==1, s=0; end
for i=s+1:n
  R(1:i-1,i)=A(:,1:i-1)'*A(:,i);          A(:,i)=A(:,i)-A(:,1:i-1)*R(1:i-1,i);
  R(i,i)=norm(A(:,i)); if R(i,i)>1e-9, A(:,i)=A(:,i)/R(i,i); else, A(:,i)=zeros(m,1); end
end                                       % Note: Q is returned in the modified A.
end % function QRcgs
```

The first two algorithms presented below (Classical Gram-Schmidt and Modified Gram-Schmidt) only determine (reduced) $\underline{QR}$ decompositions, whereas the last three algorithms presented (those based on Householder reflections, Givens rotations, and fast Givens transforms) determine (complete) QR decompositions. When column pivoting is applied [if necessary; that is, if $\text{rank}(A) < n$], the latter three algorithms thus reveal both $\underline{Q}$ and $\overline{Q}$. As mentioned above, $\underline{Q}$ is an orthogonal basis of the column space of A, and $\overline{Q}$ is an orthogonal basis of the left nullspace of A.

Determining the $\underline{QR}$ decomposition via Classical Gram-Schmidt

Perhaps the simplest method to determine an orthonormal basis of the space spanned by a set of n linearly independent vectors $\mathbf{a}^i$ (for $i = 1,\ldots n$) of order m [that is, in this subsection (only), we assume the special case that $r = \text{rank}(A) = n$, which implies that $m \geq n$] is known as **Classical Gram-Schmidt** orthogonalization, or simply as **Gram-Schmidt**. This method (see Algorithm 4.4) is initialized by taking the first vector of this orthonormal basis, $\mathbf{q}^1$, as the first vector of the original set, $\mathbf{a}^1$, scaled to be of unit norm, i.e.,

$$\mathbf{q}^1 = \mathbf{a}^1/r_{11} \quad \text{where} \quad r_{11} = \|\mathbf{a}^1\|.$$

Thereafter (for $i = 2,3,\ldots,n$), $\mathbf{q}^i$ is taken as the original vector $\mathbf{a}^i$ minus its projections in the directions of the previously-computed orthonormal basis vectors ($\mathbf{q}^k$ for $k = 1,\ldots,i-1$) and scaled to be of unit norm,

$$\mathbf{q}^i = \mathbf{z}^i/r_{ii} \quad \text{where} \quad \mathbf{z}^i = \mathbf{a}^i - \sum_{k=1}^{i-1} r_{ki}\mathbf{q}^k \quad \text{with} \quad r_{ki} = \begin{cases} (\mathbf{q}^k)^H\mathbf{a}^i & k < i \\ \|\mathbf{z}^i\| & k = i \\ 0 & k > i. \end{cases} \quad \text{Combining,} \quad \mathbf{a}^i = \sum_{k=1}^{n} r_{ki}\mathbf{q}^k.$$

Multiplying the equation above for $\mathbf{q}^i$ from the left by $(\mathbf{q}^j)^H$ for values of j ranging from 1 to i, it is easily verified that $(\mathbf{q}^j,\mathbf{q}^i) = \delta_{ji}$. The combined expression at right is easily recognized as simply

$$\begin{pmatrix} | & | & & | \\ \mathbf{a}^1 & \mathbf{a}^2 & \ldots & \mathbf{a}^n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} | & | & & | \\ \mathbf{q}^1 & \mathbf{q}^2 & \ldots & \mathbf{q}^n \\ | & | & & | \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \ldots & r_{1n} \\ & r_{22} & \ldots & r_{2n} \\ & & \ddots & \vdots \\ 0 & & & r_{nn} \end{pmatrix},$$

that is, as $A = \underline{QR}$, where A is the original matrix (with the n linearly independent vectors $\mathbf{a}^k$ as columns), $\underline{Q}$ is an $m \times n$ unitary matrix (with the n orthonormal vectors $\mathbf{q}^i$ as columns), and $\underline{R}$ is an $n \times n$ upper triangular matrix. Note that the Classical Gram Schmidt procedure builds up the matrix $\underline{R}$ one *column* at a time; that is, for each i, the r_{ki} are selected so that $\mathbf{q}^i$ is orthogonal to the previously-computed $\mathbf{q}^k$ (for $k = 1,\ldots,i-1$).

Algorithm 4.5: Compute a pivoted $\underline{QR}$ decomposition of any matrix A via Modified Gram-Schmidt.

View
Test

```
function [A,R,pi,r] = QRmgs(A,s)
% Compute an ordered complete QR decomposition A*Pi=Q*R, and rank, of ANY mxn matrix A via
% Modified Gram-Schmidt (Q is returned in the modified A).  Pivoting is implemented, but

[m,n]=size(A);  R=zeros(n,n);  pi=[1:n]'; tol=1e-8; if nargin==1, s=0; end
for i=1:n
  clear L; for j=i:n, L(j)=norm(A(:,j)); end; [LL,k]=max(L);          % Pivoting
  if LL>L(i) & i>s, R(:,[i k])=R(:,[k i]); A(:,[i k])=A(:,[k i]); pi([i k])=pi([k i]); end
  R(i,i)=LL;                          A(:,i)=A(:,i)/R(i,i);           % Modified Gram-Schmidt
  R(i,i+1:n)=A(:,i)'*A(:,i+1:n); A(:,i+1:n)=A(:,i+1:n)-(A(:,i))*(R(i,i+1:n));
end
r=n; for i=1:n, if abs(R(i,i))<tol, r=i-1; break, end, end, A=A(:,1:r); R=R(1:r,:);
if r<m, A(:,r+1:m)=randn(m,m-r); R(r+1:m,:)=zeros(m-r,n); A=QRcgs(A,r); end
end % function QRmgs
```

Determining the pivoted $\underline{QR}$ decomposition via Modified Gram-Schmidt

The **Modified Gram-Schmidt** algorithm is simply a reordering of the Classical Gram Schmidt algorithm that is better behaved numerically in terms of the orthogonality of the resulting columns of $\underline{Q}$. In constrast with the Classical Gram-Schmidt procedure, the Modified Gram-Schmidt procedure builds up the matrix $\underline{R}$ one *row* at a time; that is, for each i, the r_{ik} are selected so that $\mathbf{q}^i$ is orthogonal to the columns of the matrix from which the yet-to-be-determined $\mathbf{q}^k$ (for $k = i+1,\ldots,n$) will be determined. This is done by subtracting off from the remaining columns of A the appropriate projections on $\mathbf{q}^i$ as soon as $\mathbf{q}^i$ is determined. By so doing, future projections may be calculated more accurately, as they are based on shorter vectors (the modified columns of A) that already have several other projections subtracted off, thereby reducing round-off error. Note that, in this subsection and the three that follow, we relax the assumption made previously that $r = \text{rank}(A) = n$.

The resulting procedure, at step i, may be described as follows: assume we are constructing $A = \underline{QR}$ (where $\underline{Q}$ is unitary and $\underline{R}$ is upper triangular) and we have already determined the first $i-1$ columns of $\underline{Q}$ and the first $i-1$ rows of $\underline{R}$ (and therefore, since $\underline{R}$ is upper triangular, the first $i-1$ columns of $\underline{R}$). Writing $A - \underline{QR} = 0$ in index notation as $a_{ij} - \sum_{i'=1}^{n} q_{ii'} r_{i'j} = 0$ and moving the component of this sum corresponding to all yet-to-be-determined columns of $\underline{Q}$ and rows of $\underline{R}$ at step i to the RHS, we define A_i as the modified A matrix comprised of the following elements:

$$[A_i]_{st} \triangleq a_{st} - \sum_{i'=1}^{i-1} q_{si'} r_{i't} = \sum_{i'=i}^{n} q_{si'} r_{i't}.$$

Noting that the first $i-1$ columns of the matrix A_i are zero, we name the i'th column of the A_i matrix $\mathbf{z}^i$. As in the Classical Gram-Schmidt procedure, we now scale $\mathbf{z}^i$ appropriately to determine $\mathbf{q}^i$, taking

$$\mathbf{q}^i = \begin{cases} \mathbf{z}^i / r_{ii} & \text{if } r_{ii} > 0, \\ 0 & \text{otherwise,} \end{cases} \quad \text{where} \quad r_{ii} = \|\mathbf{z}^i\|.$$

We then subtract off from A_i the projections of the remaining columns of A_i on $\mathbf{q}^i$, taking

$$A_{i+1} = A_i - \mathbf{q}^i \begin{bmatrix} 0_{1\times(i-1)} & r_{i,i+1} & r_{i,i+2} & \ldots & r_{in} \end{bmatrix}, \quad \text{where} \quad r_{ik} = (\mathbf{q}^i)^H \mathbf{a}^k \ \text{ for } \ k > i,$$

where $\mathbf{a}^k$ refers to the k'th column of A_i. Note that the first i columns of the matrix A_{i+1} are zero. We then increment i and go to the next step. Though no more expensive than Classical Gram Schmidt, as exactly the same number of projections, vector subtractions, and normalizations are performed (that is, if pivoting

Algorithm 4.6: Compute a (full) QR decomposition of a general matrix A via Householder reflections.

View Test

```
function [A,Q,pi,r] = QRHouseholder(A)
% Compute a (full) QR decomposition A=Q*R of ANY mxn matrix A using a sequence of
% Householder reflections (R is returned in the modified A).  IF pi and r are
% requested, then pivoting is implemented and the decomposition is ordered, A*Pi=Q*R.
[m,n]=size(A); pi=[1:n]'; tol=1e-8; Q=eye(m,m);
for i=1:min(n,m-1)
  if nargout>2, for j=i:n, length(j)=norm(A(i:end,j)); end; [amax,imax]=max(length);
    if amax>length(i), A(:,[i imax])=A(:,[imax i]); pi([i imax])=pi([imax i]); end
  clear length, end
  [sig,w]=ReflectCompute(A(i:m,i)); A=Reflect(A,sig,w,i,m,i,n,'L');
  Q=Reflect(Q,sig,w,i,m,1,m,'R');
end
if nargout>2, r=min(m,n); for i=1:r, if abs(A(i,i))<tol, r=i-1; break, end, end, end
end % function QRHouseholder
```

isn't applied), the Modified Gram-Schmidt algorithm is usually a bit more accurate in terms of the resulting orthogonality of the columns of $\underline{Q}$.

The procedure of column pivoting developed in §2.2.3 (to determine the $A = PLUQ^T$ decomposition) is incorporated in Algorithm 4.5 in an almost identical fashion: before each step i, the columns of the modified A matrix are swapped to move the column with the largest norm into the i'th column, keeping track of this column swap in the permutation matrix Π (or, to reduce storage, in a permutation vector π). This ensures that $|r_{ii}|$ decreases with i, and thus that a partitioning of the form (4.18) exists even if $\text{rank}(A) < n$.

Determining the pivoted QR decomposition via Householder reflections

An approach for determining the (complete) QR decomposition which is better than modified Gram-Schmidt in terms of the orthogonality of the resulting columns of Q, but is about twice as expensive, is to use a sequence of Householder reflections in a manner very similar to that used to compute the Hessenberg decomposition in §4.4.1.

The determination of the QR decomposition of the matrix A via this approach is comprised of $p = \min(n, m-1)$ steps. The k'th step of this procedure may be described as follows: assume that the first $k-1$ columns of A have already been transformed to upper triangular form, that is,

$$(Q_{k-1}^H Q_{k-2}^H \cdots Q_1^H)A = \begin{bmatrix} \underline{R}_{(k-1)\times(k-1)} & *_{(k-1)\times(n-k+1)} \\ 0 & *_{(m-k+1)\times(n-k+1)} \end{bmatrix},$$

where the Q_i are unitary and $\underline{R}_{(k-1)\times(k-1)}$ is upper triangular (as its notation implies, $\underline{R}_{(k-1)\times(k-1)}$ turns out to be the upper-left corner of $\underline{R}_{n\times n}$). Referring to the elements of the transformed matrix as a_{ij} (performing the transformation in place in the computer memory), the k'th step focuses on reducing the k'th column to upper triangular form. Define

$$\mathbf{x}^k = \begin{pmatrix} a_{k,k} \\ a_{k+1,k} \\ \vdots \\ a_{m,k} \end{pmatrix}, \quad \{\sigma_k, \mathbf{w}^k, \nu_k\} \text{ according to (1.9b) and } H(\sigma_k, \mathbf{w}^k) \text{ according to (1.7)},$$

$$Q_k = \begin{bmatrix} I_{(k-1)\times(k-1)} & 0 \\ 0 & H(\sigma_k, \mathbf{w}^k)_{(m-k+1)\times(m-k+1)} \end{bmatrix} \quad \Rightarrow \quad (Q_k^H Q_{k-1}^H \cdots Q_1^H)A = \begin{pmatrix} \underline{R}_{k\times k} & *_{k\times(n-k)} \\ 0 & *_{(m-k)\times(n-k)} \end{pmatrix}.$$

After $p = \min(n, m-1)$ steps, defining $Q = Q_1 Q_2 \cdots Q_p$ and thus $Q^{-1} = Q_p^H \cdots Q_2^H Q_1^H$, we have $Q^{-1}A = R$ and thus $A = QR$, where $Q = Q_{m\times m}$ is unitary and $R = R_{m\times n}$ is upper triangular. As in §4.4.1, the individual

Q_i need never be computed. In order to compute both Q and R, it is sufficient to compute the σ_k and $\mathbf{w}^k$, noting (1.11), as implemented in Algorithm 4.6. Note that column pivoting is applied to this algorithm in the same manner as implemented in Algorithm 4.5. By so doing, one may easily distinguish the columns of $\underline{Q}$ from the columns of $\overline{Q}$ in the pivoted QR decomposition [of the form given in (4.18)] even when $\text{rank}(A) < n$.

Determining the QR decomposition via Givens rotations

The QR decomposition may also be built up from several, appropriately configured Givens rotation matrices G, as defined in §1.2.10. The determination of the QR decomposition of the matrix A via a series of such Givens rotations simply steps through every subdiagonal element of A, multiplying (if necessary) the entire matrix A by the appropriate Givens rotation matrix such that the transformed matrix is zero in that element, with the ordering of the loops arranged in such a manner that the subdiagonal elements of A that have already been transformed to zero by this procedure stay that way.

As an example, assume that we start with an $n \times n$ matrix A in upper Hessenberg form:

$$A = \begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{pmatrix},$$

where the x's denote the nonzero elements. We define the first rotation G_1 to put a zero in $a_{2,1}$:

$$G_1^H A = \begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & x & x & x & x \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{pmatrix},$$

where the $*$'s denote the nonzero elements that have been modified. We continue in this manner, working on each nonzero subdiagonal element in turn, until the transformed matrix reaches upper triangular form, at which point we have $Q^H A = R$, and thus $A = QR$, where $Q = G_1 G_2 \cdots G_{n-1}$. This procedure is implemented in Algorithm 4.7 for Hessenberg matrices and Algorithm 4.8 for tridiagonal matrices.

As in the Householder approach described previously, the individual G_i need never be calculated; initializing $Q = I$ and noting the simple algorithmn to compute $G^H X$ in (1.13), both Q and R can be determined efficiently by working in place in the computer memory on Q and A, applying the Givens rotations directly from the $\{i,k;c,s\}$ parameters defining each one. At the end of the procedure, A is replaced by R. Further, it is important to note that, in most algorithms that use the QR decomposition, we do not even need the matrix $Q = G_1 G_2 \cdots G_p$ explicitly; we only need to be able to apply Q to vectors or matrices (that is, collections of vectors). Thus, for efficiency, we may store the information defining each rotation matrix that combine to make up Q instead of storing Q itself. This may be done by storing the c and s associated with each rotation or, more compactly, by storing γ [from which the c and s associated with each rotation may be recomputed, as seen in (1.15)] into the element of the transformed matrix that has just been set to zero. The resulting matrix A then contains R in its upper triangular part and the various values of γ in each of its transformed subdiagonal elements. Noting that $Q = G_1 G_2 \cdots G_{n-1}$, the product XQ for some matrix X may later be computed leveraging (1.15) and (1.13b). Thus, as with the LU decomposition presented in §2.2.1, this algorithm also demonstrates the conservation of information property evident in many efficient numerical algorithms; that is, the information necessary to describe the QR decomposition of A following the minimum storage version of this algorithm (Exercise 4.7) takes precisely the same number of elements as it takes to describe A itself.

The primary advantage of the Givens rotations approach to computing the QR decomposition is that it can take advantage of any subdiagonal zeros that A already possesses by reducing the number of Givens rotations

Algorithm 4.7: Compute a *QR* decomposition of an upper Hessenberg matrix *A* via Givens rotations.

View Test

```
function [A,Q] = QRGivensHessenberg(A)
% Compute a QR decomposition A=QR by applying a sequence of min(n,m-1) Givens rotations
% to an mxn upper Hessenberg matrix A to reduce it to upper triangular form.
[m,n]=size(A);   Q=eye(m,m);        % Note: R is returned in the modified A.
for i=1:min(n,m-1)
  [c,s]=RotateCompute(A(i,i),A(i+1,i)); [A]=Rotate(A,c,s,i,i+1,i,n,'L');
  [Q]=Rotate(Q,c,s,i,i+1,1,m,'R');
end
end % function QRGivensHessenberg
```

Algorithm 4.8: Compute a *QR* decomposition of a tridiagonal matrix *A* via Givens rotations.

View Test

```
function [b,c,a,cc,ss] = QRGivensTridiag(a,b,c)
% Compute a QR decomposition of a square tridiagonal matrix A=tridiag(a,b,c) by
% applying a sequence of n-1 Givens rotations directly to the tridiagonal elements.
% The result returned is the three nonzero diagonals [b,c,a] of the upper tridiagonal
% matrix R, and the n-1 values of c and s for each of the n-1 rotations performed, which
% is more efficient in this case than returning Q itself (which is upper Hessenberg).
n=size(b); for i=1:n-1, [cc(i),ss(i)]=RotateCompute(b(i),a(i+1)); if cc(i)~=1
  b(i)   =conj(cc(i))*b(i)   -conj(ss(i))*a(i+1);        % Eqn (1.12a), row i
  temp   =conj(cc(i))*c(i)   -conj(ss(i))*b(i+1);
  a(i)   =                   -conj(ss(i))*c(i+1);
  b(i+1)=      ss(i) *c(i)   +cc(i)*b(i+1);              % Eqn (1.12a), row k=i+1
  c(i+1)=      cc(i) *c(i+1);                 c(i)=temp;
end, end
end % function QRGivensTridiag
```

required to transform *A* to an upper triangular form. In particular, if *A* is either Hessenberg (as in the above example, and implemented in Algorithm 4.7) or tridiagonal (as implemented in Algorithm 4.8), only $n-1$ Givens rotations need to be performed to compute the *QR* decomposition of *A* using this approach.

Determining the *QR* decomposition via fast Givens transforms[†]

The previous section demonstrated how to build a *QR* decomposition of *A* via a series of p (unitary) Givens rotations $(G_p^H \cdots G_2^H G_1^H)A = Q^H A = R$. The approach is particularly attractive because, unlike a Householder-based approach, it is significantly accelerated if there are already some zeros in the strictly lower-triangular part of *A*. Leveraging the following two facts, we now illustrate how to do the same thing via a series of p (nonunitary) fast Givens transforms, which, as discussed in §1.2.11, require 33% fewer flops to apply than Givens rotations due to their simpler structure.

Fact 4.10 *If $\overline{Q}^H A = \overline{R}$ is upper triangular and $\overline{Q}^H \overline{Q} = D$ is diagonal and nonsingular, then $Q = \overline{Q} D^{-1/2}$ and $R = D^{-1/2}\overline{R}$ form a QR decomposition of A (that is, Q is unitary, R is upper triangular, and $A = QR$).*

Proof: If $\overline{R}$ is upper triangular, $R = D^{-1/2}\overline{R}$ is upper triangular by inspection. To see that $Q = \overline{Q}D^{-1/2}$ is unitary, note that

$$Q^H Q = (\overline{Q}D^{-1/2})^H (\overline{Q}D^{-1/2}) = D^{-1/2} D D^{-1/2} = I.$$

Finally, to see that $A = QR$, note that

$$Q^H A = D^{-1/2}\overline{Q}^H A = D^{-1/2}\overline{R} = R. \qquad \square$$

Fact 4.11 *Initializing $\overline{Q}_0 = I$ and $D_0 = I$ and defining $\overline{Q}_j$ and D_j via a series of fast Givens transforms $\overline{Q}_j = F_1 F_2 \cdots F_j$ and $D_j = F_j^H D_{j-1} F_j$, it follows that $\overline{Q}_j^H \overline{Q}_j = D_j$ is diagonal.*

Algorithm 4.9: Compute a *QR* decomposition of an upper Hessenberg matrix A via fast Givens transforms.

View
Test

```
function [A,Q] = QRFastGivensHessenberg(A)
% Compute a QR decomposition A=QR by applying a sequence of min(n,m-1) fast Givens
% transforms to an mxn upper Hessenberg matrix A to reduce it to upper triangular form.
[m,n]=size(A);  Q=eye(m,m);  d=ones(m,1);
for i=1:min(n,m-1)
  [a,b,gamma,donothing,d([i i+1])]=FastGivensCompute(A(i,i),A(i+1,i),d(i),d(i+1));
  [A]=FastGivens(A,a,b,gamma,donothing,i,i+1,i,n,'L');
  [Q]=FastGivens(Q,a,b,gamma,donothing,i,i+1,1,m,'R');
end
for i=1:m,  dt=1/sqrt(d(i));  Q(:,i)=Q(:,i)*dt;  A(i,:)=A(i,:)*dt;  end
end % function QRFastGivensHessenberg
```

Proof (by induction): The base case $j = 0$ is clearly true. Assuming $\overline{Q}_j^H \overline{Q}_j = D_j$ is diagonal, it follows that

$$\overline{Q}_{j+1}^H \overline{Q}_{j+1} = (\overline{Q}_j F_{j+1})^H (\overline{Q}_j F_{j+1}) = F_{j+1}^H D_j F_{j+1} = D_{j+1}.$$

Noting the definition of F (see the last paragraph of §1.2.11), $F_{j+1}^H D_j F_{j+1} = D_{j+1}$ is diagonal. □

Fact 4.11 establishes a simple iterative procedure which we may use to construct a nonunitary matrix $\overline{Q}_k$ via a series of k fast Givens transforms, F_1 through F_k, such that $\overline{Q}_k^H A = \overline{R}_k$ is upper triangular, where $\overline{Q}_k = F_1 F_2 \cdots F_k$ and $\overline{Q}_k^H \overline{Q}_k = D_k$ is diagonal. Note (see §1.2.11) that the iterative procedure to determine D_k is computationally inexpensive. By Fact 4.10, it thus follows that $Q = \overline{Q}_k D_k^{-1/2}$ and $R_k = D_k^{-1/2} \overline{R}$ form a *QR* decomposition of A, as implemented for Hessenberg A in Algorithm 4.9.

4.4.3 Characterizing the Schur decomposition

Every $n \times n$ matrix A has a **Schur decomposition** $A = UTU^H$, where T is upper triangular and U is unitary with columns $\mathbf{u}^k$, sometimes referred to as **Schur vectors**. Proof of this important statement builds up from two preliminary facts that are easy to verify:

Fact 4.12 *If an $n \times n$ matrix T may be partitioned as $T = \begin{bmatrix} L_{p\times p} & C_{p\times q} \\ 0 & S_{q\times q} \end{bmatrix}$, then $\lambda(T) = \lambda(L) \cup \lambda(S)$. Further, the eigenvalues of a triangular matrix appear on its main diagonal, and the eigenvalues of a block triangular matrix, with square blocks on the main diagonal, are given by the union of the eigenvalues of the blocks on the main diagonal.*

Proof: If Λ is an eigenvalue of T, then by Property 3b of the determinant, $|\lambda I - T| = |\lambda I - L| \cdot |\lambda I - S| = 0$, where $|\lambda I - L|$ is a polynomial whose p roots are the eigenvalues of L, and $|\lambda I - S|$ is a polynomial whose q roots are the eigenvalues of S. Thus, the $p + q = n$ roots of T are given by the union of the roots of L and the roots of S. The statements about triangular and block triangular matrices follow via repeated application of this result. □

Fact 4.13 *If*

$$AX = XB, \tag{4.19}$$

where $A = A_{n\times n}$, $B = B_{p\times p}$, and $X = X_{n\times p}$ with $\text{rank}(X) = p$ *and $1 \le p < n$, then there exists a unitary Q such that*

$$Q^H A Q = \begin{bmatrix} L_{p\times p} & C_{p\times q} \\ 0 & S_{q\times q} \end{bmatrix}, \tag{4.20}$$

where $\lambda(L) = \lambda(B)$ and $q = n - p$.

Proof: By the constructions given in §4.4.2, we know a QR decomposition of X exists, which may be written

$$X_{n\times p} = Q_{n\times n}\begin{bmatrix}\underline{R}_{p\times p}\\ 0\end{bmatrix},$$

where Q is unitary and $\underline{R}$ is triangular; note also that, as $\text{rank}(X) = p$, it follows that $\text{rank}(\underline{R}) = p$ as well. Substituting this decomposition into (4.19) and multiplying by Q^H, we have

$$Q^H A Q\begin{bmatrix}\underline{R}\\ 0\end{bmatrix} = \begin{bmatrix}L_{p\times p} & C_{p\times q}\\ D_{q\times p} & S_{q\times q}\end{bmatrix}\begin{bmatrix}\underline{R}\\ 0\end{bmatrix} = \begin{bmatrix}\underline{R}\\ 0\end{bmatrix}B.$$

As $\underline{R}$ is nonsingular, the first block row of this equation may be written $\underline{R}^{-1}L\underline{R} = B$, and thus, by Fact 4.6, $\lambda(L) = \lambda(B)$. The second block row of this equation gives $D\underline{R} = 0$; as $\underline{R}$ is nonsingular, the unique solution of this equation is $D = 0$, thus establishing the existence of the decomposition (4.20). □

Fact 4.14 (The Schur Decomposition Theorem) *Every square matrix $A = A_{n\times n}$ has a Schur decomposition $A = UTU^H$, where U is unitary and T is upper triangular. Further, the matrix T in the Schur decomposition has the eigenvalues of A listed (in any desired order) on its main diagonal.*

Proof (by induction): Note first that the theorem holds (trivially) for the base case of order $n = 1$. Assume the theorem holds for order $n-1$. Now consider the case of order n: for any desired eigenvalue λ of $A = A_{n\times n}$, we can write $A\mathbf{s} = \lambda\mathbf{s}$ with $\mathbf{s} \neq 0$, as established in §4.3.1. By Fact 4.13 (with $B = \lambda$ and $X = \mathbf{s}$), there exists a unitary Q_n such that

$$Q_n^H A Q_n = \begin{bmatrix}\lambda & *\\ 0 & S\end{bmatrix},$$

where $S = S_{(n-1)\times(n-1)}$. By the induction hypothesis, a Schur decomposition exists for S, which we denote $S = U_{n-1}T_{n-1}U_{n-1}^H$, where U_{n-1} is unitary and T_{n-1} is upper triangular. Thus, we may write

$$\begin{bmatrix}1 & 0\\ 0 & U_{n-1}^H\end{bmatrix}Q_n^H A Q_n\begin{bmatrix}1 & 0\\ 0 & U_{n-1}\end{bmatrix} \triangleq U^H A U = \begin{bmatrix}\lambda & *\\ 0 & U_{n-1}^H S U_{n-1}\end{bmatrix} = \begin{bmatrix}\lambda & *\\ 0 & T_{n-1}\end{bmatrix} \triangleq T \quad\Rightarrow\quad A = UTU^H,$$

where U is unitary and T is upper triangular. By Property 3a of the determinant, as T is a triangular matrix, the equation $|\lambda I - T| = 0$ gives simply $(\lambda - t_{11})(\lambda - t_{22})\cdots(\lambda - t_{nn}) = 0$, where t_{11} to t_{nn} are the diagonal elements of T. As T and A share the same eigenvalues (Fact 4.6), it follows that the eigenvalues of A are listed on the main diagonal of T. Note further that any desired eigenvalue may be selected to be in the upper-left corner of T (and of T_{n-1}, etc.); thus, once the eigenvalues of A are known, the above relations may be used to construct a Schur decomposition with the eigenvalues appearing on the main diagonal of T in any desired ordering. □

As noted in Fact 4.8, the Schur decomposition cannot, in general, be computed exactly with a finite sequence of calculations, as could the closely-related Hessenberg decomposition discussed in §4.4.1. A family of efficient iterative algorithms to determine the Schur decomposition is presented in §4.4.5.

Note that the first column of the equation $AU = UT$ is $A\mathbf{u}^1 = \lambda_1\mathbf{u}^1$, and thus the first Schur vector is, in fact, an eigenvector of A corresponding to λ_1 (that is, $\mathbf{u}^1 = \mathbf{s}^1$).

Finally, note that, if A is Hermitian, then T is triangular and, by Fact 4.7, T is also Hermitian—that is, T is diagonal. In this case, we denote T by Λ and U by S, and the Schur decomposition reduces to the **eigen decomposition**, as described in §4.4.4, with a unitary eigenvector matrix S.

Consequences of the existence of the Schur decomposition

The existence of the Schur decomposition allows us to prove some important results, such as the following.

Fact 4.15 (The Cayley-Hamilton Theorem) *Any square matrix A satisfies its own characteristic equation.*

Proof: By the Fundamental Theorem of Algebra (Fact 4.4), the characteristic equation of A may be written

$$\lambda^n + a_{n-1}\lambda^{n-1} + \ldots + a_1\lambda + a_0\lambda^0 = (\lambda - \lambda_1)(\lambda - \lambda_2)\cdots(\lambda - \lambda_n) = 0.$$

Replacing λ in this equation with the matrix A itself, and inserting I where appropriate, consider the quantity

$$A^n + a_{n-1}A^{n-1} + \ldots + a_1A + a_0I = (A - \lambda_1 I)(A - \lambda_2 I)\cdots(A - \lambda_n I) = C. \tag{4.21}$$

By the Schur decomposition theorem, we know that $A = UTU^H$, that is, A is unitarily similar to an upper triangular matrix T with the eigenvalues of A on its main diagonal. Thus (4.21) may be written

$$\begin{aligned}(UTU^H - \lambda_1 UU^H)(UTU^H - \lambda_2 UU^H)\cdots(UTU^H - \lambda_n UU^H) &= C \\ U(T - \lambda_1 I)U^H U(T - \lambda_2 I)U^H \cdots U(T - \lambda_n I)U^H &= C \\ (T - \lambda_1 I)(T - \lambda_2 I)\cdots(T - \lambda_n I) &= U^H CU.\end{aligned}$$

The upper triangular matrix factor $(T - \lambda_i I)$ has a zero on its i'th diagonal element. Noting this fact and multiplying the upper triangular matrix factors together one by one, it is easily verified that the first column of $(T - \lambda_1 I)$ is zero, then that the first two columns of $(T - \lambda_1 I)(T - \lambda_2 I)$ are zero, etc. Thus, $U^H CU = 0 \Rightarrow C = 0$, and therefore, by (4.21), A satisfies its own characteristic equation. □

Fact 4.16 *The eigenvalues of A^{-1} are the reciprocal of the eigenvalues of A.*

Proof: By Fact 4.14, $A = UTU^H$ where T is triangular. By Fact 1.7, it follows that $A^{-1} = UT^{-1}U^H$ where, by Fact 2.1, the diagonal elements of T^{-1} are the reciprocal of the diagonal elements of T. Thus, by the Schur decomposition theorem, the eigenvalues of A^{-1} are the reciprocal of the eigenvalues of A. □

Fact 4.17 *The determinant of A is equal to the product of its eigenvalues: $|A| = \lambda_1\lambda_2\cdots\lambda_n$. In particular, $|A| = 0$ iff $\lambda_i = 0$ for at least one value of i.*

Proof: By Fact 4.14, $A = UTU^H$ with $U^H = U^{-1}$ and T upper triangular with the eigenvalues of A listed on its main diagonal. By Properties 3 and 5 of the determinant (see §4.2.1), it follows that $|A| = |UTU^H| = |U| \cdot |T| \cdot |U^H| = |T| = \lambda_1\lambda_2\cdots\lambda_n$. □

Fact 4.18 *The eigenvalues of a real matrix A come in complex conjugate pairs. That is, if $\lambda \in \lambda(A)$, then $\overline{\lambda} \in \lambda(A)$ with the same multiplicity.*

Proof: By Fact 4.14, $A = UTU^H$ with T upper triangular and the eigenvalues of A listed on its main diagonal. Since A is real, it follows that $A = \overline{A} = (\overline{U})\overline{T}(\overline{U})^H$. Since $\overline{T}$ is also triangular, the eigenvalues of A are listed on its main diagonal. □

The real Schur decomposition

Complex arithmetic is significantly more expensive than real arithmetic. If a real $n \times n$ matrix A is known to have real eigenvalues (e.g., as established in Fact 4.20 for symmetric matrices), then the matrices forming its Schur decomposition $A = UTU^H$ happen to be real, and may be computed using real arithmetic.

Unfortunately (from a computational perspective), the Schur decomposition of a general real matrix is complex. For such matrices, it is often desirable to compute a convenient eigenvalue-revealing form without resorting to complex arithmetic. This may be accomplished by computing the **real Schur decomposition** $A = U\hat{T}U^T$, where U and $\hat{T}$ are real and $\hat{T}$ is in block upper triangular form with 1×1 and 2×2 blocks (with complex conjugate eigenvalues) on the main diagonal, referred to as a **real Schur form**. Essentially, the real Schur decomposition is as close as you can get to a Schur decomposition of a general real matrix without resorting to complex arithmetic. By Facts 4.6 and Fact 4.12, the eigenvalues of A are the union of the eigenvalues of the blocks on the main diagonal of $\hat{T}$. By Fact 4.5, the eigenvalues of the 2×2 blocks are easy to compute. In particular, if the 2×2 blocks on the main diagonal are rotated into the 2×2 **standard form**

$$A = \begin{pmatrix} \alpha & \omega/k \\ -\omega k & \alpha \end{pmatrix}, \tag{4.22}$$

then $\lambda_\pm = \alpha \pm i\omega$. Most of the algorithms that follow which leverage the Schur decomposition may in fact be written to leverage the real Schur decomposition in the case that the matrices setting up the problem are real, thereby avoiding complex arithmetic for most of the computation.

4.4.4 Characterizing the eigen decomposition

If $A_{n\times n}$ has n linearly independent eigenvectors $\{\mathbf{s}^1, \mathbf{s}^2, \ldots \mathbf{s}^n\}$, then any vector $\mathbf{x}$ of order n may be uniquely decomposed in terms of contributions parallel to each eigenvector such that

$$\mathbf{x} = S\boldsymbol{\chi} = \chi_1\mathbf{s}^1 + \chi_2\mathbf{s}^2 + \ldots + \chi_n\mathbf{s}^n, \qquad \text{where} \quad S = \begin{pmatrix} | & | & & | \\ \mathbf{s}^1 & \mathbf{s}^2 & \ldots & \mathbf{s}^n \\ | & | & & | \end{pmatrix}, \quad \Lambda = \begin{pmatrix} \lambda_1 & & & 0 \\ & \lambda_2 & & \\ & & \ddots & \\ 0 & & & \lambda_n \end{pmatrix}.$$

The columns of S are, for convenience, often scaled to have unit norm. The several relations $A\mathbf{s}^\iota = \lambda_\iota \mathbf{s}^\iota$ for $\iota = 1, 2, \ldots, n$ may be assembled in matrix form as

$$\begin{pmatrix} | & | & & | \\ A\mathbf{s}^1 & A\mathbf{s}^2 & \ldots & A\mathbf{s}^n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} | & | & & | \\ \lambda_1\mathbf{s}^1 & \lambda_2\mathbf{s}^2 & \ldots & \lambda_n\mathbf{s}^n \\ | & | & & | \end{pmatrix} \quad \Leftrightarrow \quad AS = S\Lambda.$$

If the columns of S are linearly independent, then S is invertible, and we may write the result as

$$A = S\Lambda S^{-1} \quad \Leftrightarrow \quad \Lambda = S^{-1}AS,$$

where, as illustrated above, Λ is diagonal; this is known as the **eigen** (a.k.a. **spectral**) **decomposition** of A.

An eigen decomposition of a matrix A, though expensive to calculate if the matrix is large, is very revealing. As just one example, an eigen decomposition completely decouples a linear system of ODEs into its independent modes, as will be shown in §**??**. As with the Schur decomposition described in §4.4.3, the eigen decomposition cannot, in general, be computed exactly with a finite sequence of calculations; some efficient iterative algorithms to compute the eigen decomposition are thus presented in §4.4.5.

Note that an eigen decomposition of $A = A_{n\times n}$ exists if and only if A has n linearly independent eigenvectors, in which case A is said to be **nondefective**. This is often, but not always, the case. For instance, A is guaranteed to have n linearly independent eigenvectors if A has n distinct eigenvalues, if A is Hermitian, or if A is skew Hermitian, as established below. On the other hand, §4.4.7 presents an example [(4.38) with $\varepsilon = 0$] that does *not* have n linearly independent eigenvectors; such a matrix is said to be **defective**.

Fact 4.19 *The set of eigenvectors* $\{\mathbf{s}^1,\ldots,\mathbf{s}^k\}$ *corresponding to distinct eigenvalues* $\{\lambda_1,\ldots\lambda_k\}$ *is linearly independent.*

Proof (by induction): We first prove the statement is true for the set $\{\mathbf{s}^1,\mathbf{s}^2\}$: Take $\mathbf{y}=c_1\mathbf{s}^1+c_2\mathbf{s}^2=0$. Multiplying this equation by A and subtracting the result from λ_2 times this equation gives

$$\lambda_2[c_1\mathbf{s}^1+c_2\mathbf{s}^2]-A[c_1\mathbf{s}^1+c_2\mathbf{s}^2]=0 \quad\Rightarrow\quad c_1(\lambda_2-\lambda_1)\mathbf{s}^1=0 \quad\Rightarrow\quad c_1=0 \quad\Rightarrow\quad c_2=0.$$

Thus, the set $\{\mathbf{s}^1,\mathbf{s}^2\}$ is linearly independent. Now, assuming the statement is true for the set $\{\mathbf{s}^1,\ldots,\mathbf{s}^{k-1}\}$, we prove it must also be true for $\{\mathbf{s}^1,\ldots,\mathbf{s}^k\}$: Take $\mathbf{y}=c_1\mathbf{s}^1+\ldots+c_k\mathbf{s}^k=0$. Multiplying this equation by A and subtracting the result from λ_k times this equation gives

$$\begin{aligned}\lambda_k[c_1\mathbf{s}^1+\ldots+c_k\mathbf{s}^k]-A[c_1\mathbf{s}^1+\ldots+c_k\mathbf{s}^k]=0 \quad&\Rightarrow\quad c_1(\lambda_k-\lambda_1)\mathbf{s}^1+\ldots+c_{k-1}(\lambda_k-\lambda_{k-1})\mathbf{s}^{k-1}=0\\ &\Rightarrow\quad c_1=c_2=\ldots=c_{k-1}=0 \quad\Rightarrow\quad c_k=0.\end{aligned}$$

Thus, the set $\{\mathbf{s}^1,\ldots,\mathbf{s}^k\}$ is linearly independent. □

Fact 4.20 *If the matrix A is Hermitian (in the real case, symmetric), then its eigenvalues are all real and its eigenvectors may be chosen to be orthonormal. Thus, A may be written as a linear combination of rank one factors such that*

$$A=S\Lambda S^H=\lambda_1\mathbf{s}^1(\mathbf{s}^1)^H+\lambda_2\mathbf{s}^2(\mathbf{s}^2)^H+\ldots+\lambda_N\mathbf{s}^N(\mathbf{s}^N)^H.$$

Proof: By the Schur decomposition theorem, there exists a unitary matrix U and an upper triangular matrix T such that $U^HAU=T$. If $A=A^H$, then $(U^HAU)^H=U^HAU$, and thus $T^H=T$, and therefore T must be diagonal and its elements must be real. Identifying $T=\Lambda$ and $U=S$, we may write $S^HAS=\Lambda$, where the diagonal elements of Λ are the (real) eigenvalues and the columns of S are the (orthonormal) eigenvectors. □

Fact 4.21 *If the matrix A is skew-Hermitian (in the real case, skew-symmetric), then its eigenvalues are all imaginary and its eigenvectors may be chosen to be orthonormal.*

Proof: The proof follows as in the proof of Fact 4.20, where we now find that $T=-T^H$, and thus the eigenvalues must be imaginary. □

Fact 4.22 *For any matrix A, the matrix* $B=A^HA$ *is Hermitian with real, non-negative eigenvalues and eigenvectors which may be chosen to be orthonormal.*

Proof: It follows by its definition that $B=B^H$ and thus, by Fact 4.20, that its eigenvalues are real and its eigenvectors may be chosen to be orthonormal. Suppose $\mathbf{x}$ is an eigenvector of (A^HA), and thus $\|\mathbf{x}\|^2>0$. Then $\mathbf{x}^H[(A^HA)\mathbf{x}=\lambda\mathbf{x}] \;\Rightarrow\; \lambda(\mathbf{x}^H\mathbf{x})=(\mathbf{x}^HA^H)(A\mathbf{x})=\|A\mathbf{x}\|^2\geq 0$. Since $\mathbf{x}^H\mathbf{x}>0$, it follows that $\lambda\geq 0$. □

Fact 4.23 *The eigenvalues of a matrix A vary continuously as the elements of A are varied.*

Proof: Follows as an immediate consequence of Fact B.5 applied to the characteristic polynomial of A, the coefficients of which are linear combinations of various products of the elements of A. □

Fact 4.24 (The Gershgorin Circle Theorem) *Define the* **Gershgorin discs** *of* $A_{n\times n}$*, denoted here* $G(a_{\kappa\kappa},R_\kappa)$ *for* $\kappa=1,\ldots,n$*, as the n closed disks in the complex plane centered at* $a_{\kappa\kappa}$ *with radius* $R_\kappa=\sum_{i\neq\kappa}|a_{\kappa i}|$*. Then:* **(a)** *Every eigenvalue of A lies within at least one Gershgorin disc.* **(b)** *Denote by E the union of k Gershgorin discs, and by F the union of the other* $n-k$ *Gershgorin discs; if E is disjoint from F, then E contains exactly k of the eigenvalues of A, and F contains* $n-k$ *of the eigenvalues of A.*

Proof: (a) Take λ as an eigenvalue of A and $\mathbf{s}$ as a corresponding eigenvector (that is, $A\mathbf{s} = \lambda\mathbf{s}$ with $\mathbf{s} \neq 0$). Define $\kappa = \arg\max_k |s_k|$ (that is, select κ as the index of the element of $\mathbf{s}$ that is largest in magnitude). Then:

$$\sum_j a_{\kappa j} s_j = \lambda s_\kappa \;\;\Rightarrow\;\; \sum_{j\neq\kappa} a_{\kappa j} s_j = \lambda s_\kappa - a_{\kappa\kappa} s_\kappa \;\;\Rightarrow\;\; |\lambda - a_{\kappa\kappa}| = \frac{|\sum_{j\neq\kappa} a_{\kappa j} s_j|}{|s_\kappa|} \leq \sum_{j\neq\kappa} |a_{\kappa j}| \frac{|s_j|}{|s_\kappa|} \leq \sum_{j\neq\kappa} |a_{\kappa j}| = R_\kappa.$$

(b) Take D as a diagonal matrix with $d_{\kappa\kappa} = a_{\kappa\kappa}$ for $\kappa = 1,\ldots,n$, and define $B(t) = (1-t)D + tA$. By Fact 4.12, the eigenvalues of $B(0) = D$ are thus simply the diagonal elements of A. Note that, for small t, the Gershgorin discs of $B(t)$ are correspondingly small disks centered at each diagonal element of A. By part (a), the eigenvalues of $B(t)$ lie within the union of all of its Gershgorin discs. As t is increased smoothly from zero to one (that is, as $B(t)$ is converted smoothly from D into A), the eigenvalues of $B(t)$ vary continuously (by Fact 4.23); thus, as t is increased, the eigenvalues of $B(t)$ can only move from one Gershgorin disk into another if two Gershgorin disks intersect. □

Fact 4.25 *If $A = S\Lambda S^{-1}$ is an eigen decomposition of A, then the columns of $(S^{-1})^H$ are the left eigenvectors of A. Further, if the eigenvectors of A are orthonormal, the left and right eigenvectors are identical.*

Proof: Follows immediately from $S^{-1}[A = S\Lambda S^{-1}] \;\Rightarrow\; S^{-1}A = \Lambda S^{-1} \;\Rightarrow\; A^H(S^{-1})^H = (S^{-1})^H\Lambda^H$, noting the definition of a left eigenvector, and the fact that $(S^H)^H = S$. □

Fact 4.26 (The Rayleigh-Ritz Theorem) *For Hermitian B with maximum and minimum (real) eigenvalues given by λ_{max} and λ_{min}, respectively,*

$$\max_{\mathbf{x}^H\mathbf{x}=1} \mathbf{x}^H B\mathbf{x} = \max_{\mathbf{x}\neq 0} \frac{\mathbf{x}^H B\mathbf{x}}{\mathbf{x}^H\mathbf{x}} = \lambda_{\max} \quad \textit{and} \quad \min_{\mathbf{x}^H\mathbf{x}=1} \mathbf{x}^H B\mathbf{x} = \min_{\mathbf{x}\neq 0} \frac{\mathbf{x}^H B\mathbf{x}}{\mathbf{x}^H\mathbf{x}} = \lambda_{\min},$$

Proof: The fact that the max of the unnormalized expression over all $\mathbf{x}$ of unit norm is equal to the max of the normalized expression over all $\mathbf{x} \neq 0$ follows immediately by straightforward scaling arguments. To see their relation to $\lambda_{\max}$, decompose $\mathbf{x} = \sum_{i=1}^n \chi_i \mathbf{s}^i$ where λ_i and $\mathbf{s}^i$ denote the (real) eigenvalues and (orthonormal) eigenvectors of B (see Fact 4.20). Then $B\mathbf{x} = \sum_{k=1}^n \chi_k\lambda_k\mathbf{s}^k$ and

$$\max_{\mathbf{x}^H\mathbf{x}=1} \mathbf{x}^H B\mathbf{x} = \max_{\boldsymbol{\chi}^H\boldsymbol{\chi}=1} \left(\sum_{i=1}^n \chi_i\mathbf{s}^i\right)^H \left(\sum_{k=1}^n \chi_k\lambda_k\mathbf{s}^k\right) = \max_{\boldsymbol{\chi}^H\boldsymbol{\chi}=1} \sum_{i=1}^n |\chi_i|^2\lambda_i = \lambda_{\max}.$$

An analogous proof follows for the statement of the minimum. □

The final step in the equation shown above is easily understood algebraically. The sum may be interpreted as a linear combination of the several eigenvalues λ_i of B, each with a non-negative coefficient $c_i = |\chi_i|^2 \geq 0$, where the sum of these coefficients c_i is unity. To maximize this sum, the coefficient corresponding to $\lambda_{\max}$ must be one and the other coefficients zero, whereas to minimize the sum, the coefficient corresponding to $\lambda_{\min}$ should be one and the other coefficients zero.

4.4.4.1 Hermitian positive definite and Hermitian positive semidefinite matrices

A Hermitian matrix A with all positive eigenvalues is said to be **Hermitian positive definite**[7], which is commonly denoted with the abbreviated notation $A > 0$. A Hermitian matrix A with non-negative eigenvalues is said to be **Hermitian positive semidefinite**, and is commonly denoted $A \geq 0$. **Hermitian negative definite** and **Hermitian negative semidefinite** matrices are defined in an analogous fashion. A Hermitian matrix which might have both positive and negative eigenvalues is said to be **indefinite**. Note that the names **symmetric positive definite**, etc., are commonly used in the special case of real matrices.

[7] The shortened name **positive definite** is sometimes used synonymously with Hermitian positive definite in the complex case and symmetric positive definite in the real case. However, a matrix with all positive eigenvalues is not necessarily Hermitian or symmetric, and non-Hermitian / non-symmetric positive definite matrices, though rare, are occasionally encountered. Thus, this text will avoid the use of this shortened name.

Fact 4.27 *If $A \geq 0$, then $\mathbf{x}^H A\mathbf{x} \geq 0$ for all $\mathbf{x}$. If $A > 0$, then $\mathbf{x}^H A\mathbf{x} > 0$ for all $\mathbf{x} \neq 0$.*

Proof: As A is Hermitian, its eigenvector matrix S is unitary and eigenvalues λ_i are real (Fact 4.20). Thus, for any $\mathbf{x}$, we may write $\mathbf{x} = S\boldsymbol{\chi}$ (where $\boldsymbol{\chi} \neq 0$ if $\mathbf{x} \neq 0$). It follows that

$$\mathbf{x}^H A\mathbf{x} = \boldsymbol{\chi}^H S^H AS\boldsymbol{\chi} = \boldsymbol{\chi}^H S^H S\Lambda\boldsymbol{\chi} = \boldsymbol{\chi}^H \Lambda\boldsymbol{\chi} = \lambda_1|\chi_1|^2 + \lambda_2|\chi_2|^2 + \ldots + \lambda_n|\chi_n|^2.$$

For the case with $A \geq 0$, as all $\lambda_i \geq 0$, it follows that $\mathbf{x}^H A\mathbf{x} \geq 0$.
For the case with $A > 0$ with $\|\mathbf{x}\| > 0$, since $\boldsymbol{\chi} \neq 0$ and all $\lambda_i > 0$, it follows that $\mathbf{x}^H A\mathbf{x} > 0$. □

Fact 4.28 *If $A \geq 0$, then $\mathbf{x}^H B^H AB\mathbf{x} \geq 0$ for all $\mathbf{x}$ and thus $B^H AB \geq 0$. If $A > 0$ and $|B| \neq 0$, then $\mathbf{x}^H B^H AB\mathbf{x} > 0$ for all $\mathbf{x}$ and thus $B^H AB > 0$.*

Proof: The case with $A \geq 0$ follows as in the proof of Fact 4.27, taking $B\mathbf{x} = S\boldsymbol{\chi}$. The case with $A > 0$ follows directly from the case with $A \geq 0$ together with Property 5 of the determinant and Facts 4.17 and 4.27. □

Geometrically, as seen in the proof of Fact 4.27, the matrix $A > 0$ defines a family of concentric ellipsoids with $J = \mathbf{x}^H A\mathbf{x} = \boldsymbol{\chi}^H \Lambda\boldsymbol{\chi} =$ constant, where $\mathbf{x} = S\boldsymbol{\chi}$. The axes of these ellipsoids (in the space of $\mathbf{x}$) are given by the eigenvectors $\mathbf{s}^k$, whereas the extent of these ellipsoids in the direction of each of these eigenvectors is given by $\sqrt{J/\lambda_k}$. It follows easily that

Fact 4.29 *If $A > 0$ and $B > 0$, then $(A+B) > 0$ with $\mathbf{x}^H(A+B)\mathbf{x} > \mathbf{x}^H A\mathbf{x}$ for any $\mathbf{x} \neq 0$.*
If $A \geq 0$ and $B \geq 0$, then $(A+B) \geq 0$ with $\mathbf{x}^H(A+B)\mathbf{x} \geq \mathbf{x}^H A\mathbf{x}$ for any $\mathbf{x}$.

Selecting $\mathbf{x}$ to be zero in some elements and arbitrary in the other elements (denoted z_1 through z_k), it follows directly from Fact 4.27 that, for any principle submatrix B of a Hermitian positive definite matrix A, we may write $\mathbf{z}^H B\mathbf{z} > 0$ for $\mathbf{z} \neq 0$, and thus

Fact 4.30 *Any principle submatrix of a Hermitian positive definite matrix, including its leading and trailing principle submatrices as well as its individual diagonal elements, is itself Hermitian positive definite.*

Note that the first Gauss transformation $M_1 A$ of the Gaussian elimination procedure (see §2.2.1) applied to a Hermitian positive definite matrix A may be written in the form

$$A = \begin{bmatrix} \alpha_1 & \mathbf{v}_1^H \\ \mathbf{v}_1 & B_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \mathbf{v}_1/\alpha_1 & I \end{bmatrix} \begin{bmatrix} \alpha_1 & \mathbf{v}_1^H \\ 0 & B_1 - \mathbf{v}_1\mathbf{v}_1^H/\alpha_1 \end{bmatrix} \tag{4.23}$$

where, by Fact 4.30, $\alpha_1 > 0$ and $B_1 > 0$. That is, the Gaussian elimination procedure without pivoting, when applied to a Hermitian positive definite matrix, will not encounter a zero pivot during the first Gauss transformation. Defining $\beta_1 = \sqrt{\alpha_1}$, this relation may be further decomposed as

$$A = \begin{bmatrix} \beta_1 & 0 \\ \mathbf{v}_1/\beta_1 & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & B_1 - \mathbf{v}_1\mathbf{v}_1^H/\alpha_1 \end{bmatrix} \begin{bmatrix} \beta_1 & \mathbf{v}_1^H/\beta_1 \\ 0 & I \end{bmatrix} = \tilde{G}_1 \tilde{A} \tilde{G}_1^H \quad \Rightarrow \quad \tilde{G}_1^{-1} = \begin{bmatrix} 1/\beta_1 & 0 \\ -\mathbf{v}_1/\alpha_1 & I \end{bmatrix}. \tag{4.24}$$

The above decomposition may be written in the form[8] $\tilde{A} = \tilde{G}_1^{-1} A \tilde{G}_1^{-H}$ where $A > 0$, and thus $\tilde{A} > 0$ by Fact 4.28. Since $\tilde{A} > 0$, $\tilde{A}$ has all positive eigenvalues; thus, since $\tilde{A}$ has a block diagonal decomposition, it follows that $A_1 = B_1 - \mathbf{v}_1\mathbf{v}_1^H/\alpha_1$, which is Hermitian, also has all positive eigenvalues, and therefore $A_1 > 0$. Noting that (4.23) represents the first step of Gaussian elimination without pivoting, and that the matrix $A_1 = B_1 - \mathbf{v}_1\mathbf{v}_1^H/\alpha_1$ may be decomposed in an identical fashion (representing the second step of Gaussian elimination without pivoting), we conclude the following:

Fact 4.31 *The Gaussian elimination procedure without pivoting, when applied to a Hermitian positive definite matrix, will not encounter a zero pivot.*

[8] Note that $\tilde{G}_1 \tilde{G}_1^H \neq I$, and thus this is *not* a similarity transformation (that is, the eigenvalues of A and $\tilde{A}$ are different).

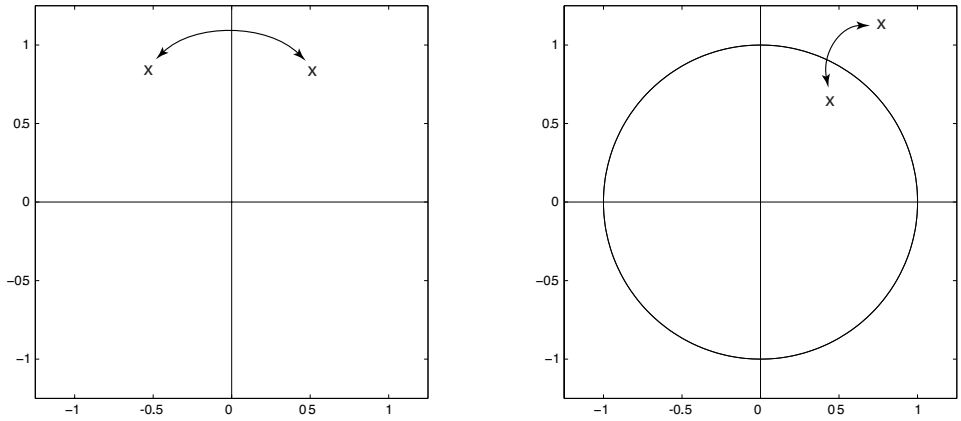


Figure 4.4: (left) The symmetric root property: if Z is Hamiltonian, then for every eigenvalue in the LHP, there is a corresponding eigenvalue in the RHP. (right) The reciprocal root property: if M is symplectic, then for every eigenvalue inside the unit circle, there is a corresponding eigenvalue outside the unit circle.

4.4.4.2 Hamiltonian and symplectic matrices

Partition Z and M and define the **symplectic identity** $J = J_{2n\times 2n}$ such that

$$Z = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}; \qquad M = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}; \qquad J = \begin{bmatrix} 0 & I \\ -I & 0 \end{bmatrix} \quad \Rightarrow \quad J^{-1} = -J = J^T.$$

The following facts follow immediately from Facts 4.6, 4.16, and 4.2:

Fact 4.32 (The Symmetric Root Property) *If $J^{-1}ZJ = -Z^H$ (that is, if Z is* **Hamiltonian***), then for each eigenvalue $\lambda = \lambda_R + i\lambda_I \in \lambda(Z)$ there is a corresponding eigenvalue $-\overline{\lambda} = -\lambda_R + i\lambda_I \in \lambda(Z)$ with the same multiplicity. Further, if Z is Hamiltonian, then $Z_{22} = -Z_{11}^H$, $Z_{12} = Z_{12}^H$, and $Z_{21} = Z_{21}^H$.*

Fact 4.33 (The Reciprocal Root Property) *If $J^{-1}MJ = M^{-H}$ (that is, if M is* **symplectic***), then for each eigenvalue $\lambda = Re^{i\theta} \in \lambda(M)$ there is a corresponding eigenvalue $1/\overline{\lambda} = (1/R)e^{i\theta} \in \lambda(M)$ with the same multiplicity. Further, if M is symplectic, then $M_{11} = M_{22}^{-H} + M_{12}M_{22}^{-1}M_{21}$ and $M_{22} = M_{11}^{-H} + M_{21}M_{11}^{-1}M_{12}$.*

The symmetric root property implies that any Hamiltonian Z has as many eigenvalues in the LHP as it has in the RHP, whereas the reciprocal root property implies that any symplectic M has as many eigenvalues inside the unit circle as it has outside the unit circle, as illustrated in Figure 4.4.

Matrices with Hamiltonian structure, which by Fact 4.32 satisfy the Symmetric Root Property (Figure 4.4a), are encountered when solving the **continuous-time algebraic Riccati equation** (**CARE**) at the heart of both the control and estimation of infinite-horizon continuous-time linear-time-invariant (LTI) systems in state-space form [see (21.13), (21.29), and (22.13)], as discussed in §4.5.2.

Matrices with symplectic structure, which by Fact 4.33 satisfy the Reciprocal Root Property (Figure 4.4b), are encountered when solving the **discrete-time algebraic Riccati equation** (**DARE**) at the heart of both the control and estimation of infinite-horizon discrete-time linear-time-invariant (LTI) systems in state-space form [see (21.43), (21.56), and (22.24)], as discussed in §4.5.4.

4.4.5 Computing the eigen and Schur decompositions†

Power methods

The simplest iterative technique available to determine the largest eigenvalue and corresponding eigenvector of a matrix A is the **power method**. This method initializes $\mathbf{x}^{(0)} \neq 0$ arbitrarily and repeatedly calculates

$$\mathbf{x}^{(k+1)} = A\mathbf{x}^{(k)}. \tag{4.25}$$

For problems in which A is known to have a complete set of linearly independent eigenvectors (e.g., if A is Hermitian, skew-Hermitian, or the eigenvalues of A are distinct; see Facts 4.20, 4.21, and 4.19), we may write $\mathbf{x}^{(0)} = S\boldsymbol{\chi}$, where S is the matrix of eigenvectors of A and $\boldsymbol{\chi}$ is a vector of coefficients; it thus follows from the relation $AS = S\Lambda$ that

$$\mathbf{x}^{(k)} = A^k\mathbf{x}^{(0)} = A^kS\boldsymbol{\chi} = S\Lambda^k\boldsymbol{\chi} = \chi_1\lambda_1^k\mathbf{s}^1 + \chi_2\lambda_2^k\mathbf{s}^2 + \ldots + \chi_n\lambda_n^k\mathbf{s}^n.$$

If $|\lambda_1| > |\lambda_2|$ and $\chi_1 \neq 0$, assuming that the remaining eigenvalues are ordered such that $|\lambda_2| \geq |\lambda_3| \geq |\lambda_4| \geq \ldots$, the magnitude of the term in the direction of $\mathbf{s}^1$ eventually dominates the magnitude of the other terms, and thus $\mathbf{x}^{(k)}$ is, eventually, approximately aligned with $\mathbf{s}^1$. The rate of convergence is ultimately dominated by the factor by which the first term grows faster than the second, that is, by $|\lambda_1|/|\lambda_2|$. If this factor is relatively small (e.g., if $|\lambda_1|/|\lambda_2| = 1.1$), convergence of the algorithm is relatively slow.

For problems in which A might not have a complete set of linearly independent eigenvectors, the conclusion is the same (again, so long as $|\lambda_1| > |\lambda_j|$ for $j \geq 2$), but the analysis is slightly more involved. We know by the Schur decomposition theorem that the decomposition $A = UTU^H$ always exists, where U is unitary, T is triangular, and the eigenvalues of A appear on the main diagonal of T in any desired order. For the sake of analysis, we again select the ordering $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots$ It follows immediately that

$$\mathbf{x}^{(k+1)} = A\mathbf{x}^{(k)} = UTU^H\mathbf{x}^{(k)} \quad \Rightarrow \quad \mathbf{y}^{(k+1)} = \frac{1}{\lambda_1}T\mathbf{y}^{(k)} = \begin{pmatrix} 1 & c_{1,2} & \ldots & c_{1,n-1} & c_{1,n} \\ & \lambda_2/\lambda_1 & \ldots & c_{2,n-1} & c_{2,n} \\ & & \ddots & \vdots & \vdots \\ & & & \lambda_{n-1}/\lambda_1 & c_{n-1,n} \\ 0 & & & & \lambda_n/\lambda_1 \end{pmatrix} \mathbf{y}^{(k)}$$

where $\mathbf{y}^{(k)} = U^H\mathbf{x}^{(k)}/\lambda_1^k$. We now examine the convergence of the components of $\mathbf{y}^{(k)}$ as k is increased:

$$\begin{aligned}
y_n^{(k)} &= \left(\frac{\lambda_n}{\lambda_1}\right) y_n^{(k-1)} = \left(\frac{\lambda_n}{\lambda_1}\right)^k y_n^{(0)} \xrightarrow[k\to\infty]{} 0 && \text{since } \left|\frac{\lambda_n}{\lambda_1}\right| < 1, \\
y_{n-1}^{(k)} &= \left(\frac{\lambda_{n-1}}{\lambda_1}\right) y_{n-1}^{(k-1)} + c_{n-1,n}y_n^{(k-1)} \xrightarrow[k\to\infty]{} 0 && \text{since } \left|\frac{\lambda_{n-1}}{\lambda_1}\right| < 1 \text{ and } y_n^{(k)} \xrightarrow[k\to\infty]{} 0, \\
&\vdots \\
y_2^{(k)} &= \left(\frac{\lambda_2}{\lambda_1}\right) y_2^{(k-1)} + c_{2,3}y_3^{(k-1)} + \ldots + c_{2,n}y_n^{(k-1)} \xrightarrow[k\to\infty]{} 0 && \text{since } \left|\frac{\lambda_2}{\lambda_1}\right| < 1 \text{ and } y_j^{(k)} \xrightarrow[k\to\infty]{} 0 \text{ for } j = 3,\ldots,n, \\
y_1^{(k)} &= y_1^{(k-1)} + c_{1,2}y_2^{(k-1)} + \ldots + c_{1,n}y_n^{(k-1)} \xrightarrow[k\to\infty]{} C && \text{since } y_j^{(k)} \xrightarrow[k\to\infty]{} 0 \text{ for } j = 2,\ldots,n.
\end{aligned}$$

†As in §1, we use the † symbol here (and elsewhere in the text) to indicate that a thorough understanding of this particular section is not necessary upon first read in order to follow the flow of the remainder of the presentation. The facts highlighted in the other sections of §4 are perhaps more immediately useful than the somewhat involved algorithms presented in §4.4.5. However, an *overview* of this section is in fact useful even from the get-go, in order to get an idea of how eigenvalues are actually determined. Be sure to come back later for a more careful read!

Thus, $\mathbf{y}^{(k)} \to C\mathbf{e}^1$ as $k \to \infty$, and therefore $\mathbf{x}^{(k)}/\lambda_1^k \to C\mathbf{u}^1$ as $k \to \infty$. In other words, unless $C = 0$ (it usually is not; for more discussion on this point, see Footnote 13 on page 100), $\mathbf{x}^{(k)}$ eventually converges towards the direction of the first Schur vector of A (that is, the eigenvector of A corresponding to λ_1).

The **inverse power method** is similar to the power method, but instead of marching the iterative equation $\mathbf{x}^{(k+1)} = A\mathbf{x}^{(k)}$, it marches the equation

$$\mathbf{x}^{(k+1)} = A^{-1}\mathbf{x}^{(k)} \quad \text{or, equivalently,} \quad A\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}. \tag{4.26}$$

The latter form may be solved efficiently using the Gaussian elimination techniques discussed in §2. Following a similar analysis as for the power method, noting that the eigenvalues of A^{-1} are the reciprocal of the eigenvalues of A (Fact 4.16), in the case of the inverse power method the magnitude of the term in the direction of $\mathbf{s}^n$ eventually dominates the magnitude of the other terms, and thus $\mathbf{x}^{(k)}$ is, for sufficiently large k, approximately aligned with $\mathbf{s}^n$. The rate of convergence is ultimately dominated by the factor by which the n'th term grows faster than the $(n-1)$'th, that is, by $|\lambda_{n-1}|/|\lambda_n|$. If this factor is relatively small (e.g., $|\lambda_{n-1}|/|\lambda_n| = 1.1$), convergence of the algorithm is relatively slow.

Finally, the **shifted inverse power method** enormously accelerates this procedure. With this method, if at the k'th iteration an approximation μ_k of the eigenvalue λ_j is available, we march the equation

$$\mathbf{x}^{(k+1)} = (A - \mu_k I)^{-1}\mathbf{x}^{(k)} \quad \text{or, equivalently,} \quad (A - \mu_k I)\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}. \tag{4.27}$$

The eigenvalues of $(A - \mu_k I)$ are just shifts of the eigenvalues of A [specifically, they are given by $(\lambda_i - \mu_k)$ for $i = 1, \ldots, n$], and the eigenvectors of $(A - \mu_k I)$ are the same as the eigenvectors of A. Thus, the factor of convergence of shifted inverse power method at each step is given by $|\lambda_i - \mu_k|/|\lambda_j - \mu_k|$, where λ_i is the next closest eigenvalue to μ_k (that is λ_i is the eigenvalue which minimizes $|\lambda_i - \mu_k|$ for all $i \neq j$). If μ_k is a good approximation of λ_j, then this factor is large, and convergence of the scheme to $\mathbf{s}^j$ is quite rapid. In fact[9,10],

- if μ_0 is an accurate approximation of an eigenvalue λ_j obtained by some other method (e.g., via computation of a Schur decomposition $A = UTU^H$ using the QR method, as developed below), convergence of the shifted inverse power method to the eigenvector $\mathbf{s}^j$ is obtained in just one or two steps.

Thus, *the primary difficulty lies with the computation of the eigenvalues*; once they are obtained, determination of the eigenvectors via the shifted inverse power method is relatively easy (see Algorithm 4.10).

Note that $\mathbf{x}^{(k)}$ in (4.25), (4.26), and (4.27) may be normalized at each step in any desired or convenient manner without altering its rate convergence towards the direction of the desired eigenvector. To clarify the analysis that follows, we will thus normalize $\mathbf{x}^{(k)}$ to be of unit norm at each iteration [$(\mathbf{x}^{(k)})^H\mathbf{x}^{(k)} = 1$].

Finally, if an accurate approximation μ of the eigenvalue λ_n is not available, the remaining question to be addressed is, at each step k of the shifted inverse power method, how should we select the approximation μ_k to the eigenvalue λ_n? Inspired by Fact 4.26, one good choice turns out to be the **Rayleigh quotient shift** $\mu_k = (\mathbf{x}^{(k)})^H A\mathbf{x}^{(k)}$. The reason this is an appropriate choice is that, as $\mathbf{x}^{(k)}$ aligns closer and closer to the direction of the eigenvector $\mathbf{s}^n$, the expression $A\mathbf{x}^{(k)}$ approaches $\lambda_n\mathbf{x}^{(k)}$, and thus $\mu_k = (\mathbf{x}^{(k)})^H A\mathbf{x}^{(k)} \to \lambda_n(\mathbf{x}^{(k)})^H\mathbf{x}^{(k)} = \lambda_n$; that is, μ_k approaches the eigenvalue λ_n.

[9] In such a problem, a consistently good choice for $\mathbf{x}^{(0)}$ is given by solving $U\mathbf{w} = \mathbf{e}$ for $\mathbf{w}$, then taking $\mathbf{x}^{(0)} = \mathbf{w}/\|\mathbf{w}\|$, where U is found via Gaussian elimination with partial pivoting applied to the matrix $(A - \mu_0 I)$ such that $P(A - \mu_0 I) = LU$, and $\mathbf{e} = \begin{pmatrix} 1 & 1 & \ldots & 1 \end{pmatrix}^T$. While other (perhaps, simpler) choices are often just as good, the reason that this choice is consistently good is that it is generally found not to be deficient in its component in the $\mathbf{s}^j$ direction. For further discussion, see Wilkinson (1965).

[10] Note also that, if μ_k is an accurate approximation of an eigenvalue λ_j of the matrix A, then the shifted inverse power method solves a problem that is nearly singular. However, μ_k is only an approximation of λ_j, and finite-precision arithmetic is used in the computations; thus, it is in fact found that the shifted inverse power method rarely fails in practice. In those isolated pathological examples in which it might fail, the techniques of §2.6 (see in particular the second bullet point of Fact 2.3) could be used instead, thereby determining the eigenvector $\mathbf{s}^j$ from the exact value of the eigenvalue λ_j.

Algorithm 4.10: Computation of the eigenvectors and Schur vectors using the shifted inverse power method.

View
Test

```
function [S,T] = ShiftedInversePower(A,mu)
% Apply two steps of the shifted inverse power method (per eigenvalue mu_k) to determine
% (if called with nargout=1) the eigenvectors S(:,k), OR (if called with nargout=2)
% the Schur vectors U(:,k) and upper-triangular T of the Schur decomposition A=U*T*U'.
n=size(A,1);
for k=1:length(mu);
  B=A-mu(k)*eye(n);                                  % Compute B=PLU (see GaussPP.m)
  for j = 1:n-1,                                     % Loop through each column j<n
    [amax,imax]=max(abs(B(j:n,j)));                  % If necessary, exchange the rows of B.
    if amax>abs(B(j,j)); B([j j-1+imax],:)=B([j-1+imax j],:); end
    B(j+1:n,j)     = - B(j+1:n,j) / B(j,j);                    % Compute m_ij.
    B(j+1:n,j+1:n) = B(j+1:n,j+1:n) + B(j+1:n,j) * B(j,j+1:n);     % Outer product update.
  end
  if B(n,n)==0,              % Eigenvalue exact! Solve Bs=0 exactly for a solution.
    S(n,k)=1;
    for i = n-1:-1:1,   S(i,k) = -B(i,i+1:n)*S(i+1:n,k) / B(i,i); end  % Backsubstitution.
  else                       % Eigenvalue approximate. Apply Shifted Inverse Power method.
    S(:,k)=ones(n,1);        % Initialize (see footnote 8). (no need to apply P to e!)
    S(n,k)=S(n,k) / B(n,n);  % Solve Ux=e (see GaussPP.m)
    for i = n-1:-1:1,   S(i,k) = (S(i,k)-B(i,i+1:n)*S(i+1:n,k)) / B(i,i); end
    for steps=1:2            % Then apply two steps of the shifted inverse power method.
      for j = 1:n-1, S(j+1:n,k) = S(j+1:n,k) + B(j+1:n,j) * S(j,k); end
      S(n,k) = S(n,k) / B(n,n);
      for i = n-1:-1:1, S(i,k) = (S(i,k)-B(i,i+1:n)*S(i+1:n,k)) / B(i,i); end
    end
  end
  if nargout>1; S(:,k) = S(:,k)-S(:,1:k-1)*S(:,1:k-1)'*S(:,k); end;  % Orthogonalize.
  S(:,k)=S(:,k)/norm(S(:,k));    % Note above that the Schur decomposition may be derived
end                              % from this method simply by orthogonalization of the S
if nargout>1; T=S'*A*S; end;     % matrix and recomputation of T.
end % function ShiftedInversePower
```

By using this expression for μ_k, the shifted inverse power method converges quickly. Putting it all together, starting from an arbitrary initial vector $\mathbf{x}^{(0)}$ of unit norm, the shifted inverse power method with Rayleigh quotient shifts is written quite simply as follows:

$$\begin{aligned} \mu_k &= (\mathbf{x}^{(k)})^H A \mathbf{x}^{(k)} \\ (A-\mu_k I)\mathbf{w} &= \mathbf{x}^{(k)} \\ \mathbf{x}^{(k+1)} &= \mathbf{w}/\|\mathbf{w}\|. \end{aligned} \tag{4.28}$$

The block power method and its equivalence to the unshifted *QR* method

Starting from some vector $\mathbf{x}^{(0)}$ and assuming $|\lambda_1| > |\lambda_j|$ for $j \geq 2$, the power method $\mathbf{x}^{(k+1)} = A\mathbf{x}^{(k)}$ presented above converges to the eigenvector corresponding to the eigenvalue of A with maximum amplitude.

Instead of iterating on a single vector, we now extend the power method by iterating on a *set* of vectors $\{\mathbf{u}^1, \mathbf{u}^2, \ldots, \mathbf{u}^n\}^{(k)}$, lined up as columns in a matrix U_k. This extended method scales each vector in the set by A, assembling the resulting vectors in the columns of a matrix Z, then orthogonalizes the resulting set of vectors (using, e.g., Modified Gram Schmidt in Algorithm 4.5), and repeats. Exactly as in the power method, the first vector $(\mathbf{u}^1)^{(k)}$ of this extended method converges to the eigenvector $\mathbf{s}^1$ corresponding to the eigenvalue λ_1 of A with maximum amplitude; this eigenvector is in fact the same as the first Schur vector $\mathbf{u}^1$ of A. By a similar argument to that given in the second paragraph of §4.4.5, assuming now that $|\lambda_2| > |\lambda_3|$, the next vector in the set $(\mathbf{u}^2)^{(k)}$, which is constrained at each iteration to be orthogonal to the first vector

in the set $(\mathbf{u}^1)^{(k)}$, converges to the second Schur vector $\mathbf{u}^2$ of A, corresponding to the eigenvalue of A with the second largest amplitude, etc.[11] We now consider the case in which we work with n vectors, which, for simplicity, we initialize as the Cartesian unit vectors. Subject to only mild assumptions[12,13] (specifically, that $|\lambda_1| > |\lambda_2| > |\lambda_3| > \ldots$ and also that the initialization chosen for each column does not happen to be lacking in the necessary components in each of the corresponding Schur vector directions), this method converges to a Schur decomposition of A. In equations, this method, called the **block power method**, is defined by

$$U_0 = I, \tag{4.29a}$$
$$Z = AU_{k-1}, \tag{4.29b}$$
$$Z = U_k R_k \quad \text{(that is, determine a } QR \text{ decomposition of } Z\text{)}. \tag{4.29c}$$

Following the insightful analysis of Trefethen & Bau (1997), we consider this method together with the supplemental matrices

$$T_k = U_k^H A U_k, \tag{4.29d}$$
$$\widetilde{R}_k = R_k R_{k-1} \cdots R_1. \tag{4.29e}$$

Fact 4.34 *The U_k and $\widetilde{R}_k$ defined by (4.29) form a QR decomposition of the k'th power of A, that is,*

$$A^k = U_k \widetilde{R}_k, \tag{4.30a}$$

and the matrix T_k defined by (4.29) is unitarily similar to A via the transformation matrix U_k, that is,

$$A = U_k T_k U_k^H. \tag{4.30b}$$

Proof: First note that (4.30b) is identical to (4.29d), so (4.30b) is verified immediately. Note also that (4.29) implies that $A^0 = U_0 = \widetilde{R}_0 = I$ and $T_0 = A$, and thus (4.30a) is verified immediately for the base case $k = 0$. Now, assuming (4.30a) is true for the case $k-1$, it follows directly from the relations in (4.29) that it must also be true for the case k:

$$A^k = AA^{k-1} = AU_{k-1}\widetilde{R}_{k-1} = Z\widetilde{R}_{k-1} = U_k R_k \widetilde{R}_{k-1} = U_k \widetilde{R}_k,$$

thereby proving Fact 4.34 by induction. □

Now consider the **unshifted** QR **method** defined by

$$T_0 = A, \tag{4.31a}$$
$$T_{k-1} = Q_k R_k \quad \text{(that is, determine a } QR \text{ decomposition of } T_{k-1}\text{)}, \tag{4.31b}$$
$$T_k = R_k Q_k, \tag{4.31c}$$

which we analyze here together with the supplemental matrices

$$U_k = Q_1 Q_2 \cdots Q_k \tag{4.31d}$$
$$\widetilde{R}_k = R_k R_{k-1} \cdots R_1. \tag{4.31e}$$

[11] In the notation of the discussion in the second paragraph of §4.4.5, defining $\mathbf{y}^{(k)}$ in this case as $\mathbf{y}^{(k)} = U^H \mathbf{x}^{(k)}/\lambda_2^k$, it is found that $\mathbf{y}^{(k)} \to C\mathbf{e}^2$ as $k \to \infty$, and therefore $\mathbf{x}^{(k)}/\lambda_2^k \to C\mathbf{u}^2$ as $k \to \infty$.

[12] The first assumption on the strict separation of the magnitudes of the eigenvalues is in fact somewhat restrictive (especially in the case of real matrices with eigenvalues in complex conjugate pairs), and will thus be relaxed when shifting is applied in later sections. Note that, in fact, *repeated* eigenvalues $\lambda_i = \lambda_{i+1} = \ldots = \lambda_{i+m}$ present no obstacle to this method, as the method will simply converge to m orthogonal vectors spanned by $\{\mathbf{u}^i, \mathbf{u}^{i+1}, \ldots, \mathbf{u}^{i+m}\}$; in this setting, any such m vectors will work.

[13] The second assumption is generally not as strict as it may first appear, as numerical errors usually work in our favor as the algorithm proceeds to introduce small components in these directions even if they are initially lacking. Convergence of this method, once the acceleration techniques presented later in this section are applied, is so rapid that such initialization of the necessary components via small numerical errors is, in fact, entirely adequate.

Fact 4.35 *The U_k, $\widetilde{R}_k$, and T_k defined by* (4.31) *also satisfy the relations given in* (4.30a) *and* (4.30b).

Proof: First note that (4.31) implies that $A^0 = U_0 = \widetilde{R}_0 = I$ and $T_0 = A$, and thus (4.30a) and (4.30b) are verified immediately for the case $k = 0$. Now, assuming (4.30a) and (4.30b) are true for the case $k-1$, it follows directly from the relations in (4.31) that they must also be true for the case k:

$$A^k = AA^{k-1} = AU_{k-1}\widetilde{R}_{k-1} = U_{k-1}T_{k-1}\widetilde{R}_{k-1} = U_{k-1}Q_kR_k\widetilde{R}_{k-1} = U_k\widetilde{R}_k$$

and

$$T_{k-1} = U_{k-1}^H AU_{k-1} \quad \Rightarrow \quad Q_k^H[Q_kR_k = U_{k-1}^H AU_{k-1}]Q_k \quad \Rightarrow \quad R_kQ_k = T_k = U_k^H AU_k,$$

thereby proving Fact 4.35 by induction. □

Together, Facts 4.34 and 4.35 establish that the two iteration schemes (4.29) and (4.31) are equivalent[14].

Accelerating the *QR* method: preliminary comments

As shown above, the block power method (4.29a)-(4.29c) ultimately converges to a Schur decomposition of A; thus, the unshifted *QR* method (4.31a)-(4.31c), which is equivalent (as established in Facts 4.34 and 4.35), converges in an identical manner. Both methods, though elegant in their simplicity, are very slow to converge. However, leveraging various facts established above, we are now in an exceptional position to accelerate tremendously the *QR* method in particular via the following four steps:

(i) Note first that, before starting the *QR* iterations to approximate the Schur decomposition $A = UTU^H$, thereby determining (in the diagonal elements of T) the eigenvalues of A, we actually don't have to start this iteration from "scratch" (that is, with the full matrix A), as indicated in (4.31a). Indeed, we have already identified an efficient technique to introduce *many* zeros into A by reducing it all the way to Hessenberg form T_0 via a unitary similarity tranformation in a finite number of steps (see §4.4.1). By Fact 4.6, the T_0 so computed has the same eigenvalues as A. Thus, the Hessenberg form T_0 provided by Algorithm 4.2 is the preferred starting point for the *QR* method. Further, as easily verified, if T_{k-1} is upper Hessenberg, then, calculating its *QR* decomposition $T_{k-1} = Q_kR_k$ via Algorithm 4.7 or 4.9, Q_k is also upper Hessenberg. As R_k is upper triangular by construction, the product $T_k = R_kQ_k$ is also upper Hessenenberg (Fact 1.10). Thus, the upper Hessenberg structure of T_k may be leveraged at each step k, both when computing the decomposition $T_{k-1} = Q_kR_k$ and when calculating the product $T_k = R_kQ_k$. It is thus seen that *the role of the QR iterations is simply to diminish the elements in the first subdiagonal of the upper Hessenberg matrix T_k to (nearly) zero*, thereby reducing it (iteratively) towards an eigenvalue-revealing Schur form.

(ii) The second step in the refinement of the unshifted *QR* method (4.31) is to note that the shifting idea discussed previously may be applied to accelerate the convergence of one of the Schur vectors being calculated (typically, the last in the set). Note that, in the case of the unshifted *QR* method, we may write

$$T_{k-1} = (Q_kR_k)Q_kQ_k^H = Q_k(R_kQ_k)Q_k^H = Q_kT_kQ_k^H.$$

Thus, T_k is unitarily similar to T_{k-1}. In a similar fashion, if we consider the **shifted QR method** defined by

$$T_0 = \text{Hessenberg form derived from a unitary similarity decomposition of } A \text{ via Algorithm 4.2}, \tag{4.32a}$$
$$(T_{k-1} - \mu_kI) = Q_kR_k \quad \text{[that is, determine a } QR \text{ decomposition of } (T_{k-1} - \mu_kI)], \tag{4.32b}$$
$$T_k = R_kQ_k + \mu_kI, \tag{4.32c}$$

[14] That is, the block power method (4.29a)-(4.29c) determines a U_k (an orthogonalization of the columns of the k'th power of A), from which the corresponding T_k may be extracted according to $T_k = U_k^H AU_k$, whereas the unshifted *QR* method (4.31a)-(4.31c) determines an essentially equivalent value of T_k directly.

then it follows similarly that

$$T_{k-1} = (Q_k R_k + \mu_k I) Q_k Q_k^H = Q_k (R_k Q_k + \mu_k I) Q_k^H = Q_k T_k Q_k^H . \tag{4.33}$$

Thus, even when such shifts are applied, T_k is unitarily similar to T_{k-1}. It follows in turn that T_k is unitarily similar to the original matrix A. If good shifts μ_k are selected (a topic that is deferred to the following three subsections), then the convergence of one of the eigenvalues [on the main diagonal of T_k] and the corresponding Schur vector [which may be reconstructed via (4.31d), if desired] is immensely accelerated, as explained in the case of the shifted inverse power method above[15].

(iii) The third step in the refinement of the *QR* method is to apply Fact 4.12 regarding the eigenvalues of a block upper triangular matrix. When one or more of the elements in the first subdiagonal of the upper Hessenberg matrix T_k is reduced to (nearly) zero, then T_k may be partitioned in block upper triangular form, and the eigenvalues of the upper Hessenberg blocks on the main diagonal of T_k may be determined separately; the eigenvalues of T_k are then given by the union of the eigenvalues of these upper Hessenberg blocks. This process of splitting the eigenvalue computation into smaller subproblems is referred to as **deflation**.

To achieve deflation efficiently in all of the cases we will consider, avoiding recursion and its associated overhead, consider a 3×3 block upper triangular partitioning of T at each iteration. Denote by T_{11}, T_{22}, and T_{33} the three square blocks on the main diagonal in this partitioning. Select T_{33} to be as large as possible while being upper triangular; that is, T_{33} contains all eigenvalues already determined in the lower-right corner on the main diagonal of T. Then, select T_{22} to be as large as possible while still being **unreduced** (that is, with no zero elements on its subdiagonal); T_{11} contains the remaining elements in the upper-left corner of T. Then, simply apply a shifted *QR* iteration to T_{22}, and repeat the entire process until T_{22} is empty.

(iv) Finally, note that, to save computational effort, we do not bother accumulating the transformation matrices U_k that complete the (nearly) triangularizing unitary similarity transformation $A = U_k T_k U_k^H$, as the computations to determine this transformation matrix during the iteration are relatively expensive. Once T_k converges to an essentially triangular form (and, thus, the λ_i are determined), computing the corresponding eigenvectors (that is, the columns of S) via the shifted inverse power method discussed previously is straightforward. If it is the corresponding Schur vectors (that is, the columns of U) that are desired, a straightfoward modification of the shifted inverse power method discussed previously may be used, subtracting off the components of $\mathbf{x}^k$ at each iteration k in the directions of each the previously computed Schur vectors (see Algorithm 4.10).

To summarize, to calculate the eigenvalues of a matrix A:

(a) Start by taking the Hessenberg decomposition using Algorithm 4.2.
(b) Identify the unreduced block T_{22} on the main diagonal of a block upper triangular partitioning of T.
(c) Apply the shifted *QR* method to T_{22}, and repeat from step (b) until T_{22} is empty.
(d) Compute the Schur vectors or eigenvectors, as necessary, using Algorithm 4.10.

To clarify the details of the acceleration of the *QR* algorithm, we isolate below three special cases:

- general (non-Hermitian complex) matrices,
- Hermitian (complex) and symmetric (real) matrices, and
- non-symmetric real matrices.

[15] Recall that the convergence of the *QR* method is identical to the convergence of the corresponding block power method.

Algorithm 4.11: Compute the eigenvalues of a general (non-Hermitian complex) matrix A.

View
Test

```
function [lam] = EigGeneral(A)
% Computes the eigenvalues of a general complex matrix A. After an
% initial Hessenberg decomposition, several explicitly shifted QR steps are taken.
A=Hessenberg(A); n=size(A,1); q=n; tol=1e-12;
while q>1          % Note: diagonal of T_22 block extends from {p,p} to {q,q} elements of T.
  for i=1:n-1; if abs(A(i+1,i))< tol*(abs(A(i,i))+abs(A(i+1,i+1))); A(i+1,i)=0; end; end
  q=1; for i=n-1:-1:1; if A(i+1,i)~=0, q=i+1; break, end, end, if q==1, continue, end
  p=1; for i=q-1:-1:1; if A(i+1,i)==0, p=i+1; break, end, end
  d=ones(n,1); mu=A(q,q);              % Initialize d and compute shift (lower-right corner).
  for i=p:q; A(i,i)=A(i,i)-mu; end  % Shift A.
  for i=p:q-1                          % Apply same algorithm as in QRFastGivensHessenberg.
    [a(i),b(i),gamma(i),dn(i),d([i i+1])]=FastGivensCompute(A(i,i),A(i+1,i),d(i),d(i+1));
    [A]=FastGivens(A,a(i),b(i),gamma(i),dn(i),i,i+1,i,q,'L');
  end
  for i=p:q-1              % Apply the postmultiplications directly to A (thus computing R*Q).
    [A]=FastGivens(A,a(i),b(i),gamma(i),dn(i),i,i+1,p,q,'R');
  end
  for i=p:q, dt=1/sqrt(d(i)); A(i,max(i-1,1):end)=A(i,max(i-1,1):end)*dt;      % Scale A.
             A(1:min(i+1,n),i)  =A(1:min(i+1,n),i)*dt;  A(i,i)=A(i,i)+mu; end % Unshift A.
end, lam=diag(A);                      % Extract eigenvalues from main diagonal of result.
end % function EigGeneral
```

Accelerating the *QR* method: the general (non-Hermitian complex) case

In this case, we may follow the *QR* method with the four acceleration steps enumerated above without further embellishment, as illustrated in Algorithm 4.11. At each iteration k, we may shift simply by the lower-right element of T_k, which is usually the first eigenvalue to converge.

Following the analysis of Golub & van Loan (1996), if at one step of the shifted *QR* method the matrix T_{k-1} has the form

$$T_{k-1} = \begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & \varepsilon & \mu_k \end{pmatrix}, \tag{4.34a}$$

where x denotes nonzero entries, $\varepsilon \ll 1$, and the subsequent shift is denoted μ_k, then it follows from (4.32b)-(4.32c) (see Exercise 4.9) that

$$T_k = \begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & \delta & \mu_{k+1} \end{pmatrix} \tag{4.34b}$$

where $\delta \propto \varepsilon^2$; that is, *convergence of the shifted QR method is quadratic*.

Algorithm 4.12: Compute the eigenvalues of a Hermitian matrix A.

View
Test

```
function [lam] = EigHermitian(A)
% Compute the eigenvalues of a Hermitian (or, if real, symmetric) matrix A.
% Leveraging the fact that Hessenberg returns a tridiagonal matrix, the QR algorithm
% with Wilkenson shifts is applied to the tridiagonal matrix T=tridiag[a,b,c] at each step.
A=Hessenberg(A); n=size(A,1); q=n; tol=1e-13;     % Note: we move the 3 nonzero diagonals to
a=[0; diag(A,-1)]; b=diag(A); c=[diag(A,1); 0];  % vectors to speeed the memory access.
while q>1          % Note: diagonal of T_22 block extends from {p,p} to {q,q} elements of T.
  for i=1:n-1; if abs(a(i+1))< tol*(abs(b(i))+abs(b(i+1))); a(i+1)=0; end; end
  q=1; for i=n-1:-1:1; if a(i+1)~=0, q=i+1; break, end, end, if q==1, continue, end
  p=1; for i=q-1:-1:1; if a(i+1)==0, p=i+1; break, end, end
  t=(b(n-1)-b(n))/2.; mu=b(n)+t-sign(t)*sqrt(abs(t)^2 + abs(a(n))^2);     % Wilkenson shift
  b=b-mu; [b,c,a,cc,ss]=QRGivensTridiag(a,b,c);       % Shift and calculate QR using Givens
  % Now calculate R*Q, given the n-1 values of [cc,ss] comprising each rotation
  % (rather than Q itself) and the nonzero diagonals [b,c,a] of the (uppertriangular) R.
  for i=1:n-1; if cc(i)~=0
    if i>1, c(i-1)=cc(i)*c(i-1)-ss(i)       *a(i-1); end        % Eqn (1.12b), column i
    temp   =        cc(i)*b(i)  -ss(i)       *c(i);
    c(i)   = conj(ss(i))*b(i)  +conj(cc(i))*c(i);               % Eqn (1.12b), column k=i+1
    b(i+1) =                    conj(cc(i))*b(i+1); b(i)=temp;
  end, end,  a(2:n)=conj(c(1:n-1)); b=b+mu;                     % Unshift A
end, lam=EigSort(real(b));              % Sort, and remove any error in the imaginary parts.
end % function EigHermitian
```

Accelerating the *QR* method: the Hermitian/symmetric case

We now consider the case in which A is Hermitian (or, if it happens to be real, symmetric). Note first that, in this case, A has real eigenvalues. The first step of (4.32) is to compute the Hessenberg decomposition of A. Note that, since the Hessenberg form T_0 is unitarily similar to A (see Fact 4.7), T_0 is also Hermitian. Thus, as T_0 is both Hessenberg and Hermitian, it must be *tridiagonal*.

Further, as easily verified, if T_{k-1} is tridiagonal then, calculating its shifted QR decomposition $(T_{k-1} - \mu_k I) = Q_k R_k$ via Algorithm 4.9, it turns out that R_k is upper tridiagonal. By Fact 1.10, T_k is upper Hessenberg. Additionally, since T_k is unitarily similar to the Hermitian matrix T_0 [see (4.30b)], T_k in this case is also Hermitian. Thus, as T_k is both Hessenberg and Hermitian, it is also tridiagonal. Thus, the tridiagonal structure of T_k may be leveraged at each step k, both when computing the decomposition $(T_{k-1} - \mu_k I) = Q_k R_k$ and when calculating $T_k = R_k Q_k + \mu_k I$. Again, the role of the QR iterations is simply to diminish the elements in the subdiagonal of the Hermitian tridiagonal matrix T_k to (nearly) zero, thereby reducing it to an eigenvalue-revealing diagonal form.

We could again shift by the lower-right element of T at each step. A better choice for the shift is given by the eigenvalue of the 2×2 submatrix in the lower-right corner of T which is closest to the element in its lower-right corner. In the present notation, this 2×2 submatrix is denoted

$$T_{k-1}(n-1:n, n-1:n) = \begin{pmatrix} b_{n-1} & \overline{a}_n \\ a_n & b_n \end{pmatrix},$$

and the corresponding eigenvalue in question, known as the **Wilkenson shift**, is given by

$$\mu_k = b_n + t - \operatorname{sgn}(t) \sqrt{|t|^2 + |a_n|^2} \quad \text{where} \quad t = (b_{n-1} - b_n)/2. \tag{4.35}$$

Implementation is given in Algorithm 4.12.

Accelerating the *QR* method: the non-symmetric real case

The non-symmetric real case is more delicate. In this case, any complex eigenvalues come in complex conjugate pairs (Fact 4.18). We may again begin by computing the Hessenberg decomposition, which is real because the initial matrix A is real. Neither of the approaches described above, however, is suitable:

- The approach of shifting each *QR* iteration by the element in the lower-right corner of T, as done in the general (non-Hermitian complex) case above, is not a good idea, because all elements during the *QR* iterations are real following this approach, though some of the eigenvalues we seek come in complex conjugate pairs. Thus, the (real) element in the lower-right corner of T at any given step will be *exactly* equidistant from the two complex eigenvalues, and shifting by it will not distinguish one eigenvector from the other. Stated another way, returning to the original description of the power method at the beginning of §4.4.5, we do not get the necessary separation of the absolute value of the eigenvalues in the shifted problem in order to ensure convergence of the corresponding Schur vectors when we use a real shift when the eigenvalues come in complex conjugate pairs.
- The approach of shifting each *QR* iteration by one of the eigenvalues of the 2×2 submatrix in the lower-right corner of T would in fact work. However, this approach unnecessarily converts a real problem into a significantly more expensive complex one. Furthermore, round-off errors accumulate following this approach such that the eigenvalues do not wind up being exact complex conjugates of each other.

An alternative approach is thus preferred. Instead of aiming to use a shifted *QR* method to iterate towards a complex upper triangular form, we instead use a **double-shift** method to iterate towards an eigenvalue-revealing *block* upper triangular form with 1×1 and 2×2 real blocks on the main diagonal, known as the **real Schur form** (see §4.4.3). At each step, we will take *two* shifted *QR* steps, one based each of the eigenvalues of the 2×2 submatrix in the lower-right corner of A. The challenge is to figure out a way to do this while keeping all arithmetic real. To accomplish this, we appeal to the following.

Fact 4.36 (The Implicit Q Theorem) *Consider two unitary similarity transformations of $T_{n\times n}$ such that $T = UGU^H = VHV^H$, where both G and H are upper Hessenberg and G is* **unreduced** *(that is, it has no zeros in its subdiagonal). If $\mathbf{v}^1 = \mathbf{u}^1$ (that is, if the first columns of V and U are equal), then it follows that $\mathbf{v}^i = \pm\mathbf{u}^i$ and $h_{i,i-1} = \pm G_{i,i-1}$ (stated loosely, the two decompositions are "essentially equivalent").*

Proof: Define a unitary $W = V^H U$ and note that $HW = WG$. It follows that, for $2 \le i \le n$,

$$H\mathbf{w}^{i-1} = \sum_{j=1}^{i-1} \mathbf{w}^j \cdot g_{j,i-1} + \mathbf{w}^i \cdot g_{i,i-1} \quad \Rightarrow \quad \mathbf{w}^i = \left(H\mathbf{w}^{i-1} - \sum_{j=1}^{i-1} \mathbf{w}^j \cdot g_{j,i-1}\right)/g_{i,i-1}.$$

Since $\mathbf{v}^1 = \mathbf{u}^1$ and U and V are unitary, it follows from $W = V^H U$ that $\mathbf{w}^1 = \mathbf{e}^1$ (that is, the first column of W is the first Cartesian unit vector). It thus follows from the above equation that W is upper triangular; since it is also unitary, it follows that $W = \text{diag}(1, \pm 1, \pm 1, \ldots)$. Since $\mathbf{w}^i = V^H\mathbf{u}^i$ and $g_{i,i-1} = (\mathbf{w}^i)^H H\mathbf{w}^{i-1}$, it also follows that $\mathbf{v}^i = \pm\mathbf{u}^i$ and $g_{i,i-1} = \pm h_{i,i-1}$. □

Leveraging the Implicit Q Theorem, we can now develop a method based on real arithmetic to perform two shifted *QR* steps based on the shifts a_1 and a_2 [given by the eigenvalues of $T(n-1:n, n-1:n)$, which are either real or a complex conjugate pair]. Noting (4.32), these two shifted *QR* steps may be written

$$(T - a_1 I) = Q_1 R_1; \qquad T_1 = R_1 Q_1 + a_1 I; \qquad (T_1 - a_2 I) = Q_2 R_2; \qquad T_2 = R_2 Q_2 + a_2 I. \tag{4.36a}$$

Pre- and post-multiplying the third relation by Q_1 and R_1 respectively, it is straightforward to show that

$$(Q_1 Q_2)(R_2 R_1) = (T - a_1 I)(T - a_2 I) = T^2 - (a_1 + a_2)T + (a_1 \cdot a_2)I \triangleq M \tag{4.36b}$$

(see Exercise 4.10). Note that $t \triangleq a_1 + a_2$ is real[16] and $d \triangleq a_1 \cdot a_2$ is real[17]; thus, M is real.

[16] By Fact 4.5, t is the sum of the eigenvalues of $T_{k-1}(n-1:n, n-1:n)$. This statement is generalized in Fact 4.44.

[17] By Fact 4.17, $d = |T_{k-1}(n-1:n, n-1:n)|$ is the product of the eigenvalues of $T_{k-1}(n-1:n, n-1:n)$.

Algorithm 4.13: Compute the eigenvalues of a real nonsymmetric matrix A.

View Test

```
function [lam] = EigReal(A)
% Compute the eigenvalues of a nonsymmetric real matrix A. After an initial Hessenberg
% decomposition, several double implicitly-shifted QR steps are taken, building up as much
% of the real Schur decomposition as necessary in order to determine the eigenvalues.
% (That is, for efficiency, we do not build the full real Schur decomposition, but rather
% work just on T_22 at each iteration).
A=Hessenberg(A); n=size(A,1); q=n; tol=1e-12;
while q>1          % Note: diagonal of T_22 block extends from {p,p} to {q,q} elements of T.
  for i=1:n-1; if abs(A(i+1,i))< tol*(abs(A(i,i))+abs(A(i+1,i+1))); A(i+1,i)=0; end; end
  q=1; for i=n-1:-1:1; if A(i+1,i)~=0, q=i+1; break, end, end, if q==1, continue, end
  p=1; for i=q-1:-1:1; if A(i+1,i)==0, p=i+1; break, end, end
  if q-p==1, a=(A(p,p)+A(q,q))/2; b=sqrt(4*A(p,q)*A(q,p)+(A(p,p)-A(q,q))^2)/2;
    A(p,p)=a+b; A(q,q)=a-b; A(q,p)=0;  % Put eigenvalues of A(q:p,q:p) on diagonal.
  continue, end
  t=A(q-1,q-1)+A(q,q);                                          % Trace of A(q-1:q,q-1:q)
  d=A(q-1,q-1)*A(q,q) - A(q-1,q)*A(q,q-1);                      % Determinant of A(q-1:q,q-1:q)
  x(1,1)=A(p,p)*A(p,p) + A(p,p+1)*A(p+1,p) - t*A(p,p) + d; % Compute first column of M.
  x(2,1)=A(p+1,p)*(A(p,p)+A(p+1,p+1)-t);
  x(3,1)=A(p+1,p)*A(p+2,p+1);
  [sig,w]=ReflectCompute(x);                                % Compute V_0
  A=Reflect(A,sig,w,p,p+2,p,q,'L');                         % Compute V_0^H T
  A=Reflect(A,sig,w,p,p+2,p,min(p+3,q),'R');                % Compute V_0^H T V_0
  for k=p:q-2;  km=min(k+3,q); kn=min(k+4,q);    % Transform the rest of T_22
    [sig,w]=ReflectCompute(A(k+1:km,k));         % via Implicit Q, returning it to
    A=Reflect(A,sig,w,k+1,km,k,q,'L');           % upper Hessenberg form.
    A=Reflect(A,sig,w,k+1,km,p,min(k+4,q),'R');
  end
end, lam=EigSort(diag(A));    % Extract eigenvalues from main diagonal of result and sort.
end % function EigReal
```

Recall from (4.33) that the shifted QR method boils down to $T_k = Q_k^H T_{k-1} Q_k$; that is, the upper Hessenberg matrix T_k is found simply by pre-multiplying T_{k-1} by a unitary matrix Q_k^H and post-multiplying the result by Q_k, where Q_k itself is found by orthogonalization of the columns of a shifted version of T_{k-1}. The idea now is similar, but instead of transforming with a single Q_k, we will effectively pre-multiply by $U^H \triangleq (Q_2^H Q_1^H)$ and postmultiply by $U \triangleq (Q_1 Q_2)$, which is why this method is said to be based on a **double shift**.

Unfortunately, computation of $M = T^2 - (a_1 + a_2)T + (a_1 \cdot a_2)I$ requires the computation of T^2, which is prohibitively expensive for large matrices. Thus, we cannot simply compute M, orthogonalize its columns (denoting the resulting matrix U), then compute $U^H T U$. Appealing to the Implicit Q Theorem, however, we may instead build up a new unitary matrix V (from a series of appropriately-constructed Householder transformation matrices embedded within identity matrices) to form a new unitary similarity transformation $V^H T V = H$. This new unitary similarity transformation will be "essentially equivalent" to the similarity transformation $U^H T U = G$ discussed above so long as the *first column* of U, denoted $\mathbf{u}^1$, matches the first column of V, denoted $\mathbf{v}^1$, and the rest of the transformation V is constructed such that H is upper Hessenberg (as is G). Thus, we perform here a **double implicitly shifted QR iteration**, achieving essentially the same effect as two shifted QR iterations but without ever explicitly performing the shifts $(T - a_1 I)$ or $(T_1 - a_2 I)$ [cf. Algorithms 4.11 and 4.12]. In addition to being substantially less computationally intensive in the present (non-symmetric real) case, for matrices whose eigenvalues span a large range of magnitudes, such an implicit shifting approach leads to a significant improvement in overall accuracy. [Thus, see also Exercise 4.11 for application of this idea to the Hermitian/symmetric case discussed previously.]

To proceed in the present (non-symmetric real) case, it is noted that $\mathbf{m}^1 = \begin{pmatrix} x & y & z & 0 & \dots & 0 \end{pmatrix}$ where

$$x = t_{11}^2 + t_{12} t_{21} - t t_{11} + d, \qquad y = t_{21}(t_{11} + t_{22} - t), \qquad z = t_{21} t_{32}.$$

We will construct V via a series of Householder transformation matrices, $V = V_0 V_1 \cdots V_{n-2}$, where

$$V_k = \text{diag}[I_{k\times k}, H^k_{3\times 3}, I_{(n-k-3)\times(n-k-3)}]; \quad \text{for example,} \quad V_1 = \begin{pmatrix} 1 & & & & & 0 \\ & x & x & x & & \\ & x & x & x & & \\ & x & x & x & & \\ & & & & 1 & \\ 0 & & & & & 1 \end{pmatrix}.$$

Note that all of the V_k matrices are $\mathbf{e}^1$ in the first column except V_0; thus, V and V_0 have identical first columns. Therefore, all we need to do is to determine $H^0_{3\times 3}$ as the upper-left block of the first Householder transformation matrix in the QR decomposition of M (see Algorithm 4.6); that is, we construct $H^0_{3\times 3}$, from which V_0 is formed, by designing a Householder reflection matrix which rotates the first three components of $\mathbf{m}^1$ into the direction $\mathbf{e}^1$. We then construct the remaining V_k to return the matrix H to upper Hessenberg form in the construction $V^H T V = H$, effectively "chasing out" the nonzero elements introduced into the second and third subdiagonals of the transformed matrix by V_0. Graphically, this process may be illustrated as follows, denoting by $*$ those elements changed by the most recent pre- and post-multiplications:

$$V_0^H T V_0 = \begin{pmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & x & x & x \\ 0 & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & x \end{pmatrix}, \quad V_1^H V_0^H T V_0 V_1 = \begin{pmatrix} x & * & * & * & x & x \\ * & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & x & x \\ 0 & 0 & 0 & 0 & x & x \end{pmatrix},$$

$$V_2^H V_1^H V_0^H T V_0 V_1 V_2 = \begin{pmatrix} x & x & * & * & * & x \\ x & x & * & * & * & x \\ 0 & * & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & x \end{pmatrix}, \quad V_3^H V_2^H V_1^H V_0^H T V_0 V_1 V_2 V_3 = \begin{pmatrix} x & x & x & * & * & * \\ x & x & x & * & * & * \\ 0 & x & x & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \end{pmatrix},$$

$$V^H T V = \begin{pmatrix} x & x & x & x & * & * \\ x & x & x & x & * & * \\ 0 & x & x & x & * & * \\ 0 & 0 & x & x & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & * & * \end{pmatrix},$$

where $V = V_0 V_1 \cdots V_{n-2}$. Implementation is relatively straightforward, as shown in Algorithm 4.13.

To conclude this section on the numerical computation of matrix eigenvalues, two convenient **wrapper** routines, `Eig.m` and `Schur.m`, are given in Algorithms 4.14 and 4.15.

Algorithm 4.14: Convenient wrapper routine for computing the eigen decomposition.

View
Test

```
function [lam,S]=Eig(S,type)
if nargin==1, type='r'; end, switch type(1)
  case 'g', lam=EigGeneral(S); case 'h', lam=EigHermitian(S); case 'r', lam=EigReal(S);
end, if nargout==2, [S]=ShiftedInversePower(S,lam); end
end % function Eig
```

Algorithm 4.15: Convenient wrapper routine for computing the Schur decomposition.

View
Test

```
function [U,T]=Schur(U,type)
if nargin==1, type='r'; end, switch type(1)
  case 'g', lam=EigGeneral(U); case 'h', lam=EigHermitian(U); case 'r', lam=EigReal(U);
end; [U,T]=ShiftedInversePower(U,lam);
end % function Schur
```

4.4.6 The Jordan decomposition†

As discussed in §4.4.4, a square matrix with a complete set of eigenvectors is similar to a diagonal matrix; to be precise, $A = S\Lambda S^{-1}$, where S is a matrix with the eigenvectors of A as columns, and Λ is a diagonal matrix with the corresponding eigenvalues of A on the main diagonal. The Jordan decomposition $A = MJM^{-1}$ represents, in a way, the closest one can get to a diagonalizing similarity transformation when the square matrix A does *not* have a complete set of eigenvectors (that is, when the matrix A is **defective**). The columns of the transformation matrix M involved in this decomposition include all of the linearly independent eigenvectors of A together with an appropriate number of so-called **generalized eigenvectors**, as defined below.

To illustrate, consider an $n \times n$ matrix A with p distinct eigenvalues[18] $\lambda_1, \ldots, \lambda_p$ found, e.g., with the QR method discussed previously, where the eigenvalue λ_i is assumed to have **multiplicity**[19] (a.k.a. **algebraic multiplicity**) m_i. The Jordan decomposition may be built as follows[20]:

1) Initialize the index $i = 1$.
2) Calculate the number of linearly independent eigenvectors[21], $v_i = n - \text{rank}(A - \lambda_i I)$, corresponding to λ_i, then solve[22] $(A - \lambda_i I)\mathbf{s}^{ij} = 0$ for the v_i independent eigenvectors $\mathbf{s}^{ij}$ for $j = 1, \ldots, v_i$.
3) Calculate the number of necessary generalized eigenvectors[23], $w_i = m_i - v_i$, corresponding to λ_i.
4) Initialize $\mathbf{g}^{i,j,0} = \mathbf{s}^{ij}$ for $j = 1, \ldots, v_i$. Also, keep track of the number of generalized eigenvectors already determined that are related to the (ij)'th eigenvector $\mathbf{s}^{ij}$ by initializing $p_{ij} = 0$, for $j = 1, \ldots, v_i$.
5) Initialize the index $j = 1$.
6) Initialize the index $k = 1$.
7) If $w_i - (p_{i1} + p_{i2} + \ldots + p_{iv_i}) = 0$, we have all the generalized eigenvectors related to λ_i. Go to step 9.
8) If $\mathbf{g}^{i,j,(k-1)} \in \text{im}(A - \lambda_i I)$, then solve $(A - \lambda_i I)\mathbf{g}^{i,j,k} = \mathbf{g}^{i,j,(k-1)}$ for the generalized eigenvector $\mathbf{g}^{i,j,k}$, increment k and p_{ij}, and repeat from step 7, else increment j and repeat from step 6.
9) If $i < p$, increment i and repeat from step 2 until finished.

The resulting transformation matrix M is then given by

$$M = \begin{bmatrix} M^{11} & M^{12} & \ldots & M^{1v_1} & M^{21} & M^{22} & \ldots & M^{2v_2} & \ldots\ldots & M^{p1} & M^{p2} & \ldots & M^{pv_p} \end{bmatrix} \tag{4.37a}$$

where M^{ij} contains the eigenvector $\mathbf{s}^{ij}$ and all the generalized eigenvectors $\mathbf{g}^{i,j,k}$ associated with it,

$$M^{ij} = \begin{bmatrix} | & | & | & & | \\ \mathbf{s}^{ij} & \mathbf{g}^{ij1} & \mathbf{g}^{ij2} & \ldots & \mathbf{g}^{ijp_{ij}} \\ | & | & | & & | \end{bmatrix}, \tag{4.37b}$$

and the corresponding block diagonal **Jordan form** J is given by

$$J = \text{diag}[J^{11}, J^{12}, \ldots, J^{1v_1}, J^{21}, J^{22}, \ldots, J^{2v_2}, \ldots\ldots, J^{p1}, J^{p2}, \ldots, J^{pv_p}], \tag{4.37c}$$

where the **Jordan block** J^{ij} is a $(p_{ij}+1) \times (p_{ij}+1)$ upper bidiagonal Toeplitz matrix of the form

$$J^{ij} = \begin{pmatrix} \lambda_1 & 1 & & 0 \\ & \ddots & \ddots & \\ & & \lambda_1 & 1 \\ 0 & & & \lambda_1 \end{pmatrix}. \tag{4.37d}$$

[18] Note that the n eigenvalues λ_i of the matrix A satisfy the characteristic polynomial $|A - \lambda_i I| = 0$, as described previously.

[19] Note also that, by the Fundamental Theorem of Algebra (Fact 4.4), $m_1 + m_2 + \ldots m_p = n$.

[20] Proof that this algorithm succeeds in producing exactly the w_i additional linearly independent vectors needed for each i is given in Horn & Johnson 1985.

[21] Note that v_i is the dimension of the nullspace of $(A - \lambda_i I)$, and is referred to as the **geometric multiplicity** of λ_i.

[22] The eigenvectors $\mathbf{s}^{ij}$ may be determined by parameterizing all solutions to $(A - \lambda_i I)\mathbf{s}^{ij} = 0$ using the techniques described in §2.6.

[23] If the **algebraic multiplicity** m_i is greater than the **geometric multiplicity** v_i of a given eigenvalue λ_i, then $w_i = m_i - v_i$ generalized eigenvectors will be required.

Multiplying out $AM = MJ$, applying the definitions in (4.37), and examining each column of the resulting equation, we arrive at two types of relations, $A\mathbf{s}^{ij} = \lambda_i \mathbf{s}^{ij}$ and $A\mathbf{g}^{i,j,k} = \lambda_i \mathbf{g}^{i,j,k} + \mathbf{g}^{i,j,(k-1)}$ where $\mathbf{g}^{i,j,0} = \mathbf{s}^{ij}$, which is consistent with the algorithm above defining the eigenvectors $\mathbf{s}^{ij}$ and generalized eigenvectors $\mathbf{g}^{i,j,k}$. Note that, if the matrix A happens to have n linearly independent eigenvectors, then the Jordan blocks are all 1×1, and the Jordan decomposition reduces immediately to the eigen decomposition[24].

A matrix is called **nonderogatory** if it has only one Jordan block associated with each eigenvalue [that is, if $v_i = n - \text{rank}(A - \lambda_i I) = 1$ for every eigenvalue λ_i of A]; otherwise, it is called **derogatory**.

For the purpose of the numerical solution of large-scale problems in science, engineering, and elsewhere, computing the Jordan decomposition is usually a bad idea; the role of this decomposition is thus deemphasized in this text. The central problem with this decomposition is that, by its definition, the Jordan form does not depend smoothly on A. For example, consider a 2×2 matrix A with eigenvalues λ_1 and $\lambda_1 + \varepsilon$ such that

$$A = \begin{pmatrix} \lambda_1 & 1 \\ 0 & \lambda_1 + \varepsilon \end{pmatrix} \quad \Rightarrow \quad \mathbf{s}^1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{s}^2 = \begin{pmatrix} 1 \\ \varepsilon \end{pmatrix}. \tag{4.38}$$

For $\varepsilon = 0$, the matrix A is already in Jordan form, and the linear independence of the two eigenvectors is lost. In this case, the Jordan decomposition is simply $A = MJM^{-1}$ with

$$J = A = \begin{pmatrix} \lambda_1 & 1 \\ 0 & \lambda_1 \end{pmatrix} \quad \text{and} \quad M = I.$$

However, for an arbitrarily small change of ε, the Jordan decomposition $A = MJM^{-1}$ sudddenly switches to the eigen decomposition such that

$$J = \Lambda = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_1 + \varepsilon \end{pmatrix} \quad \text{and} \quad M = S = \begin{pmatrix} 1 & 1 \\ 0 & \varepsilon \end{pmatrix}.$$

Because of this **ill-posedness** (that is, a lack of smooth dependence on the data in the problem formulation) in the definition of the Jordan decomposition, the Schur decomposition and/or the singular value decomposition is usually be preferred over the Jordan decomposition for most large-scale numerical computations in cases for which A is, or might be, nearly defective.

Defective matrices appear often in the study of dynamic systems. Despite the ill-posedness of the definition of the Jorden decomposition problem, as described above, deriving such matrices in Jordan form (or transforming them into Jordan form) is often instructive for the purpose of analysis. For example, consider the equation for the simple 2D rotation of a rigid body:

$$J\frac{d^2\theta}{dt^2} = \tau. \quad \text{Defining} \quad \mathbf{x} = \begin{pmatrix} \theta \\ d\theta/dt \end{pmatrix} \quad \text{yields} \quad \frac{d\mathbf{x}}{dt} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \mathbf{x} + \begin{pmatrix} 0 \\ 1/J \end{pmatrix} \tau.$$

In the first-order (a.k.a. **state-space**) form shown at right, the system matrix in the governing equation is already in Jordan form (in this case, with $\lambda_1 = 0$). The behavior of such dynamic systems is discussed further in §**??** and §20.2.

[24] That is, $J^{ij} = \lambda_i$ and $M^{ij} = \mathbf{s}^{ij}$, and thus $J = \Lambda$ and $M = S$.

4.4.7 The singular value decomposition (SVD)

As mentioned in §4.4.6, if the set of eigenvectors $\mathbf{s}^i$ of a given matrix A are not linearly independent, then they cannot be arranged into a nonsingular matrix S, and the matrix A is called defective. However, it is a remarkable fact that every $m \times n$ matrix A has a **singular value decomposition** (SVD) given by $A = U\Sigma V^H$, where $U = U_{m\times m}$ and $V = V_{n\times n}$ are unitary ($UU^H = U^H U = I_{m\times m}$ and $VV^H = V^H V = I_{n\times n}$) and $\Sigma = \Sigma_{m\times n}$ is diagonal with real, non-negative elements σ_i on the main diagonal, arranged in descending order. The σ_i are referred to as the **singular values** of A, and the columns of U and V are, respectively, the **left** and **right singular vectors** of A. *The number of nonzero singular values, denoted by r, is the rank of the matrix A*; in fact, numerically, computing the SVD is the most reliable method available to compute the rank of a matrix.

The maximum singular value of A, σ_1, is often denoted $\sigma_{\max}(A)$, whereas the minimum singular value of A is often denoted $\sigma_{\min}(A)$. Note that, if A is Hermitian positive semidefinite, then the SVD reduces to the eigen decomposition with $U = V = S$ and the $\sigma_i = \lambda_i$.

As with the QR decomposition when $m > n$, the SVD has both complete and reduced forms. Taking r as the number of nonzero singular values of A (i.e., the rank of A), we may write the SVD in block matrix form:

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^H_{n\times n} = \begin{bmatrix}\underline{U}_{m\times r} & \overline{U}_{m\times(m-r)}\end{bmatrix}\begin{bmatrix}\underline{\Sigma}_{r\times r} & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}\underline{V}_{n\times r} & \overline{V}_{n\times(n-r)}\end{bmatrix}^H = \underline{U}_{m\times r}\underline{\Sigma}_{r\times r}(\underline{V}_{n\times r})^H. \tag{4.39}$$

Note in particular that $\underline{U}^H\underline{U} = I_{r\times r}$ and $\underline{V}^H\underline{V} = I_{r\times r}$, though $\underline{U}\underline{U}^H \neq I_{m\times m}$ and $\underline{V}\underline{V}^H \neq I_{n\times n}$. The SVD is a useful generalization of the eigen decomposition introduced in §4.4.4 that is appropriate for defective and nonsquare matrices. Noting that U and V are unitary and applying Fact 2.6 to the block matrix form of the SVD given above, it is seen that

- the columns $\underline{\mathbf{u}}^i$ of the matrix $\underline{U}$ form an orthogonal basis of the column space of A,
- the columns $\overline{\mathbf{u}}^i$ of the matrix $\overline{U}$ form an orthogonal basis of the left nullspace of A,
- the columns $\underline{\mathbf{v}}^i$ of the matrix $\underline{V}$ form an orthogonal basis of the row space of A, and
- the columns $\overline{\mathbf{v}}^i$ of the matrix $\overline{V}$ form an orthogonal basis of the nullspace of A.

Note that, since $A = U\Sigma V^H$, it follows that $(A^HA) = V(\Sigma^H\Sigma)V^H$ and $(AA^H) = U(\Sigma\Sigma^H)U^H$. Recall from Fact 4.20 that an eigen decomposition of a Hermitian matrix [such as (A^HA) or (AA^H)] always exists, and from §4.4.5 that there is a very efficient numerical algorithm for computing the eigen decomposition of Hermitian matrices (see Algorithm 4.12). Thus, this algorithm may be leveraged to find the singular value decomposition of any $m \times n$ matrix A in a straightforward manner, as shown below. Note that the constructions given below also establish the existence of the SVD itself. Note also the following:

Fact 4.37 $\text{rank}(A) = \text{rank}(AA^H) = \text{rank}(A^HA)$. *Further, the nonzero eigenvalues of* (AA^H) *and* (A^HA) *are the square of the nonzero singular values of A.*

Proof: As noted previously, $(AA^H) = U(\Sigma\Sigma^H)U^H$ and $(A^HA) = V(\Sigma^H\Sigma)V^H$. Note that both $(\Sigma\Sigma^H)$ and $(\Sigma^H\Sigma)$ are diagonal, and thus these equations are in fact SVDs of (AA^H) and (A^HA). Note also that the nonzero elements on the diagonals of $(\Sigma\Sigma^H)$ and $(\Sigma^H\Sigma)$ [that is, the nonzero singular values of (AA^H) and (A^HA)] are identical and equal to the square of the nonzero singular values of A. □

Fact 4.38 *The induced 2-norm is given by the* **maximum singular value** *[i.e.,* $\|A\|_{i2} = \sigma_{max}(A)$*].*

Proof: Square both sides of the first formula in (1.20) and take $p = 2$, then apply Facts 4.26 and 4.37:

$$\|A\|^2_{i2} = \max_{\mathbf{x}\neq 0}\frac{\|A\mathbf{x}\|^2}{\|\mathbf{x}\|^2} = \max_{\mathbf{x}\neq 0}\frac{\mathbf{x}^H(A^HA)\mathbf{x}}{\mathbf{x}^H\mathbf{x}} = \lambda_{\max}(A^HA) = \lambda_{\max}(AA^H) = \sigma^2_{\max}(A). \qquad \square$$

Fact 4.39 *The singular values of A^{-1}, if it exists (that is, if $r = m = n$), are the reciprocal of the singular values of A.*

Proof: If $A = U\Sigma V^H$ is an SVD of A, then (by Fact 1.7) $A^{-1} = V^{-H}\Sigma^{-1}U^{-1}$, which itself is just an SVD of A^{-1}. The singular values of A^{-1} (that is, the diagonal elements of Σ^{-1}) are thus the reciprocal of the singular values of A (that is, the diagonal elements of Σ). □

Fact 4.40 *The 2-norm condition number (see §2.5) is given by $\kappa(A) = \|A\|_{i2}\|A^{-1}\|_{i2} = \sigma_{max}(A)/\sigma_{min}(A)$. Thus, $\kappa(A) \geq 1$ and $\kappa(A) \to \infty$ as the matrix A approaches a singular matrix [that is, as $\sigma_{min}(A) \to 0$].*

Proof: Follows directly from Facts 4.38 and 4.39. □

The SVD is also useful for determining the most energetic "modes" present in a large dataset, as discussed in §7.6.1.

Note that the SVD is not unique; for example, if $A = U_a\Sigma_a V_a^H$ is an SVD, then $A = U_b\Sigma_b V_b^H$ is also an SVD, where $U_b = -U_a$ and $V_b = -V_a$.

There are three natural constructions of the SVD, as presented below. Numerically, the third construction is far superior to the other two, as it doesn't involve multiplying A times itself, which leads to a squaring of the condition number and thus a sometimes significant loss of accuracy when using finite-precision arithmetic. However, the other two constructions are useful to be aware of from the perspective of understanding the SVD itself; we thus present all three constructions here.

(i) Construction of the SVD from the eigenvalues and eigenvectors of (A^HA)

The SVD may be constructed as follows:

- Step 1: Determine the eigenvalues of the $n \times n$ matrix A^HA. By Fact 4.22, A^HA is Hermitian and its eigenvalues are real and non-negative. Let λ_i denote these eigenvalues arranged in descending order. Determine the number r of these eigenvalues that are nonzero. Call the square root of each of these eigenvalues $\sigma_i = \sqrt{\lambda_i}$, noting that $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$, and place these nonzero σ_i (in order) in the first r elements on the main diagonal of Σ, setting the other elements of Σ to zero.
- Step 2: Determine the eigenvectors $\mathbf{v}^i$ of the matrix A^HA, and arrange them in columns in the same order as their corresponding eigenvalues in Σ to form the $n \times n$ matrix V. By Fact 4.22, V is unitary.
- Step 3: Select $\mathbf{u}^i = (1/\sigma_i)A\mathbf{v}^i$ for $i = 1, 2, \ldots r$, and define $\underline{U}$ as the matrix with these vectors as columns. By construction, we have $AV = \underline{U}\begin{bmatrix}\underline{\Sigma} & 0\end{bmatrix}$, and thus $A = \underline{U}\begin{bmatrix}\underline{\Sigma} & 0\end{bmatrix}V^H$. We have therefore constructed most of the singular value decomposition. All that remains is to find $\overline{U}$, which may be found simply by taking the (complete) QR decomposition of $\underline{U}$ and setting $\overline{U} = \overline{Q}$ in this decomposition.

Note that the $\mathbf{u}^i$ given by the above algorithm work out simply to be the orthogonal eigenvectors of AA^H, ordered appropriately such that the desired decomposition of A is satisfied by construction.

(ii) Construction of the SVD from the eigenvalues and eigenvectors of (AA^H)

An alternative construction of the SVD may be written by taking the eigenvalues of the matrix AA^H in Step 1 of construction (i) above, defining the columns of U as the corresponding eigenvectors of this matrix in Step 2, and then defining the r columns of $\underline{V}$ as $\mathbf{v}^i = (1/\sigma_i)A^H\mathbf{u}^i$, and determining the remaining columns of V via the QR decomposition of $\underline{V}$ in Step 3.

If A is rectangular, the dimensions of the square matrices A^HA and AA^H are different, and a choice between construction (i) and construction (ii) is sometimes made based on which works with the smaller of these two matrices during Steps 1 and 2. However, as mentioned previously, construction (iii) presented below is the numerically preferred algorithm for constructing the SVD.

(iii) Construction of the SVD from a bidiagonalization of A directly[†]

Following the same line of reasoning as in §4.4.1, but now pre- and post-multiplying by *different* Householder reflector matrices, it is entirely straightforward to construct a bidiagonalization $A = UB_0V^H$ where U and V are unitary and B_0 is upper bidiagonal. Indicated graphically,

$$U_1^H A = \begin{pmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix}, \quad U_1^H A V_1 = \begin{pmatrix} x & * & 0 & 0 \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \\ 0 & * & * & * \end{pmatrix}, \quad U_2^H U_1^H A V_1 = \begin{pmatrix} x & x & 0 & 0 \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{pmatrix},$$

$$U_2^H U_1^H A V_1 V_2 = \begin{pmatrix} x & x & 0 & 0 \\ 0 & x & * & 0 \\ 0 & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & * & * \end{pmatrix}, \quad U_3^H U_2^H U_1^H A V_1 V_2 = \begin{pmatrix} x & x & 0 & 0 \\ 0 & x & x & 0 \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & * \end{pmatrix}, \quad B_0 = U^H A V = \begin{pmatrix} x & x & 0 & 0 \\ 0 & x & x & 0 \\ 0 & 0 & x & x \\ 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

where $U = U_1U_2\cdots U_n$ and $V = V_1V_2\cdots V_{n-2}$. Implementation is straightforward (see Algorithm 4.16).

Starting from this bidiagonalization, we now consider the application of the shifted QR method to the tridiagonal Hermitian matrix $T_k = B_k^H B_k$. At each step of this iterative method, we may apply a Wilkenson shift based on the lower-right corner of T_k, as done in Algorithm 4.12. However, instead of explicitly shifting the matrix T_k at each step, we will apply an **implicitly shifted QR iteration**, as done in non-symmetric real case discussed previously and illustrated in Algorithm 4.13 (see also Exercise 4.11). Further, instead of working on T_k, we will apply the rotations comprising V_k that we would have applied to T_k (via both pre- and postmultiplication, applied using the implicitly shifted QR method) directly to the factor B_k (via postmultiplication, again using the implicitly shifted QR method), while applying compensating unitary rotations U_k to the factor B_k (via premultiplication) in order to keep it upper bidiagonal at each step. That is,

$$T_{k+1} = V_k^H T_k V_k = V_k^H B_k^H B_k V_k = V_k^H B_k^H U_k U_k^H B_k V_k = B_{k+1}^H B_{k+1} \quad \text{where} \quad B_{k+1} = U_k^H B_k V_k.$$

Note that B_{k+1} is upper bidiagonal (by construction), and thus T_{k+1} is tridiagonal.

As discussed in detail in the second paragraph of point (iii) on page 102, B may at each step be partitioned into a block upper triangular form with diagonal blocks B_{11}, B_{22}, and B_{33}, where B_{22} is unreduced. The algorithm described above is thus applied[25] to the B_{22} submatrix, not to all of B.

Two more clean up items must be addressed. The first is that, if there is a zero in the lower-right element of B_{22}, the entire last column of B_{22} may be zeroed by postmultiplication by an appropriate sequence of Givens rotation matrices. Indicated graphically:

$$\begin{pmatrix} x & x & 0 & 0 \\ 0 & x & x & 0 \\ 0 & 0 & x & x \\ 0 & 0 & 0 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} x & x & 0 & 0 \\ 0 & x & * & * \\ 0 & 0 & * & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} x & * & 0 & * \\ 0 & * & x & 0 \\ 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} * & x & 0 & 0 \\ 0 & x & x & 0 \\ 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Second, if there is a zero in any other of the diagonal elements of B_{22}, that entire row of B_{22} may be zeroed by premultiplication by an appropriate sequence of Givens rotation matrices. Indicated graphically:

$$\begin{pmatrix} 0 & x & 0 & 0 \\ 0 & x & x & 0 \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & * & 0 \\ 0 & * & * & 0 \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 0 & * \\ 0 & x & x & 0 \\ 0 & 0 & * & * \\ 0 & 0 & 0 & x \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & x & x & 0 \\ 0 & 0 & x & x \\ 0 & 0 & 0 & * \end{pmatrix}$$

Implementation of all of the steps described above is given in Algorithm 4.17.

[25] In order to apply the Implicit Q Theorem (Fact 4.36), the submatrix of T to which the theorem is being applied must be unreduced (that is, nonzero in its lower subdiagonal); if it isn't, the proof of this theorem breaks down, and in fact the algorithm implementing it fails. Thus, this partitioning step is essential to the success of the algorithm.

Algorithm 4.16: Compute a bidiagonalization of an $m \times n$ matrix (cf. Algorithm 4.2).

View
Test

```
function [A,U,V] = Bidiagonalization(A,m,n)
% Pre and post-multiply an mxn matrix A by a sequence of Householder reflections
% to reduce it to upper bidiagonal form B, thus computing the decomposition A=U B V'H.
V=eye(n,n); U=eye(m,m);
for i=1:min(m,n)
  if i<m, [sig,w] = ReflectCompute(A(i:m,i));
          A=Reflect(A,sig,w,i,   m,i,n,'L'); U=Reflect(U,sig,w,i,   m,1,m,'R'); end
  if i<n, [sig,w] = ReflectCompute(A(i,i+1:n)');
          A=Reflect(A,sig,w,i+1,n,i,m,'R'); V=Reflect(V,sig,w,i+1,n,2,n,'R'); end
end
end % function Bidiagonalization
```

Algorithm 4.17: Compute the reduced SVD $A = \underline{U}_{m\times r}\underline{\Sigma}_{r\times r}(\underline{V}_{n\times r})^H$.

View
Test

```
function [U,S,V,r] = SVD(A,type)
% Compute the rank and reduced (by default) or complete (if type='complete') SVD of A.
[m,n]=size(A); if m<n, [V,S,U,r]=SVD(A'); else
tol=1e-15; p=1; q=n; [A,U,V]=Bidiagonalization(A,m,n);
while q>1            % Note: diagonal of B_22 block extends from {p,p} to {q,q} elements.
  if abs(A(q,q))<tol;                                % If necessary, zero out last column...
    for i=q-1:-1:1;
     [c,s]=RotateCompute(conj(A(i,i)),conj(A(i,q)));
     [A]=Rotate(A,c,s,i,q,max(i-1,1),i,'R'); [V]=Rotate(V,c,s,i,q,1,n,'R');
    end
  end
  for k=q-1:-1:p, if abs(A(k,k))<tol, A(k,k)=0;   % ...or zero out an intermediate row.
    for j=k+1:q,
     [c,s]=RotateCompute(A(j,j),A(k,j));
     [A]=Rotate(A,c,s,j,k,j,min(j+1,q),'L'); [U]=Rotate(U,c,s,j,k,1,m,'R');
    end
  end, end,                                                  % Compute p and q
  for i=1:n-1; if abs(A(i,i+1))< tol*(abs(A(i,i))+abs(A(i+1,i+1))); A(i,i+1)=0; end; end
  q=1; for i=n-1:-1:1; if A(i,i+1)~=0, q=i+1; break, end, end, if q==1, continue, end
  p=1; for i=q-1:-1:1; if A(i,i+1)==0, p=i+1; break, end, end
  dp=A(p,p); dm=A(q-1,q-1); dq=A(q,q); fp=A(p,p+1); fm=A(q-1,q);     % Wilkenson shift of
  bq=real(dq)^2+imag(dq)^2+real(fm)^2+imag(fm)^2; aq=(dm)*conj(fm); % T=B_{22}^H B_{22}.
  if q>2, fl=A(q-2,q-1); bm=real(dm)^2+imag(dm)^2+real(fl)^2+imag(fl)^2;
  else,                  bm=real(dm)^2+imag(dm)^2; end
  t=real(bm-bq)/2.; mu=bq+t-sign(t)*sqrt(t*t+conj(aq)*aq);      % Set up first rotation
  f=real(dp)^2+imag(dp)^2-mu; g=dp*conj(fp);                    % using this shift.
  for i=p:q-1                                    % Then apply implicit Q to return the
    [c,s]=RotateCompute(f,g);                    % B_22 matrix to upperbidiagonal form.
    [A]=Rotate(A,c,s,i,i+1,max(p,i-1),i+1,'R'); [V]=Rotate(V,c,s,i,i+1,1,n,'R');
    f=A(i,i); g=A(i+1,i);
    [c,s]=RotateCompute(f,g);
    [A]=Rotate(A,c,s,i,i+1,i,min(i+2,q),'L');    [U]=Rotate(U,c,s,i,i+1,1,m,'R');
    if i<q-1, f=conj(A(i,i+1)); g=conj(A(i,i+2)); end
  end
end
s=diag(A); [scratch,index]=sort(-abs(s)); j=sqrt(-1);
if isreal(s), for i=1:n, if s(i)<0, V(:,i)=-V(:,i); end, end      % Rotate to make s(i)>0.
else,         for i=1:n, U(:,i)=U(:,i)*exp(j*atan2(imag(s(i)),real(s(i)))); end, end
s=abs(s); for r=n:-1:1, if s(index(r))>1e-7, break, end, end      % Compute rank.
if isreal(A), U=real(U); V=real(V); end
if (nargin==2 & type=='complete'), S=diag(s);
else, S=diag(s(index(1:r))); U=U(:,index(1:r)); V=V(:,index(1:r)); end % Arrange S,U,V.
end % function SVD
```

4.4.8 Decompositions related to Gaussian elimination: LDM^H, LDL^H, and Cholesky

In the course of solving $A\mathbf{x} = \mathbf{b}$ via Gaussian elimination in §2.2, a few useful matrix decompositions were already encountered. In §2.2.1, we presented a step-by-step procedure to construct an LU decomposition, $A = LU$, where L is unit lower triangular and U is upper triangular, assuming that A is nonsingular and row swaps are not required in the Gaussian elimination procedure. If row swaps were required (§2.2.2), we accounted for them with the permutaion matrix P such that $A = PLU$. If column swaps were also incorporated (§2.2.3), we accounted for them with the permutation matrix Q such that $A = PLUQ^T$.

We now discuss a few different ways to reëxpress such decompositions, assuming A is nonsingular and focusing for the remainder of this section (for simplicity) on the case with $P = Q = I$.

Starting from the LU decomposition $A = LU$ (if it exists; see, for example, Facts 2.2 & 4.31) of a square nonsingular matrix A, and defining $D = \text{diag}(u_{11}, u_{22}, \ldots, u_{nn})$, noting that D is nonsingular with $D^{-1} = \text{diag}(u_{11}^{-1}, u_{22}^{-1}, \ldots, u_{nn}^{-1})$, we may write $A = LDM^H$ where $M^H = D^{-1}U$ is unit upper triangular.

Fact 4.41 *If A is nonsingular and its LU decomposition exists (with L unit lower triangular and U upper triangular), then the LU decomposition is unique.*

Proof: If A is nonsingular, then $|A| \neq 0$, and thus, by Property 5 of the determinant, $|L| \neq 0$ and $|U| \neq 0$, so L and U are nonsingular as well. Assume L_1U_1 and L_2U_2 are two LU decompositions of A. Then $L_1U_1 = L_2U_2 \;\Rightarrow\; L_2^{-1}L_1 = U_2U_1^{-1}$. As $L_2^{-1}L_1$ is unit lower triangular and $U_2U_1^{-1}$ is upper triangular and these two expressions are equal, they must equal the identity matrix. Thus $L_2 = L_1$ and $U_2 = U_1$. □

Fact 4.42 *If A is nonsingular and Hermitian and $A = LDM^H$ where L and M are unit lower triangular and D is diagonal, then $L = M$ (that is, $A = LDL^H$) and D is real.*

Proof: If A is Hermitian ($A^H = A$) and decomposed such that $A = LDM^H$, it follows that $MD^HL^H = LDM^H$. Note that both MD^HL^H and LDM^H are LDM^H decompositions of A. Thus, by the uniqueness of the LDM^H decomposition [which follows directly from the uniqueness of the LU decomposition (Fact 4.41) from which the LDM^H decomposition is derived], $M = L$ and $D^H = D$ (that is, D is real). □

If $A > 0$, then by Facts 4.42 and 4.30 we may write $A = LDL^H$ where D is diagonal with positive diagonal elements. Thus, given an LU decomposition of A, defining $D^{1/2} = \text{diag}(u_{11}^{1/2}, u_{22}^{1/2}, \ldots, u_{nn}^{1/2})$ and $G = LD^{1/2}$, we obtain the **Cholesky decomposition** $A = GG^H$, where G is lower triangular. Alternatively, given a $\underline{QR}$ decomposition of B, $B^HB = (\underline{QR})^H\underline{QR} = \underline{R}^H\underline{R}$, and thus the Cholesky decomposition of $A = (B^HB)$ is given by $A = GG^H$ with $G = \underline{R}^H$.

Direct determination of the Cholesky decomposition

If $A > 0$, rather than performing Gaussian elimination on A and then backing out $G = LD^{1/2}$ afterwards, or determining the $\underline{QR}$ decomposition of B [where $A = (B^HB)$] and then taking $G = \underline{R}^H$, we may instead leverage the Hermitian positive definite structure of A to determine the Cholesky decomposition directly.

Consider again the decomposition of the Hermitian positive definite matrix A given by (4.24)

$$A = \begin{bmatrix} \alpha_1 & \mathbf{v}_1^H \\ \mathbf{v}_1 & B_1 \end{bmatrix} = \begin{bmatrix} \beta_1 & 0 \\ \mathbf{v}_1/\beta_1 & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & B_1 - \mathbf{v}_1\mathbf{v}_1^H/\alpha_1 \end{bmatrix} \begin{bmatrix} \beta_1 & \mathbf{v}_1^H/\beta_1 \\ 0 & I \end{bmatrix} = \tilde{G}_1\tilde{A}_1\tilde{G}_1^H,$$

recalling that $\alpha_1 > 0$, $B_1 > 0$, $\beta_1 = \sqrt{\alpha_1}$, and $A_2 = B_1 - \mathbf{v}_1\mathbf{v}_1^H/\alpha_1 > 0$.

Since $A_2 > 0$, the above decomposition may be used to reduce A_2 in a similar fashion, that is, $A_2 = \tilde{G}_2\tilde{A}_2\tilde{G}_2^H$, where $\tilde{A}_2 = \text{diag}[1, A_3]$. This process may be repeated, ultimately providing the decomposition

$$A = G_1G_2\cdots G_nG_n^H\cdots G_2^HG_1 = GG^H \quad \text{where} \quad G_k = \begin{bmatrix} I_{(k-1)\times(k-1)} & 0 \\ 0 & \tilde{G}_k \end{bmatrix} \quad \text{and} \quad G = G_1G_2\cdots G_n.$$

As they are each built from identity matrices with one column changed, the product of the G_i matrices collapses into a single matrix, as implemented in Algorithm 4.18, costing only $\sim (n^3/3)$ flops.

Algorithm 4.18: Compute the full Cholesky decomposition $A = GG^H$ of some $A > 0$.

View
Test

```
function [A] = Cholesky(A,n)
% Compute the full Cholesky decomposition A=G*G'H of some A>0.
for i=1:n
   A(i+1:n,i+1:n)=A(i+1:n,i+1:n)-A(i+1:n,i)*A(i+1:n,i)'./A(i,i);
   A(i,i)=sqrt(A(i,i)); A(i+1:n,i)=A(i+1:n,i)/A(i,i); A(i,i+1:n)=0;
end
end % function Cholesky
```

Algorithm 4.19: Approximate the Cholesky decomposition while maintaining the sparsity of A in G.

View

```
function [A] = CholeskyIncomplete(A,n)
% Compute the incomplete Cholesky decomposition G*G'H of some A>0.
for i=1:n
   for j=i+1:n, for k=i+1:n,
      if (A(j,k)~=0) A(j,k)=A(j,k)-A(j,i)*A(k,i)'./A(i,i); end;
   end; end;
   A(i,i)=sqrt(A(i,i));
   for j=i+1:n, if (A(j,i)~=0) A(j,i)=A(j,i)/A(i,i); end; end;
   A(i,i+1:n)=0;
end
end % function CholeskyIncomplete
```

Algorithm 4.20: Test the Incomplete Cholesky code on a sparse matrix given by the identity matrix minus a second-order-accurate finite-difference discretization of the 2D Laplacian, as given in (1.6).

View

```
% script <a href="matlab:CholeskyIncompleteTest">CholeskyIncompleteTest</a>
disp('Now testing Cholesky & CholeskyIncomplete on a sparse A>0.')
n=5; m=5; c=.01; A=zeros(m*n,m*n); C=-c*diag(ones(n,1),0);
B=(1+4*c)*diag(ones(n,1),0)-c*diag(ones(n-1,1),-1)-c*diag(ones(n-1,1),1);
for i=0:m-1, A(i*n+1:(i+1)*n,i*n+1:(i+1)*n)=B; end
for i=0:m-2, j=i+1; k=i+2; A(i*n+1:j*n,j*n+1:k*n)=C; A(j*n+1:k*n,i*n+1:j*n)=C; end
A(1:n,(m-1)*n+1:m*n)=C; A((m-1)*n+1:m*n,1:n)=C; n=n*m;
format +; A
Gfull = Cholesky(A,n)            % Gfull loses the sparsity structure of A.
Ginc  = CholeskyIncomplete(A,n) % Ginc retains the sparsity structure of A.
Gdiag = diag(sqrt(diag(A)))      % Gdiag is just the square root of the diagonal of A.
format short;
normA = norm(A),                    disp('This is a measure of A.')
error_Gfull= norm(Gfull*Gfull'-A), disp('This is near zero, indicating Cholesky.m works.')
error_Gdiag= norm(Gdiag*Gdiag'-A), disp('Gdiag is a zero-th order approximation of Gfull.')
error_Ginc = norm(Ginc*Ginc'-A),   disp('Ginc is a much better approximation of Gfull.')
disp(' ')
% end script CholeskyIncompleteTest
```

Approximating the Cholesky decomposition of sparse matrices

In general, the Cholesky factor G does not retain the sparsity structure of the original matrix A. However, if A is diagonally dominant, it is found, following Algorithm 4.18, that those elements of G corresponding to the zero elements of A are often *nearly* zero. As a heuristic, we may thus construct an **approximate Cholesky decomposition** of A by essentially the same algorithm, but touching only the nonzero elements of A and thereby retaining the sparsity structure of A, as implemented in Algorithm 4.19. As illustrated by the test script provided in Algorithm 4.20, this approach often gives a good approximation of the Cholesky factor of A without much computational effort. Note that any large scale implementation of this algorithm should, of course, loop through only the nonzero elements of A, as stored in vectors instead of a sparse matrix, thus skipping the time consuming `if` statements present in the sample code given in Algorithm 4.19.

4.5 Efficient solution of some important matrix equations

We now consider the solution of the unknown, X, in the (as in §20.5.1.2, forward-marching) **controllability** and (as in §20.5.2.2, forward-marching) **observability** forms of the **differential Lyapunov equation** (**DLE**)

$$\frac{dX}{dt} = AX + XA^H + Q, \quad \frac{dX}{dt} = A^H X + XA + Q,$$

and their steady-state solutions satisfying the **continuous-time algebraic Lyapunov equation** (**CALE**)

$$0 = AX + XA^H + Q, \quad 0 = A^H X + XA + Q,$$

the generalization of the CALE known as the **Sylvester equation**

$$0 = AX - XB - C,$$

and the (as in §21.1.2, backward-marching) **control** and (as in §21.1.4, forward-marching) **estimation** forms of the **differential Riccati equation** (**DRE**)

$$-\frac{dX}{dt} = A^H X + XA - XSX + Q, \quad \frac{dX}{dt} = AX + XA^H - XSX + Q$$

and their steady-state solutions satisfying the **continuous-time algebraic Riccati equation** (**CARE**)

$$0 = A^H X + XA - XSX + Q, \quad 0 = AX + XA^H - XSX + Q.$$

We similarly consider the solution of the unknown, X, in the (as in §**??**, forward-marching) **reachability** and (as in §**??**, forward-marching) **observability** forms of the **Lyapunov difference equation** (**LDE**)

$$X_{k+1} = FX_kF^H + Q, \quad X_{k+1} = F^H X_k F + Q,$$

and their steady-state solutions satisfying the **discrete-time algebraic Lyapunov equation** (**DALE**)

$$X = FXF^H + Q, \quad X = F^H XF + Q,$$

the generalization of the DALE known as the **Stein equation**

$$X = A^H XB + C,$$

and the (as in §21.2.2, backward-marching) **control** and (as in §21.2.4, forward-marching) **estimation** forms of the **Riccati difference equation** (**RDE**)

$$X_{k-1} = F^H X_k (I + SX_k)^{-1} F + Q, \quad X_{k+1} = F(I + SX_k)^{-1} X_k F^H + Q,$$

and their steady-state solutions satisfying the **discrete-time algebraic Riccati equation** (**DARE**)

$$X = F^H X(I + SX)^{-1} F + Q, \quad X = F(I + SX)^{-1} XF^H + Q.$$

The DLE, CALE, DRE, and CARE arise in the analysis, control, & estimation of continuous-time systems, whereas the LDE, DALE, RDE, and DARE arise in the analysis, control, & estimation of discrete-time systems. The DLE, DRE, LDE, and RDE may be solved simply by marching (applying, for the DLE and DRE, the appropriate time-marching techniques from §10), whereas *all* of the other equations above may be solved by clever application of the Schur decomposition, thus illustrating the power of this representation. Some of these algorithms require the eigenvalues on the main diagonal of T in the relevant Schur decomposition to be ordered in a particular way; this section thus concludes with an algorithm (based, in fact, on repeated computations of small Sylvester equations) to reorder the Schur decomposition in any desired fashion.

Algorithm 4.21: Solve the CALE, $AX + XA^H + Q = 0$.

View
Test

```
function X=CALE(A,Q)
% Compute the X that satisfies A X + X A' + Q = 0 for full A and Hermitian Q.
n=length(A); [U,T]=Schur(A'); A0=T'; Q0=U'*Q*U; X0=CALEtri(A0,Q0,n); X=U*X0*U';
end % function CALE
%%%
```

Algorithm 4.22: Solve the Sylvester equation $AX - XB = gC$.

View Test

```
function X=Sylvester(A,B,C,g,m,n)
% Compute the X=X_(mxn) that satisfies A X - X B = g C, where A=A_(mxm), B=B_(nxn), and
% C=C_(mxn) are full and g is a scaler with 0 < g <= 1.
[U,A0]=Schur(A); [V,B0]=Schur(B); C0=U'*C*V; X0=SylvesterTri(A0,B0,C0,g,m,n); X=U*X0*V';
end % function Sylvester
%%%
```

4.5.2 Solving the DRE and CARE

As developed in §21.1.2, the block system arising from the continuous-time optimal control problem may be written in the form

$$\frac{d\mathbf{z}}{dt} = H\mathbf{z} \quad \text{where} \quad H = H_{2n\times 2n} = \begin{bmatrix} A & -S \\ -Q & -A^H \end{bmatrix}, \quad \mathbf{z} = \begin{bmatrix} \mathbf{x} \\ \mathbf{r} \end{bmatrix}, \tag{4.47}$$

where $S \geq 0$, $Q \geq 0$, and H is Hamiltonian (see Figure 4.4a and Fact 4.32). Assuming $\mathbf{r} = X\mathbf{x}$ for some $X = X(t)$ (which we will seek to determine) and inserting this assumed form of the solution into the above to eliminate $\mathbf{r}$, combining rows to eliminate $d\mathbf{x}/dt$, factoring out $\mathbf{x}$ to the right, and requiring that the resulting equation holds for all $\mathbf{x}_0$, it follows that X obeys the control form of the **differential Riccati equation** (**DRE**)

$$-\frac{dX}{dt} = A^H X + XA - XSX + Q. \tag{4.48}$$

Starting from Hermitian terminal conditions, solutions $X(t)$ of this equation are Hermitian for all t, and may easily be marched backward in time using the techniques discussed in §10. The estimation form of the DRE is analogous, and may be marched forward in time in a similar fashion.

It is easily verified that H is **Hamiltonian** (see §4.4.4.2) and thus satisfies the symmetric root property (Fact 4.32); that is, for every eigenvalue of H in the LHP, there is a corresponding eigenvalue of H in the RHP. [We assume that H has no eigenvalues on the imaginary axis.] In other words, the vector space Z that $\mathbf{z}$ belongs to can be divided into two subspaces, a stable subspace Z^s and an unstable subspace Z^u. For any $\mathbf{z} \in \mathsf{Z}^s$, $\mathbf{z}(t)$ decays exponentially as $t \to \infty$.

We now seek to find an appropriate relation $\mathbf{r} = X\mathbf{x}$ that restricts the evolution of $\mathbf{z}$ in (4.47) to the stable subspace $\mathbf{z} \in \mathsf{Z}^s$. In terms of the DRE (4.48), we seek the (finite) constant value of X that (4.48) marches to as $t \to -\infty$, thereby satisfying the **continuous-time algebraic Riccati equation** (**CARE**)

$$0 = A^H X + XA - XSX + Q. \tag{4.49}$$

The solution of the CARE may be found by marching (4.48) backward in time to steady state, or determined directly via eigen or Schur decomposition of H, as discussed in the following two subsections.

Approach based on eigen decomposition

Assume first that an eigen decomposition of H is available such that

$$H = S\Lambda S^{-1} \quad \text{where} \quad S = \begin{bmatrix} S_{11} & * \\ S_{21} & * \end{bmatrix} = \begin{bmatrix} | & | & & | & | \\ \mathbf{s}^1 & \mathbf{s}^2 & \dots & \mathbf{s}^n & * \\ | & | & & | & | \end{bmatrix} \quad \text{and} \quad \mathbf{s}^i = \begin{bmatrix} \mathbf{x}^i \\ \mathbf{r}^i \end{bmatrix},$$

where the eigenvalues of H on the main diagonal of diagonal matrix Λ are enumerated such that the LHP eigenvalues appear first, followed by the RHP eigenvalues. Defining $\mathbf{y} = S^{-1}\mathbf{z}$, it follows from (4.47) that $d\mathbf{y}/dt = \Lambda\mathbf{y}$. The stable solution of $\mathbf{y}$ are thus spanned by the first n columns of Λ (that is, they are nonzero only in the first n elements of $\mathbf{y}$). Since $\mathbf{z} = S\mathbf{y}$, it follows that the stable solutions of $\mathbf{z}$ are spanned by the first n columns of S. To achieve stability of $\mathbf{z}$ via the relation $\mathbf{r} = X\mathbf{x}$ for each of these directions, denoted $\mathbf{s}^i$ and decomposed as shown above, we must have $\mathbf{r}^i = X\mathbf{x}^i$ for $i = 1 \dots n$. Assembling these equations in matrix form, we have

$$\begin{bmatrix} | & | & & | \\ \mathbf{r}^1 & \mathbf{r}^2 & \dots & \mathbf{r}^n \\ | & | & & | \end{bmatrix} = X \begin{bmatrix} | & | & & | \\ \mathbf{x}^1 & \mathbf{x}^2 & \dots & \mathbf{x}^n \\ | & | & & | \end{bmatrix} \quad \Rightarrow \quad S_{21} = XS_{11}.$$

Unfortunately, an eigen decomposition of H is *not* always available, and even when it is, it turns out that S_{11} will be nearly singular when H is nearly defective. Thus, in general, *solution of the CARE via eigen decomposition is not recommended in practice*; the method described in the followed section is preferred.

Algorithm 4.23: Solve the CARE, $A^HX + XA - XSX + Q = 0$.

View Test

```
function X=CARE(A,S,Q)
% This function finds the X that satisfies A' X + X A - X S X + Q = 0, with Q >= 0, S >= 0.
% Defining S=B R^{-1} B', we also assume (A,B) is stabilizable and (A,Q) is detectable.
n=size(A,1); [U,T]=Schur([A -S; -Q -A']);
[U,T]=ReorderSchur(U,T,'lhp'); X=GaussPP(U(1:n,1:n)',U(n+1:2*n,1:n)',n);
end % function CARE
```

Approach based on Schur decomposition

Now assume that a Schur decomposition of H is available such that

$$H = UTU^H \quad \text{where} \quad U = \begin{bmatrix} U_{11} & * \\ U_{21} & * \end{bmatrix} = \begin{bmatrix} | & | & & | & | \\ \mathbf{u}^1 & \mathbf{u}^2 & \dots & \mathbf{u}^n & * \\ | & | & & | & | \end{bmatrix} \quad \text{and} \quad \mathbf{u}^i = \begin{bmatrix} \mathbf{x}^i \\ \mathbf{r}^i \end{bmatrix}, \tag{4.50a}$$

where the eigenvalues of H on the main diagonal of the upper triangular matrix T are enumerated such that the LHP eigenvalues appear first, followed by the RHP eigenvalues. Defining $\mathbf{y} = U^H\mathbf{z}$, it follows from (4.47) that $d\mathbf{y}/dt = T\mathbf{y}$. Again, the stable solution of $\mathbf{y}$ are spanned by the first n columns of T (that is, they are nonzero only in the first n elements of $\mathbf{y}$). Since $\mathbf{z} = U\mathbf{y}$, it follows that the stable solutions of $\mathbf{z}$ are spanned by the first n columns of U. To achieve stability of $\mathbf{z}$ via the relation $\mathbf{r} = X\mathbf{x}$ for each of these directions, denoted $\mathbf{u}^i$ and decomposed as shown above, we must have $\mathbf{r}^i = X\mathbf{x}^i$ for $i = 1 \dots n$. Assembling these equations in matrix form, we have

$$\begin{bmatrix} | & | & & | \\ \mathbf{r}^1 & \mathbf{r}^2 & \dots & \mathbf{r}^n \\ | & | & & | \end{bmatrix} = X \begin{bmatrix} | & | & & | \\ \mathbf{x}^1 & \mathbf{x}^2 & \dots & \mathbf{x}^n \\ | & | & & | \end{bmatrix} \quad \Rightarrow \quad U_{21} = XU_{11} \quad \Rightarrow \quad X = U_{21}U_{11}^{-1}. \tag{4.50b}$$

In order for the last relation to be solvable for the unknown X, we must assume that U_{11} is nonsingular. To summarize, the CARE (4.49) is solvable via the above approach under the following two assumptions:

- the matrix H [see (4.47)] has no eigenvalues on the imaginary axis, and
- the upper-left factor U_{11} [see (4.50a)] in the ordered Schur decomposition of H is nonsingular.

Precise conditions under which these two assumptions are guaranteed to be satisfied are discussed in §21.

As opposed to the eigen decomposition, the Schur decomposition is guaranteed to exist (Fact 4.14), and reliable algorithms to compute it are well developed (see §4.4.5). Thus, the Schur decomposition approach discussed here is the preferred approach for solving the CARE, as implemented in Algorithm 4.23.

The estimation form of the CARE may be solved with the same code, called appropriately.

The Chandrasekhar method for approximate solution of K and L

As derived in §21.1.2 and 21.1.4, the control and estimation forms of the DRE are used to determine the **state feedback control** matrix K and the **output injection** matrix L as follows:

$$-\frac{dX(t)}{dt} = A^HX(t) + X(t)A - X(t)BQ_{\mathbf{u}}^{-1}B^HX(t) + Q_{\mathbf{x}} \quad \text{with} \quad X(T) = Q_T, \quad K(t) = -Q_{\mathbf{u}}^{-1}BX(t), \tag{4.51a}$$

$$\frac{dP(t)}{dt} = AP(t) + P(t)A^H - P(t)C^HQ_2^{-1}CP(t) + Q_1 \quad \text{with} \quad P(0) = P_0, \quad L(t) = -P(t)C^HQ_2^{-1}, \tag{4.51b}$$

where $X = X_{n\times n}$, $P = P_{n\times n}$, $K = K_{m_{\mathbf{u}}\times n}$, and $L = L_{n\times m_{\mathbf{y}}}$, where n is the **state dimension**, $m_{\mathbf{u}}$ is the **control dimension**, and $m_{\mathbf{y}}$ is the **measurement dimension**. If $n \gg m_{\mathbf{u}}$ and $n \gg m_{\mathbf{y}}$, which is typical in high-dimensional systems (that is, when $n \gg 1$), then solving Riccati equations for the $n \times n$ matrices X and P in

order to compute the $m_{\mathbf{u}} \times n$ matrix K and the $n \times m_{\mathbf{y}}$ matrix L is inefficient, as this approach computes enormous $n \times n$ Riccati matrices only to take narrow "slices" of these matrices to determine the desired feedback matrices K and L. Chandrasekhar's method (Kailath 1973) addresses this inefficiency in a clever way.

To be specific, consider the DRE for the estimator, as given in (4.51b) [the DRE for the feedback control problem in (4.51a) may be addressed in a similar fashion, as considered in Exercise 4.12]. Chandrasekhar's method solves an evolution equation for a low-dimensional factored form of $dP(t)/dt$ and another evolution equation for $L(t)$. To this end, define

$$\frac{dP(t)}{dt} = Y_1 Y_1^H - Y_2 Y_2^H = YHY^*, \quad Y = \begin{pmatrix} Y_1 & Y_2 \end{pmatrix}, \quad H = \begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix},$$

where the number of columns of Y_1 and Y_2 are the number of positive and negative eigenvalues of (dP/dt), respectively, retained in the approximation. Differentiating (4.51b) with respect to time and inserting $dP/dt = YHY^*$, assuming $\{A, B, C, Q_1, Q_2\}$ are LTI, it is easily verified that the following set of equations are equivalent to (4.51b), but much cheaper to compute if the factors Y_1 and Y_2 are low rank:

$$\begin{aligned} \frac{dL(t)}{dt} &= -Y(t)HY^H(t)C^H {Q_2}^{-1}, \quad & L(0) &= -P(0)C^H {Q_2}^{-1}, \\ \frac{dY(t)}{dt} &= [A + L(t)C]Y(t), & Y(0)HY^*(0) &= \frac{dP(t)}{dt}\Big|_{t=0}, \end{aligned}$$

where $dP/dt|_{t=0}$ is determined from the original DRE (4.51b) evaluated at $t = 0$, and $Y(0)$ is determined by its factorization. Note that Chandrasekhar's method may be used either to approximate the (time-accurate) solution of the original DRE (4.51b), or simply marched to steady state to obtain the solution of the corresponding continuous-time algebraic Riccati equation.

4.5.3 Solving the LDE and DALE

As developed in §**??**, the reachability form of the **Lyapunov difference equation** (**LDE**) is

$$X_{k+1} = FX_kF^H + Q \tag{4.52}$$

for given F, $Q \geq 0$, and $X_0 \geq 0$. Solutions X_k of this equation are Hermitian for all k, and may easily be marched forward in k. The observability form of the LDE is analogous, and may be marched forward in k in a similar fashion. The steady-state solution of (4.52), with $X_{k+1} = X_k$, satisfies the **discrete-time algebraic Lyapunov equation** (**DALE**)

$$X = FXF^H + Q \tag{4.53}$$

The solution of the DALE may be found by marching (4.52) forward in k to steady state, or determined directly, following an analogous procedure as that in §4.5.1, by performing a Schur decomposition $F^H = UF_0^HU^H$, for upper-triangular on F_0^H (that is, $F = UF_0U^H$ for lower-triangular F_0), defining $Q_0 = U^HQU$ and $X_0 = U^HXU$, and substituting into (4.53), leading to

$$X_0 = F_0X_0F_0^H + Q_0 \tag{4.54}$$

where F_0 is lower triangular and Q_0 is Hermitian. Partitioning F_0, X_0, and Q_0 according to

$$F_0 = \begin{bmatrix} f_1 & 0 \\ \mathbf{f}_1 & F_1 \end{bmatrix}, \quad X_0 = \begin{bmatrix} x_1 & \mathbf{x}_1^H \\ \mathbf{x}_1 & X_1 \end{bmatrix}, \quad Q_0 = \begin{bmatrix} q_1 & \mathbf{q}_1^H \\ \mathbf{q}_1 & \tilde{Q}_1 \end{bmatrix},$$

Algorithm 4.24: Solve the DALE, $X = FXF^H + Q$.

View
Test

```
function X=DALE(F,Q)
% Compute the X that satisfies X = F X F^H + Q for full F and Hermitian Q.
n=length(F); [U,T]=Schur(F'); F0=T'; Q0=U'*Q*U; X0=DALEtri(F0,Q0,n); X=U*X0*U';
end % function DALE
%%%
```

Under the assumption that $S = GR^{-1}G^H$ for some G and some $R > 0$, but *not* assuming that X is invertible, noting the matrix inversion lemma (Fact 1.9), the RDE (4.57a) may be written in the form

$$X_{k-1} = F^H X_k F - F^H X_k G(R + G^H X_k G)^{-1} G^H X_k F + Q. \tag{4.57c}$$

Starting from Hermitian terminal conditions, solutions X_k of this RDE (in any of the above three forms) are Hermitian for all k [a fact particularly easy to see in (4.57b) and (4.57c)], and may easily be marched backward in k (see Algorithm 4.25). The estimation forms of the RDE are analogous (see Exercise 4.15), and may be marched forward in k in a similar fashion.

It is easily verified that M is **symplectic** (see §4.4.4.2) and thus satisfies the reciprocal root property (Fact 4.33); that is, for every eigenvalue of M inside the unit circle, there is a corresponding eigenvalue of M outside the unit circle. [We assume that M has no eigenvalues on the unit circle.] In other words, the vector space Z that $\mathbf{z}$ belongs to can be divided into two subspaces, a stable subspace Z^s and an unstable subspace Z^u. For any $\mathbf{z} \in \mathsf{Z}^s$, $\mathbf{z}_k$ decays exponentially as $k \to \infty$.

We now seek to find an appropriate relation $\mathbf{r}_k = X\,\mathbf{x}_k$ that restricts the evolution of $\mathbf{z}_k$ in (4.56) to the stable subspace $\mathbf{z} \in \mathsf{Z}^s$. In terms of the RDE (4.57), we seek the (finite) constant-in-k value of X that (4.57a) marches to as $k \to -\infty$, thereby satisfying the **discrete-time algebraic Riccati equation** (**DARE**)

$$X = F^H X(I + SX)^{-1} F + Q. \tag{4.58}$$

The solution of the DARE may be found by marching one of the three forms of (4.57) backward in k to steady state, or determined directly via Schur decomposition of M: first decompose

$$M = UTU^H \quad \text{where} \quad U = \begin{bmatrix} U_{11} & * \\ U_{21} & * \end{bmatrix}, \tag{4.59a}$$

where the eigenvalues of M on the main diagonal of the upper triangular matrix T are enumerated such that those eigenvalues inside the unit circle appear first, followed by those eigenvalues outside the unit circle. Following the same line of reasoning as in §4.5.2, the resulting expression for X is given by

$$U_{21} = XU_{11} \quad \Rightarrow \quad X = U_{21}U_{11}^{-1}, \tag{4.59b}$$

as implemented in Algorithm 4.26.

The doubling algorithm for solution of the DARE

Starting at $k = 0$ **without loss of generality** (**WLOG**), the second form at left in (4.56), over 1 timestep, is written $\mathbf{z}_{-1} = M^{-1}\mathbf{z}_0$. Over 2 time steps, we may write $\mathbf{z}_{-2} = M^{-2}\mathbf{z}_0$, and over $n = 2^\ell$ time steps, we may write $\mathbf{z}_{-n} = M^{-n}\mathbf{z}_0$ where, as easily verified, $M^{-k} \cdot M^{-k} = M^{-2k}$ may be written as follows:

$$\begin{bmatrix} F_k^{-1} & F_k^{-1}S_k \\ Q_k F_k^{-1} & F_k + Q_k F_k^{-1} S_k \end{bmatrix} \cdot \begin{bmatrix} F_k^{-1} & F_k^{-1}S_k \\ Q_k F_k^{-1} & F_k + Q_k F_k^{-1} S_k \end{bmatrix} = \begin{bmatrix} F_{2k}^{-1} & F_{2k}^{-1}S_{2k} \\ Q_{2k} F_{2k}^{-1} & F_{2k} + Q_{2k} F_{2k}^{-1} S_{2k} \end{bmatrix} \tag{4.60a}$$

where $F_1 = F$, $S_1 = S$, $Q_1 = Q$, and

$$F_{2k} = F_k(I + S_k Q_k)^{-1} F_k, \quad Q_{2k} = Q_k + F_k^H (I + Q_k S_k)^{-1} Q_k F_k, \quad S_{2k} = S_k + F_k S_k (I + Q_k S_k)^{-1} F_k^H. \tag{4.60b}$$

As described previously, the (constant-in-k) solution of the DARE is simply the matrix X that, if used to relate the $\mathbf{r}$ and $\mathbf{x}$ components of $\mathbf{z}_k$ such that $\mathbf{r}_k = X\mathbf{x}_k$, restricts the evolution of $\mathbf{z}_k$ to the space spanned by the n stable eigenvectors of M, thus leading to a stable forward-in-k march of the evolution equation in (4.56). If n linearly-independent $\mathbf{z}$ spanning this subspace may be found, then the corresponding X may be

Algorithm 4.25: March the RDE (4.57) a given number of timesteps.

View Test

```
function X=RDE(X,F,S,Q,n,steps)
% March the RDE X_{k-1} = F' X_k (I+ S X_k)^{-1} F + Q a given number of steps.
for iter=1:steps; X=F'*X*GaussPP(eye(n)+S*X,F,n)+Q; end
end % function RDE
```

Algorithm 4.26: Solve the DARE (4.58) via the Schur-based algorithm.

View Test

```
function X=DARE(F,S,Q,n)
% Finds the X that satisfies X = F' X (I+ S X)^{-1} F + Q, with Q>=0, S>=0, and |F|<>0.
% This code uses an approach based on an ordered Schur decomposition.
E=inv(F'); [U,T]=Schur([F+S*E*Q -S*E; -E*Q E]); [U,T]=ReorderSchur(U,T,'unitdisk');
X=GaussPP(U(1:n,1:n)',U(n+1:2*n,1:n)',n);
end % function DARE
```

Algorithm 4.27: Solve the DARE (4.58) via the doubling algorithm, marching (4.60b) to steady state.

View

```
function Q=DAREdoubling(F,S,Q,n,steps)
% Finds the X that satisfies X = F' X (I+ S X)^{-1} F + Q, with Q>=0, S>=0.
% This code uses an elegant and efficient approach known as the doubling algorithm.
for iter=1:steps
  E=inv(eye(n)+Q*S); Fnew=F*E'*F; Qnew=Q+F'*E*Q*F; S=S+F*S*E*F'; F=Fnew; Q=Qnew;
end
end % function DAREdoubling
```

determined simply by partitioning these $\mathbf{z}$ into their $\mathbf{x}$ and $\mathbf{r}$ components, combining these relations into a single matrix form, then solving for X, as done in (4.59b).

An alternative method of determining n linearly-independent vectors which span the space spanned by the n stable eigenvectors of M, which bypasses the Schur or eigen decomposition of M, is now described. We simply initialize several convenient, linearly-independent $\mathbf{z}_0$ (that is, several $\mathbf{x}_0$ and $\mathbf{r}_0$) vectors combined into a single matrix form,

$$Z_0 = \begin{bmatrix} U_0 \\ V_0 \end{bmatrix} \quad \text{where} \quad U_0 = I, \quad V_0 = 0,$$

then march all of these vectors *backward* in k using a matrix form of (4.56), $Z_{k-1} = M^{-1} Z_k$. During this backward-in-k march, the solution components in the directions of the unstable eigenvectors of M^{-1} (that is, the stable eigenvectors of M) grow exponentially, while the the solution components in the directions of the stable eigenvectors of M^{-1} (that is, the unstable eigenvectors of M) decay exponentially toward zero. After marching a sufficient number of steps n such that the latter are negligible, the desired X may be determined simply by taking $X = V_{-n} \cdot U_{-n}^{-1}$. Noting further by (4.60a) that, for $n = 2^\ell$, we may write

$$\begin{bmatrix} U_{-n} \\ V_{-n} \end{bmatrix} = \begin{bmatrix} F_n^{-1} & F_n^{-1} S_n \\ Q_n F_n^{-1} & F_n + Q_n F_n^{-1} S_n \end{bmatrix} \begin{bmatrix} U_0 \\ V_0 \end{bmatrix}. \tag{4.60c}$$

It follows that $U_{-n} = F_n^{-1}$ and $V_{-n} = Q_n F_n^{-1}$, and thus

$$X = V_{-n} \cdot U_{-n}^{-1} = Q_n \tag{4.60d}$$

for sufficiently large n (see Algorithm 4.27). Since $n = 2^\ell$ grows rapidly with ℓ, only a few iterations of (4.60b) are required to reach convergence. [Note, e.g., that $\ell = 10$ steps of this algorithm are equivalent to $n = 2^\ell = 1024$ steps of Algorithm 4.25!] Convergence of this elegant **doubling algorithm** is so rapid that it is often preferred over the Schur-based approach described previously for rapid solution of the DARE.

4.5.5 Reordering the Schur decomposition

The algorithms to solve the CALE [in §4.5.1] and DALE [in §4.5.3] were built on *any* Schur decomposition of the matrix $A_{n\times n}$, whereas the algorithms to solve the CARE [in §4.5.2] and DARE [in §4.5.4] were built on appropriately *ordered* Schur decompositions of, respectively, a $2n\times 2n$ Hamiltonian matrix H [see (4.47)] and $2n\times 2n$ symplectic matrix M [see (4.56)], where the stable eigenvalues of these ordered Schur decompositions appear in the first n elements on the main diagonal of T, followed by the unstable eigenvalues in the last n elements on the main diagonal of T. (Recall that, for the CARE, the stable eigenvalues are in the LHP, whereas for the DARE, the stable eigenvalues are within the unit disk.) Of course, a general Schur decomposition algorithm (for example, one based on the QR method described in §4.4.5) will, in general, not return the elements of T in these desired orderings.

The key building block of an efficient algorithm to accomplish the desired reordering of a Schur decomposition is the computation of a unitary matrix $\tilde{Q}$ such that

$$\tilde{Q}^H \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix} \tilde{Q} = \begin{bmatrix} \tilde{T}_{11} & \tilde{T}_{12} \\ 0 & \tilde{T}_{22} \end{bmatrix}, \tag{4.61}$$

where T_{11} is $n_1\times n_1$ and T_{22} is $n_2\times n_2$ and the matrices T_{11}, T_{22}, $\tilde{T}_{11}$, and $\tilde{T}_{22}$ are upper triangular, with $\lambda(\tilde{T}_{11})=\lambda(T_{22})$ and $\lambda(\tilde{T}_{22})=\lambda(T_{11})$. Note that the form on the right is unitarily similar to the form on the left, though the eigenvalues of the blocks on the main diagonal have been swapped. To build this transformation, consider the identity

$$\begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix} = \begin{bmatrix} I & -X \\ 0 & gI \end{bmatrix} \begin{bmatrix} T_{11} & 0 \\ 0 & T_{22} \end{bmatrix} \begin{bmatrix} I & X/g \\ 0 & I/g \end{bmatrix} \tag{4.62}$$

for some g between zero and one[26], where X is the solution of the Sylvester equation

$$g\,T_{12} = T_{11}X - XT_{22}, \tag{4.63a}$$

which may be solved using Algorithm 4.22. Now consider the (complete) QR decomposition

$$\begin{bmatrix} -X \\ gI \end{bmatrix} = QR \quad \text{where} \quad Q = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}, \quad R = \begin{bmatrix} R_{11} \\ 0 \end{bmatrix}, \quad Q^HQ = I, \tag{4.63b}$$

and R_{11} is upper triangular. This decomposition may be solved efficiently using the methods described in §4.4.2. It follows from (4.63b) that $R_{11}^{-1} = Q_{21}/g$ and $Q_{12}^{-H} = Q_{12} - Q_{11}R_{11}Q_{22}/g$. Now multiplying (4.62) from the left by Q^H and from the right by Q and applying the relations in (4.63a) and (4.63b), it may be shown that this value for Q almost accomplishes the swap sought in (4.61), with

$$Q^H \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix} Q = \begin{bmatrix} R_{11}T_{22}R_{11}^{-1} & * \\ 0 & Q_{12}^H T_{11} Q_{12}^{-H} \end{bmatrix} \triangleq \begin{bmatrix} \tilde{T}_{11} & * \\ 0 & B \end{bmatrix}.$$

Note that the block $\tilde{T}_{11}$ of the matrix so generated is the product of upper triangular matrices, and is thus itself upper triangular. However, the block B of the matrix generated by this process is not yet upper triangular. Performing a Schur decomposition of this block, $B = V\tilde{T}_{22}V^H$, finally leads us to the desired form

$$\tilde{Q}^H \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix} \tilde{Q} = \begin{bmatrix} \tilde{T}_{11} & \tilde{T}_{12} \\ 0 & \tilde{T}_{22} \end{bmatrix} \quad \text{where} \quad \tilde{Q} = Q \begin{bmatrix} I & 0 \\ 0 & V \end{bmatrix} \quad \text{with} \quad B = V\tilde{T}_{22}V^H. \tag{4.63c}$$

[26] Note that g is introduced to scale the transformation well for large n when several swaps need to be performed, thereby keeping the norms of the blocks from getting large as several swaps are performed in succession. Any of a number of heuristic strategies for selecting g work adequately; for small n, $g=1$ is sufficient.

Algorithm 4.28: Reorder the Schur decomposition, putting the stable eigenspace first.

View Test

```
function [U,T]=ReorderSchur(U,T,type,e)
% This function reorders a Schur decomposition such that the stable eigenvalues appear
% in the first n columns and the unstable eigenvalues appear in the last n columns.
n=length(U); g=1; p1=n; p2=n; p3=n; % initialize n, g, and 3 placeholders
switch type(1)
  case 'l'    % type='lhp' for continuous-time systems (first LHP modes, then RHP)
    while p1>0 % (start from bottom right and work towards upper left)
      while p3>=1 & real(T(p3,p3))>0;  p3=p3-1; end;  p2=p3; if p3==1, break, end
      while p2>=1 & real(T(p2,p2))<0;  p2=p2-1; end;  p1=p2; if p2==0, break, end
      while p1>=1 & real(T(p1,p1))>0;  p1=p1-1; end;  [U,T]=BlockSwap(U,T,p1,p2,p3,g);
    end
  case 'u'    % type='unitdisk' for discrete-time systems (inside, then outside unit disk)
    while p1>0
      while p3>=1 & abs(T(p3,p3))>1;   p3=p3-1; end;  p2=p3; if p3==1, break, end
      while p2>=1 & abs(T(p2,p2))<1;   p2=p2-1; end;  p1=p2; if p2==0, break, end
      while p1>=1 & abs(T(p1,p1))>1;   p1=p1-1; end;  [U,T]=BlockSwap(U,T,p1,p2,p3,g);
    end
  case 'a'    % type = 'absolute', to order by the absolute value of the real part
    for i=n-1:-1:1, a=i+1; b=n;       % (see InsertionSort.m for details)
      while a<b-1; c=a+floor((b-a)/2);
        if abs(real(T(c,c))+e)<abs(real(T(i,i))+e), a=c+1; else, b=c-1; end, end
      while a<=b;
        if abs(real(T(i,i))+e)<abs(real(T(b,b))+e), b=b-1; else, a=a+2; end, end
      if b>i, [U,T]=BlockSwap(U,T,i-1,i,b,g); end % Insert record i at the correct point.
    end
end
end % function ReorderSchur
%%%
```

4.6 The trace

The **trace** of a square matrix is the sum of its diagonal elements, and is defined using summation notation as

$$\text{trace}(A) = a_{jj} = (\mathbf{e}^j)^T A \mathbf{e}^j.$$

Based on this definition, is easy to verify that the trace obeys the following:

Fact 4.43 $\text{trace}(A) = \text{trace}(A^T)$; $\text{trace}(AB) = a_{ij}b_{ji} = \text{trace}(BA) = \text{trace}(B^TA^T) = \text{trace}(A^TB^T)$;
$\text{trace}(ABC) = a_{ij}b_{jk}c_{ki} = \text{trace}(CAB) = \text{trace}(BCA) = \text{trace}(C^TB^TA^T) = \text{trace}(A^TC^TB^T) = \text{trace}(B^TA^TC^T)$.

By the Schur decomposition theorem (Fact 4.14), A may be decomposed such that $A = UTU^H$, and thus

Fact 4.44 $\text{trace}(A) = \text{trace}(UTU^H) = \text{trace}(U^HUT) = \text{trace}(T) = \lambda_1 + \lambda_2 + \ldots + \lambda_n$.
Similarly, $\text{trace}(A^k) = \text{trace}[(UTU^H)^k] = \text{trace}[UT^kU^H] = \text{trace}(U^HUT^k) = \text{trace}(T^k) = \lambda_1^k + \lambda_2^k + \ldots + \lambda_n^k$.

As discussed further in §6.1, a **covariance matrix** P is defined as the **expected value** of the outer product of two vectors. In particular, denoting $\mathcal{E}[\cdot]$ as the **expectation operator** indicating the average over a large number of statistical samples of the quantity in brackets, if we define $P = \mathcal{E}(\mathbf{x}\mathbf{x}^H)$, then it is easy to verify (noting that both the expectation and the trace are linear operators) that $\text{trace}(P) = \text{trace}(\mathcal{E}(\mathbf{x}\mathbf{x}^H)) = \mathcal{E}(\text{trace}(\mathbf{x}\mathbf{x}^H)) = \mathcal{E}(\text{trace}(\mathbf{x}^H\mathbf{x})) = \mathcal{E}(|x_1|^2 + |x_2|^2 + \ldots + |x_n|^2)$; that is, $\text{trace}(P)$ is the expected 2-norm, or "energy", of the vector $\mathbf{x}$. Thus, the trace of a covariance matrix is a measure of particular significance.

4.6.1 Identities involving the derivative of the trace of a product

Defining the derivative of a scalar $J(X)$ with respect to the matrix X as

$$\frac{\partial J}{\partial X} \triangleq \begin{pmatrix} \frac{\partial J}{\partial x_{11}} & \cdots & \frac{\partial J}{\partial x_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial x_{n1}} & \cdots & \frac{\partial J}{\partial x_{nn}} \end{pmatrix},$$

it follows that the derivative of the trace with respect to a constituent matrix X is easily computed using summation notation. For example,

$$\frac{\partial\, a_{ij}x_{jk}b_{ki}}{\partial x_{lm}} = a_{ij}b_{ki}\delta_{jl}\delta_{km} = a_{il}b_{mi} \quad \Rightarrow \quad \frac{\partial\, \text{trace}(AXB)}{\partial X} = A^TB^T, \tag{4.64a}$$

$$\frac{\partial\, x_{ji}a_{jk}x_{ki}}{\partial x_{lm}} = a_{jk}x_{ki}\delta_{jl}\delta_{im} + x_{ji}a_{jk}\delta_{kl}\delta_{im} = a_{lk}x_{km} + a_{jl}x_{jm} \quad \Rightarrow \quad \frac{\partial\, \text{trace}(X^TAX)}{\partial X} = (A + A^T)X. \tag{4.64b}$$

Several alternative forms of the above two identities may be derived by applying Fact 4.43 and setting either A or B equal to I. The derivative of the trace of other matrix expressions is computed in an analogous fashion.

4.6.2 Computing the sensitivity of an eigenvalue to matrix perturbations

Let $A\mathbf{s} = \lambda\mathbf{s}$ and $\mathbf{r}^TA = \lambda\mathbf{r}^T$ (that is, $a_{kl}s_l = \lambda s_k$ and $r_ka_{kl} = \lambda r_l$) define the right and left eigenvectors $\mathbf{s}$ and $\mathbf{r}$ corresponding to a distinct eigenvalue λ of the matrix A. The sensitivity of the eigenvalue λ to an infinitesimal perturbation of the a_{ij} component of the matrix A is given in summation notation as follows:

$$\frac{\partial}{\partial a_{ij}}\Big[a_{kl}s_l = \lambda s_k\Big] \quad \Rightarrow \quad \frac{\partial a_{kl}}{\partial a_{ij}}s_l + a_{kl}\frac{\partial s_l}{\partial a_{ij}} = \frac{\partial \lambda}{\partial a_{ij}}s_k + \lambda\frac{\partial s_k}{\partial a_{ij}} \quad \Rightarrow \quad r_k\Big[\delta_{ik}\delta_{jl}s_l + a_{kl}\frac{\partial s_l}{\partial a_{ij}} = \frac{\partial \lambda}{\partial a_{ij}}s_k + \lambda\frac{\partial s_k}{\partial a_{ij}}\Big]$$

$$r_is_j + (r_ka_{kl})\frac{\partial s_l}{\partial a_{ij}} = \frac{\partial \lambda}{\partial a_{ij}}r_ks_k + (\lambda r_l)\frac{\partial s_l}{\partial a_{ij}} \quad \Rightarrow \quad \frac{\partial \lambda}{\partial a_{ij}} = \frac{r_is_j}{r_ks_k} \quad \Rightarrow \quad \frac{\partial \lambda}{\partial A} = \frac{\mathbf{r}\mathbf{s}^T}{\mathbf{r}^T\mathbf{s}}. \tag{4.65}$$

Given a structured perturbation of the matrix A of the form $A(\alpha) = B + \alpha C + \alpha^2 D + \ldots$, we may write

$$\frac{d\lambda}{d\alpha} = \frac{\partial \lambda}{\partial a_{ij}} \frac{\partial a_{ij}}{\partial \alpha} = \frac{r_i s_j}{r_k s_k} c_{ij} \quad \Rightarrow \quad \frac{d\lambda}{d\alpha} = \frac{\mathbf{r}^T C \mathbf{s}}{\mathbf{r}^T \mathbf{s}}. \tag{4.66}$$

4.6.3 Rewriting the characteristic polynomial using the trace

Combining Fact 4.44 with Fact 4.17, we see that *the determinant of A is equal to the product of its eigenvalues, whereas the trace of A is equal to the sum of its eigenvalues*. Thus, multiplying out the characteristic polynomial (4.5) of a matrix A, we may write

$$(\lambda - \lambda_1)(\lambda - \lambda_2) \cdots (\lambda - \lambda_n) = \lambda^n - \text{trace}(A)\lambda^{n-1} + \ldots + (-1)^k a_{n-k} \lambda^{n-k} + \ldots + (-1)^n |A| = 0. \tag{4.67}$$

It is easily verified that each coefficient in this polynomial, a_{n-k} for $k = 1, \ldots, n$, is given by the sum of the products of all sets of k eigenvalues of the matrix A. For example, in the $n = 3$ case, we have

$$\lambda^3 - (\lambda_1 + \lambda_2 + \lambda_3)\lambda^2 + (\lambda_1\lambda_2 + \lambda_1\lambda_3 + \lambda_2\lambda_3)\lambda - \lambda_1\lambda_2\lambda_3 = 0,$$

and in the $n = 4$ case, we have

$$\begin{aligned}\lambda^4 - (\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4)\lambda^3 + (\lambda_1\lambda_2 + \lambda_1\lambda_3 + \lambda_1\lambda_4 + \lambda_2\lambda_3 + \lambda_2\lambda_4 + \lambda_3\lambda_4)\lambda^2 \\ - (\lambda_1\lambda_2\lambda_3 + \lambda_1\lambda_2\lambda_4 + \lambda_1\lambda_3\lambda_4 + \lambda_2\lambda_3\lambda_4)\lambda + \lambda_1\lambda_2\lambda_3\lambda_4 = 0.\end{aligned}$$

We now state and prove an intermediate fact that will help us establish a useful formula for the other coefficients of the characteristic polynomial based on the trace.

Fact 4.45 *If $p_n(z) = z^n + a_{n-1}z^{n-1} + \ldots + a_1 z + a_0$ for $n \geq 2$, then*

$$\frac{p_n(z) - p_n(\lambda)}{z - \lambda} = z^{n-1} + (\lambda + a_{n-1})z^{n-2} + (\lambda^2 + a_{n-1}\lambda + a_{n-2})z^{n-1} + \ldots + (\lambda^{n-1} + a_{n-1}\lambda^{n-2} + \ldots + a_1).$$

Proof (by induction): The base case $n = 2$ follows immediately from

$$p_2(z) - p_2(\lambda) = (z^2 + a_1 z + a_0) - (\lambda^2 + a_1\lambda + a_0) = (z - \lambda)[z + (\lambda + a_1)].$$

Assume the theorem holds for order $n-1$. Now consider the case of order n, and apply the inductive hypothesis to the underbraced term:

$$\begin{aligned}
&p_n(z) - p_n(\lambda) = (z^n + a_{n-1}z^{n-1} + \ldots + a_1 z + a_0) - (\lambda^n + a_{n-1}\lambda^{n-1} + \ldots + a_1\lambda + a_0) \\
&= z(z^{n-1} + a_{n-1}z^{n-2} + \ldots + a_1) - \lambda(\lambda^{n-1} + a_{n-1}\lambda^{n-2} + \ldots + a_1) \\
&= z[\underbrace{(z^{n-1} + a_{n-1}z^{n-2} + \ldots + a_1) - (\lambda^{n-1} + a_{n-1}\lambda^{n-2} + \ldots + a_1)}_{=p_{n-1}(z) - p_{n-1}(\lambda)}] + (z - \lambda)(\lambda^{n-1} + a_{n-1}\lambda^{n-2} + \ldots + a_1) \\
&= z[(z - \lambda)(z^{n-2} + (\lambda + a_{n-1})z^{n-3} + \ldots + (\lambda^{n-2} + a_{n-1}\lambda^{n-3} + \ldots + a_2))] + (z - \lambda)(\lambda^{n-1} + a_{n-1}\lambda^{n-2} + \ldots + a_1) \\
&= (z - \lambda)[z^{n-1} + (\lambda + a_{n-1})z^{n-2} + \ldots + (\lambda^{n-2} + a_{n-1}\lambda^{n-3} + \ldots + a_2)z + (\lambda^{n-1} + a_{n-1}\lambda^{n-2} + \ldots + a_1)].
\end{aligned}$$

□

With this fact established, we may now state and prove the following useful result:

Fact 4.46 *If* $p(z) = z^n + a_{n-1}z^{n-1} + \ldots + a_1 z + a_0 = (z-\lambda_1)(z-\lambda_2)\cdots(z-\lambda_n)$ *is the characteristic polynomial of* $A_{n\times n}$*, denoting* $T_k = \text{trace}(A^k) = \lambda_1^k + \lambda_2^k + \ldots + \lambda_n^k$*, we may write*

$$T_k + a_{n-1}T_{k-1} + \ldots + a_{n-k+1}T_1 + a_{n-k}k = 0 \quad \textit{for} \quad k = 1, 2, \ldots, n.$$

Proof: First note that

$$\begin{aligned}
p'(z) = \frac{dp(z)}{dz} &= nz^{n-1} + (n-1)a_{n-1}z^{n-2} + (n-2)a_{n-2}z^{n-3} + \ldots + a_1 \\
&= \sum_{k=1}^{n} \frac{p(z)}{z-\lambda_k} = \sum_{k=1}^{n} \frac{p(z)-p(\lambda_k)}{z-\lambda_k} \\
&= \sum_{k=1}^{n} [z^{n-1} + (\lambda_k + a_{n-1})z^{n-2} + (\lambda_k^2 + a_{n-1}\lambda_k + a_{n-2})z^{n-3} + \ldots + (\lambda_k^{n-1} + a_{n-1}\lambda_k^{n-2} + \ldots + a_1)] \\
&= nz^{n-1} + (T_1 + a_{n-1}n)z^{n-2} + (T_2 + a_{n-1}T_1 + a_{n-2}n)z^{n-3} + \ldots + (T_{n-1} + a_{n-1}T_{n-2} + \ldots + a_2T_1 + a_1 n).
\end{aligned}$$

Equating coefficients of like powers of z between the first line and the last, we obtain

$$\begin{array}{rl}
(n-1)a_{n-1} = T_1 + a_{n-1}n & 0 = T_1 + a_{n-1} \\
(n-2)a_{n-2} = T_2 + a_{n-1}T_1 + a_{n-2}n & 0 = T_2 + a_{n-1}T_1 + a_{n-2}2 \\
\vdots \qquad\qquad \Rightarrow & \vdots \\
a_1 = T_{n-1} + a_{n-1}T_{n-2} + \ldots + a_2T_1 + a_1 n & 0 = T_{n-1} + a_{n-1}T_{n-2} + \ldots + a_2T_1 + a_1(n-1).
\end{array}$$

Since each λ_k satisfies $p(\lambda_k) = 0$, we may also write

$$\sum_{k=1}^{n} p(\lambda_k) = T_n + a_{n-1}T_{n-1} + \ldots + a_1T_1 + a_0(n) = 0. \qquad \square$$

Note that we may easily rewrite the n relations of Fact 4.46 as

$$\begin{array}{ll}
a_{n-1} = -T_1 & = -\text{trace}(A) \\
a_{n-2} = -\frac{1}{2}(T_2 + a_{n-1}T_1) & = -\frac{1}{2}\text{trace}(A^2 + a_{n-1}A) \\
a_{n-3} = -\frac{1}{3}(T_3 + a_{n-1}T_2 + a_{n-2}T_1) & = -\frac{1}{3}\text{trace}(A^3 + a_{n-1}A^2 + a_{n-2}A) \\
\vdots & \\
a_1 = -\frac{1}{n-1}(T_{n-1} + a_{n-1}T_{n-2} + \ldots + a_2T_1) & = -\frac{1}{n-1}\text{trace}(A^{n-1} + a_{n-1}A^{n-2} + \ldots + a_2A) \\
a_0 = -\frac{1}{n}(T_n + a_{n-1}T_{n-1} + \ldots + a_1T_1) & = -\frac{1}{n}\text{trace}(A^n + a_{n-1}A^{n-1} + \ldots + a_1A)
\end{array} \tag{4.68}$$

This formula will be particularly useful when developing the resolvent algorithm (20.35).

4.7 The Moore-Penrose pseudoïnverse

The Moore-Penrose pseudoïnverse is a useful generalization of the matrix inverse discussed in §1.2.4 that is appropriate for singular and nonsquare matrices. It is established as follows:

Fact 4.47 *Given any $m \times n$ matrix A, the Moore-Penrose pseudoïnverse $A^+ = \underline{V\Sigma}^{-1}\underline{U}^H$ is the unique $n \times m$ matrix such that*

$$AA^+A = A, \qquad A^+AA^+ = A^+, \qquad AA^+ = (AA^+)^H, \qquad A^+A = (A^+A)^H, \tag{4.69}$$

where $A = \underline{U\Sigma V}^H$ is a reduced SVD of A.

Proof: It is trivial to verify that $A^+ = \underline{V\Sigma}^{-1}\underline{U}^H$ satisfies the four Moore-Penrose conditions given in (4.69):

$$\begin{aligned}
AA^+A &= \underline{U\Sigma V}^H\,\underline{V\Sigma}^{-1}\underline{U}^H\,\underline{U\Sigma V}^H = \underline{U\Sigma\Sigma}^{-1}\underline{\Sigma V}^H = A \\
A^+AA^+ &= \underline{V\Sigma}^{-1}\underline{U}^H\,\underline{U\Sigma V}^H\,\underline{V\Sigma}^{-1}\underline{U}^H = \underline{V\Sigma}^{-1}\underline{\Sigma\Sigma}^{-1}\underline{U}^H = A^+ \\
AA^+ &= \underline{U\Sigma V}^H\,\underline{V\Sigma}^{-1}\underline{U}^H = \underline{UU}^H = (\underline{UU}^H)^H = (AA^+)^H \\
A^+A &= \underline{V\Sigma}^{-1}\underline{U}^H\,\underline{U\Sigma V}^H = \underline{VV}^H = (\underline{VV}^H)^H = (A^+A)^H.
\end{aligned}$$

To show that this value of A^+ is unique, consider two matrices B and C that both satisfy the conditions on A^+ given in (4.69):

$$\begin{aligned}
ABA &= A, \qquad BAB = B, \qquad AB = (AB)^H, \qquad BA = (BA)^H, \\
ACA &= A, \qquad CAC = C, \qquad AC = (AC)^H, \qquad CA = (CA)^H.
\end{aligned}$$

It follows that $AB = B^HA^H = B^HA^HC^HA^H = ABC^HA^H = ABAC = AC$, and, similarly, that $BA = CA$. We thus conclude that $B = BAB = BAC = CAC = C$. □

Note that, if A is a (square) invertible matrix, then $A^+ = A^{-1}$, and if (A^HA) is invertible, then $A^+ = (A^HA)^{-1}A^H$, as may be verified by substitution into (4.69).

If A is either singular (that is, square with determinant equal zero) or nonsquare, then the dimension of either the nullspace and/or the left nullspace of A, as depicted in Figure 4.1, is nonzero. In this case, the mapping between X and Y in Figure 4.1 is not one-to-one. If the dimension of the left nullspace is nonzero, then for some $\mathbf{y} \in \mathsf{Y}$, there is no corresponding $\mathbf{x} \in \mathsf{X}$ such that $\mathbf{y} = A\mathbf{x}$. If the dimension of the nullspace is nonzero, then for some $\mathbf{y} \in \mathsf{Y}$, there are multiple $\mathbf{x} \in \mathsf{X}$ such that $\mathbf{y} = A\mathbf{x}$. However, *the mapping between the row space and the column space is still one-to-one*. Two distinguishing characteristics of the Moore-Penrose pseudoïnverse A^+ are:

- The Moore-Penrose pseudoïnverse A^+ maps the column space back to the row space in such a way that, if $\mathbf{x}_\mathsf{R}$ is in the row space, then $\mathbf{x}_\mathsf{R}$ gets mapped back to itself when operated on first by A and then by A^+, that is,
$$A^+(A\mathbf{x}_\mathsf{R}) = [\underline{V\Sigma}^{-1}\underline{U}^H][\underline{U\Sigma V}^H]\mathbf{x}_\mathsf{R} = \underline{VV}^H\mathbf{x}_\mathsf{R} = \underline{\mathbf{v}}^j(\underline{\mathbf{v}}^j, \mathbf{x}_\mathsf{R}) = \mathbf{x}_\mathsf{R},$$
as $\mathbf{x}_\mathsf{R}$ is in the row space of A, for which the $\underline{\mathbf{v}}^j$ form an orthogonal basis.
- If $\mathbf{y}_\mathsf{L}$ is in the left nullspace (that is, the subspace of Y that cannot be reached by the operation $A\mathbf{x}$ for any $\mathbf{x}$), then the Moore-Penrose pseudoïnverse simply maps $\mathbf{y}_\mathsf{L}$ back to zero; that is,
$$A^+\mathbf{y}_\mathsf{L} = \underline{V\Sigma}^{-1}\underline{U}^H\mathbf{y}_\mathsf{L} = 0,$$
as $\mathbf{y}_\mathsf{L}$ is in the left nullspace of A, to which the $\underline{\mathbf{u}}^j$ (that is, the columns of $\underline{U}$) are all orthogonal.

Thus, if $\mathbf{y} = A\mathbf{x}$ where A is possibly singular or nonsquare, the Moore-Penrose pseudoïnverse does the best job possible, in a particular well-defined sense (discussed further below), at mapping a given value of $\mathbf{y}$ back to the corresponding $\mathbf{x}$ via the mapping $\mathbf{x} = A^+\mathbf{y}$, as illustrated graphically in Figure 4.3.

4.7.1 Inconsistent and/or underdetermined systems

Given A and $\mathbf{b}$, the "best" solution $\{\mathbf{x}, \boldsymbol{\varepsilon}\}$ to the problem

$$A\mathbf{x} = \mathbf{b} + \boldsymbol{\varepsilon} \tag{4.70}$$

with m equations and n unknowns (that is, "best" in a **least-squares** sense, minimizing the length of the error vector $\boldsymbol{\varepsilon}$) in the inconsistent case (with two or more equations which are impossible to satisfy simultaneously for any $\mathbf{x}$ when $\boldsymbol{\varepsilon} = 0$) and/or the underdetermined case (with fewer independent equations than unknowns) may be found directly using the Moore-Penrose pseudoïnverse. In order to ensure that $\boldsymbol{\varepsilon}$ is as small as possible [in the inconsistent case, a solution $\mathbf{x}$ to (4.70) is not possible with $\boldsymbol{\varepsilon} = 0$], we seek the value of $\mathbf{x}$ such that $\boldsymbol{\varepsilon}$ is at least orthogonal to all vectors that may be reached by the operation $A\mathbf{x}$ for any $\mathbf{x}$. That is, we want $\boldsymbol{\varepsilon}$ to be in the left nullspace, $\mathcal{N}(A^H)$, from which it follows that

$$A^+\boldsymbol{\varepsilon} = 0. \tag{4.71}$$

This may be enforced by taking $A^H\boldsymbol{\varepsilon} = A^H(A\mathbf{x} - \mathbf{b}) = 0$, from which we deduce that

$$A^HA\mathbf{x} = A^H\mathbf{b}. \tag{4.72}$$

If (A^HA) is invertible, the solution is seen immediately to be[27]

$$\mathbf{x} = (A^HA)^{-1}A^H\mathbf{b} = A^+\mathbf{b}. \tag{4.73}$$

In the underdetermined case, however, $B = (A^HA)$ will not be invertible. In this case, there are several vectors $\mathbf{x}$ that satisfy (4.72); following a similar least-squares mindest, we may seek the smallest one. That is, we want $\mathbf{x}$ to be in the row space of B (and, therefore, in the row space of A). We may ensure this by selecting an $\mathbf{x}$ such that

$$A^+A\mathbf{x} = \mathbf{x}. \tag{4.74}$$

Premultiplying (4.70) by A^+ and applying (4.74) and (4.71), we again recover the answer $\mathbf{x} = A^+\mathbf{b}$, where A^+ must now be determined from the reduced SVD of A; that is, $A^+ = \underline{V}\underline{\Sigma}^{-1}\underline{U}^H$ (Fact 4.47).

4.7.1.1 The $\underline{QR}$ approach to inconsistent systems

If (A^HA) is invertible (that is, if $A = A_{m\times n}$ has rank n), there is a more economical way of finding the least-squares solution than by using A^+ or by applying Gaussian elimination to (4.72). Starting from (4.72) and performing the (reduced) decomposition $A = \underline{Q}\underline{R}$ described in §4.4.2 (noting that $\underline{R}$ also has rank n) we have

$$\underline{R}^H\underline{Q}^H\underline{Q}\underline{R}\mathbf{x} = \underline{R}^H\underline{Q}^H\mathbf{b} \quad \Rightarrow \quad \underline{R}\mathbf{x} = \underline{Q}^H\mathbf{b}. \tag{4.75}$$

Thus, if $A = A_{m\times n}$ has rank n and is inconsistent (with $m > n$), and we perform the decomposition $A = \underline{Q}\underline{R}$, we may satisfy $A\mathbf{x} = \mathbf{b}$ as well as possible by solving (4.75) instead of calculating $\mathbf{x} = A^+\mathbf{b}$. Recall that computing the $\underline{QR}$ decomposition is much less expensive than computing the SVD upon which the pseudoïnverse is based. Further, as $\underline{R}$ is triangular, the efficient backsubstitution technique (introduced in §2.1 and described in detail in §2.2.1) may be used to solve (4.75).

A significant benefit of the $\underline{QR}$ approach to this problem, over that given in (4.73), is that the $\underline{QR}$ approach effectively solves (4.72) without ever performing the product A^HA, thereby avoiding the loss of information caused by performing this product on a machine with finite-precision arithmetic.

[27] In the case that $m \geq n = r$ and thus (A^HA) is invertible, $B = A^+ = (A^HA)^{-1}A^H$ may also be referred to as a **left inverse** of the (square or tall) matrix $A = A_{m\times n}$, as $BA = I$. Alternatively, in the case that $n \geq m = r$ and thus (AA^H) is invertible, $C = A^H(AA^H)^{-1}$ may be referred to as a **right inverse** of the (square or fat) matrix $A = A_{m\times n}$, as $AC = I$. Note that the left inverse of tall matrices and the right inverse of fat matrices, when they exist, are not unique. In particular, any column vector in the nullspace of a fat matrix A (which necessarily has dimension greater than zero) may be added to any column of a right inverse C without changing the fact that $AC = I$. Similarly, any row vector in the left nullspace of the tall matrix A (which necessarily has dimension greater than zero) may be added to any row of a left inverse B without changing the fact that $BA = I$.

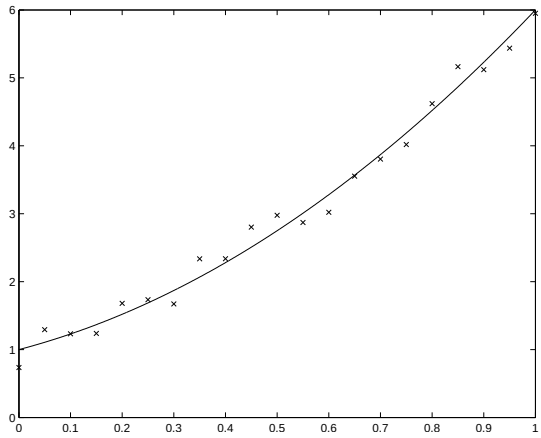


Figure 4.5: The data fitting problem [cf. the interpolation problem in Figure 7.11]: tune a small number of adjustable parameters to pass a smooth curve (———) as close as possible to the (perhaps, numerous) available datapoints (×).

4.7.2 The least-squares solution to the data fitting problem

The problem of **data fitting** may be described as the problem of adjusting a relatively small number, n, of undetermined coefficients in a relatively large number, m, of realizations of an equation in order to fit a proposed mathematical model to the available experimental data as accurately as possible. If the model is linear in these undetermined coefficients, such a problem may be written in the form (4.70), where $\mathbf{x}$ is the vector of undetermined coefficients, A and $\mathbf{b}$ are related to the data taken, and $\boldsymbol{\varepsilon}$ is a vector containing the (unknown) measurement errors; the least squares solution to this problem then gives one "fit" of the model to the data by minimizing the cost function $J = \|A\mathbf{x}-\mathbf{b}\|^2 = \|\boldsymbol{\varepsilon}\|^2$. For example, if an n'th-order polynomial model $y = a_0 + a_1 x + \ldots + a_n x^n$ is proposed to fit a set of $(m+1)$ datapoints $\{x_j, y_j\}$ for $j = 0, 1, \ldots, m$, as illustrated with $n = 2$ in Figure 4.5, then the problem can be written in the form

$$\begin{pmatrix} 1 & x_0 & \ldots & x_0^n \\ 1 & x_1 & \ldots & x_1^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & \ldots & x_m^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix} + \boldsymbol{\varepsilon} \quad \Leftrightarrow \quad A\mathbf{x} = \mathbf{b} + \boldsymbol{\varepsilon}.$$

If a sufficient amount of data is taken in such an experiment [that is, if $(m+1) > (n+1)$, where $(m+1)$ is the number of datapoints and $(n+1)$ is the number of coefficients in the model being adjusted], then A is tall [often, $m \gg n$] and, if the experiments are well chosen, $\text{rank}(A) = (n+1)$. That is, the system is inconsistent but not underdetermined, in which case the coefficients that minimize J, thereby reconciling the data with the model with the smallest additional term $\boldsymbol{\varepsilon}$, may be determined uniquely according to (4.73). If the number of coefficients to be determined is large, the most efficient way to solve this problem is the *QR* approach described in §4.7.1.1.

Weighted least squares

If some measurements are expected to be disrupted by more or less measurement error than others, or if the error of some measurements is expected to be correlated with the error of other measurements, we may account for this knowledge in the formulation of the data fitting problem, which we now denote $\overline{A}\mathbf{x} = \overline{\mathbf{b}} + \overline{\boldsymbol{\varepsilon}}$, by weighting the norm used in the cost function to be minimized such that $J = \|\overline{A}\mathbf{x} - \overline{\mathbf{b}}\|_Q^2 = \|\overline{\boldsymbol{\varepsilon}}\|_Q^2 = \overline{\boldsymbol{\varepsilon}}^H Q \overline{\boldsymbol{\varepsilon}}$,

where $Q = W^{-1}$ and W is the **covariance matrix** describing (that is, modeling) the expected statistics of measurement errors such that $W = \mathscr{E}[\overline{\boldsymbol{\varepsilon}}\,\overline{\boldsymbol{\varepsilon}}^H]$, where, again, $\mathscr{E}[\cdot]$ denotes the expectation operator indicating the average over a large number of statitical samples of the quantity in brackets. In the case that W is diagonal, the weighted cost function takes the simple form $J = \sum_{j=1}^{m} |\varepsilon_j|^2 / \mathscr{E}[|\varepsilon_j|^2]$, thereby reflecting the fact that such a formulation effectively "normalizes" the significance of each available measurement in the calculation by the inverse of the error expected in the measurement.

The weighted least squares problem may be solved in a straightforward manner simply by defining $A = Q^{1/2}\overline{A}$, $\mathbf{b} = Q^{1/2}\overline{\mathbf{b}}$, and $\boldsymbol{\varepsilon} = Q^{1/2}\overline{\boldsymbol{\varepsilon}}$, and determining $\mathbf{x}$ according to the unweighted least-squares algorithm described previously.

4.8 Chapter summary

Linear algebra forms the foundation upon which efficient methods may be built to solve many important classes of problems numerically. At first blush, the subject seems like it must certainly be quite simple and dull, as all it amounts to is the organization of addition, subtraction, multiplication, and division of blocks of numbers. Indeed, in a very real sense, that's all that linear algebra really is. However, upon further inspection, it is seen that this seemingly simple subject is in fact quite deep. The facts that

- not all systems of equations have any solutions at all,
- those that do might have multiple solutions,
- any solutions sought using a computer must be calculated using finite-precision arithmetic, and
- that arithmetic can sometimes take an unacceptably long time to complete,

are all facets of the explanation of why a deep understanding of linear algebra is both difficult and important to obtain. In particular, we find there are several natural ways to decompose a matrix into the product of other matrices with special structure (Hessenberg, tridiagonal, triangular, diagonal, unitary, etc.). These decompositions may in fact be put to very good use by numerical algorithms that use such matrices. The most important of these decompositions are the *LU*/*PLU*/Cholesky, Hessenberg, *QR*, Schur, real Schur, eigen, Jordan, and singular value decompositions, all of which the reader should become familiar with before proceeding. Various useful measures of matrices have also been presented, including the determinant, trace, matrix norms, and condition number. A wide variety of useful facts have also been noted along the way, each of which is used later in this text.

With this foundation set, we have seen that linear algebra can be used right away for a wide variety practical problems, including determining the modes of oscillation of a dynamic system and efficiently solving the problem $A\mathbf{x} = \mathbf{b}$ (even if this system is inconsistent and/or underdetermined).

The present chapter summarizes several topics and results from linear algebra which are used heavily in the remainder of this study. The reader will likely need to review this dense chapter multiple times in order to digest it completely; rest assured that this effort will not be in vain.

We conclude this chapter by remarking that the subject of numerical linear algebra is a rich and fascinating subject that may (indeed, should) be studied at a deeper level than the present succinct review of this subject can possibly achieve. Of particular interest is the analysis and quantification of both the rate of convergence and the accumulation of numerical errors by various schemes that have been proposed to solve the several linear-algebraic problems laid out in this chapter, and how these schemes may be implemented efficiently in modern parallel computer systems with cache-based memory architectures. It is hoped that the introduction provided here will help you to set your compass as you navigate your way through the vast and creative literature on this subject.

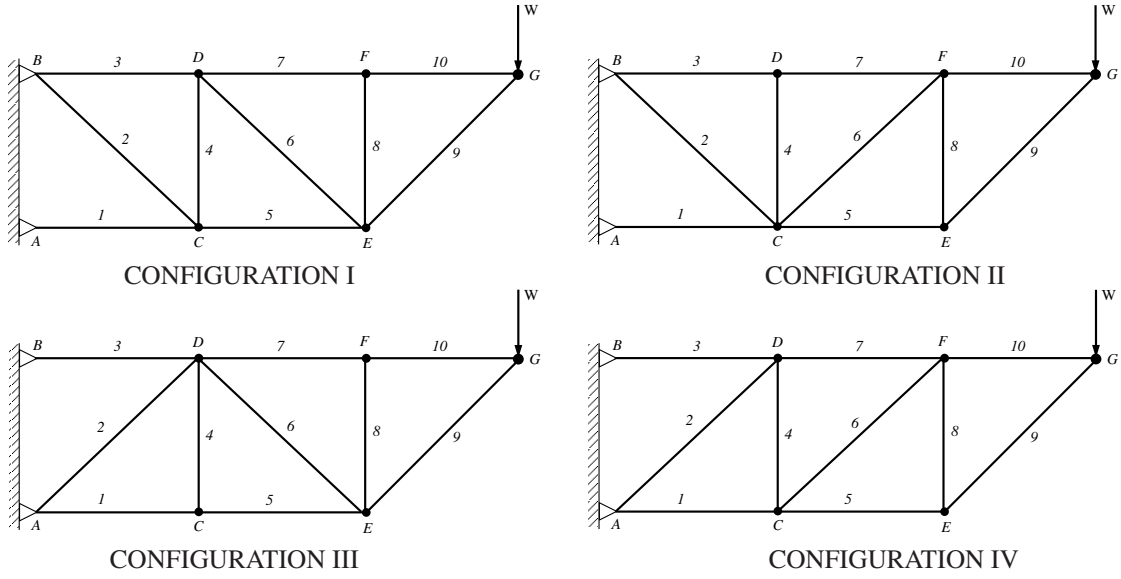


Figure 4.6: Cantilevered trusses attached at a vertical wall at $\{x_A, y_A\} = \{0,0\}$ and $\{x_B, y_B\} = \{0,1\}$, designed to support a weight at $\{x_G, y_G\} = \{3,1\}$. All structural members are assumed to be joined by frictionless pins and are assumed to be confined to a 2D plane (these idealizations simplify the computations). SI units are used throughout (forces in Newtons, distances in meters, etc.).

Exercises

Problems 4.1, 4.2, and 4.3 consider the static loading, displacement under load, and dynamic modes of vibration of a simple cantilevered truss designed to support a weight a fixed distance from a vertical wall, as illustrated in Figure 4.6. Choose one of the four configurations illustrated for all three of these questions (indicate clearly which one you are considering if turning this in for a class!).

Exercise 4.1 Static loading of the structure

We first determine the forces $\mathbf{f}$ in the truss when $W = 1000\ N$, assuming the displacement of each node is negligible and the mass of each member is negligible. We begin by writing the equations for static equilibrium of the loaded system. Note that the nodes C, D, E, F, and G are free to move in both the horizontal (x) and vertical (y) directions. At equilibrium, the sum of the forces in both the x and y directions at each of these nodes must be exactly zero. Define positive forces $f_i > 0$ as members under compression and negative forces $f_i < 0$ as members under tension. Also, define the 10 angles θ_i as the angles each rod makes from horizontal [e.g., noting (B.1), $\theta_9 = \text{atan2}(y_G - y_E, x_G - x_E)$]. The equations of **static equilibrium** are:

$$\text{forces at node C}: \begin{cases} \sum \text{Forces}_x = \ldots = 0, \\ \sum \text{Forces}_y = \ldots = 0, \end{cases} \qquad \text{forces at node F}: \begin{cases} \sum \text{Forces}_x = \ldots = 0, \\ \sum \text{Forces}_y = \ldots = 0, \end{cases}$$

$$\text{forces at node D}: \begin{cases} \sum \text{Forces}_x = \ldots = 0, \\ \sum \text{Forces}_y = \ldots = 0, \end{cases} \qquad \text{forces at node G}: \begin{cases} \sum \text{Forces}_x = f_9 \cos(\theta_9) + f_{10} \cos(\theta_{10}) = 0, \\ \sum \text{Forces}_y = f_9 \sin(\theta_9) + f_{10} \sin(\theta_{10}) = W. \end{cases}$$

$$\text{forces at node E}: \begin{cases} \sum \text{Forces}_x = \ldots = 0, \\ \sum \text{Forces}_y = \ldots = 0, \end{cases}$$

There are 10 unknowns in this system, $\{f_1,\ldots,f_{10}\}$. The system is **statically determinant**, meaning that the forces in the members may be determined by the conditions of static equilibrium. Assume that the nominal configuration of the structure is: $\{x_C,y_C\}=\{1,0\}$, $\{x_D,y_D\}=\{1,1\}$, $\{x_E,y_E\}=\{2,0\}$, $\{x_F,y_F\}=\{2,1\}$.

(**a**) Defining a geometry vector $\mathbf{g}=[x_C\ y_C\ x_D\ y_D\ x_E\ y_E\ x_F\ y_F]^T$, write a function `[f]=TrussForces(g)` which sets up the system of 10 equations listed above (in the order given) as $A\mathbf{f}=\mathbf{b}$, then solves this system for the member forces using one of the Gaussian elimination routines of §2. Is pivoting required to solve this system? For $\mathbf{g}=[1\ 0\ 1\ 1\ 2\ 0\ 2\ 1]^T$ and $W=1000$, what are the forces in each member? Compute the condition number of A. Do you trust this numerical result? Which members are under tension, and which are under compression? Physically, does this result make sense?

The members in the truss which bear tensile forces when the load is applied will be built with cables with negligible mass. The members which bear compressive forces will be built with `I` beams, with size (and mass) selected appropriately to bear the required load without failing, subject to an appropriate safety factor.

Exercise 4.2 Displacement of the structure under load

The displacements of the nodes from the nominal design, $\mathbf{x}$, due to both the applied load W and the acceleration due to gravity, are now computed. The displacements are assumed to be small, so the configuration of the structure is close to nominal (this assumption will be verified *a posteriori*). Note that Exercise 4.1 identified which members are `I` beams (with mass) and which members are tendons (with negligible mass) in the structure considered. The weight of the structure itself is modeled with a **lumped mass** approximation, accounting for half of the mass of each `I` beam as a point mass at each end of the beam. Stock beams are initially chosen with a mass per unit length of $\rho=5\,\text{kg/m}$ (this could be optimized once the nominal forces under load are calculated), and the acceleration due to gravity is $g=9.81\,\text{m/s}^2$. Assume that the mass per unit length of a member is $\rho_i=5\,\text{kg/m}$ if the member i is under compression ($f_i>0$) in the loaded structure, and $\rho_i=0\,\text{kg/m}$ if the member i is under tension ($f_i<0$) in the loaded structure.

The vector $\mathbf{x}$ is referred to as the **configuration** of the system. Together, $\mathbf{x}$ and $d\mathbf{x}/dt$ are referred to as the **state** of the system. When combined with the equation of motion, as derived in the following section, initial conditions on the state completely specify how the system will evolve in time. The **potential energy** PE of the system, due both to the applied external forces and the weight of the structure itself, the **deformation energy** DE of the system, due to the compression or extension of the (elastic) members, and the **kinetic energy** KE of the system, due to the motion of the members, are functions of $\mathbf{x}$ and $d\mathbf{x}/dt$. Our first task is to represent these energies in matrix form. Note that, when the (unmodeled) damping in the system causes the loaded structure to approach a static equilibrium, the **total energy** $TE=PE+DE+KE$ of the system is minimum and the kinetic energy $KE=0$.

A configuration vector $\mathbf{x}$ containing the displacements of the nodes with respect to their nominal positions, a mass vector $\mathbf{m}$ containing the lumped masses associated with the elements of $\mathbf{x}$, and a load vector $\mathbf{z}$ describing the forces associated with the elements of $\mathbf{x}$ may now be defined such that[28]

$$\mathbf{x}=\begin{pmatrix} x'_C\\ y'_C\\ x'_D\\ y'_D\\ x'_E\\ y'_E\\ x'_F\\ y'_F\\ x'_G\\ y'_G \end{pmatrix},\qquad \mathbf{m}=m_0+\frac{1}{2}\begin{pmatrix} \rho_1 l_1+\rho_2 l_2+\rho_4 l_4+\rho_5 l_5\\ \rho_1 l_1+\rho_2 l_2+\rho_4 l_4+\rho_5 l_5\\ \rho_3 l_3+\rho_4 l_4+\rho_6 l_6+\rho_7 l_7\\ \rho_3 l_3+\rho_4 l_4+\rho_6 l_6+\rho_7 l_7\\ \vdots \end{pmatrix},\qquad \mathbf{z}=\begin{pmatrix} 0\\ m_2 a\\ 0\\ m_4 a\\ 0\\ m_6 a\\ 0\\ m_8 a\\ 0\\ m_{10}a+W \end{pmatrix},$$

[28] Note that the mass vector shown is for Configuration I; the mass vectors for the other three configurations are slightly different.

where, for example, x'_C and y'_C denote (respectively) the horizontal and vertical displacements of node C from their nominal (unloaded) positions, and ρ_i is the mass per unit length of member i. Note that $m_0 = 0.1\,\text{kg}$ is the mass of the hardware at each joint. Based on these definitions, the potential energy due to displacements of the nodes (accounting both for the weight of the structure and for the applied external loads) may be expressed in matrix form as

$$PE = \mathbf{z}^T \mathbf{x}.$$

Note that upward (positive y'_i) displacements of the nodes result in increases of the potential energy.

The elastic deformation energy due to small deformations of the structure is

$$DE = \sum_{i=1}^{10} \frac{1}{2} c_i d_i^2 = \frac{1}{2} \mathbf{d}^T C \mathbf{d},$$

where $C = \text{diag}(\mathbf{c})$, c_i is the **spring constant** of member i, and d_i is the difference in length of member i from its nominal length ℓ_i. For the present analysis, assume that $c_i = 10^7/\ell_i\ \text{kg/s}^2$. Taking the leading terms only (that is, assuming small deformations), we have

$$d_1 = (x'_C - x'_A)\cos(\theta_1) + (y'_C - y'_A)\sin(\theta_1), \qquad \ldots$$

Applying the definition of $\mathbf{x}$ given above, we may write this system of equations in matrix form as

$$\mathbf{d} = D\mathbf{x},$$

where D is a 10×10 matrix with at most 4 nonzero entries per row. The deformation energy of the entire structure is

$$DE = \frac{1}{2} \mathbf{d}^T C \mathbf{d} = \frac{1}{2} \mathbf{x}^T D^T C D \mathbf{x} = \frac{1}{2} \mathbf{x}^T K \mathbf{x},$$

where $K = D^T C D$ is a symmetric 10×10 "stiffness matrix". Note that K is positive definite since (in this problem) for any nonzero set of displacements $\mathbf{x}$, DE is positive.

The total energy of the truss at equilibrium may now be written

$$TE = PE + DE + KE = \mathbf{z}^T \mathbf{x}_e + \frac{1}{2} \mathbf{x}_e^T K \mathbf{x}_e + 0.$$

(**a**) Minimizing the above expression for TE at equilibrium with respect to $\mathbf{x}_e$, show (using index notation) that equilibrium is achieved at

$$\mathbf{z} + K\mathbf{x}_e = 0.$$

Note that this equation may be solved for $\mathbf{x}_e$ once K and $\mathbf{z}$ are computed. The forces in the members of the perturbed structure at equilibrium are then given by $\mathbf{f}_e = -C\mathbf{d}_e = -CD\mathbf{x}_e$.

(**b**) Write a function `[K]=TrussK(g)` to compute K and a function `[xe,fe]=TrussLoading(g,z)` to compute the displacements $\mathbf{x}_e$ and the structural forces $\mathbf{f}_e$ at equilibrium due to the applied external loads $\mathbf{z}$. For $\mathbf{g} = [1\ 0\ 1\ 1\ 2\ 0\ 2\ 1]^T$, $W = 1000$, and the loads due to the weight of the structure (as discussed above), what are $\mathbf{x}_e$ and $\mathbf{f}_e$? Plot the shape of the displacement of the perturbed structure in Matlab, magnifying the perturbation enough in the plot so the shape of the perturbation may easily be seen (indicate how much such magnification is used). Is the "small perturbation" assumption mentioned previously valid?

Exercise 4.3 Vibration modes of the structure.

We now examine the modes of vibration of the structure. As the vibration problem is dynamic, we must now include the kinetic energy KE in the computation of the total energy. The kinetic energy of the entire structure may be expressed in matrix form as

$$KE = \frac{1}{2}\left[\frac{d\mathbf{x}}{dt}\right]^T M \frac{d\mathbf{x}}{dt},$$

where $M = \text{diag}(\mathbf{m})$. The potential and deformation energies, PE and DE, have the same form as in the previous section. The total energy of the structure in oscillation (neglecting damping) is:

$$TE = PE + DE + KE = \mathbf{z}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T K\mathbf{x} + \frac{1}{2}\left[\frac{d\mathbf{x}}{dt}\right]^T M \frac{d\mathbf{x}}{dt}.$$

(**a**) Setting the time derivative of the above expression for the total energy TE equal to zero for arbitrary $d\mathbf{x}/dt$, show (using index notation) that

$$M\frac{d^2\mathbf{x}}{dt^2} + K\mathbf{x} = -\mathbf{z}.$$

This is referred to as the **equation of motion** of our idealized, undamped system. The dynamics of the response do not depend on the steady-state deformation of the structure, which satisfy $K\mathbf{x}_e = -\mathbf{z}_e$, due to the applied loading $\mathbf{z}$. Thus, subtracting this equation from that given above, and defining $\mathbf{x}'(t) = \mathbf{x}(t) - \mathbf{x}_e$, we now focus on the dynamic modes $\mathbf{x}'(t)$ which satisfy the homogeneous equation

$$M\frac{d^2\mathbf{x}'(t)}{dt^2} + K\mathbf{x}'(t) = 0.$$

We can obtain useful information by extracting the frequencies and shapes of the normal modes of vibration in this idealization. Following the SOV approach, we seek modes of this second-order differential equation of the following form:

$$\mathbf{x}(t) = \mathbf{y}_\kappa e^{\mathrm{i}\omega_\kappa t}, \tag{4.76}$$

where ω_κ is the (temporal) frequency of vibration mode κ, and $\mathbf{s}_\kappa$ is the corresponding natural mode of vibration of the structure. This leads to a problem of the form

$$K\mathbf{y}_\kappa = \omega_\kappa^2 M\mathbf{y}_\kappa, \tag{4.77}$$

which is a **generalized eigenvalue problem** for the eigenvalues ω_κ^2 and the corresponding eigenvectors $\mathbf{y}_\kappa$; note that K and M are symmetric.

(**b**) Since the matrix M is nonsingular, one could simply multiply (4.77) from the left by M^{-1} to convert this problem into a regular eigenvalue problem; however, this would destroy the symmetry of the matrices involved, leading to a more difficult eigenvalue problem to solve. Instead, define an intermediate vector $\mathbf{s}$ such that $\mathbf{s} = T\mathbf{y}$. How should T be selected such that this definition transforms (4.77) into a regular eigenvalue problem of the form $A\mathbf{y} = \lambda\mathbf{y}$ where A is symmetric? What is the corresponding equation for A? Is it positive definite? What can you say about the eigenvalues λ_κ and the corresponding ω_κ (that is, are they real, imaginary, positive, ...)? Noting (4.76), does $\mathbf{x}'(t)$ oscillate in time? If so, do these oscillations decay or grow in time?

(**c**) For $\mathbf{g} = [1\ 0\ 1\ 1\ 2\ 0\ 2\ 1]^T$, find the natural modes of vibration of the structure $\mathbf{y}_\kappa = T^{-1}\mathbf{s}_\kappa$ and the corresponding frequencies of vibration ω_κ. Plot (in Matlab) the structure deformed into the five modes with the lowest frequencies. (These modes are usually the ones that are the least damped in the actual structure, and are thus generally the most significant in practice.) Discuss.

Exercise 4.4 The PDE governing a longitudinal wave in a 100 cm long aluminum bar is given by

$$\frac{\partial^2 f}{\partial t^2} = \frac{1}{\rho A}\frac{\partial}{\partial x}\left(EA\frac{\partial f}{\partial x}\right),$$

where the density $\rho = 2.7$ g/cm^3, Young's modulus $E = 6.9 \times 10^{11}$ dyne/cm^2, and the bar is assumed to be fixed at both ends. Consider three cases: (a) the area of the bar $A = 1$ cm^2 is constant; (b) the area of the bar varies linearly, from $A = 2$ cm^2 at one end to $A = 0$ at the other end; and (c) the area of the bar, shaped like a right circular cone, varies quadratically, from $A = 3$ cm^2 at one end to $A = 0$ at the other end. Calculate the first five mode shapes, and the corresponding frequencies of oscillation, in all five cases.

Exercise 4.5 Cyclic reduction.

(a) Recalling the checkerboard matrix illustrated in (3.9a), the rearrangement of this matrix illustrated in (3.9b), and leveraging, m times, the relation given in (4.2b), write a (serial) recursive code for solution of the problem $A\mathbf{x} = \mathbf{b}$ for tridiagonal A (without pivoting), via solution of 2^m smaller problems.

(b) Using the parallel Matlab coding constructs illustrated in Algorithm 2.12, rewrite the code written in part (a) so that it actually runs in parallel on 2^m threads on your machine. Time its execution while running in an appropriate number of threads; does parallelizing this code actually give you a decrease in run time for execution of the algorithm on large problems on your machine? Discuss.

(c) Rewrite the (serial) code developed in part (a) to solve directly, via 4^m smaller problems, the 2D Poisson problem set up in Example 3.3, in which A has the block tridiagonal Toeplitz structure illustrated in (1.6).

(d) Rewrite the code written in part (c) so that it actually runs in parallel on 4^m threads on your machine. Again, time its execution while running in an appropriate number of threads for your machine, and discuss.

Exercise 4.6 Convergence of the Jacobi and Gauss-Seidel methods (see §3.2.1).

(**a**) Writing (2.8) for the elements of the matrix (λA) rather than the elements of A, establish that, if A is strictly diagonally dominant and $|\lambda| \geq 1$, then $B \triangleq \lambda D + L + U$ is strictly diagonally dominant (and, thus, nonsingular) as well.

(**b**) If A is strictly diagonally dominant, note that the diagonal elements of $M \triangleq D$ are nonzero as a consequence of Fact 2.2. Consider now the characteristic polynomial of $P \triangleq -D^{-1}(L+U)$ and, leveraging part (a), establish that $|\lambda I - P| \neq 0$ if $|\lambda| \geq 1$. This implies that all eigenvalues λ of P satisfy $|\lambda| \leq 1$ and thus, by Fact 4.38, that $\|P\|_{i2} = \sigma_{\max}(P) < 1$ as well, thereby proving that the Jacobi method is convergent [see (3.7)] if A is strictly diagonally dominant.

(**c**) Repeat parts (a) and (b) taking $B \triangleq \lambda(D+L)+U$ and $C \triangleq D+L$ and $P \triangleq -(D+L)^{-1}U$, thereby proving that the Gauss-Seidel method is convergent if A is strictly diagonally dominant.

Exercise 4.7 Following closely the code `QRGivensHessenberg.m` in Algorithm 4.7, and the description on the top of page 88, write a streamlined Matlab function `[A]=QRGivensMinStorage(A)`, and test script `QRGivensMinStorageTest.m`, implementing a minimum storage variant of the Givens-based QR algorithm for full (not Hessenberg) matrices (hint: you have to order the rotations carefully!) and demonstrating that the information necessary to describe the entire resulting QR decomposition of A (that is, both Q and R) is efficiently contained in the modified value of the A matrix returned by this algorithm.

Exercise 4.8 The goal in this project is to better understand why, in general, solvers for the algebraic eigenvalue problem must be iterative.

(**a**) Consider first a 2×2 matrix A together with a Givens rotation matrix G such that

$$G^H = \begin{pmatrix} \overline{c} & -\overline{s} \\ s & c \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad G = \begin{pmatrix} c & \overline{s} \\ -s & \overline{c} \end{pmatrix}.$$

For simplicity, we will restrict our attention to real symmetric A; in this case, s and c are real and

$$c^2 + s^2 = 1. \tag{4.78}$$

Performing the product $B = G^H AG$ by hand, compute b_{21} in terms of the components of A and G. Taking $b_{21} = 0$, together with (4.78), you now have two simple nonlinear equations for the two unknowns c and s. Rearranging (4.78) to express s in terms of c, and inserting this expression into your formula for b_{21} gives a single equation for the unknown c. Set $b_{21} = 0$ and solve this equation, thereby determining precisely a rotation that makes $G^H AG$ diagonal. [Hint: note that there will be a couple of different cases to check, because the roots of a quadratic will be involved. However, one of these roots for c, and the corresponding s, should ultimately give a rotation G that will make $b_{21} = 0$.] Implement in code, and test on various randomly-generated real symmetric 2×2 matrices A.

(**b**) Now, based on the above method of calculating c and s to rotate a 2×2 matrix into a diagonal form, attempt to build a direct approach to compute all of the eigenvalues of an $n \times n$ real symmetric matrix A in a finite number of steps for any n (!). Start, of course, by taking a Hessenberg decomposition of the matrix A in order to reduce it to a tridiagonal form (in a finite number of steps). Then, attempt to compute just $n-1$ additional Givens rotations, using the results of part (A) appropriately embedded into identity matrices, in order to drive the entire first subdiagonal to zero. Such an idea might at first seem plausible, but clearly flies in the face of the Abel-Ruffini theorem. Find the flaw in this proposed idea, and clearly describe why it doesn't work.

Exercise 4.9 Show that (4.34b) follows from (4.34a) with $\delta \propto \varepsilon^2$, thus establishing that convergence of the shifted QR method is quadratic.

Exercise 4.10 Verify (4.36b).

Exercise 4.11 Noting the Implicit Q Theorem (Fact 4.36) and the development of the implicitly shifted QR iteration applied in Algorithm 4.13, modify Algorithm 4.12 to apply implicitly shifted QR iterations rather than explicit shifts. Apply as many techniques as possible to make this algorithm maximally efficient, and discuss each of them. Test your code on a wide variety of singular and nonsingular matrices to make sure it works. The resulting code will in fact be superior to Algorithm 4.12 for the Hermitian eigenvalue problem.

Exercise 4.12 Develop Chandrasekhar's method for the feedback control problem in (4.51a), in a manner analogous to Chandrasekhar's method for the estimation problem (4.51b), as developed in the text.

Exercise 4.13 Leveraging Algorithm 2.10 and the discussion in §4.2.2, write an efficient code to calculate the determinant of a large Circulant matrix.

Exercise 4.14 Extending Algorithm 4.24, write an efficient code to solve the Stein equation $X = A^H XB + C$.

Exercise 4.15 Verify algebraically all three of the forms given in (4.57).

Exercise 4.16 Recalling (1.20), Fact 1.12, and Fact 4.40, derive an expression providing a bound on the 2-norm condition number of both $A^H A$ and AA^H.

Exercise 4.17 Consider a complete SVD $A = U\Sigma V^H$ of (for convenience) a square matrix A, and the eigen decomposition

$$\mathscr{A}S = S\Lambda \quad \Leftrightarrow \mathscr{A} = S\Lambda S^{-1} \quad \text{where} \quad \mathscr{A} = \begin{bmatrix} 0 & A^H \\ A & 0 \end{bmatrix}.$$

(a) Is the matrix $\mathscr{A}$ Hermitian? What can you say about it's eigenvalues λ_i and eigenvectors $\mathbf{s}^i$?

(b) Determine S and Λ in terms of U, Σ, and V.

Hint: assume S and Λ may be written in the form $S = \begin{bmatrix} B & B \\ C & -C \end{bmatrix}$ and $\Lambda = \begin{bmatrix} D & 0 \\ 0 & -D \end{bmatrix}$, and determine $\{B, C, D\}$.

(c) Based on this formula, describe (in detail) a fourth construction of the SVD of A which is an alternative to the three constructions presented in §4.4.7.

(d) Is the eigen decomposition of $\mathscr{A}$ given above also an SVD of $\mathscr{A}$? Why or why not? If it is not, describe how the SVD of $\mathscr{A}$ may easily be constructed given the information presented thus far in this problem.

(e) Compute the 2-norm condition number of $\mathscr{A}$. Noting your answer to Exercise 4.16 above, discuss why the new method suggested in (c) might be superior to the SVD constructions based on the eigen decompositions of (AA^H) and (A^HA).

Exercise 4.18 Calculate (by hand) the Moore-Penrose pseudoïnverse of:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Exercise 4.19 Is a nilpotent matrix always singular? What property of the determinint establishes this fact most directly? Explain.

References

Amrein, WO, Andreas M. Hinz, AM, & Pearson, DB, editors (2005) *Sturm-Liouville Theory: Past and Present*. Birkhäuser.

Golub, GH, & Van Loan, CF (1996) *Matrix Computations*. Johns Hopkins.

Higham, NJ (2002) *Accuracy and Stability of Numerical Algorithms*. SIAM.

Horn, RA, & Johnson, CR (1990) *Matrix Analysis*. Cambridge.

Lancaster, P, & Tismenetsky, M (1985) *The Theory of Matrices*. Academic Press.

Pesic, P (2004) *Abel's proof: an essay on the sources and meaning of mathematical unsolvability*. MIT Press.

Postnikov, MM (2004) *Foundations of Galois Theory*. Dover.

Press, WH, Teukolsky, SA, Vetterling, WT, & Flannery, BP (2007). *Numerical Recipes, the Art of Scientific Computing*. Cambridge.

Stoer, J, & Bulirsch, R (1980) *Introduction to Numerical Analysis*. Springer-Verlag.

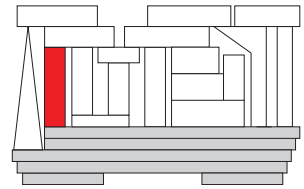
Strang, G (2006) *Linear Algebra and its Applications*. Brooks/Cole.

Trefethen, LN, & Bau, D (1997) *Numerical Linear Algebra*. SIAM.

Wilkenson, JH (1965) *The Algebraic Eigenvalue Problem*. Oxford.

Kailath, T (1973) Some New Algorithms for Recursive Estimation in Constant Linear Systems, *IEEE Trans. Information Theory*, **19**, 750-760.

Chapter 5



Spectral methods, fast transforms, and the Dirac delta

Contents

5.1 The orthogonality of sines, cosines, and complex exponentials . . . 143
5.2 Infinite Fourier series: continuous functions on bounded domains . . . 144
5.2.1 Differentiation using Fourier representations . . . 144
5.2.2 The Dirac delta on a bounded domain . . . 145
5.2.3 The completeness of the infinite Fourier series . . . 146
5.3 Infinite Fourier integral: continous functions on infinite domains . . . 147
5.3.1 The Dirac delta on an infinite domain . . . 147
5.3.2 The infinite Fourier integral expansion of a Gaussian function . . . 148
5.3.3 The Dirac delta on an infinite domain approximated as a Gaussian distribution . . . 149
5.3.4 The Dirac delta on an infinite domain approximated as a gamma distribution . . . 149
5.4 Finite Fourier series: discrete functions on bounded domains . . . 150
5.4.1 The fast Fourier transform (FFT) . . . 151
5.4.2 Properties of the coefficients of a finite Fourier series . . . 154
5.4.3 Approximating differentiation using a finite Fourier series . . . 157
5.5 Special properties of the Fourier expansions of real functions . . . 158
5.6 Convolution sums and nonlinear products . . . 161
5.7 Aliasing due to nonlinear products and the 2/3 dealiasing rule . . . 163
5.8 Two-point correlations and Parseval's theorem . . . 163
5.9 Fourier expansions of nonsmooth functions: Gibbs phenomenon . . . 164
5.9.1 Sobolev spaces and the quantification of smoothness . . . 166
5.10 Fourier series in multiple dimensions . . . 167
5.11 Sine series and cosine series . . . 168
5.11.1 The fast sine transform (FST) . . . 171
5.11.2 The fast cosine transform (FCT) . . . 172
5.12 Extending finite Fourier series to stretched grids . . . 174
5.13 Chebyshev representations . . . 176

5.14 Fourier-Bessel representations 180
Exercises 181

In many problems (see, e.g., §4.3.2), it is convenient to represent a function of a physical coördinate (space or time, denoted below as x) as a linear combination of an infinite number of smooth **basis functions** $b_n(x)$ or $b(k,x)$ that each satisfy simple boundary conditions, e.g.,

$$u(x) = \sum_n v_n b_n(x) \tag{5.1a}$$

$$u(x) = \int v(k)\, b(k,x)\, dk. \tag{5.1b}$$

As we will show, expansions of the form (5.1a) are appropriate for functions defined on **bounded** domains[1]. {e.g., $x \in [-L/2, L/2]$}, whereas expansions of the form (5.1b) are appropriate for functions defined on **semi-infinite** {e.g., $x \in [0,\infty)$} or **infinite** {e.g., $x \in (-\infty,\infty)$} domains. It is often said that $u(x)$ represents the function in **physical space**, whereas the coefficients v_n or $v(k)$ represent the function in **transform space**.

We are guaranteed that an expansion of such a form exists [stated precisely, for any $u(x)$ in an appropriate Hilbert space H, corresponding v_n or $v(k)$ may be found such that (5.1a) or (5.1b) is satisfied] if the basis functions are **complete** [that is, if the functions $b_n(x)$ or $b(k,x)$ form a **basis** for H]. This statement is analogous to the corresponding, perhaps more obvious statement in the finite-dimensional setting (see §1.3), and is established for the expansions of interest in the present work in §5.2.2.

Expansions of this form are especially convenient when the basis functions may be differentiated exactly, as the derivative of any linear combination $u(x)$ of these basis functions may then be determined immediately by taking, e.g., the derivative of (5.1a):

$$u'(x) = \sum_n v_n b_n'(x), \quad u''(x) = \sum_n v_n b_n''(x), \quad \text{etc.} \tag{5.2}$$

Further, expansions with **orthogonal basis functions** are particularly convenient; such basis functions satisfy the orthogonality property

$$\langle b_n(x), b_m(x) \rangle = \delta_{nm} \tag{5.3}$$

for an appropriately-defined continuous inner product $\langle \cdot, \cdot \rangle$. Using such a basis, the coefficients v_n in, e.g., the expansion (5.1a) are particularly easy to determine from $u(x)$: taking the inner product of (5.1a) with $b_m(x)$ and applying (5.3), it is seen immediately that

$$\left\langle \left[u(x) = \sum_n v_n b_n(x) \right], b_m(x) \right\rangle \quad \Rightarrow \quad v_m = \langle u(x), b_m(x) \rangle.$$

In order to make calculations based on such expansions numerically feasible, we ultimately restrict our attention to a finite-dimensional approximation of $u(x)$ evaluated at a suitable number of gridpoints x_j (appropriately distributed over the domain of interest), and an identical number of terms in the expansion,

$$u(x_j) \triangleq u_j = \sum_{n=1}^{N} v_n b_n(x_j) \qquad \text{for } j = 1 \ldots N. \tag{5.4}$$

Note that, even though we must truncate the series (5.1a) to handle it computationally, we may still leverage (5.2) to calculate derivatives when necessary.

To make such transform methods practical, an efficient algorithm to convert between the discretization of the function (u_j for $j = 1 \ldots N$) and the coefficients of the expansion (v_n for $n = 1 \ldots N$) is necessary. If the

[1] The notation $x \in [a,b)$ means x is considered between point a on the left and point b on the right, including a but not including b.

basis is in some way derived from complex exponential functions, such an algorithm is provided by the **fast Fourier transform** (**FFT**), as presented in §5.4.1. This chapter focuses on expansions that can, once discretized, be determined using this remarkable algorithm, commonly referred to as **spectral methods**. This chapter also weaves in the development of the Dirac delta, which is a very useful tool in engineering mathematics, especially in the derivation of numerical methods [see, e.g., the verification of the inverse Laplace transform formula (17.47b)]. The Dirac delta is a delicate and often misused construct; by presenting it in parallel with the Fourier transform, we hope to emphasize its interpretation in terms of its spectral decomposition.

Two additional transform techniques presented later in the text, the Laplace transform (§17.3) and the Z transform (§17.4), build on the theory of the Fourier transform and prove useful in the understanding of continuous-time linear systems and discrete-time linear systems.

As introduced above, **spectral methods** refer to expansions based in some way or another on complex exponential functions, leveraging (in the numerical setting) the FFT to transform between physical space and transform space. There are several variants of such transforms, which are described in detail in this chapter.

5.1 The orthogonality of sines, cosines, and complex exponentials

Complex exponential functions $e^{\mathrm{i}kx} = \cos(kx) + \mathrm{i}\sin(kx)$ with $\mathrm{i} = \sqrt{-1}$ (see Appendix A) and with appropriate choices for the **wavenumber** k form the foundation for spectral methods. Such functions obey two important orthogonality properties. In the spatially-continuous setting, for m, n integers and $k_n = 2\pi n/L$,

$$\frac{1}{L}\int_{-L/2}^{L/2} e^{\mathrm{i}k_n x} e^{-\mathrm{i}k_m x}\,dx = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{otherwise.} \end{cases} \tag{5.5a}$$

In the spatially-discrete setting, for m,n integers between $-N/2$ and $N/2$, $k_n = 2\pi n/L$, and $x_j = j(L/N)$ for $j = 0,\ldots,N-1$,

$$\frac{1}{N}\sum_{j=0}^{N-1} e^{\mathrm{i}k_n x_j} e^{-\mathrm{i}k_m x_j} = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{otherwise.} \end{cases} \tag{5.5b}$$

For sine functions, in the spatially-continuous setting with m, n integers and $k_n = 2\pi n/L$,

$$\frac{1}{L}\int_{-L/2}^{L/2} \sin(k_n x)\sin(k_m x)\,dx = \begin{cases} 1/2 & \text{if } n = m \neq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5.6a}$$

In the spatially-discrete setting, for m,n integers between $-N/2$ and $N/2$, $k_n = 2\pi n/L$, and $x_j = jL/(2N)$ for $j = 1,\ldots,N-1$,

$$\frac{1}{N}\sum_{j=1}^{N-1} \sin(k_n x_j)\sin(k_m x_j) = \begin{cases} 1/2 & \text{if } n = m \neq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5.6b}$$

For cosine functions, in the spatially-continuous setting with m, n integers and $k_n = 2\pi n/L$,

$$\frac{1}{L}\int_{-L/2}^{L/2} \cos(k_n x)\cos(k_m x)\,dx = \begin{cases} 1 & \text{if } n = m = 0 \\ 1/2 & \text{if } n = m \neq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{5.7a}$$

In the spatially-discrete setting, for m, n integers between $-N/2$ and $N/2$, $k_n = 2\pi n/L$, and $x_j = jL/(2N)$ for $j = 0,\ldots,N$,

$$\frac{1}{N}\sum_{j=0}^{N} \frac{1}{c_j}\cos(k_n x_j)\cos(k_m x_j) = \begin{cases} c_m/2 & \text{if } n = m \\ 0 & \text{otherwise,} \end{cases} \quad \text{with} \quad c_j \triangleq \begin{cases} 2 & \text{if } j = 0 \text{ or } j = N \\ 1 & \text{otherwise.} \end{cases} \tag{5.7b}$$

5.2 Infinite Fourier series: continuous functions on bounded domains

If $u(x)$ is a smooth, continuous, possibly complex function that is periodic on the interval $[-L/2, L/2]$ [that is, if $u(-L/2) = u(L/2)$], we may express $u(x)$ as an **infinite Fourier series** expansion of the form (5.1a) with complex exponentials as basis functions (with the wavenumbers k_n selected such that each basis function $b_n(x)$ itself satisfies the same periodic boundary conditions that $u(x)$ satisfies) such that

$$u(x) = \sum_{n=-\infty}^{\infty} \hat{u}_n e^{\mathrm{i}k_n x} \quad \text{with} \quad k_n = \frac{2\pi n}{L}, \tag{5.8a}$$

where the $\hat{u}_n$ are referred to as **Fourier series coefficients**. When the continuous function $u(x)$ is referenced outside the range $x \in [-L/2, L/2)$, a **periodic extension** of this function is assumed: $u(x + mL) = u(x)$ for $x \in [-L/2, L/2)$ and all integers m. Multiplying the above expression by $(1/L)e^{-\mathrm{i}k_m x}$ and integrating over the interval $(-L/2, L/2)$, assuming u is sufficiently smooth (specifically, that the magnitude of its Fourier series coefficients eventually decay exponentially with $|n|$) such that Fubini's theorem[2] applies, then applying (5.5a), we find that

$$\frac{1}{L}\int_{-L/2}^{L/2} \left[u(x) = \sum_{n=-\infty}^{\infty} \hat{u}_n e^{\mathrm{i}k_n x}\right] e^{-\mathrm{i}k_m x}\, dx \quad \Rightarrow \quad \hat{u}_m = \frac{1}{L}\int_{-L/2}^{L/2} u(x) e^{-\mathrm{i}k_m x}\, dx. \tag{5.8b}$$

We will refer to a **truncated Fourier series approximation** $u^M(x)$ of the function $u(x)$ as the continuous function given by the series expansion in (5.8a) with all Fourier coefficients outside $m \in [-M, M]$ set to zero; that is,

$$u^M(x) = \sum_{m=-M}^{M} \hat{u}_m e^{\mathrm{i}k_m x}. \tag{5.9}$$

5.2.1 Differentiation using Fourier representations

If we take $f_j = \frac{du}{dx}\big|_{x=x_j}$ and $g_j = \frac{d^2u}{dx^2}\big|_{x=x_j}$ and expand u_j, f_j, and g_j with an infinite Fourier series as defined in (5.8a), it follows that

$$f_j = \left.\frac{du}{dx}\right|_{x=x_j} = \sum_{n=-\infty}^{\infty} [\mathrm{i}k_n \hat{u}_n] e^{\mathrm{i}k_n x_j} = \sum_{n=-\infty}^{\infty} [\hat{f}_n] e^{\mathrm{i}k_n x_j} \quad \forall j \quad \Rightarrow \quad \hat{f}_n = \mathrm{i}k_n \hat{u}_n, \tag{5.10a}$$

$$g_j = \left.\frac{d^2u}{dx^2}\right|_{x=x_j} = \sum_{n=-\infty}^{\infty} [-k_n^2 \hat{u}_n] e^{\mathrm{i}k_n x_j} = \sum_{n=-\infty}^{\infty} [\hat{g}_n] e^{\mathrm{i}k_n x_j} \quad \forall j \quad \Rightarrow \quad \hat{g}_n = -k_n^2 \hat{u}_n. \tag{5.10b}$$

Thus, the *exact* derivative of an expression represented as an infinite Fourier series is straightforward to determine: the first derivative is obtained by multiplying the Fourier coefficients of the original expansion by $\mathrm{i}k_n$, the second derivative is obtained by multiplying the Fourier coefficients by $-k_n^2$, etc.

[2] **Fubini's theorem** states that, if $\int_A \int_B |f(x,y)|\, dy\, dx$ is finite for any bounded or unbounded domains A and B, then $\int_A \left(\int_B f(x,y)\, dy\right) dx = \int_B \left(\int_A f(x,y)\, dx\right) dy$ — that is, you can swap the order of integration. Note that, defining the dependence of the function f in x and/or y as piecewise constant, one or both of the integrals may be converted to sums and the same result applies. At every point in this chapter that we swap the order of integration and/or summation, we do so assuming the quantity being integrated or summed satisfies the necessary relation (usually, because it is sufficiently smooth) such that this theorem may be applied.

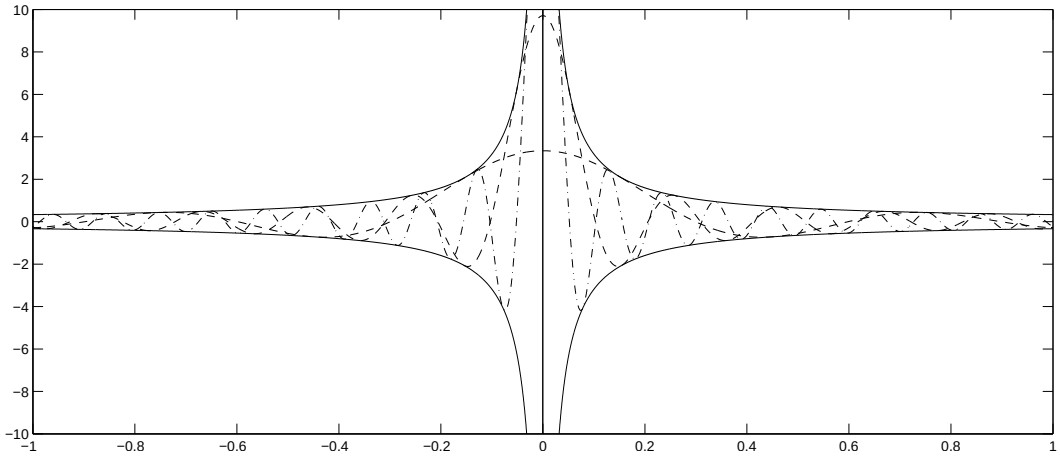


Figure 5.1: The function $\delta^M(x)$ for $L = 2\pi$ (plotted on $x \in [-1,1]$ for clarity of detail), taking $M = 10$ and $M = 30$ (dashed) and $M = 60$ (dot-dashed), and the envelope containing these oscillations (solid). Note that it is *not* correct to say that $\delta^M(x) \to 0$ as $M \to \infty$ for $x \neq 0$. However, it is true that, for any $\varepsilon > 0$, one can find a sufficiently large M such that any point x is within an ε neighborhood of a zero of the function $\delta^M(x)$; that is, as M is made large, the zeros of $\delta^M(x)$ become **dense**.

5.2.2 The Dirac delta on a bounded domain

Consider now the periodic function on the domain $x \in [-L/2, L/2]$ defined by[3]

$$\delta^M(x) \triangleq \sum_{m=-M}^{M} \frac{1}{L} e^{ik_m x} = \cdots = \frac{1}{L} \frac{\sin[(M+1/2)(2\pi x/L)]}{\sin(\pi x/L)}, \tag{5.11a}$$

as illustrated in Figure 5.1. Note that

$$\hat{\delta}_m^M = \begin{cases} 1/L & \text{for } -M \leq m \leq M \\ 0 & \text{otherwise;} \end{cases} \tag{5.11b}$$

that is, the function $\delta^M(x)$ is defined in terms of a truncated Fourier series expansion, with all of its nonzero Fourier coefficients equal. Note also the following

Fact 5.1 *For any $u(x)$ such that $u'(x)$ exists and is bounded on $x \in [a,b]$, $\int_a^b u(x)\sin(Mx)\,dx \xrightarrow[M\to\infty]{} 0$.*

Proof: Integrating by parts and taking the absolute value, it follows that

$$\int_a^b u(x)\sin(Mx)\,dx = \Big[-u(x)\frac{1}{M}\cos(Mx)\Big]_a^b + \frac{1}{M}\int_a^b u'(x)\cos(Mx)\,dx,$$

$$\Rightarrow \quad \Big|\int_a^b u(x)\sin(Mx)\,dx\Big| \leq \frac{1}{M}\Big[2\max_{x\in[a,b]}|u(x)| + |b-a|\max_{x\in[a,b]}|u'(x)|\Big] \xrightarrow[M\to\infty]{} 0. \qquad \square$$

[3] The form on the right follows after a minor amount of algebra and applying the identities (B.53) and (B.50); see Exercise 5.1.

As an easy corollary, it follows similarly that $\int_a^b u(x)\sin[(M+\alpha)(x+\beta)+\varepsilon]\,dx \xrightarrow[M\to\infty]{} 0$. With this fact established, it is easy to confirm that $\delta^M(x)$ satisfies the following three properties[4]

$$\int_{-L/2}^{L/2} \delta^M(x)\,dx = 1 \quad \text{for any } L \text{ and } M, \tag{5.12a}$$

$$\lim_{M\to\infty} \int_a^b \delta^M(x)dx = 0 \quad \text{for } a\cdot b > 0, \tag{5.12b}$$

$$\lim_{M\to\infty} \int_{-L/2}^{L/2} u(x')\,\delta^M(x-x')\,dx' \triangleq \int_{-L/2}^{L/2} u(x')\,\delta(x-x')\,dx' = u(x) \quad \text{for sufficiently smooth}^5\ u(x). \tag{5.12c}$$

That is,

(a) for any L and M, $\delta^M(x)$ is a function of unit area on $[-L/2, L/2]$,

(b) as $M\to\infty$, the integral of $\delta^M(x)$ over any interval that does *not* contain the origin approaches 0, and

(c) as $M\to\infty$, the integral of $\delta^M(x-x')$ multiplied by a smooth[5] function $u(x')$ approaches $u(x)$.

The often misused construct $\delta(x)$, called the **Dirac delta**, is defined in (5.12c) on the finite domain $\Omega = [-L/2, L/2]$ by considering the $M\to\infty$ limit of the action of the **approximating function** $\delta^M(x)$ under the integral sign. As developed according to this construction, note that you can only actually plot the approximating function $\delta^M(x)$ for some M, as illustrated in Figure 5.1; you can not plot the construct $\delta(x)$ itself, as $\delta^M(x)$ does *not* converge to a normal function as $M\to\infty$. That is, *the Dirac delta is not a function*, and thus should never be referred to as such (though many authors make this mistake). Stated precisely, the Dirac delta is referred to as a **distribution**[6] or **generalized function**; it is a construct that only makes sense when used inside an integral over its argument, as illustrated by its definition in (5.12c). This definition of the Dirac delta is generalized in §5.3.1 and §5.3.3 and discussed further in §5.9.1.

5.2.3 The completeness of the infinite Fourier series

The property of the completeness of the complex exponentials, which allows us to perform the expansion (5.8a) for any sufficiently smooth[5] $u(x)$, may be confirmed by substituting (5.8b) into the RHS of a truncated version of (5.8a), applying Fubini's theorem, and seeing that $u(x)$ is indeed recovered as the number of terms in the expansion is increased to infinity:

$$\begin{aligned}\lim_{M\to\infty} \sum_{m=-M}^{M} \Big[\frac{1}{L}\int_{-L/2}^{L/2} u(x')e^{-ik_m x'}\,dx'\Big] e^{ik_m x} &= \lim_{M\to\infty} \int_{-L/2}^{L/2} u(x')\Big[\sum_{m=-M}^{M} \frac{1}{L} e^{ik_m(x-x')}\Big]dx' \\ &= \lim_{M\to\infty} \int_{-L/2}^{L/2} u(x')\,\delta^M(x-x')\,dx' = u(x),\end{aligned} \tag{5.13}$$

thereby establishing the completeness of the infinite Fourier series expansion for the (sufficiently smooth) functions of interest in the present work.

[4] Property (5.12a) is easily verified by inserting the sum in (5.11a) into the LHS of (5.12a) and integrating, and property (5.12b) follows directly from Fact 5.1. Property (5.12c) then follows by noting that, due to (5.12b), the entire contribution to the integral in (5.12c) must come from the vicinity of $x-x'=0$, where the integrand is scaled by $u(x)$.

[5] That is, for any $u(x)$ such that $u'(x)$ exists and is bounded on the interval under consideration.

[6] A distribution such as $\delta(x)$ is defined precisely on a compact domain Ω in terms of its mapping from any **test function** $u\in C_\infty(\Omega)$ [that is, any smooth function $u(x)$ defined on Ω] into R; in particular, the Dirac delta $\delta(x)$ is defined here such that $\int_{-L/2}^{L/2} u(x)\delta(x)\,dx = u(0)$ for all test functions $u\in C_\infty([-L/2, L/2])$. The alternative definition/description given in this text, in terms of the approximating functions $\delta^M(x)$ as M is increased, help to provide further intuition concerning the properties of the Dirac delta.

5.3 Infinite Fourier integral: continous functions on infinite domains

Next, we consider the limit of the infinite Fourier series (5.8) as $L \to \infty$. To pass to this limit more easily, we rearrange the coefficients of (5.8a) and multiply (5.8b) by $L/(2\pi)$, resulting in

$$u(x) = \sum_{n=-\infty}^{\infty} \left[\frac{L\hat{u}_n}{2\pi}\right] e^{\mathrm{i}k_n x} \left[\frac{2\pi}{L}\right] \quad \text{with} \quad k_n = \frac{2\pi n}{L}, \tag{5.14a}$$

$$\left[\frac{L\hat{u}_m}{2\pi}\right] = \frac{1}{2\pi}\int_{-L/2}^{L/2} u(x) e^{-\mathrm{i}k_m x}\, dx. \tag{5.14b}$$

Defining a continuous **Fourier integral coefficient function** $\hat{u}(k)$ such that $\hat{u}(k = k_n) = L\hat{u}_n/2\pi$, and defining $\Delta k = k_n - k_{n-1} = 2\pi/L$, we may rewrite (5.14a) as

$$u(x) = \sum_{n=-\infty}^{\infty} \hat{u}(k_n) e^{\mathrm{i}k_n x} \Delta k \quad \text{with} \quad k_n = \frac{2\pi n}{L}.$$

Interpreting the above expression as a rectangular-rule approximation of an integral over k and taking the limit as $L \to \infty$ (and thus $\Delta k \to 0$), the sum converts to an integral, and we obtain [cf. (5.8)]

$$u(x) = \int_{-\infty}^{\infty} \hat{u}(k) e^{\mathrm{i}kx} dk \quad \text{for } x \in (-\infty, \infty), \tag{5.15a}$$

$$\hat{u}(k) = \frac{1}{2\pi}\int_{-\infty}^{\infty} u(x) e^{-\mathrm{i}kx}\, dx \quad \text{for } k \in (-\infty, \infty). \tag{5.15b}$$

This is best referred to as the **infinite Fourier integral** expansion, to distinguish it from the **finite Fourier integral** expansion developed in Exercise 5.2. However, since the latter is relatively uncommon, the above expansion is commonly (though a bit ambiguously) referred to simply as "the" **Fourier integral** expansion.

5.3.1 The Dirac delta on an infinite domain

We now apply the same sequence of operations described in §5.3 to the approximating function $\delta^M(x)$ (defined on the interval $x \in [-L/2, L/2]$ in §5.2.2) to create a new approximating function δ^K on $x \in (-\infty, \infty)$ with Fourier integral coefficient function $\hat{\delta}^K(k = k_m) = L\hat{\delta}^M_m/(2\pi) = 1/(2\pi)$ for $-M \le m \le M$ with $k_m = 2\pi m/L$. In particular, taking the limit that $L \to \infty$ and $M \to \infty$ in such a way that $K = 2\pi M/L$ remains constant results in a function $\delta^K(x)$ on the domain $x \in (-\infty, \infty)$ given by

$$\delta^K(x) \triangleq \int_{-K}^{K} \frac{1}{2\pi} e^{\mathrm{i}kx} dk \tag{5.16a}$$

with

$$\hat{\delta}^K(k) = \begin{cases} 1/(2\pi) & \text{for } -K \le k \le K, \\ 0 & \text{otherwise;} \end{cases} \tag{5.16b}$$

that is, the new approximating function $\delta^K(x)$ is defined in terms of its infinite Fourier integral expansion, with its Fourier integral coefficient function constant over the specified range. As in the bounded case, it is easy to verify that $\delta^K(x)$ satisfies the following properties:

$$\int_{-\infty}^{\infty} \delta^K(x)\, dx = 1 \quad \text{for any } K, \tag{5.17a}$$

$$\lim_{K\to\infty} \int_a^b \delta^K(x) dx = 0 \quad \text{for } a \cdot b > 0, \tag{5.17b}$$

$$\lim_{K\to\infty} \int_{-\infty}^{\infty} u(x')\, \delta^K(x - x')\, dx' \triangleq \int_{-\infty}^{\infty} u(x')\, \delta(x - x')\, dx' = u(x) \quad \text{for sufficiently smooth } u(x). \tag{5.17c}$$

That is,

(a) for any K, $\delta^K(x)$ is a function of unit area,

(b) as $K \to \infty$, the integral of $\delta^K(x)$ over any interval that does not contain the origin approaches 0, and

(c) as $K \to \infty$, the integral of $\delta^K(x-x')$ multiplied by a smooth function $u(x')$ approaches $u(x)$.

As in the case of the finite domain considered in §5.2.2, the construct $\delta(x)$ on the infinite domain $\Omega = (-\infty, \infty)$ may be defined as shown in (5.17c) by considering the $K \to \infty$ limit of the action of the approximating function $\delta^K(x)$ under the integral sign. Again, $\delta(x)$ is not a function, and only makes sense when kept inside an integral over its argument.

As K is made large in (5.16b), it is seen that the spectral content of $\delta^K(x-x')$ (or a superposition of many such functions) is equal over a broad range of frequencies; this is akin to the fact that white light has an approximately uniform spectral content over the visible spectrum. Thus, a random signal $w(x) = \sum_{x'} c(x')\delta^K(x-x')$ or $w(x) = \int_{-\infty}^{\infty} c(x')\delta^K(x-x')\,dx'$ created as a superposition of many functions $\delta^K(x-x')$ with random coefficients $c(x')$ is often referred to as **white noise**. Note that, mathematically, white noise is something of an idealization in continuous time, as the functions $\delta^K(x-x')$ upon which it is based must take K as large but finite in order for the result to be a bounded function.

5.3.2 The infinite Fourier integral expansion of a Gaussian function

We now consider the infinite Fourier integral expansion of the **Gaussian function**

$$u(x) = Ce^{-x^2/(2\sigma^2)} \quad \text{for } x \in (-\infty, \infty). \tag{5.18a}$$

Taking d/dx of this equation, then computing its infinite Fourier integral expansion as in (5.15b), we may manipulate this expression as follows:

$$\frac{1}{2\pi}\int_{-\infty}^{\infty}\left\{\frac{d}{dx}\left(u(x) = Ce^{-x^2/(2\sigma^2)}\right)\right\}e^{-\mathrm{i}kx}dx \quad \Rightarrow$$

$$\mathrm{i}k\hat{u}(k) = \frac{-1}{2\pi}\int_{-\infty}^{\infty}\left\{C\frac{x}{\sigma^2}e^{-\frac{x^2}{2\sigma^2}}\right\}e^{-\mathrm{i}kx}dx = \frac{-1}{\sigma^2 2\pi}\int_{-\infty}^{\infty}xu(x)e^{-\mathrm{i}kx}dx = \frac{-\mathrm{i}}{\sigma^2}\frac{d}{dk}\left[\frac{1}{2\pi}\int_{-\infty}^{\infty}u(x)e^{-\mathrm{i}kx}dx\right] = \frac{-\mathrm{i}}{\sigma^2}\frac{d}{dk}\hat{u}(k)$$

$$\Rightarrow \quad \int_0^k\left[\frac{\frac{d}{dk'}\hat{u}(k')}{\hat{u}(k')} = -k'\sigma^2\right]dk' \quad \Rightarrow \quad \exp\left[\log\hat{u}(k) - \log\hat{u}(0) = -\frac{\sigma^2k^2}{2}\right] \quad \Rightarrow \quad \frac{\hat{u}(k)}{\hat{u}(0)} = e^{-\sigma^2k^2/2}.$$

By (5.18a) and (5.15b)[7], $\hat{u}(0) = \frac{C}{2\pi}\int_{-\infty}^{\infty}e^{-x^2/(2\sigma^2)}\,dx = \frac{C\sigma}{\pi\sqrt{2}}\int_{-\infty}^{\infty}e^{-y^2}\,dy = C\sigma/\sqrt{2\pi}$, and thus

$$\hat{u}(k) = \frac{C\sigma}{\sqrt{2\pi}}e^{-\sigma^2k^2/2} \quad \text{for } k \in (-\infty, \infty). \tag{5.18b}$$

Note that σ^2 is in the denominator of the exponent in (5.18a), but the numerator of the exponent in (5.18b). To summarize,

Fact 5.2 *The Fourier transform of a Gaussian function $u(x)$ is itself a Gaussian function $\hat{u}(k)$. Moreover, the narrower the width of $u(x)$, the broader the width of $\hat{u}(k)$.*

[7] Note that the integral of a Gaussian, $s = \int_{-\infty}^{\infty}e^{-x^2}dx$, may be determined as follows:

$$s^2 = \left(\int_{-\infty}^{\infty}e^{-x^2}dx\right)\left(\int_{-\infty}^{\infty}e^{-y^2}dy\right) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}e^{-(x^2+y^2)}dxdy = \int_0^{2\pi}\int_0^{\infty}e^{-r^2}rdrd\theta = 2\pi\left[\frac{-e^{-r^2}}{2}\right]_0^{\infty} = \pi, \quad \Rightarrow \quad s = \sqrt{\pi}.$$

5.3.3 The Dirac delta on an infinite domain approximated as a Gaussian distribution

It is easy to verify that the function $\delta^\sigma(x) \triangleq \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/(2\sigma^2)}$ satisfies the following three properties [cf. (5.17)]:

$$\int_{-\infty}^{\infty} \delta^\sigma(x)\,dx = 1 \quad \text{for any } \sigma > 0, \tag{5.19a}$$

$$\lim_{\sigma\to 0} \int_a^b \delta^\sigma(x)dx = 0 \quad \text{for } a\cdot b > 0, \tag{5.19b}$$

$$\lim_{\sigma\to 0} \int_{-\infty}^{\infty} u(x')\,\delta^\sigma(x-x')\,dx' \triangleq \int_{-\infty}^{\infty} u(x')\,\delta(x-x')\,dx' = u(x) \quad \text{for sufficiently smooth } u(x). \tag{5.19c}$$

That is,

(a) for any σ, $\delta^\sigma(x)$ is a function of unit area,

(b) as $\sigma \to 0$, the integral of $\delta^\sigma(x)$ over any interval that does not contain the origin approaches 0, and

(c) as $\sigma \to 0$, the integral of $\delta^\sigma(x-x')$ multiplied by a smooth function $u(x')$ approaches $u(x)$.

As an alternative to (5.17), the construct $\delta(x)$ on the infinite domain $\Omega = (-\infty,\infty)$ may instead be defined as in (5.19), by considering the $\sigma \to 0$ limit of the action of the approximating function $\delta^\sigma(x)$ under the integral sign. For small σ, the approximating function $\delta^\sigma(x)$ is a tall, narrow Gaussian "bump" of unit area that is symmetric around the origin; we thus refer to $\delta^\sigma(x)$, for small but finite σ, as a **continuous two-sided unit impulse**. Again, $\delta(x)$ is not a function, and only makes sense when kept inside an integral over its argument, as $\delta^\sigma(x)$ does *not* converge to a regular function as $\sigma \to 0$. By (5.18b),

$$\hat{\delta}^\sigma(k) = e^{-\sigma^2 k^2/2}/(2\pi); \tag{5.20}$$

note in particular that, as the width of $\delta^\sigma(x)$ narrows as σ is reduced, the width of $\hat{\delta}^\sigma(k)$ broadens, approaching $\hat{\delta}^\sigma(k) = 1/(2\pi)$ over an increasingly broad range of frequencies [cf. (5.16b)].

5.3.4 The Dirac delta on an infinite domain approximated as a gamma distribution

Noting (B.69a), it is also easy to verify that the function $\delta^{\lambda,m}(x) \triangleq h_1(x)\lambda^m x^{m-1} e^{-\lambda x}/\Gamma(m)$, for $\lambda > 0$ and any integer $m \geq 0$, satisfies the same three properties:

$$\int_{-\infty}^{\infty} \delta^{\lambda,m}(x)\,dx = 1 \quad \text{for any } \lambda > 0, \tag{5.21a}$$

$$\lim_{\lambda\to 0} \int_a^b \delta^{\lambda,m}(x)dx = 0 \quad \text{for } a\cdot b > 0, \tag{5.21b}$$

$$\lim_{\lambda\to 0} \int_{-\infty}^{\infty} u(x')\,\delta^{\lambda,m}(x-x')\,dx' \triangleq \int_{-\infty}^{\infty} u(x')\,\delta(x-x')\,dx' = u(x) \quad \text{for sufficiently smooth } u(x). \tag{5.21c}$$

That is,

(a) for any $\lambda > 0$ (and any integer $m \geq 0$), $\delta^{\lambda,m}(x)$ is a function of unit area,

(b) as $\lambda \to \infty$, the integral of $\delta^{\lambda,m}(x)$ over any interval that does not contain the origin approaches 0, and

(c) as $\lambda \to \infty$, the integral of $\delta^{\lambda,m}(x-x')$ multiplied by a smooth function $u(x')$ approaches $u(x)$.

As alternative to (5.17) and (5.19), $\delta(x)$ on the infinite domain $\Omega = (-\infty,\infty)$ may instead be defined as in (5.21), by considering the $\lambda \to \infty$ limit of the action of the approximating function $\delta^{\lambda,m}(x)$ under the integral sign for any integer $m \geq 0$. For large λ and any integer $m \geq 0$, the approximating function $\delta^{\lambda,m}(x)$ is a tall, narrow "bump" of unit area that is zero to the left of origin and infinitely differentiable to the right of the origin; further, for $m > 0$, the function and its first $m-1$ derivatives are continuous at the origin. We thus refer to $\delta^{\lambda,m}(x)$, for any integer $m \geq 0$ and for large but finite λ, as a **continuous one-sided unit impulse**.

Again, $\delta(x)$ is not a function, and only makes sense when kept inside an integral over its argument, as $\delta^{\lambda,m}(x)$ does not converge to a regular function as $\lambda \to \infty$. It is straightforward to verify for any integer $m > 0$ that

$$\hat{\delta}^{\lambda,m}(k) = \frac{1}{2\pi} \frac{\lambda^m}{(\lambda + \mathrm{i}k)^m}; \tag{5.22}$$

note in particular that, as the width of $\delta^{\lambda,m}(x)$ narrows as λ is increased, the width of $\hat{\delta}^{\lambda,m}(k)$ broadens, approaching $\hat{\delta}^{\lambda,m}(k) = 1/(2\pi)$ over an increasingly broad range of frequencies [cf. (5.16b) and (5.20)].

The key point is this: *the integral properties in* (5.17) *and* (5.19) *and* (5.21) *which define the Dirac delta are identical, though the approximating functions* δ^K *and* δ^σ *and* $\delta^{\lambda,m}$ *look quite different. It is its integral properties that define the Dirac delta, and, in this regard, the three constructions are equivalent.*

5.4 Finite Fourier series: discrete functions on bounded domains

As mentioned in the introduction, in order to make the Fourier series expansion numerically feasible, the logical thing to do is to discretize the continuous periodic function $u(x)$ considered in (5.8) on a finite number of gridpoints, which we will space equally over the domain[8] $x \in [0, L)$ such that $x_j = jL/N$ for $j = 0 \ldots N-1$. *Note that, in the remainder of this chapter, for convenience, all vectors and matrices will be indexed from* 0. As with the continuous function $u(x)$ expanded by the infinite Fourier series in §5.2, this discrete function $u_j = u(x_j)$ is also assumed to be periodic (that is, $u_0 = u_N$); when the discrete function u_j is referenced outside the range $j \in [0, N-1]$, a periodic extension of this function is assumed: $u_{j+mN} = u_j$ for $j \in [0, \ldots, N-1]$ and all integers m. We may express this discrete function as a **finite Fourier series** (a.k.a. **discrete Fourier series**) with N complex coefficients[9] times mutually orthogonal complex exponential functions such that [cf. (5.8a)]

$$u_j = \sum_{n=0}^{N-1} \hat{u}_n e^{\mathrm{i}k_n x_j} = \sum_{n=0}^{N-1} \hat{u}_n e^{\mathrm{i}2\pi jn/N} \quad \text{with} \quad k_n = \frac{2\pi n}{L}. \tag{5.23a}$$

Multiplying the above expression by $(1/N)e^{-\mathrm{i}k_m x_j}$ and summing over the values $j = 0, \ldots, N-1$, applying in this case (5.5b), we find that [cf. (5.8b)]

$$\frac{1}{N} \sum_{j=0}^{N-1} \left[u_j = \sum_{n=0}^{N-1} \hat{u}_n e^{\mathrm{i}k_n x} \right] e^{-\mathrm{i}k_m x} \quad \Rightarrow \quad \hat{u}_m = \frac{1}{N} \sum_{j=0}^{N-1} u_j e^{-\mathrm{i}k_m x_j} = \frac{1}{N} \sum_{j=0}^{N-1} u_j e^{-\mathrm{i}2\pi jm/N}. \tag{5.23b}$$

The determination of the Fourier coefficients $\hat{u}_m$ from the function values u_j, which may be found using the relation given in (5.23b), is referred to as taking the **Fourier transform** (that is, it is said that $\hat{\mathbf{u}}$ is the Fourier transform of $\mathbf{u}$). Conversely, the determination of the function values u_j from the Fourier coefficients $\hat{u}_n$, which may be found using the relation given in (5.23a), is referred to as taking the **inverse Fourier transform** (that is, it is said that $\mathbf{u}$ is the inverse Fourier transform of $\hat{\mathbf{u}}$). The vector $\mathbf{u}$ is referred to as a **physical space** representation, whereas the vector $\hat{\mathbf{u}}$ is referred to as a **Fourier space** representation. Note that the formula to compute the Fourier transform [that is, the equation to determine the $\hat{u}_m$ given the u_j, as given in (5.23b)] is almost exactly the same as the formula to compute the inverse Fourier transform [that is, the equation to determine the u_j given the $\hat{u}_m$, as given in (5.23a)]. Thus,

Fact 5.3 *The same numerical code can be used for both the forward and inverse Fourier transform; to calculate a forward Fourier transform using a code that calculates an inverse Fourier transform, simply replace the Fourier coefficients* $\hat{u}_n$ *in the function call with the function values* u_n*, replace* i *by* $-i$ *in the algorithm, and scale the result by* $1/N$.

[8] For convenience, we have shifted the domain over which we consider $u(x)$ to $x \in [0, L)$; due to its assumed periodicity, this is equivalent to considering it over $x \in [-L/2, L/2)$.

[9] Note the **conservation of information** in the representation of N complex function values u_j in terms of N complex coefficients $\hat{u}_n$ in the finite Fourier series. The special case of real functions u_j is considered in §5.5.

Note that the finite Fourier series relationships given in (5.23) may be written in matrix form for any positive integer N as[10]

$$\mathbf{u} = F\hat{\mathbf{u}} \quad \text{with} \quad f_{jn} = e^{\mathrm{i}k_n x_j} = w_N^{jn} \quad \text{where} \quad w_N = e^{\mathrm{i}2\pi/N}, \quad \text{and} \tag{5.24a}$$

$$\hat{\mathbf{u}} = G\mathbf{u} \quad \text{with} \quad g_{mj} = \frac{1}{N} e^{-\mathrm{i}k_m x_j} = \frac{1}{N} \overline{w}_N^{mj}. \tag{5.24b}$$

The matrices F and G are often referred to as **discrete Fourier transform** (**DFT**) matrices. Note that, by the definitions in (5.24) and (5.5b), $GF = FG = I$; that is, $G = F^{-1} = F^H/N$. Note also that we often denote w, F, and G with the subscript N to indicate the size of the transform we imply. In matrix form, applying the facts that $w_N^N = 1$, $w_N^{N/2} = -1$, and $w_N^{N/4} = \mathrm{i}$ (see Appendix B) and writing $w = w_8$, we may write F_N in the cases $N = 2$, $N = 4$, and $N = 8$ as

$$F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \end{pmatrix}, \quad F_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w & w^4 & w^7 & w^2 & w^5 \\ 1 & w^4 & 1 & w^4 & 1 & w^4 & 1 & w^4 \\ 1 & w^5 & w^2 & w^7 & w^4 & w & w^6 & w^3 \\ 1 & w^6 & w^4 & w^2 & 1 & w^6 & w^4 & w^2 \\ 1 & w^7 & w^6 & w^5 & w^4 & w^3 & w^2 & w \end{pmatrix}.$$

Stating (5.24) in words, one technique to convert the vector of physical space function values, $\mathbf{u}$, to the vector of Fourier coefficients, $\hat{\mathbf{u}}$, is simply to premultiply the vector $\mathbf{u}$ by the matrix G; to convert back to physical space, premultiply $\hat{\mathbf{u}}$ by F. Each of these (full) matrix/vector products costs $\sim N^2$ complex multiplications and $\sim N^2$ complex additions, which is equivalent[11] to $\sim 8N^2$ real flops. A much faster numerical algorithm to compute both of these transforms, called the fast Fourier transform, is presented in the following section.

5.4.1 The fast Fourier transform (FFT)

The discrete Fourier transform matrices F and G in (5.24) are full but exhibit a very distinct structure. By leveraging this structure [as done, e.g., in the Householder matrix/vector product given in (1.11)], discrete Fourier transforms can be computed via an algorithm that is much cheaper than full matrix/vector multiplication. Such an algorithm is referred to as a **fast Fourier transform** (**FFT**). For $N = 2^s$, the structure of the matrix F_N (or G_N) becomes apparent by rearranging it (via postmultiplication by a permutation matrix P_N^{eo}; see §1.2.5) to list its even columns first, followed by its odd columns. For example,

$$F_4 P_4^{eo} = \left(\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & \mathrm{i} & -\mathrm{i} \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & -\mathrm{i} & \mathrm{i} \end{array}\right) = \begin{bmatrix} F_2 & \Omega_2 F_2 \\ F_2 & -\Omega_2 F_2 \end{bmatrix} = \begin{bmatrix} I & \Omega_2 \\ I & -\Omega_2 \end{bmatrix} \begin{bmatrix} F_2 & 0 \\ 0 & F_2 \end{bmatrix},$$

[10] Note that there is flexibility in the definition of the Fourier series and Fourier integral expansions. In particular, the exponent in the complex exponential may be either positive or negative in (5.8a), (5.15a), and (5.23a), with the opposite sign chosen in the complex exponential in (5.8b), (5.15b), and (5.23b). Also, the product of the coefficient of the sum in (5.8a) and the coefficient of the integral in (5.8b) needs to be $1/L$, the product of the coefficient of the integral in (5.15a) and the coefficient of the integral in (5.15b) needs to be $1/(2\pi)$, and the product of the coefficient of the sum in (5.23a) and the coefficient of the sum in (5.23b) needs to be $1/N$; it is sometimes more convenient algebraically to make these formulae more symmetric by taking the coefficients in both (5.8a) and (5.8b) as $1/\sqrt{L}$, the coefficients in both (5.15a) and (5.15b) as $1/\sqrt{2\pi}$, and the coefficients in both (5.23a) and (5.23b) as $1/\sqrt{N}$; the latter case makes the matrix F unitary in (5.24) (that is, $G = F^{-1} = F^H$). *Be aware that different authors use different conventions.*

[11] Note that a complex addition costs two real flops [that is, $(a+b\mathrm{i})+(c+d\mathrm{i}) = e+f\mathrm{i}$ where $e = a+c$ and $f = b+d$], and a complex multiplication costs six real flops [that is, $(a+b\mathrm{i})(c+d\mathrm{i}) = g+h\mathrm{i}$ where $g = ac-bd$ and $h = ad+bc$].

and, again denoting $w = w_8$,

$$F_8 P_8^{eo} = \left(\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} & w & w^3 & w^5 & w^7 \\ 1 & -1 & 1 & -1 & w^2 & w^6 & w^2 & w^6 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} & w^3 & w & w^7 & w^5 \\ \hline 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} & -w & -w^3 & -w^5 & -w^7 \\ 1 & -1 & 1 & -1 & -w^2 & -w^6 & -w^2 & -w^6 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} & -w^3 & -w & -w^7 & -w^5 \end{array}\right) = \begin{bmatrix} F_4 & \Omega_4 F_4 \\ F_4 & -\Omega_4 F_4 \end{bmatrix} = \begin{bmatrix} I & \Omega_4 \\ I & -\Omega_4 \end{bmatrix} \begin{bmatrix} F_4 & 0 \\ 0 & F_4 \end{bmatrix},$$

where

$$\Omega_2 \triangleq \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad \Omega_4 \triangleq \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & w & 0 & 0 \\ 0 & 0 & w^2 & 0 \\ 0 & 0 & 0 & w^3 \end{pmatrix}, \quad \text{etc.},$$

and

$$P_4^{eo} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad P_8^{eo} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad \text{etc.}$$

In general,

$$F_N P_N^{eo} = \begin{bmatrix} F_{N/2} & \Omega_{N/2} F_{N/2} \\ F_{N/2} & -\Omega_{N/2} F_{N/2} \end{bmatrix} = \begin{bmatrix} I & I \\ I & -I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & \Omega_{N/2} \end{bmatrix} \begin{bmatrix} F_{N/2} & 0 \\ 0 & F_{N/2} \end{bmatrix},$$

where $\Omega_{N/2} = \mathrm{diag}([1, w_N, w_N^2, w_N^3, \ldots, w_N^{N/2-1}])$ and P_N^{eo} is the permutation matrix such that the product $F_N P_N^{eo}$ reorders all of the even columns of F_N first, then all of the odd columns (and, therefore, the product $(P_N^{eo})^T \mathbf{x}$ reorders all of the even rows of $\mathbf{x}$ first, then all of the odd rows).

Based on these relationships, note that $F_N \mathbf{x}$ may be determined from $F_{N/2}\mathbf{x}^e$ and $F_{N/2}\mathbf{x}^o$, where $\mathbf{x}^e$ and $\mathbf{x}^o$ are the vectors containing the even and odd elements of $\mathbf{x}$, according to the following important identity:

$$F_N \mathbf{x} = \{F_N P_N^{eo}\}\{(P_N^{eo})^T \mathbf{x}\} = \underbrace{\begin{bmatrix} I & I \\ I & -I \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} I & 0 \\ 0 & \Omega_{N/2} \end{bmatrix}}_{B} \underbrace{\begin{bmatrix} F_{N/2}\mathbf{x}^e \\ F_{N/2}\mathbf{x}^o \end{bmatrix}}_{\mathbf{c}}. \tag{5.25}$$

This calculation requires only $\sim N/2$ complex multiplications (to calculate $B\mathbf{c}$) and $\sim N$ complex additions (to calculate A times the result); that is, it requires $\sim 5N$ real flops. The two vectors $F_{N/2}\mathbf{x}^e$ and $F_{N/2}\mathbf{x}^o$, in turn, may each be determined based on $F_{N/4}$ FFT's applied to the vectors containing their own even and odd elements, etc., for a total of $\log_2(N) = s$ stages. Each stage of this calculation requires $\sim 5N$ real flops, so the total cost is $\sim 5Ns = 5N\log_2(N)$ real flops[12,13]. For $N = 2^3 = 8$, there are three such stages, and the resulting

[12] The difference between the $\sim 5N\log_2(N)$ cost of the FFT and the $\sim 8N^2$ cost of the equivalent full matrix/vector multiplication, as descripted at the end of the previous section, is quite significant. For $N = 16$, the latter requires over 6 times as many flops as the former. For $N = 1024$, this factor is over 160, and for larger N, it is even greater.

[13] Note also that a few more flops can in fact be shaved off by writing special algorithms for the first two stages (furthest to the left in Figure 5.2), where all of the multiplies are actually just multiplication by ± 1 and $\pm \mathrm{i}$ (which only flip real and imaginary parts and flip sign bits, and thus don't count as flops). When coded carefully, this reduces the cost of both of the first two stages to $\sim 4N$ real flops.

algorithm may be represented as the **block diagram** illustrated in Figure 5.2. The resulting algorithm (referred to as the **Cooley-Tukey** variant of the fast Fourier transform, in honor of its inventors) may be arranged as a simple recursive code, as illustrated in Algorithm 5.1, or as a slightly more involved direct calculation, as illustrated in Algorithm 5.2. Note that the latter of these two algorithms involves a reordering of the inputs into bit reversed order during every FFT call, as described in the caption of Figure 5.2. This can be avoided by keeping the physical space representation in standard order and the Fourier space representation in bit reversed order everywhere in the code, as illustrated in Figure 5.3 and implemented in Algorithm 5.3. [14]

Other FFT algorithms. If the order of $\mathbf{x}$ is not a power of 2, then instead of relating the output vectors to twice as many input vectors of half the order at each stage, one may instead[15] relate the output vectors to p_j times as many input vectors of $1/p_j$ the order at each stage j, where the p_j for $j = 1 \ldots s$ are the various **prime factors** of N (some of which may be repeated). Following this strategy for an N with a prime factor of 3, the following identity may be determined [cf. (5.25); (5.26) is derived in an analogous fashion]

$$F_N \mathbf{x} = \underbrace{\begin{bmatrix} I & I & I \\ I & w_3 I & \overline{w}_3 I \\ I & \overline{w}_3 I & w_3 I \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} I & & 0 \\ & \Omega_{N/3} & \\ 0 & & \Omega_{N/3}^2 \end{bmatrix}}_{B} \underbrace{\begin{bmatrix} F_{N/3}\mathbf{x}^a \\ F_{N/3}\mathbf{x}^b \\ F_{N/3}\mathbf{x}^c \end{bmatrix}}_{\mathbf{c}}, \tag{5.26}$$

where $w_3 = e^{\mathrm{i}2\pi/3}$, $w_N = e^{\mathrm{i}2\pi/N}$, $\Omega_{N/3} = \mathrm{diag}([1, w_N, \ldots, w_N^{N/3-1}])$, $\mathbf{x}^a$ is the vector containing the $\{0,3,6,9,\ldots\}$ rows of $\mathbf{x}$, $\mathbf{x}^b$ is the vector containing the $\{1,4,7,10,\ldots\}$ rows of $\mathbf{x}$, and $\mathbf{x}^c$ is the vector containing the $\{2,5,8,11,\ldots\}$ rows of $\mathbf{x}$. For example, if $N = 96$, five of the stages relate the ouput vectors to twice as many vectors of half the order via (5.25), whereas one of the stages relates the output vectors to three times as many vectors of one third their order via (5.26). A stage implementing (5.26) requires $\sim 2N/3$ complex multiplications (to calculate $B\mathbf{c}$), plus $\sim 4N/3$ complex multiplications and $\sim 2N$ complex additions (to calculate A times the result). Note, however, that the two complex products $w \cdot z = (a+b\mathrm{i})(c+d\mathrm{i}) = (ac-bd)+(ad+bc)\mathrm{i}$ and $\overline{w} \cdot z = (a-b\mathrm{i})(c+d\mathrm{i}) = (ac+bd)+(ad-bc)\mathrm{i}$ may be calculated with a total of 8 real flops (not 12). Leveraging this fact, (5.26) may be calculated with a total of $\sim 40N/3 \approx 13.33N$ real flops[16] (for implementation, see Exercise 5.4). Thus, due to the relative complexity of (5.26), each stage corresponding to a prime factor of 3 is significantly more expensive than even two stages corresponding to a prime factor of 2. However, it is typical in most problems which make extensive use of FFT calls, such as the numerical simulation of PDE systems, that we may in fact allow the user to select N. In such problems, it is thus strongly advised to stick with FFTs of order $N = 2^s$ or, if necessary, $N = 3 \cdot 2^s$; *the remainder of this chapter assumes* $N = 2^s$.

The Fastest Fourier Transform in the West. Many variations of the basic FFT algorithm described above exist, and each differs from the others in an assortment of seemingly minor details (execution order, etc.) that can significantly affect the efficiency of the cache usage on different CPU architectures, and thus significantly impact the overall execution speed of the FFT algorithm. For this reason, one FFT algorithm can not be depended on to be the fastest FFT algorithm (even for a particular value of N) in all situations. To address this fact, an ingenious FFT package called the **Fastest Fourier Transform in the West** (**FFTW**) was developed, and is available for free download at `http://www.fftw.org`. This package implements not one but *all* of the major variations of the FFT algorithm. When a particular code is going to be run on a particular machine

[14] Note that the bitswap of the exponent vector in Algorithm 5.3 is, unfortunately, very slow to execute in the current (as of this writing) version of Matlab. An attractive alternative is to save all the `wv` variables during an FFT "setup" call, and then reuse them for all subsequent calls to this algorithm. The coding necessary to accomplish this is addressed in Exercise 5.3.

[15] This approach is efficient only for N with small prime factors, as such general FFT algorithms suffer a significant performance penalty if any of the prime factors of N is significantly larger than 2; in the extreme worst case that N is itself large and prime, the cost of the FFT reverts to the $\sim 8N^2$ real flops of full complex matrix/vector multiplication.

[16] Note that, if N has only a single prime factor of three, executing it as the first stage, and implementing carefully, again allows a few flops to be shaved off, thus allowing the calculation of this stage to be executed with only $\sim 28N/3 \approx 9.33N$ real flops.

Algorithm 5.1: The recursive Cooley-Tukey variant of the fast Fourier transform.

View Test

```
function [x] = FFTrecursive(x,N,g)
% Compute the forward FFT (g=-1) or inverse FFT (g=1) of a vector x of order N=2^s.  The
% entire algorithm is simply repeated application of Eqn (5.20).  NOTE: to get the
% forward transform, you must divide the result by N outside this function. Note also that
% the wavenumber vector corresponding to the Fourier representation should be defined
% (outside this routine) as:  k=(2*pi/L)*[[0:N/2]';[-N/2+1:-1]']; note in particular that
% k(1)=0 (recall that Matlab indexes from 1, not 0).
if N>1                                          % If N=1, do nothing, else...
   M=N/2; w=exp(g*2*pi*i/N); xe=x(1:2:N-1); xo=x(2:2:N);
   xe=FFTrecursive(xe,M,g);                     % Compute FFTs of the even and odd parts.
   xo=FFTrecursive(xo,M,g);
   for j=1:M; xo(j,1)=w^(j-1)*xo(j,1); end      % Split each group in half and combine
   x=[xe+xo; xe-xo];                            % as in Eqn (5.25)
end
end % function FFTrecursive
```

Algorithm 5.2: The direct Cooley-Tukey variant of the fast Fourier transform.

View Test

```
function x=FFTdirect(x,N,g)
% Compute the forward FFT (g=-1) or inverse FFT (g=1) of a vector x of order N=2^s.
% At each stage, defining Ns=2^stage, the elements divide into N/Ns groups, each with Ns
% elements.  Each group is split in half and combined as in Fig 5.2.
% The corresponding wavenumber vector is:  k=(2*pi/L)*[[0:N/2]';[-N/2+1:-1]'].
s=log2(N);                  % number of stages
ind=bin2dec(fliplr(dec2bin([0:N-1]',s))); x=x(ind+1); % Put input into bit reversed order.
for stage=1:s               % For each stage...
   Ns=2^stage;              % Determine size of the transform computed at this stage.
   M=Ns/2; w=exp(g*2*pi*i/Ns);  % Calculate w
   for m=0:M-1              % Split each group in half to combine as in Fig 5.2:
      wm=w^m;               % Calculate the necessary power of w.
      for n=1:Ns:N          % Determine each pair of elements to be combined and combine them.
         a=m+n; b=M+a; x([a b])=x(a)+[1; -1]*x(b)*wm;
      end
   end
end
if g==-1, x=x/N; end                % Scale the forward version of the transform by 1/N.
end % function FFTdirect
```

with a particular cache structure, and execute billions of FFTs of a particular order on this machine, FFTW first executes an assortment of different FFT algorithms of this order on this machine, thereby determining the execution time required for each experimentally. FFTW then selects whichever algorithm happened to run the fastest during this test for all subsequent FFT calculations. This self-optimizing software library thus achieves that often elusive goal of **portable speed** (that is, a single code that gives nearly optimal speed on a wide range of different computer architectures). It is therefore no exaggeration to say that, as a result, FFTW is the FFT package of choice for most modern high-performance numerical codes.

5.4.2 Properties of the coefficients of a finite Fourier series

Note that, at $k_n = 0$, we have [by (5.23b)]

$$\hat{u}_0 = \frac{1}{N}\sum_{j=1}^{N} u_j. \tag{5.27}$$

That is, $\hat{u}_0$ is simply the average of the u_j over the N samples. This property is illustrated in several of the test codes provided with the FFT algorithms in this chapter.

Algorithm 5.3: The direct Cooley-Tukey variant of the fast Fourier transform without reordering.

View
Test

```
function x=FFTnonreordered(x,N,g)
% Compute the FFT (g=-1) or inverse FFT (g=1) of a vector x of order N=2^s.  NOTE: as
% opposed to Algorithm 5.2, this algorithm does not bother reordering the Fourier
% representation out of, and then back into, bit reversed order, thereby reducing both
% storage and execution time (see Figure 5.3).  The corresponding wavenumber vector is
% t=(2*pi/L)*[[0:N/2]';[-N/2+1:-1]']; k=t(1+bin2dec(fliplr(dec2bin([0:N-1]',s))))
s=log2(N);          % Number of stages.
if g==-1            % FORWARD TRANSFORM (physical space input expected in standard order).
  for stage=1:s     % (This section similar to Algorithm 5.2, modified as in Figure 5.3.)
    Ns=2^stage; d=2*N/Ns; M=N/Ns; w=exp(-2*pi*i/Ns);
    wv=w.^(bin2dec(fliplr(dec2bin([0:Ns/2-1]))));          % Bitswap the exponent vector.
    for m=0:Ns/2-1
      md=m*d;
      for n=1:M     % Determine each pair of elements to be combined, and combine them.
        a=md+n; b=a+M; x([a b])=x(a)+[1; -1]*x(b)*wv(m+1);
      end
    end
  end               % Note: Fourier space output is returned in bit reversed order.
  x=x/N;
else                % INVERSE TRANSFORM (Fourier space input expected in bit reversed order).
  for stage=1:s     % (This section as in Algorithm 5.2.)
    Ns=2^stage; M=Ns/2; w=exp(2*pi*i/Ns); wv=w.^([0:Ns/2-1]);  % Standard order used here.
    for m=0:M-1
      for n=1:Ns:N
        a=m+n; b=a+M; x([a b])=x(a)+[1; -1]*x(b)*wv(m+1);
      end
    end
  end               % Note: Physical space output is returned in standard order.
end
end % function FFTnonreordered
```

Note also that, since $k_N x_j = \frac{2\pi N}{L} \cdot \frac{jL}{N} = 2\pi j$,

$$e^{ik_{m+N}x_j} = e^{ik_m x_j}. \tag{5.28a}$$

By (5.23b), it thus follows for the finite Fourier series that

$$\hat{u}_{m+N} = \frac{1}{N}\sum_{j=0}^{N-1} u_j e^{-ik_{m+N}x_j} = \frac{1}{N}\sum_{j=0}^{N-1} u_j e^{-ik_m x_j} = \hat{u}_m. \tag{5.28b}$$

That is, *the coefficients of a finite Fourier series are periodic*. Thus, applying (5.28a) and (5.28b), the following expansions are completely equivalent[17]

$$u_j = \underbrace{\sum_{n=-N/2}^{N/2-1} \hat{u}_n e^{ik_n x_j}}_{(A)} = \underbrace{\sum_{n=-N/2+1}^{N/2} \hat{u}_n e^{ik_n x_j}}_{(B)} = \underbrace{\sum_{n=0}^{N-1} \hat{u}_n e^{ik_n x_j}}_{(C)} = \underbrace{\sum_{n=1}^{N} \hat{u}_n e^{ik_n x_j}}_{(D)} = \underbrace{\sum_{n=-p}^{N-1-p} \hat{u}_n e^{ik_n x_j}}_{(E)} \tag{5.29}$$

for any integer p. One important implication/interpretation of (5.29) is that $\sin(k_n x_j)$ and $\sin(k_{n+pN} x_j)$, for any integer p, are indistinguishable on a discrete grid with N gridpoints, as are $\cos(k_n x_j)$ and $\cos(k_{n+pN} x_j)$,

[17] Note that the FFT algorithms given in §5.4.1 return an expansion in the form (C), with the indices shifted by $+1$ (because Matlab indexes from 1, not 0).

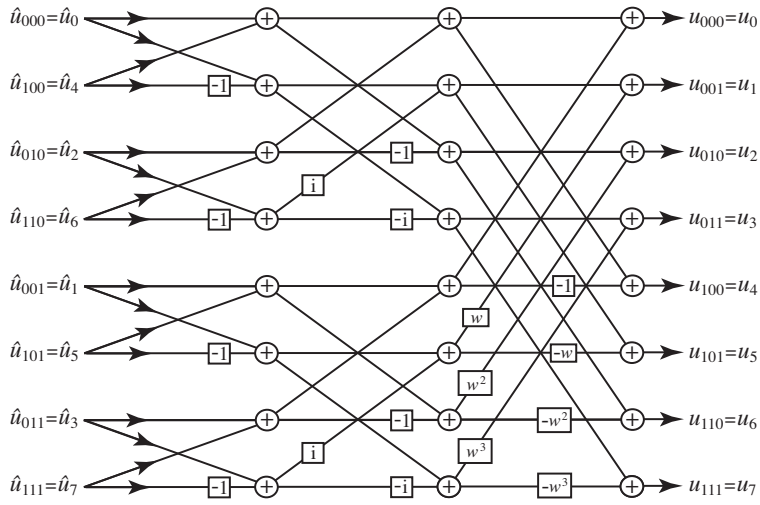


Figure 5.2: Half butterfly graph illustrating the Cooley-Tukey variant of the inverse FFT $\mathbf{u} = F_N\hat{\mathbf{u}}$ (5.24a) for $N = 8$, where $w = e^{i2\pi/N}$. The recursive form (Algorithm 5.1) may be seen by working backwards from the output vector on the right, which is related to the $F_{N/2}$ transforms of its even and odd components via (5.25), which in turn (working to the left) are related to the $F_{N/4}$ transforms of their even and odd components, etc. The direct form (Algorithm 5.2), in contrast, works from left to right, performing all the necessary linear combinations in order; to accomplish this, the input must be arranged in **bit reversed order** (that is, regular counting order with bits reversed in a binary representation). In either form, this algorithm costs $\sim 5N\log_2(N)$ real flops. Note that the forward transform $\hat{\mathbf{u}} = G_N\mathbf{u}$ [see (5.24b)] may be accomplished via the same graph, replacing i with $-$i and w with $\overline{w}$ and dividing the output by N; in such a case, the function values u_j are input in bit reversed order on the left and coefficients $\hat{u}_n$ are output in standard order on the right.

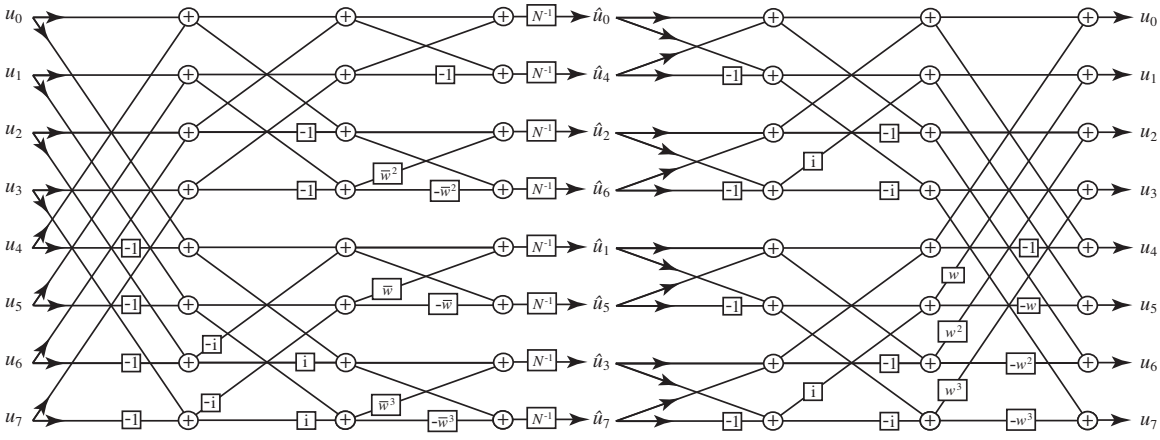


Figure 5.3: Full butterfly graph illustrating both forward and inverse FFTs (Algorithm 5.3). The inverse transform on the right is identical to Figure 5.2. The forward transform on the left is formed by performing the manipulations discussed in the caption of Figure 5.2 to the half butterfly in Figure 5.2 and then just repositioning the rows of nodes to place the inputs in standard order and the outputs in bit-reversed order. The remarkable feature of this graph is that it highlights the fact that there is no need to rearrange the coefficients of the Fourier-space represenation out of, and then back into, bit reversed order; with care in the rest of the numerical code (e.g., defining the vector containing the wavenumbers k_n in bit-reversed order), the Fourier-space representation may be kept in bit-reversed order everywhere in the entire code, thereby avoiding the significant computation time required to perform these reorderings during every FFT call.

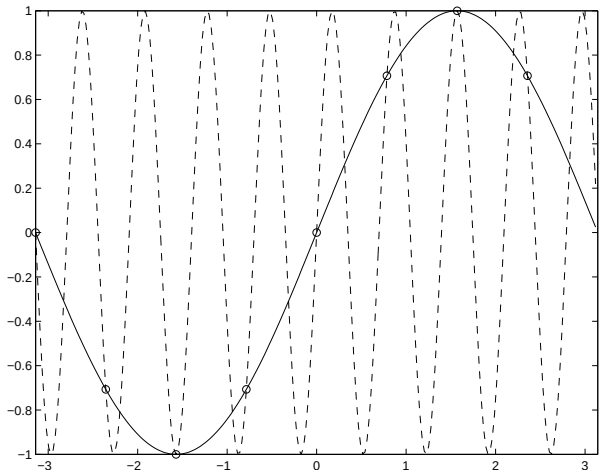


Figure 5.4: Two different signals, $\sin(x)$ and $\sin(9x)$, with exactly the same samples on $N = 8$ points on $-\pi \leq x < \pi$, illustrating that these two curves are indistinguishable with an $N = 8$ discrete representation. This effect is referred to as **aliasing**.

as illustrated in Figure 5.4. Thus, if a continuous signal is to be sampled and represented with a finite Fourier series, it is important to filter the continuous signal first to remove its wavenumber components outside the range $-k_{N/2} < k < k_{N/2}$, where $k_{N/2} = \pi N/L = \pi/\Delta x$ is referred to as the **Nyquist** frequency; §5.4.3 and §5.5 discuss this matter further.

5.4.3 Approximating differentiation using a finite Fourier series

Now rewrite the infinite Fourier series expansion of a function $u(x)$ in terms of an infinite sum of sines and cosines:

$$u(x) = \sum_{m=-\infty}^{\infty} \hat{u}_m e^{\mathrm{i}k_m x} = \sum_{m=-\infty}^{\infty} \hat{u}_m [\cos(k_m x) + \mathrm{i}\sin(k_m x)] \triangleq \sum_{m=0}^{\infty} [\hat{u}^c_m \cos(k_m x) + \hat{u}^s_m \sin(k_m x)] \tag{5.30a}$$

$$\Rightarrow \quad \begin{cases} \hat{u}^c_m = \hat{u}_m + \hat{u}_{-m} \\ \hat{u}^s_m = (\hat{u}_m - \hat{u}_{-m})\mathrm{i}. \end{cases} \tag{5.30b}$$

If we want to determine uniquely the sinusoidal components of the continuous function $u(x)$ (that is, the coefficients $\hat{u}^c_m$ and $\hat{u}^s_m$) over a finite range of wavenumber $k \in [0, k_1, \ldots, k_M]$, then we need to know both the positive *and* negative Fourier coefficients of its infinite Fourier series over this range; that is, we need to know the $\hat{u}_m$ over the range $m \in [-M, M]$, as suggested in (5.9). For this reason, interpreting the finite Fourier series expansions as summing over the wavenumber range indicated by the form (A) or (B) in (5.29) is preferred. For the FFT routines listed in Algorithms 5.1-5.3, this may be accomplished simply by defining the wavenumber vector as listed in the preamble of each of these algorithms, it is not necessary to actually reorder the coefficients of the series into the order implied by the form (A) or (B) in (5.29). Significantly, noting (5.10), defining the wavenumber vector `k` in this manner allows one to approximate differentiation in Fourier space with the simple commands

```
fhat=i*k.*uhat;   ghat=-k.^2.*uhat;
```

etc. Thus, in the remainder of this text, for simplicity, we will continue to write the Fourier series expansion as introduced in (5.23a) [that is, form (C) in (5.29)], though we assume the wavenumber vector k_n is defined

such that, noting (5.28), the sum is actually, in effect, in form (A) or (B) in (5.29) — that is, approximately centered over the zero wavenumber.

Interpreted another way, it does not matter what lower limit on the sum is used when *representing* a finite Fourier series, as all four of the natural choices are mathematically equivalent, as noted in (5.29). However, applying the operations in (5.10a) and (5.10b) to a truncated approximation of the infinite Fourier series, noting that $k_m = 2\pi m/L$ and thus $k_{m+N} \neq k_m$ for the infinite Fourier series, it *does* matter what lower limit is used when *differentiating* a finite Fourier series approximation of a function, as by so doing the spatially-continuous interpretation of the function $u(x)$ sampled by the discrete function u_j is significant. To differentiate all of the lowest-wavenumber sine and cosine components of the corresponding continuous function $u(x)$ correctly, *the wavenumber vector should be defined such that the finite Fourier series expansion actually sums over the wavenumber range indicated by the form* (A) *or* (B) *in* (5.29).

5.5 Special properties of the Fourier expansions of real functions

If u is real, then[18] $u(x) = u^*(x)$, and thus, by the infinite Fourier series expansion (5.8), defining $m' = -n$, we have

$$u(x) = \sum_{m=-\infty}^{\infty} [\hat{u}_m] e^{\mathrm{i}k_m x} = u^*(x) = \sum_{n=-\infty}^{\infty} \hat{u}_n^* e^{-\mathrm{i}k_n x} = \sum_{m'=-\infty}^{\infty} [\hat{u}_{-m'}^*] e^{\mathrm{i}k_{m'} x} \quad \forall x \quad \Rightarrow \quad \hat{u}_m = \hat{u}_{-m}^*. \tag{5.31}$$

This property is evident in the test codes accompanying Algorithms 5.1-5.3. Thus, when expanding a real function as a Fourier series, only the Fourier coefficients at the zero and positive wavenumbers need to be determined, as the Fourier coefficients at the negative wavenumbers may be determined directly from (5.31).

In the discretized setting, if u_j is real, two wavenumbers in the finite Fourier series expansion (5.23) deserve special attention. Note first that, by (5.27), $\hat{u}_0$ is simply the average of the u_j over the N samples. Thus, *if the* u_j *are real, then* $\hat{u}_0$ *is also real*. Note also that, by (5.28b), $\hat{u}_{N/2} = \hat{u}_{-N/2}$, and, by (5.31), $\hat{u}_{N/2} = \hat{u}_{-N/2}^*$. Thus, $\hat{u}_{N/2} = \hat{u}_{N/2}^*$; that is, *if the* u_j *are real, then* $\hat{u}_{N/2}$ *is also real*. We therefore see that we again satisfy the curious recurrent **conservation of information** property: in the representation of N real function values as a finite Fourier series, there are N independent real pieces of information: the two real Fourier coefficients $\hat{u}_0$ and $\hat{u}_{N/2}$ and the $N/2-1$ complex coefficients $\hat{u}_1$ to $\hat{u}_{N/2-1}$ (each of which consists of two real numbers).

Dealing with the Nyquist frequency. The frequency $k_{N/2} = \pi N/L$ which is constrained to have a real Fourier coefficient if the original u_j is real is called the **Nyquist frequency** or **oddball wavenumber**[19]. The fact that, if u_j is real, the Fourier coefficient must be real at the Nyquist frequency in the finite Fourier series expansion causes a peculiar problem when trying to represent a numerical approximation of an odd derivative of real function u. Specifically, if the discretization u_j of a continuous real function $u(x)$ has a nonzero coefficient $\hat{u}_{N/2}$ at the Nyquist frequency in its finite Fourier series expansion, then the discretization f_j of the continuous function defined as an odd derivative of $u(x)$ [that is, $f(x) = d^p u(x)/dx^p$, where p is odd] should have an imaginary coefficient at the same frequency. However, if f is expanded with a finite Fourier series with the same number of terms as used in the expansion of u, this would be an imaginary Fourier coefficient $\hat{f}_{N/2}$ at the Nyquist frequency, which would mean that the corresponding f_j would not be real [that is, (5.28b) and (5.31) would fail to both hold simultaneously for $\hat{f}_{N/2}$], which contradicts the fact that both $f(x)$ and its discrete approximation should be real. We are thus forced to select $\hat{f}_{N/2} = 0$ regardless of $\hat{u}_{N/2}$; that is, the finite Fourier series expansion is not suitable for differentiation of those oscillations of u at (or, for that matter,

[18] For the remainder of §5, in order to simplify the notation, the symbol $()^*$ is used to denote the complex conjugate.

[19] As the independent coördinate x in this chapter may be interpreted as either a spatial coördinate or time, we use for k the words "wavenumber" (typically reseverved for the spatial interpretation) and "frequency" (typically reserved for the temporal interpretation) essentially interchangeably; following the temporal interpretation, the independent coördinate is typically denoted t and the frequency denoted ω.

above) the Nyquist frequency $k_{N/2} = \pi N/L$. The practical conclusion is this: *oscillations in the numerical representation of a function at the Nyquist frequency should be removed from the simulation before spectral differentiation in order for the simulation to behave correctly*[20]. In practice, we thus, usually, simply constrain the coefficients at the Nyquist frequency to be zero at the end of every Fourier transform in order to remove these rapid oscillations from the numerical code entirely, thereby "smoothing" the resulting calculation.

The FFT of a real or complex vector u_j may be calculated using Algorithms 5.1-5.3. However, in the case that u_j is real, we know that the Fourier transform will return a vector $\hat{u}_n$ exhibiting the symmetry $\hat{u}_n = \hat{u}^*_{-n}$ [see (5.31)], and that the subsequent inverse transform will return a vector u_j with zero imaginary part. We should be able to exploit these facts to compute the transform of a real vector more efficiently; the following discussion shows how.

Computing the fast Fourier transform of two real vectors. Consider first the finite Fourier series expansion of two real vectors u_j and v_j of order N. By (5.31), we have

$$\hat{u}_n = \hat{u}^*_{-n}, \quad \hat{v}_n = \hat{v}^*_{-n}.$$

Now define a complex auxiliary function w_j such that

$$w_j = u_j + \mathrm{i}v_j \quad \Rightarrow \quad \hat{w}_n = \hat{u}_n + \mathrm{i}\hat{v}_n. \tag{5.32a}$$

It follows that

$$\hat{w}^*_{-n} = \hat{u}^*_{-n} - \mathrm{i}\hat{v}^*_{-n} \quad \Rightarrow \quad \hat{w}^*_{-n} = \hat{u}_n - \mathrm{i}\hat{v}_n. \tag{5.32b}$$

Taking [(5.32a) + (5.32b)]/2 and [(5.32a) − (5.32b)]/(2i) gives

$$\hat{u}_n = (\hat{w}_n + \hat{w}^*_{-n})/2, \quad \hat{v}_n = (\hat{w}_n - \hat{w}^*_{-n})/(2\mathrm{i}). \tag{5.33}$$

Thus, the Fourier transform of *two* real vectors u_j and v_j of order N may be computed simultaneously via a *single* order N FFT by combining the two vectors into a complex vector $w_j = u_j + \mathrm{i}v_j$, taking its FFT, then extracting $\hat{u}_n$ and $\hat{v}_n$ according to (5.33), as implemented in Algorithm 5.4. All of the steps of the forward transform are invertible, so the inverse transform is easy to develop simply by inverting the steps of the forward transform and doing them in reverse, as implemented in Algorithm 5.5.

Computing the fast Fourier transform of a single real vector. Now consider the finite Fourier series expansion of a single real vector u_j of order N. By the identity (5.25), we may write

$$\hat{u}_n = \hat{u}^e_n + e^{-2\pi \mathrm{i}n/N}\hat{u}^o_n \quad \text{for} \quad n = 0, 1, \ldots, N-1, \tag{5.34}$$

where u^e_j is the vector containing the even elements of u_j and u^o_j is the vector containing the odd elements of u_j, both of which are order $N/2$, and $\hat{u}^e_n$ and $\hat{u}^o_n$ are the (periodic) coefficients of their finite Fourier transforms. Thus, the Fourier transform of a single real vector u_j of order N may be computed via a single order $N/2$ FFT of its (real, order $N/2$) even and odd parts using Algorithm 5.4, then combining to determine $\hat{u}_n$ using (5.34), as implemented in Algorithm 5.6. As a significant streamlining step, one may insert Algorithm 5.4 into Algorithm 5.6 (once both of these codes are debugged) and then simplify the resulting code, which results in Algorithm 5.7. Again, all of the steps of the forward transform are invertible, so the inverse transform is easy to develop simply by inverting the steps of the forward transform and doing them in reverse, as implemented in Algorithm 5.8.

[20] In the finite-difference literature, oscillations at the Nyquist frequency are commonly referred to as **two-delta waves**. One encounters a similar difficulty differentiating a two-delta wave using finite difference methods (see §8) as one has differentiating fluctuations at the Nyquist frequency using spectral methods.

Algorithm 5.4: The FFT of two real functions.

View
Test

```
function [uhat,vhat]=RFFT2(u,v,N)
% INPUT: u and v are real arrays of order N=2^s.
% OUTPUT: uhat and vhat are complex arrays of order N/2, containing half of the FFTs
% of u and v, for wavenumbers 0 to N/2.  As u and v are real, their 0 and N/2 Fourier
% coefficients are real, so the N/2 coefficients (at the Nyquist frequency)
% are stored in the imaginary part of the the 0 coefficients.
% To remove them, just set, e.g., uhat(1)=real(uhat(1)).
w=u+i*v;                                    % Combine u and v into a single complex
what=FFTdirect(w,N,-1);                     % vector and transform to Fourier space.
M=N+2;
for j=2:N/2
   uhat(j,1)=(what(j)+conj(what(M-j)))/2;   % Extract uhat and vhat from the result.
   vhat(j,1)=(what(j)-conj(what(M-j)))/(2*i); % via (5.33)
end
uhat(1,1)=real(what(1)) +i*real(what(N/2+1)); % Handle the zero and Nyquist frequency
vhat(1,1)=imag(what(1)) +i*imag(what(N/2+1)); % separately, noting they are both real.
end % function RFFT2
```

Algorithm 5.5: The inverse FFT of two real functions.

View

```
function [u,v]=RFFT2inv(uhat,vhat,N)
% INPUT: uhat & vhat are complex arrays of length N/2 containing half of the ffts of u & v,
% for wavenumbers 0 to N/2, with the N/2 coefficients stored in the imaginary part of the
% 0 coefficients.
% OUTPUT: u and v are real arrays of length N=2^s.
% This routine was written by inverting the steps of RFFT2 and doing them in reverse.
what(1)     =real(uhat(1)) +i*real(vhat(1));
what(N/2+1)=imag(uhat(1)) +i*imag(vhat(1));
M=N+2;
for j=2:N/2
   what(j)   =uhat(j)+i*vhat(j);            % Combine uhat+i*vhat
   what(M-j)=conj(uhat(j))+i*conj(vhat(j));
end
w=FFTdirect(what,N,1);                      % Transform to physical space.
u=real(w)';  v=imag(w)';                    % Extract u and v from the result.
end % function RFFT2inv
```

Algorithm 5.6: The FFT of a single real function.

View
Test

```
function [uhat]=RFFT1(u,N)
% INPUT: u is a real array of length N=2^s.
% OUTPUT: uhat is a complex array of length N/2.  It contains half of the fft of u,
% for the wavenumbers 0 to N/2.  As u is real, its 0 and N/2 coefficients are real,
% so the N/2 coefficient (at the "oddball wavenumber") is stored in the imaginary
% part of the the 0 coefficient.  To remove it, just set uhat(1)=real(uhat(1)).
[uehat,uohat]=RFFT2(u(1:2:N-1),u(2:2:N),N/2);  % Compute FFTs of the even and
M=N/2+2;                                       % odd parts of u
for n=2:N/4
  uhat(n,1)   =    (uehat(n)+exp(-2*pi*i*(n-1)/N)*uohat(n))/2;  % Combine result
  uhat(M-n,1)=conj(uehat(n)-exp(-2*pi*i*(n-1)/N)*uohat(n))/2;  % as in (5.34)
end
uhat(1,1)=(real(uehat(1))+real(uohat(1)))/2 + i*(real(uehat(1))-real(uohat(1)))/2;
uhat(N/4+1,1)=(imag(uehat(1))-i*imag(uohat(1)))/2;
end % function RFFT1
```

Algorithm 5.7: The FFT of a single real function, simplified.

View
Test

```
function [uh]=RFFT(u,N)
% This routine was written by substituting RFFT2 into RFFT1 and simplifying.
wh=FFTdirect(u(1:2:N-1)+i*u(2:2:N),N/2,-1);
M=N/2+2;
for n=2:N/4
   uh(n,1)   =(wh(n)+conj(wh(M-n))-i*exp(-2*pi*i*(n-1)/N)*(wh(n)-conj(wh(M-n))))/4;
   uh(M-n,1)=(conj(wh(n))+wh(M-n)-i*exp( 2*pi*i*(n-1)/N)*(conj(wh(n))-wh(M-n)))/4;
end
uh(1,1)=(real(wh(1))+imag(wh(1)))/2 + i*(real(wh(1))-imag(wh(1)))/2;
uh(N/4+1,1)=(real(wh(N/4+1))-i*imag(wh(N/4+1)))/2;
end % function RFFT
```

Algorithm 5.8: The inverse FFT of a single real function.

View

```
function [u]=RFFTinv(uh,N)
% This routine was written by inverting the steps of RFFT and doing them in reverse.
wh(1)=real(uh(1,1))+imag(uh(1,1)) + i*(real(uh(1,1))-imag(uh(1,1)));
wh(N/4+1)=(real(uh(N/4+1,1)) -i*imag(uh(N/4+1,1)))*2;
M=N/2+2;
for n=2:N/4
   wh(n)=uh(n,1)+conj(uh(M-n,1))+(uh(n,1)-conj(uh(M-n,1)))*i*exp(2*pi*i*(n-1)/N);
   wh(M-n)=conj(uh(n,1))+uh(M-n,1)+(conj(uh(n,1))-uh(M-n,1))*i*exp(-2*pi*i*(n-1)/N);
end
w=FFTdirect(wh,N/2,1);
u(1:2:N-1,1)=real(w)';   u(2:2:N,1)=imag(w)';
end % function RFFTinv
```

5.6 Convolution sums and nonlinear products

Consider now the **convolution sum** defined in physical space (for real u_j, v_j) by

$$s_m = \frac{1}{N}\sum_{j=0}^{N-1} u_j v_{m-j} \quad \text{for } m = 0,\ldots,N-1, \tag{5.35a}$$

where the periodic extension of v_j is assumed (see §5.4). Note that $\sim 2N^2$ real flops are required to calculate the s_m for $m = 0,\ldots,N-1$. Expanding s_m, u_j, and v_{m-j} with finite Fourier series (5.23a), we find

$$\begin{aligned}\sum_{p=0}^{N-1}[\hat{s}_p]e^{ik_p x_m} &= \frac{1}{N}\sum_{j=0}^{N-1}\left(\sum_{n=0}^{N-1}\hat{u}_n e^{ik_n x_j}\right)\left(\sum_{p=0}^{N-1}\hat{v}_p e^{ik_p x_{m-j}}\right)\\ &= \sum_{p=0}^{N-1}\sum_{n=0}^{N-1}\hat{u}_n\hat{v}_p\underbrace{\left(\frac{1}{N}\sum_{j=0}^{N-1}e^{i(k_n-k_p)x_j}\right)}_{=\delta_{np}\text{ by (5.5b)}}e^{ik_p x_m} = \sum_{p=0}^{N-1}[\hat{u}_p\hat{v}_p]e^{ik_p x_m} \quad \forall m,\end{aligned}$$

from which we deduce that

$$\hat{s}_p = \hat{u}_p\hat{v}_p \quad \text{for } p = 0,\ldots,N-1. \tag{5.35b}$$

We thus see that the Fourier coefficients $\hat{s}_p$ can be computed as a simple product; noting (5.31), only $\sim N/2$ complex flops are required to calculate all of the (complex) $\hat{s}_p$. In other words, *it is much cheaper to compute* **s** *in Fourier space* (where its computation requires a simple product at each wavenumber) *than it is to compute it in physical space* (where its computation requires a convolution sum at each gridpoint).

Now consider the **nonlinear product** of two discretized functions in physical space:

$$w_p = u_p v_p \quad \text{for } p = 0,\ldots,N-1. \tag{5.36a}$$

Expanding w_p, u_p, and v_p with finite Fourier series (5.23a), noting (5.29), we find that

$$\begin{aligned}\sum_{m=0}^{N-1}\Big[\hat{w}_m\Big]e^{\mathrm{i}k_m x_p} &= \sum_{j=0}^{N-1}\hat{u}_j e^{\mathrm{i}k_j x_p}\sum_{m=0}^{N-1}\hat{v}_m e^{\mathrm{i}k_m x_p} = \sum_{j=0}^{N-1}\hat{u}_j\left(\sum_{m=-j}^{N-1-j}\hat{v}_m e^{\mathrm{i}k_m x_p}\right)e^{\mathrm{i}k_j x_p}\\ &= \sum_{j=0}^{N-1}\hat{u}_j\left(\sum_{m=0}^{N-1}\hat{v}_{m-j}e^{\mathrm{i}k_{m-j}x_p}\right)e^{\mathrm{i}k_j x_p} = \sum_{m=0}^{N-1}\left[\sum_{j=0}^{N-1}\hat{u}_j\hat{v}_{m-j}\right]e^{\mathrm{i}k_m x_p},\end{aligned}$$

from which we deduce that

$$\hat{w}_m = \sum_{j=0}^{N-1}\hat{u}_j\hat{v}_{m-j} \quad \text{for } m = 0,\ldots,N-1. \tag{5.36b}$$

We thus see the converse of what we saw in the computation of $\mathbf{s}$; that is, *it is much cheaper to compute* $\mathbf{w}$ *in physical space* (where its computation requires a simple product at each gridpoint) *than it is to compute it in Fourier space* (where its computation requires a convolution sum at each wavenumber). To summmarize,

Fact 5.4 *Products at each gridpoint in physical space correspond to convolution sums at each wavenumber in Fourier space, whereas products at each wavenumber in Fourier space correspond to convolution sums at each gridpoint in physical space. Products are much cheaper to compute than convolution sums.*

Note that derivations such as those given above extend to infinite Fourier series in a straightforward fashion. Consider, for example, the **convolution integral** defined in physical space (for real u, v) by

$$s(x) = \frac{1}{L_x}\int_0^{L_x} u(x')v(x-x')\,dx', \tag{5.37a}$$

where the periodic extension of $v(x)$ is assumed. Expanding $s(x)$, $u(x')$, and $v(x-x')$ with infinite Fourier series (5.8a), we find

$$\begin{aligned}\sum_{p=-\infty}^{\infty}[\hat{s}_p]e^{\mathrm{i}k_p x_m} &= \frac{1}{L_x}\int_0^{L_x}\Big(\sum_{n=-\infty}^{\infty}\hat{u}_n e^{\mathrm{i}k_n x'}\Big)\Big(\sum_{p=-\infty}^{\infty}\hat{v}_p e^{\mathrm{i}k_p x_{m-j}}\Big)\\ &= \sum_{p=-\infty}^{\infty}\sum_{n=-\infty}^{\infty}\hat{u}_n\hat{v}_p\underbrace{\Big(\frac{1}{L_x}\int_0^{L_x}e^{\mathrm{i}(k_n-k_p)x'}dx'\Big)}_{=\delta_{np}\text{ by (5.5a)}}e^{\mathrm{i}k_p x} = \sum_{p=-\infty}^{\infty}[\hat{u}_p\hat{v}_p]e^{\mathrm{i}k_p x} \quad \forall x,\end{aligned}$$

from which we deduce that

$$\hat{s}_p = \hat{u}_p\hat{v}_p \quad \text{for all } p. \tag{5.37b}$$

Again, it is seen that a convolution in physical space corresponds to a product at each wavenumber in Fourier space. Following a similar derivation for the infinite Fourier integral expansion (see §5.3) of a continuous function on the interval $(-\infty,\infty)$ leads to

$$s(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty}u(x')v(x-x')\,dx' \quad \Leftrightarrow \quad \hat{s}(k) = \hat{u}(k)\hat{v}(k). \tag{5.38}$$

5.7 Aliasing due to nonlinear products and the 2/3 dealiasing rule

We now revisit the problem of taking the nonlinear product of two functions in physical space, as considered in (5.36). Imagine the functions u_p and v_p, each discretized on the gridpoints $p = 0, \ldots, N-1$, have nonzero Fourier coefficients $\hat{u}_m$ and $\hat{v}_m$ only over the range $j = -M, \ldots, M$, where $M \ll N$. Examining the Fourier coefficients of their product w_p, as given in (5.36b), we see that $\hat{w}_m$ is nonzero over the range $m = -2M, \ldots, 2M$. Put in words, we may say that

Fact 5.5 *Nonlinear products scatter energy to both lower and higher wavenumbers.*

If $M \ll N$, then the calculation of the w_p in physical space [that is, where it may be calculated cheaply using (5.36a)], followed by the transformation of the result to Fourier space gives the expected result: that is, $\hat{w}_j$ will have energy in higher wavenumber components than both $\hat{u}_j$ and $\hat{v}_j$. However, a problem arises when M is not sufficiently small as compared with N. For example, if all of the coefficients in the finite Fourier series expansions of the u_p and v_p on the N gridpoints are nonzero (except for the Nyquist frequency, which we generally set to zero as discussed at the end of §5.5), then $M = N_{\max} = N/2 - 1$. In this case, due to the periodicity of the coefficients of the finite Fourier series expansion as established in (5.28b), the sum on the RHS of (5.36b) [which is exactly equivalent to the corresponding expression in physical space, (5.36a)] picks up extra contributions at certain wavenumbers that it wouldn't normally get if N were larger. These extra contributions to several of the $\hat{w}_m$ coefficients are referred to as **aliasing**. Aliasing may be thought of as a direct result of the fact that the fields are not expanded with finite Fourier series that are long enough to capture the scattering of energy to higher wavenumbers due to nonlinear products.

Recall that, in order to ensure that the derivative of a function u is representable with the same type of Fourier series as the original function, we suggested in §5.5 simply to "smooth" the function u a bit, insisting that any field that we consider obey $\hat{u}_{N/2} = \hat{u}_{-N/2} = 0$. This simple idea effectively solved the "oddball wavenumber problem".

The solution to the "aliasing problem" is similar: before calculating any nonlinear product $u \cdot v$, we simply ensure that u and v are sufficiently smooth (that is, that M is sufficiently small as compared with N) that (5.36b) doesn't pick up any extra contributions due to the periodicity of the coefficients of the finite Fourier series expansions, as given in (5.28b). This may be accomplished by filtering u and v (that is, setting their higher-order Fourier coefficients to zero) such that $M = (2/3)N_{\max}$ in their finite Fourier series expansions, which is referred to as the **2/3 dealiasing rule**. (Taking the opposite perspective, taking N sufficiently large as compared with M, this idea is sometimes referred to as the **3/2 dealiasing rule**.) The reason that the factor is only $2/3$ and not $1/2$ is, again, the periodicity of the Fourier coefficients (5.28b), which implies that setting the top $1/3$ of the Fourier coefficients to zero (that is, setting $\hat{u}_n = \hat{v}_n = \hat{w}_n = 0$ for $n \in [N/3, N/3+1, \ldots N/2]$ and $n \in [-N/2, -N/2+1, \ldots -N/3]$) effectively results in $\hat{u}_n = \hat{v}_n = \hat{w}_n = 0$ for both the range $n \in [N/3, N/3+1, \ldots 2N/3]$ and the range $n \in [-2N/3, -2N/3+1, \ldots -N/3]$, which is (just) sufficient to insure that (5.36b) does not pick up any extra contributions due to the periodicity of the coefficients of the finite Fourier series expansions.

5.8 Two-point correlations and Parseval's theorem

Consider now the **two-point correlation** defined in physical space by

$$r_m = \frac{1}{N} \sum_{j=0}^{N-1} u_j^* v_{j+m} \quad \text{for } m = 0, \ldots N-1, \tag{5.39a}$$

where the periodic extension of v_j is assumed. This quantity, which is similar but different than the convolution sum defined in (5.35a), is often a useful statistic to characterize the relationship between u and v.

Note that $\sim 2N^2$ real flops are required to calculate all the r_m. Expanding r_m, u_j, and v_{m-j} with finite Fourier series (5.23a), we find

$$\sum_{p=0}^{N-1}[\hat{r}_p]e^{ik_p x_m} = \frac{1}{N}\sum_{j=0}^{N-1}\Big(\sum_{n=0}^{N-1}\hat{u}_n e^{ik_n x_j}\Big)^*\Big(\sum_{p=0}^{N-1}\hat{v}_p e^{ik_p x_{j+m}}\Big)$$
$$= \sum_{p=0}^{N-1}\sum_{n=0}^{N-1}\hat{u}_n^*\hat{v}_p\underbrace{\Big(\frac{1}{N}\sum_{j=0}^{N-1}e^{-i(k_n-k_p)x_j}\Big)}_{=\delta_{np}\text{ by (5.5b)}}e^{ik_p x_m} = \sum_{p=0}^{N-1}[\hat{u}_p^*\hat{v}_p]e^{ik_p x_m} \quad \forall m,$$

from which we deduce [cf. (5.35b)] that

$$\hat{r}_p = \hat{u}_p^*\hat{v}_p \quad \text{for } p = 0,\ldots N-1. \tag{5.39b}$$

As with the computation of the convolution sum $\mathbf{s}$ in (5.35), it is much cheaper to compute $\mathbf{r}$ in Fourier space than it is to compute it in physical space.

The special case of (5.39) with $m = 0$ is known as **Plancherel's theorem**. Noting (5.39a), (5.23a), and (5.39b),

$$r_0 = \frac{1}{N}\sum_{j=0}^{N-1}u_j^*v_j, \quad r_0 = \sum_{p=0}^{N-1}\hat{r}_p = \sum_{p=0}^{N-1}\hat{u}_p^*\hat{v}_p \quad \Rightarrow \quad \frac{1}{N}\sum_{j=0}^{N-1}u_j^*v_j = \sum_{p=0}^{N-1}\hat{u}_p^*\hat{v}_p. \tag{5.40}$$

The important special case of Plancherel's theorem with $v = u$ is known as **Parseval's theorem**. In words, Parseval's theorem states that *the mean square of the magnitude of u_j in physical space is equal to the sum of the squares of the magnitude of the Fourier coefficients $\hat{u}_p$*; that is,

$$\frac{1}{N}\sum_{j=0}^{N-1}u_j^*v_j = \sum_{p=0}^{N-1}\hat{u}_p^*\hat{v}_p \quad \Rightarrow \quad \frac{1}{N}\sum_{j=0}^{N-1}|u_j|^2 = \sum_{p=0}^{N-1}|\hat{u}_p|^2. \tag{5.41a}$$

This property is illustrated in several of the test codes provided with the FFT algorithms in this chapter. Following a similar derivation as above for the infinite Fourier series expansion (see §5.2) of a continuous function on the interval $x \in [-L/2, L/2]$ leads to

$$\frac{1}{L}\int_{-L/2}^{L/2}u^*(x)\,v(x)\,dx = \sum_{p=-\infty}^{\infty}\hat{u}_p^*\hat{v}_p \quad \Rightarrow \quad \frac{1}{L}\int_{-L/2}^{L/2}|u(x)|^2\,dx = \sum_{p=-\infty}^{\infty}|\hat{u}_p|^2, \tag{5.41b}$$

whereas, interpreting the independent variable as time and following a similar derivation as above for the infinite Fourier integral expansion (see §5.3) of a continuous function on the interval $t \in (-\infty, \infty)$ leads to

$$\underbrace{\frac{1}{2\pi}\int_{-\infty}^{\infty}u^*(t)\,v(t)\,dt}_{\triangleq\langle u(t),v(t)\rangle} = \underbrace{\int_{-\infty}^{\infty}\hat{u}^*(\omega)\,\hat{v}(\omega)\,d\omega}_{\triangleq\langle\hat{u}(\omega),\hat{v}(\omega)\rangle} \quad \Rightarrow \quad \underbrace{\frac{1}{2\pi}\int_{-\infty}^{\infty}|u(t)|^2\,dt}_{\triangleq\|u(t)\|_2^2} = \underbrace{\int_{-\infty}^{\infty}|\hat{u}(\omega)|^2\,d\omega}_{\triangleq\|\hat{u}(\omega)\|_2^2}. \tag{5.41c}$$

5.9 Fourier expansions of nonsmooth functions: Gibbs phenomenon

If the function $u(x)$ being expanded and/or its required derivatives (e.g., $f = du/dx$ and $g = d^2u/dx^2$) are either discontinuous at certain point(s) on the domain $x \in [-L/2, L/2)$ or smooth on the domain $x \in [-L/2, L/2)$ but not periodic (and, therefore, the periodic connection of $u(x)$ is discontinuous at $x = L/2$), then Fourier methods (which attempt to expand $u(x)$ with smooth periodic basis functions) are ill suited. This is illustrated in Figure 5.5; note in particular that, as the number of modes retained in the truncated Fourier series

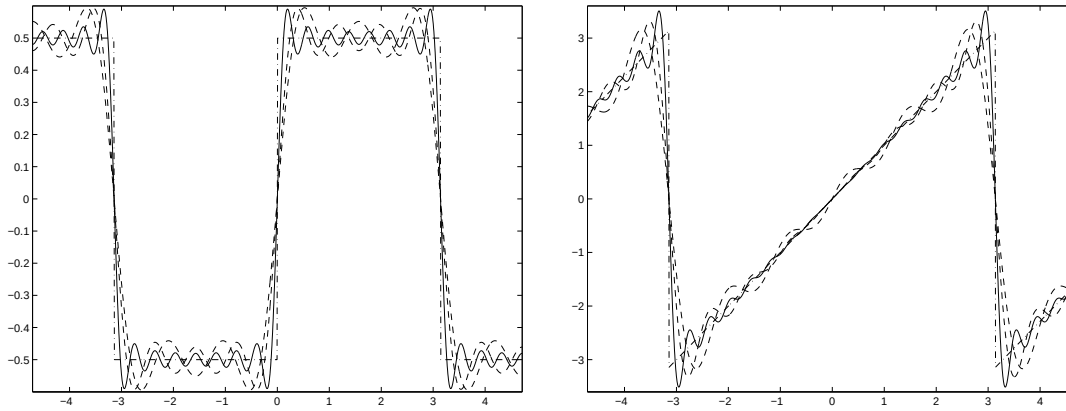


Figure 5.5: Demonstration of Gibbs phenomenon, that is, the overshoot or "ringing" present when attempting to represent a discontinuous function $u(x)$ (left, dot-dashed) or smooth {on $x \in [-\pi,\pi)$} but nonperiodic function $v(x)$ (right, dot-dashed) with a truncated Fourier series approximations $u^M(x)$ and $v^M(x)$ [see (5.9)] taking $M = 4$ and $M = 8$ (dashed) and $M = 16$ (solid).

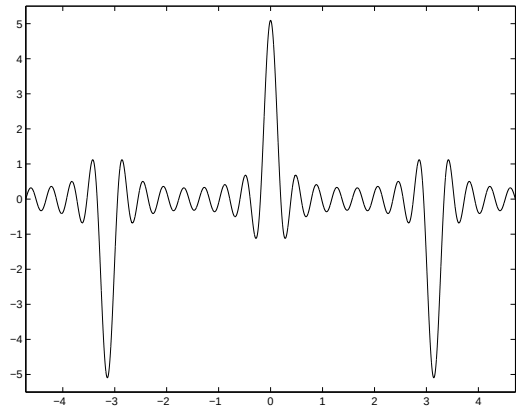


Figure 5.6: The derivative $f^M(x) = du^M(x)/dx$ of the truncated Fourier series approximation $u^M(x)$ of the step function $u(x)$, as illustrated by the solid line in Figure 5.5a, taking $M = 16$.

approximation (5.9) is increased, the maximum overshoot of the truncated Fourier series approximation of the discontinuous functions illustrated move closer to the discontinuity but remain approximately constant in magnitude. This is referred to as **Gibbs phenomenon**.

The main problem revealed by Figure 5.5 is that the slope and curvature of the truncated Fourier series approximations $u^M(x)$ and $v^M(x)$ are, over a large portion of the domain (especially in the vicinity of the discontinuities), substantially different than the slope and curvature of the functions $u(x)$ and $v(x)$ which they purport to approximate (e.g., see the first derivative of $u^M(x)$ with $M = 16$ in Figure 5.6). Thus,

- Fourier methods are not suitable for differentiation of nonsmooth and/or nonperiodic functions.

This difficulty is a manifestation of the fact that the infinite Fourier series of nonsmooth functions have substantial high wavenumber components, the truncation of which has a correspondingly substantial effect. Further, differentiating a function with substantial high wavenumber components p times magnifies the amplitude of these high wavenumber components by k^p [see (5.10)], thereby exacerbating the problem significantly, as quantified further in the following section.

5.9.1 Sobolev spaces and the quantification of smoothness

Consider a function $u(x)$, periodic on $x \in [-L/2, L/2)$, defined in terms of its infinite Fourier series (5.8a), and the related function $u^M(x)$ formed by truncating this Fourier series, as illustrated in (5.9). We now define the quantity

$$I = \lim_{M\to\infty} \sum_{m=-M}^{M} |\hat{u}_m|^2. \tag{5.42a}$$

When I is finite, we will say that the function $u(x)$ is L^2 **integrable**, or that $u(x)$ is in L^2, and it follows by Parseval's theorem (5.41b) that

$$\frac{1}{L}\int_{-L/2}^{L/2} |u(x)|^2\, dx = I. \tag{5.42b}$$

Now consider the function $h(x)$ defined in terms of its infinite Fourier series such that $\hat{h}_m = (ik_m)^p \hat{u}_m$; by (5.10a), $h(x)$ corresponds to the p'th derivative (if $p > 0$) or the $(-p)$'th antiderivative (if $p < 0$) of $u(x)$. If $h(x)$ is L^2 integrable according to the above definition, it is said that that $u(x)$ is "in" the **Sobolev space** H^p. Note that $H^0 = L^2$, and $H^2 \subset H^1 \subset L^2 \subset H^{-1}$. Note also that, in general, if $u(x)$ is in H^p, then $f(x) = du(x)/dx$ is in H^{p-1}. Identifying what Sobolev space a function $u(x)$ is in is valuable in a practical sense because, by identifying how many derivatives of $u(x)$ are L^2 integrable, one quantifies the eventual rate of "roll-off" of the magnitude of the Fourier coefficients, $|\hat{u}_m|$, as m is made large, thereby quantifying the "smoothness" of the function in a tangible manner.

As an example, consider the discontinuous function shown in Figure 5.5a and the coefficients of its infinite Fourier series expansion

$$u(x) = \begin{cases} -1/2 & -L/2 < x < 0 \\ 0 & x = -L/2, 0, L/2 \\ 1/2 & 0 < x < L/2 \end{cases} \quad \Leftrightarrow \quad \hat{u}_m(x) = \frac{1}{L}\int_{-L/2}^{L/2} u(x) e^{-ik_m x} dx = \ldots = \begin{cases} -2\mathrm{i}/(Lk_m) & m \text{ odd} \\ 0 & m \text{ even.} \end{cases}$$

It is easily verified that this function is L^2 integrable according to the above definition, with $I = 1/4$ in both (5.42a) and (5.42b). Now consider the first derivative $f^M(x) = du^M(x)/dx$ of the truncated Fourier series approximation $u^M(x)$ of $u(x)$. Noting the relation between the Fourier coefficients of a function and its derivatives [see (5.10)], $f^M(x)$ (for M even) may be defined in terms of the coefficients of its truncated Fourier series expansion:

$$\hat{f}_m^M = \begin{cases} 2/L & m = -M+1, -M+3, \ldots, M-3, M-1 \\ 0 & \text{otherwise} \end{cases} = 2(\hat{\delta}_m^M - \hat{\delta}_{2m}^M),$$

where $\hat{\delta}_m^M$ is defined as in (5.11b). Proceeding as in (5.42a), it is seen that $I = \infty$, and thus the $\hat{f}_m$ are *not* L^2 integrable; we thus refrain from even calling the corresponding $f(x)$ a function. In fact, $f(x)$ is the sum of two Dirac deltas, one of period L and integral 2 in the vicinity of the origin, the other of period $L/2$ and integral -1 in the vicinity of the origin (for a plot of the truncated Fourier series approximation $f^M(x)$ of $f(x)$ with $M = 16$, see Figure 5.6). To summarize,

Fact 5.6 *The derivative of a step function is a Dirac delta; the step is in L^2, whereas the Dirac is in H^{-1}.*

The relationship between the Dirac delta and the step function is further elucidated in (B.69a).

5.10 Fourier series in multiple dimensions

Fourier transforms of physical systems in multiple dimensions are straightforward. Consider a problem in three orthogonal coördinates[21], which we will call $\{x,y,z\}$. The (real) data is first transformed along each line (that is, for each discrete value of y and z) in the first dimension (x). The resulting "partially transformed" Fourier coefficients are complex, and functions of $\{k_x,y,z\}$. This (complex) data is then transformed along each line (that is, for each discrete value of k_x and z) in the second dimension (y). The result is a function of $\{k_x,k_y,z\}$. Finally, this (complex) data is transformed along each line (that is, for each discrete value of k_x and k_z) in the last dimension (z). The resulting **finite Fourier series in three dimensions** may be written

$$u_{i,j,k} = \sum_{p=0}^{N_x-1}\sum_{q=0}^{N_y-1}\sum_{r=0}^{N_z-1} \hat{u}_{p,q,r} e^{\mathrm{i}(k_{x_p}x_i + k_{y_q}y_j + k_{z_r}z_k)} \quad \text{with} \quad k_{x_p} = \frac{2\pi p}{L_x},\ k_{y_q} = \frac{2\pi q}{L_y},\ k_{z_r} = \frac{2\pi r}{L_z}. \tag{5.43a}$$

Multiplying the above expression by $\frac{1}{N_xN_yN_z} e^{-\mathrm{i}((k_{x_p}x_i + k_{y_q}y_j + k_{z_r}z_k)}$ and summing over the values $i = 0,\dots,N_x-1$, $j = 0,\dots,N_y-1$, and $k = 0,\dots,N_z-1$, and applying (5.5b), we find that

$$\hat{u}_{p,q,r} = \frac{1}{N_xN_yN_z} \sum_{i=0}^{N_x-1}\sum_{j=0}^{N_y-1}\sum_{k=0}^{N_z-1} u_{i,j,k} e^{-\mathrm{i}((k_{x_p}x_i + k_{y_q}y_j + k_{z_r}z_k)}. \tag{5.43b}$$

Note that the same considerations hold here as discussed in §5.4.3; that is, to facilitate an accurate approximation of differentiation of the continuous function $u(x,y,z)$ using this discrete approximation $u_{i,j,k}$, the sums in (5.43a) should be interpreted as in form (A) or (B) of (5.29). Again, it is not necessary to actually reorder the coefficients of the series in order to achieve this; rather it is sufficient to define the wavenumber vectors k_{x_p}, k_{y_q}, and k_{z_r} appropriately, as will be illustrated in Algorithm 5.11.

If the original data from the physical system is real, $u(x,y,z) = u^*(x,y,z)$. Thus, following a similar derivation as in (5.31), it follows that

$$\hat{u}_{p,q,r} = \hat{u}^*_{-p,-q,-r}. \tag{5.44}$$

That is, only half of the coefficients in the 3D domain $\{p \in [0,N_x-1], q \in [0,N_y-1], r \in [0,N_z-1]\}$ need to be saved in the computer, as the rest can be recovered using (5.44).

The resulting algorithm to compute the forward and inverse FFT of a real function in three dimensions is illustrated in Algorithm 5.9.

Example 5.1 Example: removing the divergence of a 3D vector field using spectral methods

We now diverge briefly to illustrate the power of spectral methods to solve an otherwise difficult problem. Define a 3D rectangular domain Ω of size $L_x \times L_y \times L_z$ and consider a three-component, three-dimensional[22] (3C,3D) vector field $\vec{v}(x,y,z)$ in Ω with periodic boundary conditions in all three directions. We now consider the problem of *projecting $\vec{v}$ onto a divergence-free manifold*[23]; that is, finding a nearby vector field $\vec{u}(x,y,z)$ (with the same periodic boundary conditions as on $\vec{v}$) that is divergence free, that is (see §B.4 for an abbreviated review of vector calculus),

$$\nabla \cdot \vec{u} = 0. \tag{5.45a}$$

This problem is easily solved by defining a **Poisson equation** for an auxiliary field q such that

$$\Delta q = \frac{1}{c} \nabla \cdot \vec{v} \tag{5.45b}$$

[21] Extension to problems in two dimensions (e.g., $\{z,\theta\}$) and four dimensions (e.g., $\{x,y,z,t\}$) follow accordingly.
[22] That is, at each point within the 3D domain Ω, $\vec{v}$ has three components.
[23] Note that this problem is of significant importance in the code developed in §13.

Algorithm 5.9: The FFT of a real 3D function.

View
Test

```
function [uhat]=RFFT3D(u,NX,NY,NZ)
% Compute the 3D FFT of the input u, setting all oddball wavenumber coefficients
% equal to zero.  Note that this code was written to emphasize the simplicity of this
% operation: more efficient codes would involve many fewer function calls by using
% specialized versions of RFFT and FFTdirect that compute many FFTs simultaneously.
% This code also computes the 2D FFT of u if NZ=1.
for J=1:NY,    for K=1:NZ, uhat(:,J,K)=RFFT(u(:,J,K),NX); end, end
uhat(1,:,:)=real(uhat(1,:,:));
for I=1:NX/2, for K=1:NZ, uhat(I,:,K)=FFTdirect(uhat(I,:,K),NY,-1); end, end
uhat(:,NY/2+1,:)=0;
if NZ>1;
  for I=1:NX/2, for J=1:NY, uhat(I,J,:)=FFTdirect(uhat(I,J,:),NZ,-1); end, end
  uhat(:,:,NZ/2+1)=0;
end
end % function RFFT3D
```

Algorithm 5.10: The inverse FFT of a real 3D function.

View

```
function [u]=RFFT3Dinv(uhat,NX,NY,NZ)
% Compute the inverse 3D FFT of the input uhat.
if NZ>1; for I=1:NX/2; for J=1:NY; uhat(I,J,:)=FFTdirect(uhat(I,J,:),NZ,1); end; end; end
for I=1:NX/2; for K=1:NZ;          uhat(I,:,K)=FFTdirect(uhat(I,:,K),NY,1); end; end;
for J=1:NY;   for K=1:NZ;          u   (:,J,K)=RFFTinv  (uhat(:,J,K),NX);   end; end;
end % function RFFT3Dinv
```

for any nonzero constant c. We then define $\vec{u}$ such that

$$\vec{u} = \vec{v} - c\nabla q. \tag{5.45c}$$

Taking the divergence of (5.45c) [that is, calculating $\nabla\cdot(\vec{u}=\vec{v}-c\nabla q)$], noting that $\nabla\cdot\nabla q = \Delta q$, it follows from (5.45b) that the desired condition (5.45a) is satisfied; that is, the field $\vec{u}$ so defined is divergence-free. As seen in Algorithm 5.11, implementing this procedure in code is straightforward in Fourier space, where each Fourier mode can be handled independently. In problems for which spectral methods can not be used, solving the multidimensional Poisson equation at the heart of this problem is much more difficult, and in general must be solved iteratively, as discussed further in §3.2.

5.11 Sine series and cosine series

In the spirit of the example presented in §4.3.2, if a real function $u(x)$ on a bounded domain, taken here without loss of generality to be $x\in[0,L/2]$, is known to be zero at both ends, then it is natural to expand it with an **infinite sine series** with the wavenumbers k_n selected so that the basis functions match the (Dirichlet) boundary conditions on u:

$$u(x) = \sum_{n=0}^{\infty} \hat{u}_n^s \sin(k_n x) \quad \text{with} \quad k_n = \frac{2\pi n}{L}. \tag{5.46a}$$

Similarly, if a real function $u(x)$ on $x\in[0,L/2]$ is known to have zero slope at both ends, then it is natural to expand it with an **infinite cosine series** with the wavenumbers k_n selected so that, again, the basis functions match the (Neumann) boundary conditions on u:

$$u(x) = \sum_{n=0}^{\infty} \hat{u}_n^c \cos(k_n x) \quad \text{with} \quad k_n = \frac{2\pi n}{L}. \tag{5.46b}$$

Algorithm 5.11: Codes to remove and compute the divergence of a 3D vector field, and a simple test code.

View

```
function [v1hat,v2hat,v3hat]=RemoveDivergence(v1hat,v2hat,v3hat,NX,NY,NZ,KX,KY,KZ)
% Remove the divergence of a periodic 3D vector field on a uniform grid.
% The input and output are in Fourier space, where the operations performed are simple.
for I=1:NX/2; for J=1:NY; for K=1:NZ
  if I*J*K > 1
    qhat(I,J,K)=(i*KX(I)*v1hat(I,J,K)+i*KY(J)*v2hat(I,J,K)+...
                 i*KZ(K)*v3hat(I,J,K))/(-KX(I)^2-KY(J)^2-KZ(K)^2);
  else
    qhat(I,J,K)=0;
  end
  v1hat(I,J,K)=v1hat(I,J,K)-i*KX(I)*qhat(I,J,K);
  v2hat(I,J,K)=v2hat(I,J,K)-i*KY(J)*qhat(I,J,K);
  v3hat(I,J,K)=v3hat(I,J,K)-i*KZ(K)*qhat(I,J,K);
end; end; end;
end % function RemoveDivergence
```

View

```
function [div]=ComputeDivergence(u1hat,u2hat,u3hat,NX,NY,NZ,KX,KY,KZ)
% Compute the divergence of a periodic 3D vector field on a uniform grid.
% Input and output are in Fourier space, where the operations performed are quite simple.
div=0;
for I=1:NX/2; for J=1:NY; for K=1:NZ
  div=div+i*KX(I)*u1hat(I,J,K)+i*KY(J)*u2hat(I,J,K)+i*KZ(K)*u3hat(I,J,K);
end; end; end;
end % function ComputeDivergence.m
```

View

```
% script <a href="matlab:RemoveDivergenceTest">RemoveDivergenceTest</a>
disp('Now testing RemoveDivergence on a random 3D vector field')
NX=16; NY=32; NZ=64; LX=1.0; LY=2.0; LZ=3.0;  % First, set up physical
KX=(2*pi/LX)*[[0:NX/2-1]'];                    % domain and the wavenumbers
KY=(2*pi/LY)*[[0:NY/2]';[-NY/2+1:-1]'];        % within it.
KZ=(2*pi/LZ)*[[0:NZ/2]';[-NZ/2+1:-1]'];
v1=rand(NX,NY,NZ); v2=rand(NX,NY,NZ); v3=rand(NX,NY,NZ); N=NX*NY*NZ; % Initialize v
for i=1:2
  v1hat=RFFT3D(v1(:,:,:),NX,NY,NZ);       % Transform v to Fourier space
  v2hat=RFFT3D(v2(:,:,:),NX,NY,NZ);
  v3hat=RFFT3D(v3(:,:,:),NX,NY,NZ);       % Now, remove the divergence.
  [u1hat,u2hat,u3hat]=RemoveDivergence(v1hat,v2hat,v3hat,NX,NY,NZ,KX,KY,KZ);
  u1=RFFT3Dinv(u1hat(:,:,:),NX,NY,NZ);    % Transform uhat back to Physical space
  u2=RFFT3Dinv(u2hat(:,:,:),NX,NY,NZ);
  u3=RFFT3Dinv(u3hat(:,:,:),NX,NY,NZ);    % u should now be divergence free.
  if i==1, disp('Divergence before and after pass 1:')
    v_divergence=ComputeDivergence(v1hat,v2hat,v3hat,NX,NY,NZ,KX,KY,KZ)
    u_divergence=ComputeDivergence(u1hat,u2hat,u3hat,NX,NY,NZ,KX,KY,KZ)
    v1=u1; v2=u2; v3=u3; clear u1 u2 u3;
  else, disp('Amount u changes when removing divergence again in pass 2:')
    norm(reshape(u1-v1,N,1))+norm(reshape(u2-v2,N,1))+norm(reshape(u3-v3,N,1)), disp(' ')
  end
end
% end script RemoveDivergenceTest
```

In the case of the sine expansion, proceeding as in the derivation of the finite Fourier series in §5.4, we expand the discretized function u_j as a **finite sine series** (a.k.a. a **discrete sine transform**) such that

$$u_j = \sum_{n=1}^{N-1} \hat{u}_n^s \sin(k_n x_j) = \sum_{n=1}^{N-1} \hat{u}_n^s \sin\frac{\pi jn}{N} \quad \text{with} \quad k_n = \frac{2\pi n}{L}. \tag{5.47a}$$

Multiplying the above expression by $(1/N)\sin(k_m x_j)$, summing over $j = 1,\ldots,N-1$, and applying (5.6b),

we find that

$$\frac{1}{N}\sum_{j=1}^{N-1}\left[u_j=\sum_{n=1}^{N-1}\hat{u}_n^s\sin(k_nx_j)\right]\sin(k_mx_j)\quad\Rightarrow\quad\hat{u}_m^s=\frac{2}{N}\sum_{j=1}^{N-1}u_j\sin(k_mx_j)=\frac{2}{N}\sum_{j=1}^{N-1}u_j\sin\frac{\pi jm}{N}. \tag{5.47b}$$

Note that the formula to compute the forward sine transform [that is, to determine the $\hat{u}_m^s$ given the u_j, as shown in (5.47b)] is almost the same as the formula to compute the inverse sine transform [that is, to determine the u_j given the $\hat{u}_m^s$, as shown in (5.47a)]. Thus,

Fact 5.7 *The same numerical code can be used for both the forward and inverse sine transform; to calculate an inverse sine transform using a code that calculates a forward sine transform, simply replace the function values u_j in the function call with the sine coefficients $\hat{u}_j^s$ and scale the result by $N/2$.*

In the case of the cosine expansion, we expand the discretized function u_j as a **finite cosine series** (a.k.a. a **discrete cosine transform**) such that

$$u_j=\sum_{n=0}^{N}\hat{u}_n^c\cos(k_nx_j)=\sum_{n=0}^{N}\hat{u}_n^c\cos\frac{\pi jn}{N}\quad\text{with}\quad k_n=\frac{2\pi n}{L}. \tag{5.48a}$$

Multiplying the above expression by $1/(c_jN)\cos(k_mx_j)$ where c_j is defined in (5.7b), summing over the values $j=0,\ldots,N$, and applying (5.7b), we find that

$$\frac{1}{N}\sum_{j=0}^{N}\frac{1}{c_j}\left[u_j=\sum_{n=0}^{N}\hat{u}_n^c\cos(k_nx_j)\right]\cos(k_mx_j)\quad\Rightarrow\quad\hat{u}_m^c=\frac{2}{c_mN}\sum_{j=0}^{N}\frac{u_j}{c_j}\cos(k_mx_j)=\frac{2}{c_mN}\sum_{j=0}^{N}\frac{u_j}{c_j}\cos\frac{\pi jm}{N}. \tag{5.48b}$$

Note that, again, the formula to compute the forward cosine transform [that is, to determine the $\hat{u}_m^c$ given the u_j, as shown in (5.48b)] is almost the same as the formula to compute the inverse cosine transform [that is, to determine the u_j given the $\hat{u}_n^c$, as shown in (5.48a)]. Thus,

Fact 5.8 *The same numerical code can be used for both the forward and inverse cosine transform; to calculate an inverse cosine transform using a code that calculates a forward cosine transform, simply replace the function values u_j in the function call with the scaled cosine coefficients $c_j\hat{u}_j^c$ and scale the result by $c_mN/2$, where c_j is defined in (5.7b).*

Returning our attention to the infinite series (5.46), we now show that there is a close relationship between the infinite sine and cosine series (5.46) and the infinite Fourier series (5.8a). Indeed, in the case of the sine series, if an **odd extension** of a function u with homogeneous Dirichlet boundary conditions at $x=0$ and $x=L/2$ is constructed by taking $u(-x)=-u(x)$, the resulting function is odd about $x=0$ and, significantly, *periodic* on $x\in[-L/2,L/2]$, thereby rendering it amenable to expansion via a Fourier series. Similarly, in the case of the cosine series, if an **even extension** of a function u with homogeneous Neumann boundary conditions at $x=0$ and $x=L/2$ is constructed by taking $u(-x)=u(x)$, the resulting function is even about $x=0$ and periodic on $x\in[-L/2,L/2]$, thereby, again, rendering it amenable to expansion via a Fourier series[24]. In either case, the coefficients of the sine or cosine series may be related to the Fourier expansion of

[24]Beware that, though these approaches construct continuous real odd and even functions with continuous first derivatives, the second (and higher) derivatives of the functions so constructed with odd extensions, and the third (and higher) derivatives of the functions so constructed with even extensions, are *not* necessarily continuous, so these approaches do not, in general, completely cure the problem of Gibbs phenomenon discussed in §5.9 for smooth functions with homogeneous Dirichlet or homogeneous Neumann (rather than periodic) boundary conditions, especially in situations in which the derivatives of these functions will be required. This issue is thus revisited in §5.13, where it is found that the appropriate clustering of gridpoints near the boundaries, as implied by Chebyshev methods (see Figure 5.10), is the most suitable method available to eliminate Gibbs phenomenon in the spectral representation and differentiation of nonperiodic functions.

the (odd or even, as appropriate) extension of the original function $u(x)$ on $x \in [0, L/2]$ via the relation given in (5.30), which is repeated here for convenience:

$$u(x) = \sum_{n=-\infty}^{\infty} \hat{u}_n e^{\mathrm{i}k_n x} = \sum_{n=-\infty}^{\infty} \hat{u}_n [\cos(k_n x) + \mathrm{i}\sin(k_n x)] \triangleq \sum_{n=0}^{\infty} [\hat{u}_n^c \cos(k_n x) + \hat{u}_n^s \sin(k_n x)] \tag{5.49a}$$

$$\Rightarrow \quad \begin{cases} \hat{u}_n^c = \hat{u}_n + \hat{u}_{-n} \\ \hat{u}_n^s = (\hat{u}_n - \hat{u}_{-n})\mathrm{i}. \end{cases} \quad \Leftrightarrow \quad \begin{cases} \hat{u}_n = (\hat{u}_n^c - \mathrm{i}\hat{u}_n^s)/2 \\ \hat{u}_{-n} = (\hat{u}_n^c + \mathrm{i}\hat{u}_n^s)/2. \end{cases} \tag{5.49b}$$

A real, even function expanded with a linear combination of sines and cosines, as seen in the form on the right in (5.49a), must have a zero coefficient in front of the odd basis functions (that is, $\hat{u}_n^s = 0$). By the lower-left relation in (5.49b), this implies that $\hat{u}_n = \hat{u}_{-n}$; combined with the fact that $\hat{u}_n = \hat{u}_{-n}^*$ because u is real [see (5.31)], this implies that,

- if $u(x)$ is real and even about $x = 0$, its Fourier coefficients $\hat{u}_n$ are real.

Similarly, a real, odd function expanded with a linear combination of sines and cosines must have a zero coefficient in front of the even basis functions (that is, $\hat{u}_n^c = 0$). By (5.49b), this implies that $\hat{u}_n = -\hat{u}_{-n}$; combined with the fact that $\hat{u}_n = \hat{u}_{-n}^*$, this implies that,

- if $u(x)$ is real and odd about $x = 0$, its Fourier coefficients $\hat{u}_n$ are imaginary.

Truncating the sums in (5.49a) and evaluating the (odd or even) extended function on a finite number of equispaced gridpoints, it is straightforward to use the relations in (5.49b) to calculate the coefficients of a finite sine series or finite cosine series from an FFT of the extended real function using Algorithm 5.7. However, in the case of the even extension, we know by the above discussion that the resulting Fourier series will be real, and in the case of the odd extension, we know by the above discussion that the resulting Fourier series will be imaginary. Similarly, during the subsequent inverse transforms, we know the resulting function values will be either even or odd about $x = 0$. We should be able to exploit these facts to compute the sine or cosine transform of real vectors even more efficiently; the following two sections show how.

5.11.1 The fast sine transform (FST)

As described above, the sine transform (5.47b) may be computed by performing an odd extension of the real data u_j onto $2N$ grid points, computing the FFT of this extended real function via an order $2N$ real FFT, then extracting the sine coefficients $\hat{u}_n^s$ from the result. As a more economical alternative, we now define a carefully constructed real auxiliary function w_j on N grid points, compute its Fourier expansion coefficients $\hat{w}_n$ via an order N real FFT, then leverage trigonometric identities to relate the Fourier coefficients $\hat{w}_n$ of the result to the desired sine coefficients of the original function, $\hat{u}_n^s$. Proceeding in this manner, defining a real auxiliary function (with both odd and even parts about $j = N/2$) from the original $N-1$ function values $u_1, u_2, \ldots, u_{N-1}$ such that

$$w_j = \begin{cases} 0 & \text{for } j = 0 \\ (u_{N-j} - u_j) + (u_j + u_{N-j}) \sin \dfrac{j\pi}{N} & \text{for } j = 1, \ldots, N-1, \end{cases} \tag{5.50a}$$

denoting the real and imaginary parts of its Fourier transform [see (5.23b)] as

$$\hat{w}_n = \hat{w}_n^R + \mathrm{i}\hat{w}_n^I = \frac{1}{N} \sum_{j=0}^{N-1} w_j e^{-\mathrm{i}k_n x_j} = \left[\frac{1}{N} \sum_{j=0}^{N-1} w_j \cos(k_n x_j)\right] - \mathrm{i}\left[\frac{1}{N} \sum_{j=0}^{N-1} w_j \sin(k_n x_j)\right]$$

Algorithm 5.12: The fast sine transform.

View Test

```
function [uhatS]=RFST(u,N)
% INPUT: u is a real column vector of length N-1 where N=2^s.
% OUTPUT: uhatS is a real column vector of length N-1.
% This code combines the u_j according to (5.47a), computes its RFFT,
% then extracts the uhat^s_n according to (5.47b).
w(1)=0; w(2:N)=(u(N-1:-1:1)-u)+(u+u(N-1:-1:1)).*sin([1:N-1]'*pi/N); what=RFFT(w,N);
uhatS(1,1)=real(what(1)); uhatS(2:2:N-2,1)=imag(what(2:N/2));
for n=3:2:N-1; uhatS(n,1)=uhatS(n-2,1)+2*real(what((n-1)/2+1)); end
end % function RFST
```

Algorithm 5.13: The inverse fast sine transform.

View

```
function [u]=RFSTinv(uhatS,N)
% Compute the inverse FST via applciation of Fact 5.6.
u=RFST(uhatS,N)*N/2;
end % function RFSTinv
```

with $k_n = 2\pi n/L$ and $x_j = jL/N$, and noting (5.47b), the identity $2\sin A\cos B = \sin(A+B)+\sin(A-B)$, and applying the symmetries about $j = N/2$, we may write

$$\hat{w}_n^R = \frac{1}{N}\sum_{j=0}^{N-1} w_j \cos\frac{2\pi jn}{N} = \frac{1}{N}\sum_{j=1}^{N-1}(u_j + u_{N-j})\sin\frac{j\pi}{N}\cos\frac{2\pi jn}{N} = \frac{2}{N}\sum_{j=1}^{N-1} u_j \sin\frac{j\pi}{N}\cos\frac{2\pi jn}{N}$$

$$= \begin{cases} \dfrac{2}{N}\displaystyle\sum_{j=1}^{N-1} u_j \sin\frac{j\pi}{N} & = \hat{u}_1^s \qquad \text{for } n = 0, \\ \dfrac{1}{N}\displaystyle\sum_{j=1}^{N-1} u_j \left[\sin\frac{(2n+1)j\pi}{N} - \sin\frac{(2n-1)j\pi}{N}\right] = \frac{\hat{u}_{2n+1}^s - \hat{u}_{2n-1}^s}{2} & \text{for } n = 1,2,\ldots,\dfrac{N}{2}-1, \end{cases}$$

and

$$\hat{w}_n^I = \frac{-1}{N}\sum_{j=0}^{N-1} w_j \sin\frac{2\pi jn}{N} = \frac{1}{N}\sum_{j=1}^{N-1}(u_j - u_{N-j})\sin\frac{2\pi jn}{N} = \frac{2}{N}\sum_{j=1}^{N-1} u_j \sin\frac{2\pi jn}{N} = \hat{u}_{2n}^s \quad \text{for } n = 1,2,\ldots,\frac{N}{2}-1.$$

The $\hat{u}_n^s$ may thus be determined from the $\hat{w}_n^s$ as follows:

$$\hat{u}_n^s = \begin{cases} \hat{w}_0^R & n = 1, \\ \hat{w}_{n/2}^I & n = 2,4,\ldots,N-2, \\ \hat{u}_{n-2}^s + 2\hat{w}_{(n-1)/2}^R & n = 3,5,\ldots,N-1. \end{cases} \tag{5.50b}$$

Equations (5.50a) and (5.50b) are implemented directly in the **fast sine transform** (FST) in Algorithm 5.12. For the inverse FST, note Fact 5.7 and its implementation in Algorithm 5.13.

5.11.2 The fast cosine transform (FCT)

As described in §5.11, the cosine transform (5.48b) may be computed by performing an even extension of the real data u_j onto $2N$ grid points, computing the FFT of this extended real function via an order $2N$ real FFT, then extracting the cosine coefficients $\hat{u}_n^c$ from the result. As a more economical alternative, as in §5.11.1 for the fast sine transform, we now define a (different) real auxiliary function w_j on N grid points, compute its

Algorithm 5.14: The fast cosine transform.

View
Test

```
function [uhatC]=RFCT(u,N)
% INPUT: u is a real column vector of length N+1 where N=2^s.
% OUTPUT: uhatC is a real column vector of length N+1.
% This code combines the u_j according to (5.48a), computes its RFFT,
% then extracts the uhat^c_n according to (5.48b).
w(1:N)=(u(1:N)+u(N+1:-1:2))+(u(N+1:-1:2)-u(1:N)).*sin([0:N-1]'*pi/N); what=RFFT(w,N);
uhatC(1,1)=real(what(1))/2; uhatC(N+1,1)=imag(what(1))/2; uhatC(3:2:N-1)=real(what(2:N/2));
u(1)=u(1)/2; u(N+1)=u(N+1)/2; uhatC(2,1)=(2/N)*cos(pi*[0:N]/N)*u;
for n=3:2:N-1; uhatC(n+1,1)=uhatC(n-1,1)-2*imag(what((n-1)/2+1)); end
end % function RFCT
```

Algorithm 5.15: The inverse fast cosine transform.

View

```
function [u]=RFCTinv(uhatC,N)
% Compute the inverse FCT via applciation of Fact 5.7.
uhatC(1)=2*uhatC(1); uhatC(N+1)=2*uhatC(N+1); u=RFCT(uhatC,N)*N/2;
u(1)=2*u(1); u(N+1)=2*u(N+1);
end % function RFCTinv
```

Fourier expansion coefficients $\hat{w}_n$ via an order N real FFT, then leverage trigonometric identities to relate the Fourier coefficients $\hat{w}_n$ of the result to the desired cosine coefficients of the original function, $\hat{u}^c_n$. Proceeding in this manner, defining in this case a real auxiliary function (with both even and odd parts about $j = N/2$) from the original $N+1$ function values $u_0, u_1, \ldots, u_N$ such that [cf. (5.50a)]

$$w_j = (u_j + u_{N-j}) + (u_{N-j} - u_j)\sin\frac{j\pi}{N} \quad \text{for } j = 0,\ldots,N-1, \tag{5.51a}$$

denoting the real and imaginary parts of its Fourier transform as

$$\hat{w}_n = \hat{w}^R_n + \mathrm{i}\hat{w}^I_n = \frac{1}{N}\sum_{j=0}^{N-1} w_j e^{-\mathrm{i}k_n x_j} = \Big[\frac{1}{N}\sum_{j=0}^{N-1} w_j \cos(k_n x_j)\Big] - \mathrm{i}\Big[\frac{1}{N}\sum_{j=0}^{N-1} w_j \sin(k_n x_j)\Big]$$

with $k_n = 2\pi n/L$, $x_j = jL/N$, and c_j defined as in (5.7b), and noting (5.48b), the identity $2\sin A\sin B = \cos(A-B) - \cos(A+B)$, and applying the symmetries about $j = N/2$, we may write

$$\hat{w}^R_n = \frac{1}{N}\sum_{j=0}^{N-1} w_j \cos\frac{2\pi jn}{N} = \frac{1}{N}\sum_{j=0}^{N-1}(u_j + u_{N-j})\cos\frac{2\pi jn}{N} = \frac{2}{N}\sum_{j=0}^{N-1}\frac{u_j}{c_j}\cos\frac{2\pi jn}{N} = c_{2n}\hat{u}^c_{2n} \quad \text{for } n = 0,1,\ldots,\frac{N}{2}.$$

and

$$\begin{aligned}\hat{w}^I_n &= \frac{-1}{N}\sum_{j=0}^{N-1} w_j \sin\frac{2\pi jn}{N} = \frac{1}{N}\sum_{j=1}^{N-1}(u_j - u_{N-j})\sin\frac{j\pi}{N}\sin\frac{2\pi jn}{N} = \frac{2}{N}\sum_{j=0}^{N-1} u_j \sin\frac{j\pi}{N}\sin\frac{2\pi jn}{N} \\ &= \begin{cases} 0 & \text{for } n = 0, \dfrac{N}{2}, \\ \dfrac{1}{N}\displaystyle\sum_{j=0}^{N-1} u_j\Big[\cos\frac{(2n-1)j\pi}{N} - \cos\frac{(2n+1)j\pi}{N}\Big] = \frac{\hat{u}^c_{2n-1} - \hat{u}^c_{2n+1}}{2} & \text{for } n = 1,2,\ldots,\dfrac{N}{2}-1. \end{cases}\end{aligned}$$

The $\hat{u}^c_n$ may thus be determined from the $\hat{w}_k$ as follows [cf. (5.50b)]:

$$\hat{u}^c_n = \begin{cases} \hat{w}^R_{n/2}/c_n & n = 0,2,\ldots,N, \\ \text{(5.48b)} & n = 1, \\ \hat{u}^c_{n-2} - 2\hat{w}^I_{(n-1)/2} & n = 3,5,\ldots,N-1. \end{cases} \tag{5.51b}$$

Unfortunately, it is not possible to extract $\hat{u}_1^c$ from the $\hat{w}_n$; we thus resort to the "brute force" equation (5.48b) for this coefficient. Equations (5.51a) and (5.51b) are implemented directly in the **fast cosine transform** (FCT) in Algorithm 5.14. For the inverse FCT, note Fact 5.8 and its implementation in Algorithm 5.15.

5.12 Extending finite Fourier series to stretched grids

The discretization of physical problems, such as those in cylindrical coördinates, often leads to smooth systems that are periodic in one or more directions, rendering computational approaches based on finite Fourier series well suited. Due to the physics of such problems, however, one often desires to **stretch** the numerical grid (see Figure 5.7, and further discussion in §8.1.3) in order to cluster gridpoints in areas of particular physical significance, such as the wake of a flow past a cylinder. Following Avital, Sandham, & Luo (2000), the present section illustrates a natural and smooth way to accomplish such clustering of gridpoints (other than the particular clustering implied by the Chebyshev method discussed in §5.13—see Figure 5.8 for an example) while preserving the exact differentiation capability of spectral methods.

Define x as the physical coördinate and s as a transformed coördinate via the stretching function

$$\frac{ds}{dx} = \alpha + \beta \sin^2(\pi s) = \alpha + \frac{\beta}{2} - \frac{\beta}{4}\left(e^{\mathrm{i}2\pi s} + e^{-\mathrm{i}2\pi s}\right) \quad \Rightarrow \quad x = \frac{\operatorname{atan}\left[\tan(\pi s)\sqrt{1+\beta/\alpha}\,\right]}{\pi\sqrt{\alpha\,(\alpha+\beta)}} \tag{5.52}$$

where, taking $\alpha = (-\beta + \sqrt{\beta^2 + 4/L^2})/2$, the transformed domain $s \in [0,1]$ corresponds to the physical domain $x \in [0,L]$ (further, taking $\beta = 0$ corresponds to an unstretched grid, and taking small $\beta > 0$ corresponds to a gentle clustering of gridpoints in the vicinity of $x = L/2$, as illustrated in Figure 5.7). Then, expand $u(x)$ in terms of the transformed coördinate $s \in [0,1]$ via a finite Fourier series expansion (see §5.4)

$$u[x(s)] = \sum_{n=0}^{N-1} \hat{u}_n e^{\mathrm{i}k_n s},$$

where the transformed wavenumber $k_n = 2\pi n$. [That is, for a given function $u(x)$ and a given value of N, define a uniform grid $s_j = jL/N$ for $j = 0,\ldots,N-1$, compute the corresponding stretched grid $x_j = x(s_j)$ via (5.52), then compute the corresponding $u_j = u(x_j) = u(x(s_j))$; the $\hat{u}_n$ for $n = 0,\ldots,N-1$ may then be determined from the u_j for $j = 0,\ldots,N-1$ via the FFT (see §5.4.1)].

Applying the chain rule to compute $f = du/dx$, we find that

$$f = \frac{du}{ds}\cdot\frac{ds}{dx} = \sum_{n=0}^{N-1} \mathrm{i}k_n\hat{u}_n e^{\mathrm{i}k_n s}\left[\alpha + \frac{\beta}{2} - \frac{\beta}{4}\left(e^{\mathrm{i}2\pi s} + e^{-\mathrm{i}2\pi s}\right)\right] = \sum_{n=0}^{N-1} \mathrm{i}k_n\hat{u}_n\left[\left(\alpha+\frac{\beta}{2}\right)e^{\mathrm{i}k_n s} - \frac{\beta}{4}\left(e^{\mathrm{i}k_{n-1}s} + e^{\mathrm{i}k_{n+1}s}\right)\right].$$

Writing out this sum and collecting all terms multiplying $e^{\mathrm{i}k_n s}$ in the result, we may write the first derivative of u in terms of the expansion

$$f = \frac{du}{dx} = \sum_{n=0}^{N-1} \hat{f}_n e^{\mathrm{i}k_n s} \tag{5.53a}$$

where

$$\hat{f}_n = -\left[\mathrm{i}k_{n-1}\frac{\beta}{4}\right]\cdot\hat{u}_{n-1} + \left[\mathrm{i}k_n\left(\alpha+\frac{\beta}{2}\right)\right]\cdot\hat{u}_n - \left[\mathrm{i}k_{n+1}\frac{\beta}{4}\right]\hat{u}_{n+1}. \tag{5.53b}$$

In the unstreched case (with $\alpha = 1$ and $\beta = 0$), we recover the relation $\hat{f}_n = \mathrm{i}k_n\hat{u}_n$, as noted in the unstretched Fourier representation in (5.10a). When $\beta \neq 0$ in this representation, we see that differentiation scatters energy to the neighboring wavenumbers in the series. Thus, to be able to represent the derivative of a function u with the same series as that which is used to expand u itself, we must insist that u be "smoothed" a bit (that is, that

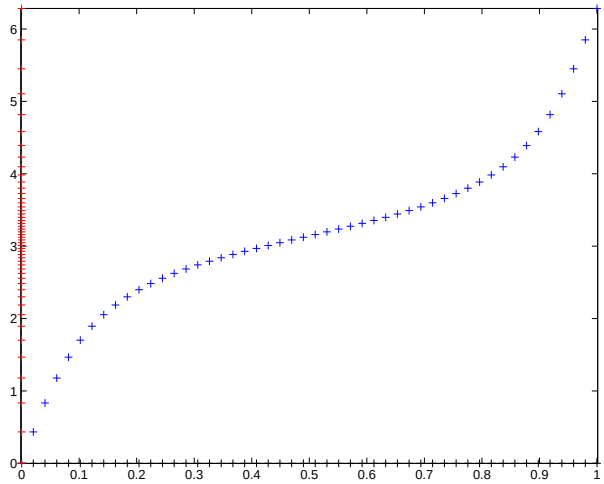


Figure 5.7: Grid stretching function defined by (5.52) with $\beta = 0.5$, $L = 2\pi$, and $N = 50$.

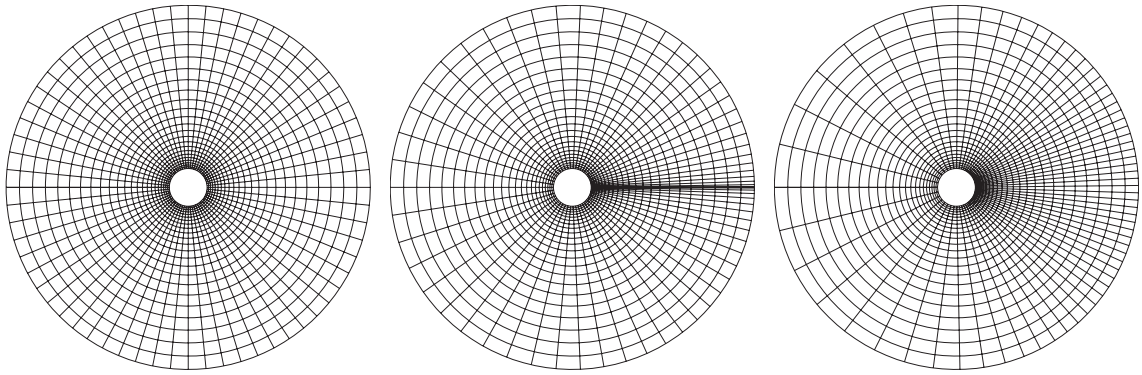


Figure 5.8: Example of the stretching of a cylindrical-coördinate grid with $N_\theta = 64$ in the azimuthal direction: (a) uniform in θ; (b) Chebyshev in θ; (c) smooth stretching in θ, as defined by (5.52) with $\beta = 0.3$.

the coefficients $\hat{u}_k$ corresponding to both the Nyquist frequency $k_{N/2} = \pi N/L$ and the next smaller frequency in the series, $k_{N/2-1} = 2\pi(N/2-1)/L$, be set to zero) before calculating its derivative.

Higher derivatives may be computed in an analogous fashion. For example, defining

$$g = \frac{d^2u}{dx^2} = \sum_{n=0}^{N-1} \hat{g}_n e^{ik_n s}, \tag{5.54a}$$

it follows in a similar fashion that

$$\begin{aligned}\hat{g}_n = &-\left[k_{n-2}(k_{n-2}+2\pi)\frac{\beta^2}{16}\right]\cdot\hat{u}_{n-2} + \left[k_{n-1}(k_{n-1}+\pi)\frac{2\alpha\beta+\beta^2}{4}\right]\cdot\hat{u}_{n-1} - \left[k_n^2(\alpha^2+\alpha\beta+3\beta^2/8)\right]\cdot\hat{u}_n \\ &+ \left[k_{n+1}(k_{n+1}-\pi)\frac{2\alpha\beta+\beta^2}{4}\right]\cdot\hat{u}_{n+1} - \left[k_{n+2}(k_{n+2}-2\pi)\frac{\beta^2}{16}\right]\cdot\hat{u}_{n+2}.\end{aligned} \tag{5.54b}$$

In the unstreched case ($\alpha = 1$ and $\beta = 0$), we recover the relation $\hat{g}_n = -k_n^2\hat{u}_n$, as noted in (5.10b). An illustration of how to use (5.52), (5.53), and (5.54) is given in `StretchedFourierTest.m` in the *NRC*.

Algorithm 5.16: A code for computing the Chebyshev function and its derivatives.

View Test

```
function [x]=Chebyshev(n,x,derivative,kind)
% Compute the Chebyshev polynomial of the 1st kind, T, and its 1st & 2nd derivatives, as
% well as the Chebyshev polynomial of the 2nd kind, U, and its derivative (see Wikipedia).
if nargin<4, kind=1; if nargin<3, derivative=0; end, end
switch kind
  case 1, switch derivative
    case 0, T(1)=1; T(2)=x; for j=3:n+1, T(j)=2*x*T(j-1)-T(j-2); end, x=T(n+1);
    case 1, x=n*Chebyshev(n-1,x,0,2);
    case 2, switch x
       case  1, x=(n^4-n^2)/3;
       case -1, x=(-1)^n*(n^4-n^2)/3;
       otherwise, x=n*((n+1)*Chebyshev(n,x,0,1)-Chebyshev(n,x,0,2))/(x^2-1); end
    otherwise, x=0; disp('Case not yet implemented'); end
  case 2, switch derivative
    case 0, U(1)=1; U(2)=2*x; for j=3:n+1, U(j)=2*x*U(j-1)-U(j-2); end, x=U(n+1);
    case 1, x=((n+1)*Chebyshev(n+1,x,0,1)-x*Chebyshev(n,x,0,2))/(s^2-1);
    otherwise, x=0; disp('Case not yet implemented'); end
end
end % function Chebyshev
```

5.13 Chebyshev representations

As illustrated in §5.9, Fourier methods are ill suited for the expansion of non-periodic functions. As presented in §5.11, this issue can be partially addressed in the case of functions with homogeneous Dirichlet or homogeneous Neumann boundary conditions by performing the appropriate sine or cosine expansion of an odd or even extension of the function itself. Unfortunately, such approaches fail to reconcile the problem completely, as the *derivatives* of functions with homogeneous Dirichlet or homogeneous Neumann boundary conditions are generally discontinuous under such extensions, and thus Gibbs phenomenon (see §5.9) will spoil the calculation of the derivatives of such functions even if the extended function itself is continuous.

A more powerful technique to leverage spectral methods for smooth but non-periodic functions is thus required. The method of choice is built on **Chebyshev polynomials** (a.k.a. **Chebyshev polynomials of the first kind**), which are functions on the interval $x \in [-1,1]$ that may be defined recursively as follows:

$$T_0(x) = 1, \tag{5.55a}$$
$$T_1(x) = x, \tag{5.55b}$$
$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x) \qquad \text{for } n = 2,3,\ldots \tag{5.55c}$$

So defined (and implemented in Algorithm 5.16), the next several Chebyshev polynomials are:

$$T_2(x) = 2x^2 - 1, \qquad T_6(x) = 32x^6 - 48x^4 + 18x^2 - 1,$$
$$T_3(x) = 4x^3 - 3x, \qquad T_7(x) = 64x^7 - 112x^5 + 56x^3 - 7x,$$
$$T_4(x) = 8x^4 - 8x^2 + 1, \qquad T_8(x) = 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1,$$
$$T_5(x) = 16x^5 - 20x^3 + 5x, \qquad T_9(x) = 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x,$$

as plotted in Figure 5.9. Note that $T_n(x)$ is a polynomial in x of degree n. The Chebyshev polynomials have many interesting properties, only three more of which will be mentioned here [in (5.56), (5.58), and (5.59)].

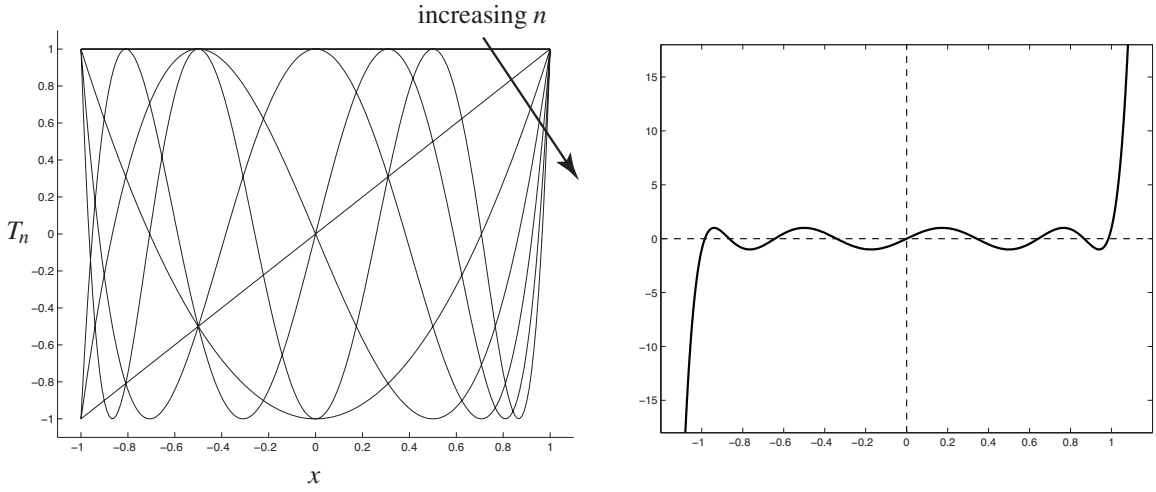


Figure 5.9: (left) The first seven Chebyshev polynomials $T_n(x)$ for $n = 0$ to 6 (increasing in the direction of the arrow), and (right) $T_9(x)$. Note that the Chebyshev polynomials are reminiscent of sines and cosines on the interior of the domain, but the higher-order Chebyshev polynomials get steep near the boundaries of the domain $x \in [-1,1]$; this renders such polynomials well suited for the expansion of nonperiodic functions with a reduced number of modes. Note also that the Chebyshev polynomials $T_n(x)$ for $n > 0$ increase monotonically for $x > 1$, with $T_n(x) \to \infty$ as $x \to \infty$, and they increase or decrease monotonically for $x < -1$, with $T_n \to \infty$ as $x \to -\infty$ for n even and $T_n \to -\infty$ as $x \to -\infty$ for n odd.

Perhaps the most important property that Chebyshev polynomials obey is the relation[25,26]

$$T_n(x) = \cos[n\,\theta(x)] \quad \text{where} \quad \theta = \operatorname{acos}(x) \quad \Rightarrow \quad \frac{d\theta}{dx} = \frac{-1}{\sin(\theta)}; \tag{5.56}$$

that is, if one defines the new coördinate $\theta = \operatorname{acos}(x)$ and considers this coördinate on the interval $\theta \in [0,\pi]$, then the Chebyshev polynomials are simply cosine functions in the new coördinate θ, that is, $T_n(x) = \cos[n\theta]$. Thus, once the physical coördinate x is stretched appropriately, one may think of the Chebyshev polynomials as just cosine functions in the stretched coördinate θ.

Chebyshev polynomials are particularly useful when used as basis functions in a **finite Chebyshev series** expansion of the form

$$u(x_j) \triangleq u_j = \sum_{n=0}^{N} \check{u}_n T_n(x_j) \quad \text{for} \quad x_j = \cos(j\pi/N) \in [-1,1] \quad \text{and} \quad j = 0,1,\ldots,N, \tag{5.57a}$$

which is equivalent to saying that cosine functions are used as a basis in a truncated series expansion in the stretched coördinate θ of the form

$$u_j = \sum_{n=0}^{N} \check{u}_n \cos(n\theta_j) \quad \text{on} \quad \theta_j = j\pi/N \in [0,\pi] \quad \text{and} \quad j = 0,1,\ldots,N. \tag{5.57b}$$

[25] The Chebyshev polynomials are sometimes said to be *defined* by (5.56), from which the recursive formulae in (5.55) follow. Whichever viewpoint you prefer, the equivalence between (5.55) and (5.56) is clear: the base cases (5.55a) and (5.55b) are equivalent to (5.56) with $n = 0$ and $n = 1$, and the recursive property (5.55c) is equivalent to the trigonometric identity $2\cos A\cos B = \cos(A+B)+\cos(A-B)$, taking $A = (n-1)x$ and $B = x$ and applying (5.56).

[26] Note that, as a direct result of (5.56), it follows that $|T_n(x)| \le 1$ on $x \in [-1,1]$ and $T_n(\pm 1) = (\pm 1)^n$.

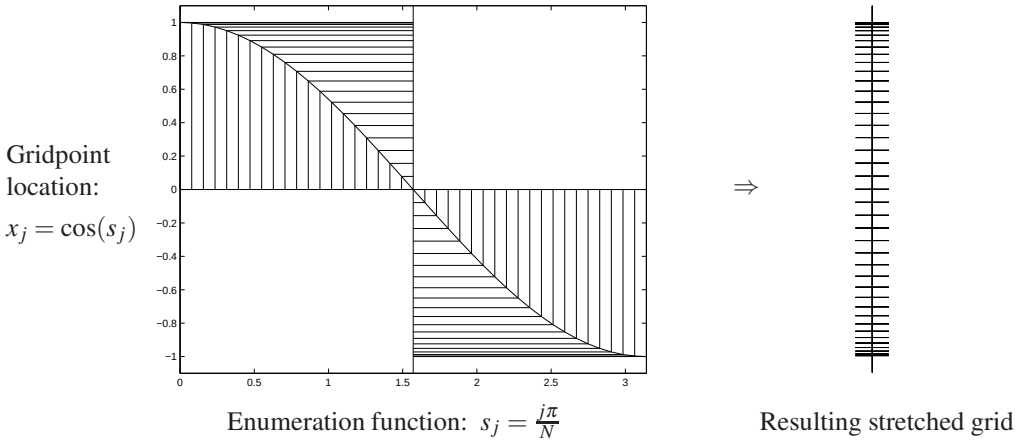


Figure 5.10: The **Chebyshev-Gauss-Lobatto stretching function** (a.k.a. **cosine grid**), which clusters the gridpoints enumerated from $j = 0$ to $j = N$ near the boundaries of the computational domain $x \in [-1,1]$, taking $N = 40$. Note that $j = 0$ corresponds to *upper* boundary, $x_0 = 1$, and $j = N$ corresponds to the lower boundary, $x_N = -1$.

Note that the finite Chebyshev series expansion (5.57b), written in terms of the stretched coördinate θ, is *identical* to the finite cosine series expansion (5.48a), and thus the inverse of the relationship (5.57b) (that is, to determine the coefficients $\breve{u}_n$ from the function values u_j) is given *exactly* by the inverse of the finite cosine series in (5.48b). It is only the interpretation of where the function is evaluated in physical space that has now changed. Note also that the variation of θ_j from 0 to π corresponds to the variation of x_j from 1 to -1; that is, opposite to the order you might otherwise anticipate. To summarize, once the function of interest is discretized on a uniform grid in the stretched coördinate, θ_j, which amounts to a so-called **cosine grid** in the physical coördinate, x_j (as illustrated in Figure 5.10)[27], then the fast cosine transform (FCT), as presented in §5.11.2, may be used *without modification* to transform back and forth between the physical space function values u_j and the corresponding transform space coefficients $\breve{u}_n$. We may thus, conveniently, refer to the FCT in Algorithm 5.14 as a **fast Chebyshev transform**.

The second important property of Chebyshev polynomials is the **weighted orthogonality** property

$$\frac{1}{\pi}\int_{-1}^{1} T_n(x)\,T_m(x)\,w(x)\,dx = \begin{cases} 1 & n = m = 0 \\ 1/2 & n = m \neq 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{where} \quad w(x) = 1/\sqrt{1-x^2}. \tag{5.58}$$

This weighted orthogonality property follows immediately from the transformation $x = \cos(\theta)$, which by (5.56) renders $T_n(x) = \cos(n\theta)$ and $d\theta = dx/\sqrt{1-x^2}$, and the orthogonality of the cosine functions in (5.7a).

The final important property of Chebyshev polynomials that we leverage is a useful relationship between the polynomials themselves and their derivatives, namely[28]

$$2T_n = \frac{1}{n+1}T'_{n+1} - \frac{1}{n-1}T'_{n-1}. \tag{5.59}$$

[27] Again, the subject of grid stretching is discussed much further in §8.1.3.

[28] Noting (5.56) and that $d\theta/dx = -1/\sin(\theta)$, we have $T'_n(x) = dT_n(x)/dx = n\sin(n\theta)/\sin(\theta)$; applying this result and (5.56) to the trigonometric identity $2\sin A\cos B = \sin(A+B) + \sin(A-B)$ with $A = \theta$ and $B = n\theta$, (5.59) follows immediately.

We now expand both a function $u(x)$ and its derivative $f(x) = u'(x)$ with finite Chebyshev series such that

$$u(x) = \sum_{n=0}^{N} \check{u}_n T_n(x) \quad \text{and} \quad f(x) = \sum_{n=0}^{N-1} \check{f}_n T_n(x), \tag{5.60}$$

noting that $f(x) = u'(x)$ must be a polynomial in x of degree $N-1$ and thus

$$\check{f}_N = 0, \quad \check{f}_{N+1} = 0. \tag{5.61a}$$

By (5.60) and the fact that $T_0' = 0$, it follows from $f(x) = u'(x)$ that

$$\sum_{n=0}^{N-1} \check{f}_n T_n(x) = \sum_{n=1}^{N} \check{u}_n T_n'(x).$$

On the LHS, substituting $T_0 = T_1'$ and $T_1 = T_2'/4$ [both easily verified using (5.55)], substituting (5.59) for T_n for $n = 2,3,\ldots,N-1$, then matching the coefficients of like powers of T_n' for $n = 1,2,\ldots,N$, it follows that

$$c_{n-1}\check{f}_{n-1} - \check{f}_{n+1} = 2n\check{u}_n \quad \text{for} \quad n = 1,2,\ldots,N, \qquad \text{where} \quad c_j \triangleq \begin{cases} 2 & \text{if } j = 0 \text{ or } j = N \\ 1 & \text{otherwise.} \end{cases} \tag{5.61b}$$

The relation (5.61) may be used two different ways. If the coefficients $\check{u}_n$ of the expansion of $u(x)$ are known, then the coefficients $\check{f}_n$ of the expansion of its derivative $f(x) = u'(x)$ may be found from (5.61b) evaluated first for $n = N$ [noting (5.61a)], then for $n = N-1$, $n = N-2$, etc. Conversely, if the coefficients $\check{f}_n$ of the expansion of $f(x)$ are known, then the coefficients $\check{u}_n$ of the expansion of its integral,

$$u(x) = \int_{-1}^{x} f(x')\,dx', \tag{5.62}$$

may be found directly from (5.61b) for $n = 1,2,\ldots,N$; the constant of integration $\check{u}_0$ may then be found by evaluating (5.62) at $x = -1$, which, inserting the expansion for $u(-1)$ from (5.60), gives

$$0 = \sum_{n=0}^{N} \check{u}_n (-1)^n \quad \Rightarrow \quad \check{u}_0 = -\sum_{n=1}^{N} \check{u}_n (-1)^n.$$

As a final note, we mention here that there exists wide assortment of orthogonal polynomials[29] available in the literature, most of which are named after famous mathematicians (Bessel, Jacobi, Legendre, Laguerre, Hermite, etc.). The reason that Chebyshev polynomials are the most important of these from a numerical perspective is the existence of the Fast Chebyshev Transform technique described above, which allows a Chebyshev transform to be calculated in $\sim 5N\log(N)]$ flops instead of $\sim 8N^2$ flops; for large N, as discussed in Footnote 12 on page 152, this distinction is remarkable.

Shifted Chebyshev representations

As described above, Chebyshev polynomials are defined over the interval $[-1,1]$. However, it is straightforward to shift these orthogonal polynomials to any finite domain $[a,b]$ via the transformation

$$T_n^*(x) = T_n[y(x)] \quad \text{where} \quad y(x) = \frac{2x-(a+b)}{b-a}. \tag{5.63}$$

[29] **Orthogonal polynomials** are sets of functions $f_n(x)$ which are polynomials of order n [like the functions $T_n(x)$ defined in (5.55)] which, when integrated over the appopriate domain $[a,b]$ with the appropriate weighting function $w(x)$, satisfy a weighted orthogonality property like that in (5.58); see also the last paragraph of §5.14.

The resulting **shifted Chebyshev polynomials** $T_n^*(x)$ are themselves polynomials in x of degree n and satisfy the weighted orthogonality property

$$\int_a^b T_n^*(x)\,T_m^*(x)\,w^*(x)\,dx = 0 \;\text{ if }\; n \neq m, \qquad \text{where} \quad w^*(x) = 1/\sqrt{1-[y(x)]^2}. \tag{5.64}$$

In the special case that we shift to the domain $[0,1]$, the shifted Chebyshev polynomials are given by $T_n^*(x) = T_n(2x-1)$, the first eight of which are thus

$$\begin{aligned}
T_0^*(x) &= 1, & T_4^*(x) &= 128x^4 - 256x^3 + 160x^2 - 32x + 1,\\
T_1^*(x) &= 2x - 1, & T_5^*(x) &= 512x^5 - 1280x^4 + 1120x^3 - 400x^2 + 50x - 1,\\
T_2^*(x) &= 8x^2 - 8x + 1, & T_6^*(x) &= 2048x^6 - 6144x^5 + 6912x^4 - 3584x^3 + 840x^2 - 72x + 1.\\
T_3^*(x) &= 32x^3 - 48x^2 + 18x - 1, & T_7^*(x) &= 8192x^7 - 28672x^6 + 39424x^5 - 26880x^4 + 9408x^3 - 1568x^2 + 98x - 1.
\end{aligned}$$

Plotted, these polynomials look just like those in Figure 5.9 shifted to the domain $[0,1]$.

5.14 Fourier-Bessel representations

Noting the general comments at the beginning of §5, it is readily seen that the infinite Fourier series expansion given in (5.8) may be generalized to incorporate a variety of different sets of orthogonal basis functions; such expansions are sometimes referred to **generalized Fourier series**. One convenient such expansion for functions $h(r)$ derived from systems written in polar coördinates [see, e.g., (11.34)] and endowed with homogeneous Dirichlet boundary conditions at $r = r_{\max}$ (while constrained to be finite near $r = 0$), called an **infinite Fourier-Bessel series**, may be written

$$h(r) = \sum_{p=0}^{\infty} \hat{h}_p J_\alpha\Big(\frac{\lambda_p r}{r_{\max}}\Big) \quad \text{with} \quad J_\alpha(\lambda_p) = 0 \quad \text{for} \quad p = 1,2,\ldots, \tag{5.65a}$$

where λ_p is the p'th zero of the Bessel function of the first kind of order α, denoted J_α, and the $\hat{h}_p$ are referred to as **Fourier-Bessel series coefficients**. Defining the normalization factor $b_q = [J_{\alpha+1}(\lambda_q)]^2/2$, multiplying the above expression by $(1/b_q)J_\alpha(\lambda_q r/r_{\max})\,r$, and integrating over the interval $(0, r_{\max})$, assuming $h(r)$ is sufficiently smooth (specifically, that the magnitude of its Fourier-Bessel series coefficients eventually decay exponentially with $|p|$) such that Fubini's theorem applies, then applying the orthogonality of the Bessel functions[30], we find that

$$\frac{1}{b_q}\int_0^{r_{\max}} \Big[h(r) = \sum_{p=0}^{\infty} \hat{h}_p J_\alpha\Big(\frac{\lambda_p r}{r_{\max}}\Big)\Big] J_\alpha\Big(\frac{\lambda_q r}{r_{\max}}\Big)\,r\,dr \quad \Rightarrow \quad \hat{h}_q = \frac{1}{b_q}\int_0^{r_{\max}} h(r) J_\alpha\Big(\frac{\lambda_q r}{r_{\max}}\Big)\,r\,dr. \tag{5.65b}$$

We refer to a **truncated Fourier-Bessel series approximation** $h^M(r)$ of the function $h(r)$ as the continuous function given by the expansion in (5.65a) with all Fourier-Bessel series coefficients with $p > M$ set to zero:

$$h^M(r) = \sum_{p=0}^{M} \hat{h}_p J_\alpha\Big(\frac{\lambda_p r}{r_{\max}}\Big). \tag{5.66}$$

As when passing from the Fourier series expansion to the Fourier integral expansion in §5.3, we now consider the limit of the infinite Fourier series (5.65) as $r_{\max} \to \infty$. To pass to this limit more easily, defining

[30] That is, $\frac{1}{b_p}\int_0^{r_{\max}} J_\alpha(\lambda_p r/r_{\max})\,J_\alpha(\lambda_q r/r_{\max})\,r\,dr = \delta_{pq}$ where $b_q = \frac{[J_{\alpha+1}(\lambda_q)]^2}{2}$.

$\Delta\lambda_q = (\lambda_{q+1} - \lambda_{q-1})/2$, we rearrange the coefficients of (5.65a) and multiply (5.65b) by $1/(\lambda_q \Delta\lambda_q)$:

$$h(r) = \sum_{p=0}^{\infty} \left[\frac{\hat{h}_p}{\lambda_q \Delta\lambda_q}\right] J_\alpha\left(\frac{\lambda_p r}{r_{\max}}\right) \left[\lambda_q \Delta\lambda_q\right] \quad \text{with} \quad J_\alpha(\lambda_p) = 0 \quad \text{for} \quad p = 1, 2, \ldots, \tag{5.67a}$$

$$\left[\frac{\hat{h}_p}{\lambda_q \Delta\lambda_q}\right] = \frac{1}{b_q \lambda_q \Delta\lambda_q} \int_0^{r_{\max}} h(r) J_\alpha\left(\frac{\lambda_p r}{r_{\max}}\right) r\, dr. \tag{5.67b}$$

Noting that $b_q \lambda_q \Delta\lambda_q = [J_{\alpha+1}(\lambda_q)]^2 \lambda_q (\lambda_{q+1} - \lambda_{q-1})/4 = 1$ and defining a continuous **Fourier-Bessel integral coefficient function** $\hat{h}(\lambda)$ such that $\hat{h}(\lambda = \lambda_p) = \hat{h}_p/(\lambda_q \Delta\lambda_q)$, interpreting the resulting expression for (5.67a) as a rectangular-rule approximation of an integral over λ, and taking the limit as $r_{\max} \to \infty$ (and thus $\Delta\lambda_q \to 0$), the sum converts to an integral, and we obtain [cf. (5.65) and (5.15)]

$$h(r) = \int_0^{\infty} \hat{h}(\lambda) J_\alpha(\lambda r) \lambda\, d\lambda \quad \text{for } r \in [0, \infty), \tag{5.68a}$$

$$\hat{h}(\lambda) = \int_0^{\infty} h(r) J_\alpha(\lambda r) r\, dr \quad \text{for } \lambda \in [0, \infty). \tag{5.68b}$$

This Fourier-Bessel integral expansion is commonly referred to as a **Hankel transform**.

Other generalized Fourier series may be developed based on the many available sets of orthogonal functions, most of which (including the **Bessel functions** as well as the **Chebyshev**, **Legendre**, **Hermite**, and **Laguerre polynomials**) may be derived from an associated Sturm-Liouville system [for further discussion, see footnote 1 on page 75]. Of these many available generalized Fourier series representations, only the Chebyshev series (see §5.13) is used to any significant degree in large-scale simulations, due primarily to the present lack of an available fast transform technique (akin to the FFT) for these alternative representations.

Exercises

Exercise 5.1 Verify (5.11a).

Exercise 5.2 Swapping the interpretation of k and x in (5.5a), for m, n integers and $x_n = 2\pi n/K$, we have

$$\frac{1}{K} \int_{-K/2}^{K/2} e^{\mathrm{i}kx_n} e^{-\mathrm{i}kx_m}\, dk = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{otherwise.} \end{cases} \tag{5.69}$$

Reviewing (5.8), (5.15), and (5.23), we may now complete the tetralogy by proposing what might be termed the forward and inverse **finite Fourier integral** expansion, defined for discrete functions $u_j = u(x_j)$ on $x_j = jh$ for $j = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ with $k \in [-\pi/h, \pi/h]$, as

$$\hat{u}(k) = \frac{h}{2\pi} \sum_{n=-\infty}^{\infty} u_n e^{-\mathrm{i}kx_n} \quad \Leftrightarrow \quad u_j = \int_{-\pi/h}^{\pi/h} \hat{u}(k) e^{\mathrm{i}kx_j}\, dk. \tag{5.70}$$

Substituting the first equation above into the second and applying (5.69) with $K = 2\pi/h$ and thus $x_n = hn$, verify that the above transform pair is valid. Note that, taking the limit that $h \to 0$, this transform pair reduces immediately to the infinite Fourier integral expansion in (5.15). [Note also that the complete **tetralogy** of Fourier transform pairs so generated is summarized succinctly in the introduction of §17.] Then, verify that Plancherel's and Parseval's theorems (5.41) may be written for this transform pair as

$$\frac{h}{2\pi} \sum_{j=-\infty}^{\infty} u_j^* v_j = \int_{-\pi/h}^{\pi/h} \hat{u}^*(k) \hat{v}(k)\, dk \qquad \Rightarrow \qquad \frac{h}{2\pi} \sum_{j=-\infty}^{\infty} |u_j|^2 = \int_{-\pi/h}^{\pi/h} |\hat{u}(k)|^2\, dk. \tag{5.71}$$

Exercise 5.3 As mentioned in Footnote 14 on 153, the "setup" operations (specifically, the bitswap) significantly reduce the execution speed of Algorithm 5.3 (the direct Cooley-Tukey variant of the FFT without reordering). Thus, write appropriate `FFTnonreorderedInit.m` and `FFTnonreorderedModified.m` routines which pull all of these "setup" calculations into an initialization routine. Compare the execution speed of the `FFTnonreordered.m` code provided and your new `FFTnonreorderedModified.m` (using the `tic/toc` commands) to measure the factor by which this adjustment accelerates your code for $N = 256$.

Exercise 5.4 Leveraging (5.26) in addition to (5.25), rewrite Algorithm 5.1 in such a way to handle $N = 2^s \cdot 3^t$ for any integer s and t. Compare the execution speed of this code for $N = 243$ and $N = 256$, and for $N = 2187$ and $N = 2048$. Does this comparison match your theoretical prediction? Discuss.

Exercise 5.5 Discretize the real functions $u(x) = \sin(2x)$, $v(x) = \sin(3x)$, and $w(x) = u(x) \cdot v(x)$ over the interval $x \in [0, L)$, for $L = 2\pi$, on the gridpoints $x_j = jL/N$ for $j = 0 \dots N-1$ where $N = 16$. Compute and plot (in Fourier space) the magnitude of the Fourier coefficients $\hat{u}_j$, $\hat{v}_j$, and $\hat{w}_j$ using the real FFT algorithm described in the text. Is Fact 5.5 evident? Discuss.

Exercise 5.6 Discretize the real functions $u(x) = \sin(2x)$, $v(x) = \sin(3x/2)$, and $w(x) = \sin(x/4)$ over the interval $x \in [0, L)$, for $L = 2\pi$, on the gridpoints $x_j = jL/N$ for $j = 0 \dots N-1$. Compute and plot (in physical space) the first and second derivatives of these functions via transformation using the real FFT, FST, and FCT algorithms described in the text, taking $N = 32$, $N = 64$, and $N = 128$ in each case, and compare with the exact solution. (Note that the derivative of a sine series is a cosine series, and the derivative of a cosine series is a sine series.) Discuss.

Exercise 5.7 Extend Parseval's theorem (5.41a) to the cases of the finite sine series (5.47a) & finite cosine series (5.48a), and verify your derivations with the test codes accompanying Algorithms 5.12 & 5.14.

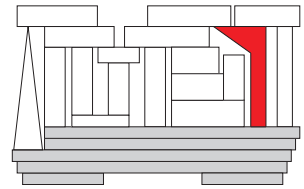
Exercise 5.8 Discretize the real functions $u(x) = \sin(\pi(x+1))$, $v(x) = \sin(3\pi(x+1)/4)$, $w(x) = \sin(\pi(x+1)/8)$ over the interval $x \in [-1, 1)$, on the gridpoints $x_j = \cos(j\pi/N)$ for $j = 0 \dots N-1$. Compute and plot the first and second derivatives of these functions using the Chebyshev methods described in the text, using the FCT algorithm described in the text to take the Chebyshev transform, taking $N = 32$, $N = 64$, and $N = 128$ in each case, and compare with the exact solution. Discuss.

References

Avital, EJ, Sandham, ND, & Luo, KH (2000) Stretched Cartesian grids for solution of the incompressible Navier-Stokes equations. *Int. J. Numer. Meth. Fluids* **33**, 897-918.

Boyd, JP (2001) *Chebyshev and Fourier Spectral Methods*. Dover.

Canuto, C, Hussaini, MY, Quarteroni, A, & Zang, TA (2006) *Spectral Methods: Fundamentals in Single Domains* & (2007) *—: Evolution to Complex Geometries & Applications to Fluid Mechanics.* Springer.



Chapter 6

Statistical representations

Contents

6.1 Random variables . . . 183
6.1.1 Conditional probability measures and Bayes' rule . . . 186
6.2 Statistical models . . . 186
6.2.1 The Gaussian distribution . . . 186
6.2.2 Other distributions . . . 187
6.3 Continuous-time random processes . . . 189
6.3.1 The joint description of two continuous-time random processes . . . 190
6.4 Discrete-time random processes . . . 190
6.4.1 The joint description of two discrete-time random processes . . . 191
Exercises . . . 192

6.1 Random variables

The **cumulative distribution function** (**CDF**) of a **random real vector** $\mathbf{x}$, denoted $f_{\mathbf{x}}(\underline{\mathbf{x}})$, is a mapping from $\underline{\mathbf{x}} \in \mathbb{R}^n$ to the real interval $[0,1]$ that monotonically increases in each of the components of $\underline{\mathbf{x}}$, and is defined

$$f_{\mathbf{x}}(\underline{\mathbf{x}}) = P(x_1 \leq \underline{x}_1, x_2 \leq \underline{x}_2, \ldots, x_n \leq \underline{x}_n),$$

where $\underline{\mathbf{x}}$ is some particular value of the random vector $\mathbf{x}$ and $P(S)$ denotes a **probability measure** that the conditions stated in S are true. In the scalar case, for example, $f_x(1) = 0.6$ means that it is 60% likely that the random variable x satisfies the condition $x \leq 1$. The CDF of a random complex vector $\mathbf{z}$, denoted $f_{\mathbf{z}}$, is defined in the same manner, with the real and imaginary component of each complex number treated separately; that is, $f_{\mathbf{z}}(\underline{\mathbf{z}}) = f_{\mathbf{x}}(\underline{\mathbf{x}})$ where $x_1 = \Re[z_1]$, $x_2 = \Im[z_1]$, $x_3 = \Re[z_2]$, etc. We will thus not distinguish between the real and complex cases in the discussion that follows.

For any random vector $\mathbf{x}$ whose CDF is differentiable everywhere (most are), the **probability density function** (**PDF**, a.k.a. **likelihood function**) $p_{\mathbf{x}}(\mathbf{x}') \geq 0$ is a scalar function of $\mathbf{x}'$ defined such that

$$f_{\mathbf{x}}(\underline{\mathbf{x}}) = \int_{-\infty}^{\underline{x}_1} \int_{-\infty}^{\underline{x}_2} \cdots \int_{-\infty}^{\underline{x}_n} p_{\mathbf{x}}(\mathbf{x}')\, dx'_1\, dx'_2 \cdots dx'_n \quad \Leftrightarrow \quad p_{\mathbf{x}}(\mathbf{x}') = \frac{\partial^n f_{\mathbf{x}}(\underline{\mathbf{x}})}{\partial \underline{x}_1\, \partial \underline{x}_2 \cdots \partial \underline{x}_n}\bigg|_{\underline{\mathbf{x}}=\mathbf{x}'}.$$

For small $|\Delta\mathbf{x}'|$, the quantity $p_{\mathbf{x}}(\mathbf{x}')\Delta x_1' \Delta x_2' \cdots \Delta x_n'$ represents the probability that the random vector $\mathbf{x}$ takes some value within a small rectangular region centered at the value $\mathbf{x}'$ and of width $\Delta x_i'$ in each coördinate direction $\mathbf{e}_i$. Note that the integral of $p_{\mathbf{x}}(\mathbf{x}')$ over all possible values of $\mathbf{x}'$ is unity, that is

$$\int_{\mathbb{R}^n} p_{\mathbf{x}}(\mathbf{x}')\, d\mathbf{x}' = 1.$$

If $\mathbf{x}$ is a random vector, it is often convenient to represent its CDF $f_{\mathbf{x}}(\underline{\mathbf{x}})$ and PDF $p_{\mathbf{x}}(\mathbf{x}')$ as a **joint CDF** and **joint PDF** by splitting $\mathbf{x}$ up into components. For example, if

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}, \quad \text{then} \quad f_{\mathbf{x}_1,\mathbf{x}_2}(\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2) \triangleq f_{\mathbf{x}}(\underline{\mathbf{x}}) \quad \text{and} \quad p_{\mathbf{x}_1,\mathbf{x}_2}(\mathbf{x}_1', \mathbf{x}_2') \triangleq p_{\mathbf{x}}(\mathbf{x}').$$

If the probability distribution is **separable** in $\mathbf{x}_1$ and $\mathbf{x}_2$, then we may write

$$f_{\mathbf{x}_1,\mathbf{x}_2}(\underline{\mathbf{x}}_1, \underline{\mathbf{x}}_2) = f_{\mathbf{x}_1}(\underline{\mathbf{x}}_1) f_{\mathbf{x}_2}(\underline{\mathbf{x}}_2) \quad \text{and} \quad p_{\mathbf{x}_1,\mathbf{x}_2}(\mathbf{x}_1', \mathbf{x}_2') = p_{\mathbf{x}_1}(\mathbf{x}_1') p_{\mathbf{x}_2}(\mathbf{x}_2'),$$

and we say that the random vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ are **uncorrelated** or **independent**.

Now consider a problem in which $\mathbf{r} = \mathbf{s} + \mathbf{t}$ where $\mathbf{s}$ and $\mathbf{t}$ are uncorrelated random vectors of order n, and thus a joint PDF may be written $p_{\mathbf{s},\mathbf{t}}(\mathbf{s}', \mathbf{t}') = p_{\mathbf{s}}(\mathbf{s}') p_{\mathbf{t}}(\mathbf{t}')$. Then

$$p_{\mathbf{r}}(\mathbf{r}') = \int_{\mathbb{R}^n} p_{\mathbf{s},\mathbf{t}}(\mathbf{s}', \mathbf{r}' - \mathbf{s}')\, d\mathbf{s}' = \int_{\mathbb{R}^n} p_{\mathbf{s}}(\mathbf{s}') p_{\mathbf{t}}(\mathbf{r}' - \mathbf{s}')\, d\mathbf{s}',$$

and thus, by (5.38), the coefficients of the Fourier integral expansion of $p_{\mathbf{r}}$ may be represented in terms of the coefficients of the Fourier integral expansions of $p_{\mathbf{s}}$ and $p_{\mathbf{t}}$ such that

$$\widehat{p_{\mathbf{r}}} = \frac{1}{(2\pi)^n} \widehat{p_{\mathbf{s}}} \widehat{p_{\mathbf{t}}}$$

at each wavenumber $\mathbf{k}$. Note that the Fourier integral expansion of a probability density function is often referred to as a **characteristic function**.

The **expected value** of a function $\mathbf{g}(\mathbf{x})$ of a random vector $\mathbf{x}$ is given by[1]

$$\mathscr{E}\{\mathbf{g}(\mathbf{x})\} = \int_{\mathbb{R}^n} \mathbf{g}(\mathbf{x}')\, p_{\mathbf{x}}(\mathbf{x}')\, d\mathbf{x}'$$

The expected value may be interpreted as the average of the quantity in question over many **experiments** (a.k.a. **realizations** or **ensembles**). In particular, the **mean** $\overline{\mathbf{x}}$ of the random vector $\mathbf{x}$ is defined as the expected value of $\mathbf{x}$ itself, that is,

$$\overline{\mathbf{x}} \triangleq \mathscr{E}\{\mathbf{x}\} = \int_{\mathbb{R}^n} \mathbf{x}'\, p_{\mathbf{x}}(\mathbf{x}')\, d\mathbf{x}'.$$

Note that a **zero-mean** random vector is one for which $\overline{\mathbf{x}} = 0$. The **covariance** $P_{\mathbf{x}}$ of a random vector $\mathbf{x}$ is defined as

$$P_{\mathbf{x}} \triangleq \mathscr{E}\{(\mathbf{x} - \overline{\mathbf{x}})(\mathbf{x} - \overline{\mathbf{x}})^H\} = \int_{\mathbb{R}^n} (\mathbf{x}' - \overline{\mathbf{x}})(\mathbf{x}' - \overline{\mathbf{x}})^H\, p_{\mathbf{x}}(\mathbf{x}')\, d\mathbf{x}'.$$

If a random vector $\mathbf{x}$ has a diagonal covariance matrix, its individual components are said to be **uncorrelated**. The k'th diagonal element of $P_{\mathbf{x}}$ is often denoted σ_k^2; the quantity σ_k^2 is called the **variance** of the k'th component of $\mathbf{x}$, and quantity σ_k is called its **standard deviation**. In the scalar case, we denote $P_x = \sigma^2$.

Consider now a random scalar distribution with mean $\overline{x}$ and variance σ^2 which is sampled n times. Though the expected value of each sample x_i is $\mathscr{E}\{x_i\} = \overline{x}$, each sample x_i in fact differs from the mean such that

$$x_i = \overline{x} + \delta_i \quad \text{where} \quad \mathscr{E}\{\delta_i\} = 0, \quad \mathscr{E}\{\delta_i \delta_j\} = \sigma^2 \delta_{ij}. \tag{6.1a}$$

[1] Note that the expected value of $\mathbf{g}(\mathbf{x})$ is sometimes deonted $\langle \mathbf{g}(\mathbf{x}) \rangle$ instead of $\mathscr{E}\{\mathbf{g}(\mathbf{x})\}$.

The average of these n samples thus differs from the mean as well, that is,

$$\frac{1}{n}\sum_{i=1}^{n} x_i = \overline{x} + \varepsilon. \tag{6.1b}$$

Subtracting $\overline{x}$ from both sides of (6.1b), applying (6.1a), and taking the expected value of the result reveals that $\mathscr{E}\{\varepsilon\} = 0$, whereas taking the expected value of the square of the result reveals that $\mathscr{E}\{\varepsilon^2\} = \sigma^2/n$. Thus,

$$\begin{aligned}\mathscr{E}\Big\{\sum_{i=1}^{n}\Big[x_i - \frac{1}{n}\sum_{k=1}^{n} x_k\Big]^2\Big\} &= \sum_{i=1}^{n}\mathscr{E}\{[x_i - (\overline{x}+\varepsilon)]^2\} = \sum_{i=1}^{n}\mathscr{E}\{(x_i-\overline{x})^2 - 2(x_i-\overline{x})\varepsilon + \varepsilon^2\} \\ &= n\sigma^2 - 2\mathscr{E}\{\underbrace{\textstyle\sum_{i=1}^{n}(x_i-\overline{x})}_{=n\varepsilon}\varepsilon\} + \sigma^2 = n\sigma^2 - 2\sigma^2 + \sigma^2 = (n-1)\sigma^2.\end{aligned}$$

The above analysis implies that, if a random scalar distribution with unknown mean and variance is sampled n times, the correct formulae to estimate the mean and variance are given by the **ensemble averages**

$$\overline{x} \approx \frac{1}{n}\sum_{k=1}^{n} x_k, \qquad \sigma^2 \approx \frac{1}{n-1}\sum_{i=1}^{n}\Big(x_i - \frac{1}{n}\sum_{k=1}^{n} x_k\Big)^2; \tag{6.2}$$

note in particular the $(n-1)$ term in the denominator of the second expression, which arises as a result of the fact that the value of $\overline{x}$ used in this expression is only approximate. Note also that the expected squared error in the first expression is σ^2/n [see discussion after (6.1b)]; that is, the error of this approximation decreases fairly slowly with the number of samples n. It follows similarly in the vector case that

$$\overline{\mathbf{x}} \approx \frac{1}{n}\sum_{k=1}^{n}\mathbf{x}_k, \qquad P_x \approx \frac{1}{n-1}\sum_{i=1}^{n}\Big(\mathbf{x}_i - \frac{1}{n}\sum_{k=1}^{n}\mathbf{x}_k\Big)\Big(\mathbf{x}_i - \frac{1}{n}\sum_{k=1}^{n}\mathbf{x}_k\Big)^H. \tag{6.3}$$

In the scalar case, a few other characterizations of probability distributions are common. The **mode** of a scalar distribution is the value of x for which the PDF reaches its maximum (if the PDF has more than one maximum, it is called **bimodal**, **trimodal**, or, in general, **multimodal**), whereas the **median** of a scalar distribution is the value of x for which the CDF equals 0.5; in general the mean, mode, and median of a scalar distribution are different. The k'th **central moment** of a scalar distribution is defined as

$$\mu_k \triangleq \mathscr{E}\{(x - \mathscr{E}\{x\})^k\} = \int_{-\infty}^{\infty}(x'-\overline{x})^k\, p_{\mathbf{x}}(x')\,dx'.$$

The second central moment is just the variance defined above, i.e., $\mu_2 = \sigma^2$. The third and fourth central moments are used to define the **skewness**,

$$\gamma_1 = \frac{\mu_3}{\sigma^3},$$

and the **kurtosis** (a.k.a. **excess kurtosis**), which is defined here such that

$$\gamma_2 = \frac{\mu_4}{\sigma^4} - 3.$$

Skewness is a measure of the asymmetry of a probability distribution, with postive skewness indicating a more elongated tail to the right than to the left, as exhibited by the **chi-squared**, **exponential**, and **gamma** distributions described in §6.2.2. Kurtosis is a measure of the "peakedness" of a probability distribution, with negative kurtosis indicating less elongated tails than the ubiquitous **Gaussain** distribution (see §6.2.1), and positive kurtosis indicating more elongated tails than the Gaussain distribution. [See in particular the **uniform** distribution in §6.2.2 with no tails whatsoever and a kurtosis of -1.2, and the **Laplace** distribution in §6.2.2 with long exponential tails and a kurtosis of $+3$.] Distributions with postive, negative, and aproximately zero kurtosis measures are termed **leptokurtic**, **platykurtic**, and **mesokurtic** respectively.

6.1.1 Conditional probability measures and Bayes' rule

As in §6.1, we again denote by $P(S)$ the probability that condition S is true. We also denote by $P(\overline{S}) = 1 - P(S)$ the probability that condition S is false, and introduce the notation $P(S|T)$ as the **conditional probability measure**—that is, the probability that condition S is true *given* that condition T is true. It follows that the probability that both condition S *and* condition T are true is $P(S,T) = P(S|T)P(T) = P(T|S)P(S)$, and thus

$$P(S|T) = \frac{P(T|S)\,P(S)}{P(T)}; \tag{6.4}$$

this is known as **Bayes' rule**. This formula is useful, and requires some interpretation to appreciate fully. As a prototypical example, suppose condition S is that you have a certain Sickness, whereas condition T is that you Tested positive for this sickness at the hospital. Then, by (6.4), the probability that you have this particular sickness given that you tested positive for it [that is, $P(S|T)$] is equal to $P(T|S)$ [that is, the probability that the test is positive if one indeed actually has the sickness—that is, 1 minus the "false negative" probability of the test, denoted $P(\overline{T}|S)$] times $P(S)$ [that is, the total probability that one has the sickness irrespective of the test] divided by $P(T)$ [that is, the total probability that you test positive for the sickness irrespective of whether or not you are actually sick]. Inserting some representative numbers, suppose:

- $P(S) = 0.0001$ [i.e., the probability that someone in your demographic has this sickness is only 0.01%];
- $P(\overline{T}|S) = 1 - P(T|S) = 0.001$ [i.e., the probability of a false negative test is 0.1%];
- $P(T|\overline{S}) = 0.0005$ [i.e., the probability of a false positive test is 0.05%];
- $P(T) = P(T|S)\,P(S) + P(T|\overline{S})\,P(\overline{S}) = 0.999 * 0.0001 + 0.0005 * 0.9999 \approx 0.0006$ [i.e., the overall probability of positive test is 0.06%].

At first glance, the second and third bullet points above appear to imply that this a very reliable test, with the odds of it being wrong only 0.1% for those who are sick, and only 0.05% for those who are not sick. However, by (6.4), the probability that you have the sickness given that you test positive for it is only $P(S|T) = 0.999 * 0.0001/0.0006 \approx 1/6$; that is, the test is not nearly as conclusive as it might first seem! This result is logical because on average, out of every 10000 people that are tested, $10000 * P(S) = 1$ actually has the sickness, but $10000 * P(T) = 6$ test positive; thus, if you are one of the six that test positive, the odds that you actually have the sickness are only $1/6$. Therefore, if only one test is to be performed, any conclusive test needs to have the probability of a false positive, $P(T|\overline{S})$, much closer to zero than in the above example, thus highlighting the extraördinary care that is often required to interpret statistics correctly.

Note that, in the special case that the conditions S and T are uncorrelated, $P(S|T) = P(S|\overline{T}) = P(S)$ and $P(T|S) = P(T|\overline{S}) = P(T)$ and thus, as mentioned in §6.1, the probability that both conditions S and T are true is simply $P(S,T) = P(S)P(T)$. Note also that Bayes rule extends immediately from the condition S being a logical state [e.g., 0 (sick) or 1 (not sick)] to a condition S related to a CDF (e.g., $x_1 \le \underline{x}_1$).

6.2 Statistical models

6.2.1 The Gaussian distribution

The nature of random physical systems very often leads to PDFs that are well approximated by an appropriately-normalized Gaussian function [see (5.18a)]. In the scalar case, the PDF $p_x(x')$ and CDF $f_x(x')$ of a **Gaussian distribution** (a.k.a. **normal distribution**) may be written[2]

$$p_x(x') = \frac{1}{(2\pi)^{1/2}\sigma}\exp\Big[-\frac{(x'-\overline{x})^2}{2\sigma^2}\Big] \qquad \Leftrightarrow \qquad f_x(x') = \frac{1}{2}\Big[1 + \mathrm{erf}\Big(\frac{x'-\overline{x}}{\sigma\sqrt{2}}\Big)\Big].$$

[2]The integral of the Gaussian function e^{-x^2} is commonly called the **error function**, and is denoted $\mathrm{erf}(y) = \frac{2}{\sqrt{\pi}}\int_0^y e^{-x^2}dx$.

It follows directly from these formulae that $\int_{-\infty}^{\infty} p_x(x')\,dx' = 1$, $\mathscr{E}\{x\} = \overline{x}$, and $\mathscr{E}\{(x-\overline{x})^2\} = \sigma^2$. The quantities $\overline{x}$ and σ^2 are thus identified as the mean and variance of this distribution. The notation $x = N(\overline{x}, \sigma^2)$ is sometimes used to denote such a Gaussian distribution for the random scalar x. Note that the mean, median, and mode of a scalar Gaussian distribution coincide, whereas the skewness and kurtosis of a scalar Gaussian distribution are zero.

The idea of a Gaussian "bump" extends naturally to random vectors; in two dimensions, $p_{\mathbf{x}}(\mathbf{x}')$ may be visualized as a localized "bump" of a deformed carpet. In the multivariate case, a Gaussian PDF may be written[3]

$$p_{\mathbf{x}}(\mathbf{x}') = \frac{1}{(2\pi)^{n/2}|P_{\mathbf{x}}|^{1/2}} \exp\Big[-\frac{1}{2}(\mathbf{x}'-\overline{\mathbf{x}})^H P_{\mathbf{x}}^{-1}(\mathbf{x}'-\overline{\mathbf{x}})\Big]. \tag{6.5}$$

It follows directly from this formula that $\int_{\mathbb{R}^n} p_{\mathbf{x}}(\mathbf{x}')\,d\mathbf{x}' = 1$, $\mathscr{E}\{\mathbf{x}\} = \overline{\mathbf{x}}$, and $\mathscr{E}\{(\mathbf{x}-\overline{\mathbf{x}})(\mathbf{x}-\overline{\mathbf{x}})^H\} = P_{\mathbf{x}}$. The quantities $\overline{\mathbf{x}}$ and $P_{\mathbf{x}}$ are thus identified as the mean and covariance of this distribution. The notation $\mathbf{x} = N(\overline{\mathbf{x}}, P_{\mathbf{x}})$ is sometimes used to denote such a Gaussian distribution for the random vector $\mathbf{x}$.

6.2.2 Other distributions

There are literally dozens of random distributions other than the Gaussian distribution introduced above that are appropriate for various problems in physics and engineering. We mention briefly just five of them here.

The chi-squared distribution

Some random variables are necessarily positive. As an example, consider the number x formed as the sum of the squares of an integer number k of random variables z_i, each of which itself is a random number with a Gaussian distribution with mean 0 and standard deviation 1, i.e.,

$$x = \sum_{i=1}^{k} z_i^2 \quad \text{where} \quad z_i = N(0,1).$$

In this case (as may verified, with some effort, via the appropriate integrations), x itself may be considered a random number with a **chi-squared distribution** defined by a PDF and CDF, with $k > 0$, given by

$$p_x(x;k) = \begin{cases} x^{k/2-1}e^{-x/2}/[2^{k/2}\Gamma(k/2)] & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases} \quad \Leftrightarrow \quad f_x(x;k) = \begin{cases} \gamma(k/2,x/2)/\Gamma(k/2) & \text{for } x \geq 0 \\ 0 & \text{for } x < 0, \end{cases}$$

where Γ denotes the **gamma function** $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt$ [note that $\Gamma(n) = (n-1)!$ when n is an integer] and $\gamma(z,x)$ denotes the **lower incomplete gamma function** $\gamma(z,x) = \int_0^x t^{z-1}e^{-t}\,dt$. The mean of the chi-squared distribution is $\overline{x} = k$, its mode is 0 for $k < 2$ and $k-2$ for $k \geq 2$, its variance is $\sigma^2 = 2k$, its skewness is $\gamma_1 = \sqrt{8/k}$, and its kurtosis is $\gamma_2 = 12/k$.

The exponential distribution

The **exponential distribution** is another common distribution for positive random variables, defined by a PDF and CDF given by

$$p_x(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases} \quad \Leftrightarrow \quad f_x(x;\lambda) = \begin{cases} 1-e^{-\lambda x} & \text{for } x \geq 0 \\ 0 & \text{for } x < 0, \end{cases}$$

where $\lambda > 0$ is referred to as the **rate parameter**. The mean of the exponential distribution is $\overline{x} = 1/\lambda$, its mode is 0, its variance is $\sigma^2 = 1/\lambda^2$, its skewness is $\gamma_1 = 2$, and its kurtosis is $\gamma_2 = 6$.

Note that a chi-squared distribution with $k = 2$ and an exponential distribution with $\lambda = 1/2$ coincide.

[3] Unfortunately, there is no convenient closed-form expression for the CDF of a multivariate Gaussian distribution.

The gamma distribution

Consider now the number x formed as the sum of an integer number m of random variables z_i, each of which itself is a random number with an exponential distribution with rate parameter λ, i.e.,

$$x = \sum_{i=1}^{m} z_i.$$

In this case (as may verified, with some effort, via the appropriate integrations), x itself may be considered a random number with a **gamma distribution** defined by a PDF and CDF, with $m > 0$ and $\lambda > 0$, given by

$$p_x(x;m,\lambda) = \begin{cases} \lambda^m x^{m-1} e^{-\lambda x}/\Gamma(m) & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases} \quad \Leftrightarrow \quad f_x(x;m,\lambda) = \begin{cases} \gamma(m,x\lambda)/\Gamma(m) & \text{for } x \geq 0 \\ 0 & \text{for } x < 0. \end{cases}$$

The mean of the gamma distribution is $\overline{x} = m/\lambda$, its mode is 0 for $m < 1$ and $(m-1)/\lambda$ for $m \geq 1$, its variance is $\sigma^2 = m/\lambda^2$, its skewness is $\gamma_1 = \sqrt{4/m}$, and its kurtosis is $\gamma_2 = 6/m$. Note also that, as m is made large for a given λ, the gamma distribution approaches a Gaussian distribution with mean and variance given above, and skewness and kurtosis approaching zero.

Note that the chi-squared distribution is a special case the gamma distribution with $m = k/2$ and $\lambda = 1/2$, whereas the exponential distribution is a special case of the gamma distribution with $m = 1$.

The Laplace distribution

Consider now the number x formed as the *difference* of two random variables z_1 and z_2, each of which itself is a random number with an exponential distribution with rate parameter λ, i.e.,

$$x = \overline{x} + z_1 - z_2.$$

In this case (as may verified, with some effort, via the appropriate integrations), x itself may be considered a random number with a **Laplace distribution** (a.k.a. **double exponential distribution**) defined by a PDF and CDF, with $\lambda > 0$, given by

$$p_x(x;\lambda) = \lambda e^{-\lambda|x-\overline{x}|}/2 \quad \Leftrightarrow \quad f_x(x;\lambda) = \begin{cases} e^{\lambda(x-\overline{x})}/2 & \text{for } x < \overline{x} \\ 1 - e^{-\lambda(x-\overline{x})}/2 & \text{for } x \geq \overline{x}. \end{cases}$$

The mean of the Laplace distribution is $\overline{x}$, its mode is $\overline{x}$, its variance is $\sigma^2 = 2/\lambda^2$, its skewness is $\gamma_1 = 0$, and its kurtosis is $\gamma_2 = 3$.

The uniform distribution

The **uniform distribution** is defined by a PDF and CDF given by

$$p_x(x;a,b) = \begin{cases} 1/(b-a) & \text{for } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases} \quad \Leftrightarrow \quad f_x(x;a,b) = \begin{cases} 0 & \text{for } x < a \\ (x-a)/(b-a) & \text{for } a \leq x \leq b \\ 1 & \text{for } x > b. \end{cases}$$

The mean of the uniform distribution is $\overline{x} = (a+b)/2$, its variance is $\sigma^2 = (b-a)^2/12$, its skewness is $\gamma_1 = 0$, and its kurtosis is $\gamma_2 = -1.2$.

6.3 Continuous-time random processes

A random vector $\mathbf{w}$ that is a function of the continuous time variable t is called a **continuous-time random process**, denoted $\mathbf{w}(t)$. The PDF of a continuous-time random process $\mathbf{w}(t)$ is denoted $p_{\mathbf{w}}(\mathbf{w}';t)$. For any given time t, the PDF $p_{\mathbf{w}}(\mathbf{w}';t)$ describes the probability distribution of $\mathbf{w}(t)$ as described in §6.1, and thus

$$\int_{\mathbb{R}^n} p_{\mathbf{w}}(\mathbf{w}';t)\,d\mathbf{w}' = 1.$$

The PDF $p_{\mathbf{w}}(\mathbf{w}';t)$ alone does not completely describe a continuous-time random process. We must also quantify the expected *time correlation* of the vector $\mathbf{w}$ with itself, which may be accomplished with the **autocorrelation** $R_{\mathbf{w}}(\tau;t)$ defined such that

$$R_{\mathbf{w}}(\tau;t) \triangleq \mathscr{E}\{\mathbf{w}(t+\tau)\,\mathbf{w}^H(t)\}.$$

Note that $R_{(\mathbf{w}-\overline{\mathbf{w}})}(0;t) = P_{\mathbf{w}}(t)$; thus, the autocorrelation may be thought of as an appropriate generalization of the covariance for a continuous-time random process $\mathbf{w}(t)$ which also describes its time correlation.

The PDF $p_{\mathbf{w}}(\mathbf{w}';t)$ and autocorrelation $R_{\mathbf{w}}(\tau;t)$ may vary with t, in which case the random process $\mathbf{w}(t)$ is said to be **nonstationary**. In many cases of interest, however, the PDF and autocorrelation of a continuous-time random process $\mathbf{w}(t)$ (and, thus, the mean and covariance of the PDF) do *not* vary in time (even though the random variable $\mathbf{w}(t)$ itself does); in such cases, the random process $\mathbf{w}(t)$ is said to be **stationary**, and its PDF and autocorrelation are denoted $p_{\mathbf{w}}(\mathbf{w}')$ and $R_{\mathbf{w}}(\tau)$. In general, if a random process $\mathbf{w}(t)$ is nonstationary, its expected value must be estimated by **ensemble averaging**, as defined in §6.1. In the special case that the random process $\mathbf{w}(t)$ is stationary, its expected value may be estimated instead by **time averaging** of a single member of the ensemble for a long time:

$$\mathscr{E}\{\mathbf{g}[\mathbf{w}]\} = \lim_{T\to\infty}\left[\frac{1}{T}\int_0^T \mathbf{g}[\mathbf{w}(t)]\,dt\right];$$

this property is commonly referred to as **ergodicity**.

As with the PDF of a random variable, the nature of random physical systems often leads to autocorrelations that are well approximated by a Gaussian-in-time behavior. Further, in many systems, an autocorrelation that decouples into the following separable form is representative:

$$R_{(\mathbf{w}-\overline{\mathbf{w}})}(\tau;t) = S_{\mathbf{w}}(t)\,\delta^{\sigma}(\tau) \quad \text{with} \quad \delta^{\sigma}(\tau) \triangleq \frac{1}{\sigma\sqrt{2\pi}}e^{-\tau^2/(2\sigma^2)}, \tag{6.6}$$

where $S_{\mathbf{w}}(t)$ is called the **spectral density**[4] of the continuous-time random process $\mathbf{w}(t)$, and the definition of the normalized Gaussian $\delta^{\sigma}(\tau)$ (with unit area) is identical to that given in §5.3.3. The notation $\mathbf{w}(t) = N(\overline{\mathbf{w}}, S_{\mathbf{w}}, \sigma)$ will sometimes be used to denote a continuous-time random process $\mathbf{w}(t)$ with both a Gaussian PDF and a decoupled autocorrelation of the above separable form.

Note that, at any time t, the covariance of a zero-mean random process $\mathbf{w}(t)$ with a decoupled autocorrelation of the above form is given by $P_{\mathbf{w}}(t) = S_{\mathbf{w}}(t)\,\delta^{\sigma}(0) = S_{\mathbf{w}}(t)/(\sigma\sqrt{2\pi})$. *This is where the analysis of continuous-time random processes gets delicate, and special attention is warranted.* To simplify certain important derivations, such as that leading to the continuous-time Kalman filter in §22.1, it will be quite tempting to idealize the autocorrelation of a certain random process $\mathbf{w}(t)$ as uncorrelated in time, which implies that $R_{\mathbf{w}}(\tau;t) = 0$ for $\tau \neq 0$; specifically, it will be tempting to try to accomplish this by taking the $\sigma \to 0$ limit of the autocorrelation described above, thereby idealizing $\mathbf{w}(t)$ as a **white noise** processes with uniform spectral content across all frequencies (see §5.3.1). Indeed, many otherwise respectable texts give in to this

[4] Occasionally, the spectral density of a continuous-time random process is mistakenly called its covariance. To avoid confusion, this sloppy practice should be avoided.

temptation. We assert here, however, that one must resist this temptation, because by so doing the covariance $P_{\mathbf{w}}(t)$ of the random vector $\mathbf{w}(t)$ at any time t, which reflects the expected "energy" in each component of $\mathbf{w}(t)$ at any instant, would be infinite, which is not physical.

To avoid a nonphysical modelization[5] of this sort, we may simply modelize[5] the random process $\mathbf{w}(t)$ as **nearly white**. That is, $R_{\mathbf{w}}(\tau;t)$ may be assumed to be described by an autocorrelation of the form given above for small but finite σ, and thus $R_{\mathbf{w}}(\tau;t) \approx 0$ for $|\tau| \gtrsim c \cdot \sigma$ for some $c = O(10)$. It is found that the actual value of σ selected in this model is completely inconsequential, so long as it is substantially smaller than any of the significant characteristic response times of the dynamic system under consideration.

6.3.1 The joint description of two continuous-time random processes

When characterizing the relationship between two continuous-time random processes $\mathbf{v}(t)$ and $\mathbf{w}(t)$, the **joint PDF** $p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';\tau,t)$ may be used. Extending the definition of the PDF presented in §6.1, for small $|\Delta\mathbf{v}'|$ and $|\Delta\mathbf{w}'|$, the quantity $p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';\tau,t)\Delta v'_1 \cdots \Delta v'_m \Delta w'_1 \cdots \Delta w'_n$ represents the probability that the $\mathbf{v}(t+\tau)$ takes a value within a rectangular region centered at $\mathbf{v}'$ and of width $\Delta v'_i$ in each coördinate direction while $\mathbf{w}(t)$ takes a value within a rectangular region centered at $\mathbf{w}'$ and of width $\Delta w'_i$ in each coördinate direction. It follows as before that, for any given τ and t,

$$\int_{\mathbb{R}^n}\int_{\mathbb{R}^m} p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';\tau,t)\,d\mathbf{v}'\,d\mathbf{w}' = 1.$$

The expected value of a function $\mathbf{g}(\mathbf{v},\mathbf{w})$ of two continuous-time random processes $\mathbf{v}(t)$ and $\mathbf{w}(t)$ is

$$\mathscr{E}\{\mathbf{g}[\mathbf{v}(t+\tau),\mathbf{w}(t)]\} = \int_{\mathbb{R}^n}\int_{\mathbb{R}^m} \mathbf{g}[\mathbf{v}',\mathbf{w}']\,p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';\tau,t)\,d\mathbf{v}'\,d\mathbf{w}'.$$

In particular, the **cross correlation** of $\mathbf{v}(t)$ and $\mathbf{w}(t)$ is defined by

$$R_{\mathbf{vw}}(\tau;t) \triangleq \mathscr{E}\{\mathbf{v}(t+\tau)\,\mathbf{w}^H(t)\}.$$

Note that the autocorrelation of $\mathbf{w}(t)$ is just the cross correlation of $\mathbf{w}(t)$ with itself, that is, $R_{\mathbf{w}}(\tau;t) = R_{\mathbf{ww}}(\tau;t)$. If $R_{\mathbf{vw}}(\tau;t) = 0$ for all τ and t, the random processes $\mathbf{v}(t)$ and $\mathbf{w}(t)$ are said to be **uncorrelated**.

The joint PDF $p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';\tau,t)$ and cross correlation $R_{\mathbf{vw}}(\tau;t)$ may, in general, depend on t. In many cases of interest, however, the joint PDF and cross correlation of two continuous-time random processes $\mathbf{v}(t)$ and $\mathbf{w}(t)$ do not vary in time; in such cases, the two random processes $\mathbf{v}(t)$ and $\mathbf{w}(t)$ are said to be **jointly stationary**, and their joint PDF and cross correlation are denoted $p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';\tau)$ and $R_{\mathbf{vw}}(\tau)$.

6.4 Discrete-time random processes

A random vector $\mathbf{w}$ that is a function of the discrete time index k is called a **discrete-time random process**, denoted $\mathbf{w}_k$. The PDF of a discrete-time random process $\mathbf{w}_k$ is denoted $p_{\mathbf{w}}(\mathbf{w}';k)$. For any given k, the PDF $p_{\mathbf{w}}(\mathbf{w}';k)$ describes the probability distribution of $\mathbf{w}_k$ as described in §6.1, and thus

$$\int_{\mathbb{R}^n} p_{\mathbf{w}}(\mathbf{w}';k)\,d\mathbf{w}' = 1.$$

[5] The British and the Americans have long debated whether the noun meaning "the devising or use of abstract or mathematical models" is spelled **modelling** or **modeling**, with the Brits preferring the former and the Yanks preferring the latter. Many French, on the other hand, mistakenly employ the word **modelization**, and the corresponding verb form **modelize**. In fact, as far as the author is aware, neither of these forms appears, as yet, in any dictionary of the English language. However, both forms have a certain irresistible (and characteristically French) charm. Only time will tell if their repeated (yet, still, techincally incorrect) use by the French-speaking scientific community will cause these forms to evolve into bona fide English words.

To quantify the expected time correlation of the vector $\mathbf{w}$ with itself, the **autocorrelation** $R_{\mathbf{w}}(j;k)$ is defined such that

$$R_{\mathbf{w}}(j;k) \triangleq \mathscr{E}\{\mathbf{w}_{k+j}\mathbf{w}_k^H\}.$$

Note that $R_{(\mathbf{w}-\overline{\mathbf{w}})}(0;k) = P_{\mathbf{w}}(k)$; thus, the autocorrelation may be thought of as an appropriate generalization of the covariance for a discrete-time random process $\mathbf{w}_k$ which also describes its time correlation.

The PDF $p_{\mathbf{w}}(\mathbf{w}';k)$ and autocorrelation $R_{\mathbf{w}}(j;k)$ may vary with k, in which case the random process $\mathbf{w}_k$ is said to be **nonstationary**. In many cases of interest, however, the PDF and autocorrelation of a discrete-time random process $\mathbf{w}_k$ do *not* vary with k (even though the random variable $\mathbf{w}_k$ itself does); in such cases, the random process is said to be **stationary**, and its PDF and autocorrelation are denoted $p_{\mathbf{w}}(\mathbf{w}')$ and $R_{\mathbf{w}}(j)$. In general, if a random process $\mathbf{w}_k$ is nonstationary, its expected value must be estimated by **ensemble averaging**, as defined in §6.1. In the special case that the random process $\mathbf{w}_k$ is stationary, its expected value may be estimated instead by **time averaging** of a single member of the ensemble for a long time:

$$\mathscr{E}\{\mathbf{g}(\mathbf{w})\} = \lim_{N\to\infty}\Big[\frac{1}{N}\sum_{k=0}^{N}\mathbf{g}(\mathbf{w}_k)\Big];$$

this property is commonly referred to as **ergodicity**.

In many discrete-time systems, an autocorrelation of the following separable form is representative [cf. the more delicate treatment required in the continuous-time case in (6.6)]:

$$R_{(\mathbf{w}-\overline{\mathbf{w}})}(j,k) = P_{\mathbf{w}}(k)\delta_{j0}, \tag{6.7}$$

where $P_{\mathbf{w}}(k)$ is the covariance of the discrete-time random process $\mathbf{w}_k$ and δ_{j0} is the Kronecker delta (see §1.2.3). A discrete-time random process of this form is said to be **white**. The notation $\mathbf{w}_k = N(\overline{\mathbf{w}}, P_{\mathbf{w}}(k), \delta)$ will sometimes be used to denote a discrete-time random process $\mathbf{w}_k$ with both a Gaussian PDF and a white autocorrelation of the above simple form.

6.4.1 The joint description of two discrete-time random processes

When characterizing the relationship between two discrete-time random processes $\mathbf{v}_k$ and $\mathbf{w}_k$, the **joint PDF** $p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';j,k)$ may be used. For small $|\Delta\mathbf{v}'|$ and $|\Delta\mathbf{w}'|$, the quantity $p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';j,k)\Delta v'_1\cdots\Delta v'_m\Delta w'_1\cdots\Delta w'_n$ represents the probability that the $\mathbf{v}_{k+j}$ takes a value within a rectangular region centered at $\mathbf{v}'$ and of width $\Delta v'_i$ in each coördinate direction while $\mathbf{w}_k$ takes a value within a rectangular region centered at $\mathbf{w}'$ and of width $\Delta w'_i$ in each coördinate direction. It follows as before that, for any given j and k,

$$\int_{\mathbb{R}^n}\int_{\mathbb{R}^m} p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';j,k)\,d\mathbf{v}'\,d\mathbf{w}' = 1.$$

The expected value of a function $\mathbf{g}(\mathbf{v},\mathbf{w})$ of two discrete-time random processes $\mathbf{v}_k$ and $\mathbf{w}_k$ is given by

$$\mathscr{E}\{\mathbf{g}[\mathbf{v}_{k+j},\mathbf{w}_k]\} = \int_{\mathbb{R}^n}\int_{\mathbb{R}^m}\mathbf{g}[\mathbf{v}',\mathbf{w}']\,p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';j,k)\,d\mathbf{v}'\,d\mathbf{w}'.$$

In particular, the **cross correlation** of $\mathbf{v}_k$ and $\mathbf{w}_k$ is defined by

$$R_{\mathbf{vw}}(j;k) \triangleq \mathscr{E}\{\mathbf{v}_{k+j}\mathbf{w}_k^H\}.$$

Note that if $R_{\mathbf{vw}}(j;k) = 0$ for all j and k, the random processes $\mathbf{v}_k$ and $\mathbf{w}_k$ are said to be **uncorrelated**.

The joint PDF $p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';j,k)$ and cross correlation $R_{\mathbf{vw}}(j;k)$ may, in general, depend on k. In many cases of interest, however, the joint PDF and cross correlation of two discrete-time random processes $\mathbf{v}_k$ and $\mathbf{w}_k$ do not vary in time; in such cases, the two random processes $\mathbf{v}_k$ and $\mathbf{w}_k$ are said to be **jointly stationary**, and their joint PDF and cross correlation are denoted $p_{\mathbf{vw}}(\mathbf{v}',\mathbf{w}';j)$ and $R_{\mathbf{vw}}(j)$.

Exercises

Exercise 6.1 A statistics class with 1000 equally-gifted students has two exams, each with 10 questions, each question worth a maximum of 10 points (no extra credit), with varying amounts of partial credit awarded for partially correct answers. Student k's exam score is denoted x_k, for $k = 1,\ldots,1000$, and is considered here as a **pseudo-random variable**—that is, each exam score is in fact strongly correlated to several quantifiable factors (motivation, focus, study habits, class attendance, background, caffeine intake, etc.) that are simply not considered in the present mathematical model. In Exercise (6.1), we assume all components of x_k are uncoupled and distributed equally, and consider the distribution of a single component of $\mathbf{x}$, which for simplicity we denote as x.

(a) The mean on the first exam is 65 points and standard deviation is 5 points. Assuming an approximate Gaussian distribution of exam scores, plot the PDF $p_x(x')$ and corresponding CDF $f_x(x')$ for this exam. What are the median and mode of this distribution? How many students scored above 80 points?

(b) The mean on the second exam is 95 points and the standard deviation is 5 points. Is a Gaussian distribution of exam scores an adequate model in this case? Explain. Defining $y = 100 - x$, and assuming a Gamma distribution of y, plot the PDF $p_x(x')$ and corresponding CDF $f_x(x')$ for this exam. What are the appropriate values of the model parameters m and λ in this problem? What are the median and mode of this distribution? How many students scored above 80 points?

Exercise 6.2 Now assume that the statistics class mentioned in Exercise 6.1 is full of habitual cheaters (hopefully, yours is not). Indeed, the situation is so bad that, during the first exam, it turns out that each student copied, verbatim, the answer to exactly half of the questions from the student sitting on his left (for the purpose of this exercise, imagine all the students are sitting in a row). Describe accurately the feature of the corresponding covariance matrix $P_{\mathbf{x}}$ that might have alerted the attentive grader to this widespread cheating problem.

Exercise 6.3 Consider the discrete-time sequence $x_{k+1} = x_k + w_k$ where $x_0 = 0$ and $w_k = N(0,1,\delta)$. Derive and plot the distributions of w_5 and w_{1000}.

References

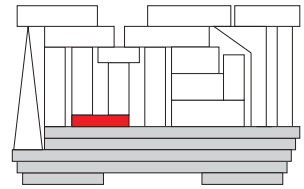
Bendat, JS, & Piersol, AG (1986) *Random Data: Analysis and Measurement Procedures*. Wiley.

Gill, J (2002) *Bayesian Methods: A Social and Behavioral Sciences Approach*. CRC Press.

Knuth, DE (1997) *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley.

Press, WH, Teukolsky, SA, Vetterling, WT, & Flannery, BP (2007). *Numerical Recipes, the Art of Scientific Computing*. Cambridge.

Chapter 7



Data manipulation: sorting, interpolation, & compression

Contents

7.1 Sorting algorithms . . . 194
7.1.1 Insertion sort . . . 195
7.1.2 Block insertion sort . . . 195
7.1.3 Cocktail sort . . . 196
7.1.4 Merge sort . . . 197
7.1.5 Quick sort . . . 197
7.1.6 Heap sort . . . 199
7.1.7 Sorting using a binary search tree . . . 200
7.1.8 Sorting networks . . . 202
7.2 Quantifying the distance between strings . . . 206
7.3 Interpolation over a single variable . . . 208
7.3.1 Linear spline interpolation . . . 208
7.3.2 Lagrange interpolation . . . 208
7.3.3 Cubic spline interpolation . . . 210
7.4 Multivariate interpolation of structured data . . . 214
7.5 Multivariate interpolation of unstructured data . . . 218
7.5.1 Interpolation via inverse distance . . . 218
7.5.2 Polyharmonic spline interpolation . . . 219
7.5.3 Kriging interpolation† . . . 222
7.6 Compression . . . 225
7.6.1 Dataset compression based on the SVD (a.k.a. POD, PCA,...)† . . . 225
7.6.2 Neural networks . . . 229
Exercises . . . 233

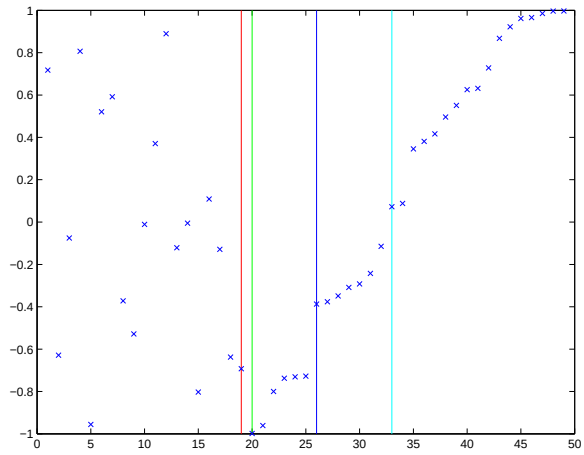


Figure 7.1: Execution of the insertion sort. The element being placed is marked by the red line, and is currently being compared to the element marked by the dark blue line; after each such comparison, a bisection algorithm is used to refine the window of possible locations where the element in question might belong, which is somewhere between the green and light blue lines.

Algorithm 7.1: Implementation of the insertion sort.

View
Test

```
function [D,index]=InsertionSort(D,v,n)
% Reorder a matrix D based on the elements in its first column using an insertion sort.
if nargout==2; D=[D, [1:n]']; end, for i=n-1:-1:1, a=i+1; b=n;
  % The following 2 lines search the ordered part of the list, [a,b], using a bisection
  % algorithm to find the appropriate point of insertion for record i.
  while a<b-1; c=a+floor((b-a)/2);
    if D(c,1)<D(i,1), a=c+1; else, b=c-1; end, end
  while a<=b;
    if D(i,1)<D(b,1), b=b-1; else, a=a+2; end, end
  D(i:b,:)=[D(i+1:b,:); D(i,:)];     % Insert record i at the designated point.
end
if nargout==2, index=round(D(:,end)); D=D(:,1:end-1); end
end % function InsertionSort
```

7.1 Sorting algorithms

A **list** refers to a collection of **records**, each containing some amount of **data**. If there are many records in the list, it is often necessary to **sort** (a.k.a. **order**) the list, based on an appropriately-chosen **marker** on each record, in order to access a given record of the list quickly. At various times, it is necessary

(a) to sort such a list from scratch,
(b) to add a few records to a previously sorted list, or
(c) to update a list that has gotten slightly out of order.

The present section discusses a few of the many available **sorting algorithms** to address these problems. Of the techniques presented, the merge sort, quick sort, and heap sort algorithms are best suited to problem (a), the insertion sort is best suited for problem (b), and the cocktail sort is best suited for problem (c).

In addition to the marker, each record might contain, for example, data about a particular book in a library, or at least a pointer to such data. Note that many lists may be maintained for a given collection of data to facilitate searches on different characteristic markers (title, author, call number, etc.).

The test codes provided with the sorting algorithms in §7.1 produce animations which are essential in understanding how these sorting algorithms work; *the reader is thus advised to run each of these test codes.*

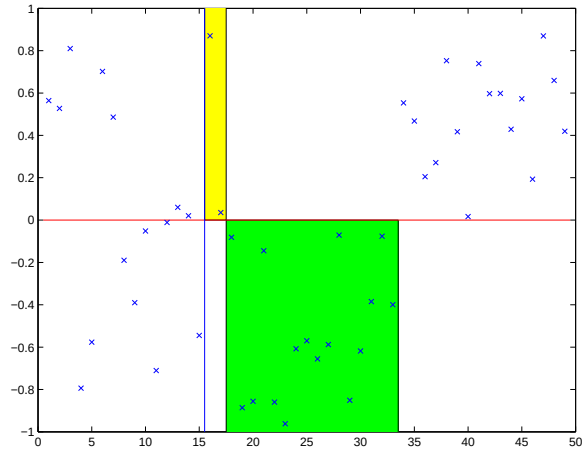


Figure 7.2: Execution of the block insertion sort. Blocks of positive elements (yellow) and negative elements (green) are identified and swapped until the negative elements come first, followed by the positive elements.

Algorithm 7.2: Implementation of the block insertion sort.

View
Test

```
function [D,index]=BlockInsertionSort(D,v,n)
% Reorder a matrix D based on the elements in its first column using a block insertion sort.
n=size(D,1); if nargout==2, D=[D, [1:n]']; end
k=n; while k>1 & D(k,1)>=0, k=k-1; end, j=k-1; i=j;
if k>1, while i>0                          % Determine a<0 block [j+1:k] and a>=0 block [i+1:j]
  while j>0
    if D(j,1)<0, j=j-1; else, break, end, end
  if j==1, i=j; else, i=j-1; end
  while i>0
    if D(i,1)>=0, i=i-1; else, break, end, end
  if i<0, break, end, D(i+1:k,:)=[D(j+1:k,:); D(i+1:j,:)];
k=i+k-j; j=i-1; end, end
if nargout==2, index=round(D(:,end)); D=D(:,1:end-1); end
end % function BlockInsertionSort
```

7.1.1 Insertion sort

The insertion sort is straightforward: working from one end of the list to the other, it simply picks up one of the records from the unsorted portion of the list at a time, scans through the sorted portion of the list to determine where this element belongs, then opens the sorted portion of the list to insert this element into the correct position. The scan through the sorted portion of the list is most efficiently accomplished with a discretized version of the bisection algorithm introduced in §3.1.2. An efficient implementation of the insertion sort is given in Algorithm 7.1, with a corresponding visualization in Figure 7.1. The insertion sort is particularly well suited to the problem of adding a few records to a previously sorted list; it is far outperformed by the merge sort, quick sort, and heap sort, introduced below, for most other applications of sorting algorithms.

7.1.2 Block insertion sort

In some applications of sorting algorithms, one only needs to separate the list into a relatively small number of blocks, such as a "stable" block and an "unstable" block (see, e.g., §4.5.5). For such problems, the insertion sort introduced above can be accelerated significantly, as the records within each individual block do not need to be sorted. Further, records can often be moved a group at a time into the appropriate blocks. An implementation is given in Algorithm 7.2, with a corresponding visualization in Figure 7.2.

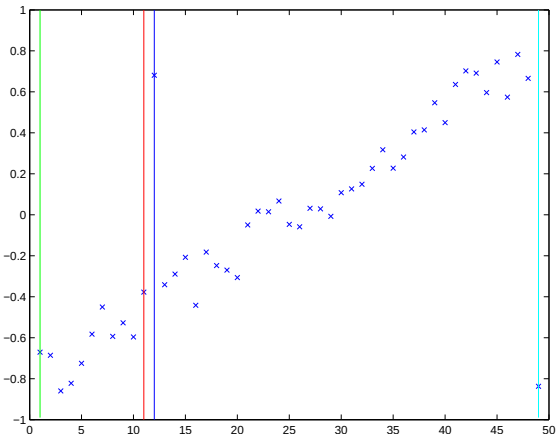


Figure 7.3: Execution of the cocktail sort.

Algorithm 7.3: Implementation of the cocktail sort.

View
Test

```
function [D,index]=CocktailSort(D,v,n)
% Reorder a matrix D based on the elements in its first column using a cocktail sort.
n=size(D,1); if nargout==2, D=[D, [1:n]']; end, l=1; r=n;
while r>l
  b=l; for i=l:r-1, if D(i,1)>D(i+1,1), D(i:i+1,:)=[D(i+1,:); D(i,:)]; b=i; end
  end, r=b; if r<=l, break; end
  a=r; for i=r-1:-1:l, if D(i,1)>D(i+1,1), D(i:i+1,:)=[D(i+1,:); D(i,:)]; a=i; end
  end, l=a+1;
end
if nargout==2, index=round(D(:,end)); D=D(:,1:end-1); end
end % function CocktailSort
```

7.1.3 Cocktail sort

A **cocktail sort** (a.k.a. a **bidirectional bubble sort** or **shaker sort**) simply sweeps from a left boundary to the right and back, comparing neighboring values and swapping to put the smaller value on the left after each comparison, then updating the boundaries appropriately after each sweep if the sweep indicates a portion of the list is completely sorted, as implemented in Algorithm 7.3 and visualized in Figure 7.3.

The cocktail sort, which alternates between left-to-right sweeps and right-to-left sweeps, achieves a significant acceleration of the simpler **bubble sort**, which only uses left-to-right sweeps but is otherwise identical. The bubble sort has a peculiar weakness worth noting (in order to avoid). If there is a record on the left with a relatively large marker that needs to be moved a long way to the right, a bubble sort can accomplish this with a single sweep (we thus refer to such an outlier as a **rabbit**). However, if there is a record on the right with a relatively small marker that needs to be moved a long way to the left, a bubble sort can only move this record a single position during each left-to-right sweep, and thus such a record takes an enormous time to move a significant number of records to the left (we thus refer to such an outlier as a **turtle**). The possible existance of turtles render the bubble sort unsuitable for application. However, by incorporating both left-to-right sweeps and right-to-left sweeps, the cocktail sort becomes competitive with other more sophisticated algorithms for those problems in which the list is known to be only slightly out of order, as indicated in Figure 7.3.

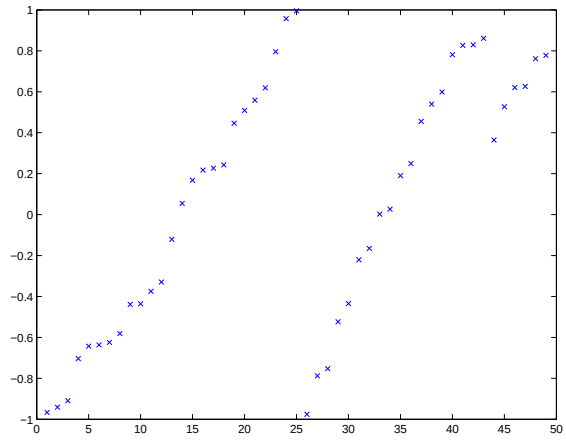


Figure 7.4: Execution of the merge sort. The entire list is divided in half, then in half again, etc., until the resulting blocks have a single record. Then these blocks are merged two at a time, then the resulting blocks merged two at a time, etc., until the entire list is sorted.

Algorithm 7.4: Implementation of the merge sort using a **divide and conquer** paradigm.

View
Test

```
function [D,index]=MergeSort(D,v,a,b)
% Reorder a matrix D based on the elements in its first column using a merge sort.
if nargout==2, D=[D, [1:size(D,1)]']; end, if nargin==3, b=a; a=1; end
if b-a > 0
  b1 = a + floor((b-a)/2); a1=b1+1; D=MergeSort(D,v,a,b1); D=MergeSort(D,v,a1,b);
  while b1-a >= 0 & b-a1 >= 0
    if D(a1,1) < D(a,1); D(a:a1,:)=[D(a1,:); D(a:a1-1,:)];
      a1=a1+1; b1=b1+1; end; a=a+1;
end; end
if nargout==2, index=round(D(:,end)); D=D(:,1:end-1); end
end % function MergeSort
```

7.1.4 Merge sort

The merge sort is based on the idea that it is much faster to interweave appropriately two sorted groups of records of length $m/2$, to make a single sorted group of records of length m, than it is to sort m records from scratch. The merge sort applies this concept repeatedly, starting by sorting individual records (of length 1) 2 at a time, then merging the resulting small sorted groups (of length 2) 2 at a time, then merging the resulting larger groups (of length 4) 2 at a time, etc., until the entire list is sorted. An implementation is given in Algorithm 7.4 and a visualization in Figure 7.4.

7.1.5 Quick sort

The quick sort is based on repeated application of a block insertion sort to successively smaller groups of records. First, a block insertion sort is performed to arrange the entire list into three blocks:

- The middle block contains a single record from the original list. Its marker is referred to as the **pivot**.
- The block on the left contains all records with markers less than the pivot.
- The block on the right contains all records with markers greater than or equal to the pivot.

After this first sort, the record containing the pivot is in its final location. Next, block insertion sorts are used to sort both the left and right blocks individually in an identical fashion, repeating until the entire list is sorted.

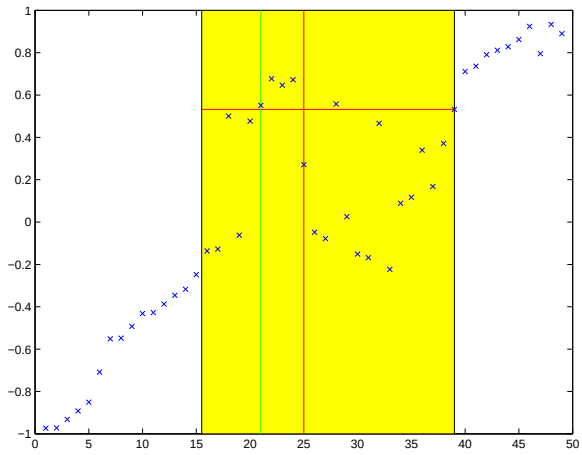


Figure 7.5: Execution of the quick sort. The markers in each (yellow) block are first compared against the pivot, which is determined via a median-of-three approach and moved temporarily to the right edge of the block. The corresponding records are then sorted with a block insertion sort into two sub-blocks. The pivot is then moved to its final location near the middle of the yellow block, and the sub-block to the left and right of the pivot are then sorted in an identical fashion.

Algorithm 7.5: Implementation of the quick sort, using the **median-of-three** approach to determine the pivot.

View
Test

```
function [D,index]=QuickSort(D,v,i,k)
% Reorder a matrix D based on the elements in its first column using a quick sort.
if nargout==2, D=[D, [1:size(D,1)]']; end, if nargin==3, k=i; i=1; end
if k>i  % Begin by identifying a pivot.  Take the median of i=left, j=middle, and k=right.
  j=i+floor((k-i)/2);  a=D(i,1);  b=D(j,1);  c=D(k,1);
  if a>b, if b>c, pivot=j; elseif a>=c, pivot=k; else, pivot=i; end;
  else    if b<c, pivot=j; elseif c>=a, pivot=k; else, pivot=i; end; end
  if      pivot==j; D([j k],:)=D([k j],:);                    % Store pivot at right, for now.
  elseif pivot==i; D([i k],:)=D([k i],:); end;
  value=D(k,1); pivot=i; % Now scan from i to k-1 to determine new pivot value, separating
  for l=i:k-1,           % the entries below and above the specified pivot value.
    if D(l,1) <= value; D([pivot l],:)=D([l pivot],:);  pivot=pivot+1; end
  end
  D([k pivot],:)=D([pivot k],:);                % Move pivot (stored at k) to where it belongs.
  D=QuickSort(D,v,i,pivot-1); D=QuickSort(D,v,pivot+1,k); % Sort to left & right of pivot.
end
if nargout==2, index=round(D(:,end)); D=D(:,1:end-1); end
end % function Quicksort
```

Algorithm 7.6: A convenient wrapper routine to sort complex records based on the Quicksort.

View
Test

```
function [D,index]=SortComplex(D,Trait,Alg)
% Sort a matrix D based on the 'absolute value' or 'real part' (specified by Trait) of the
% complex elements in its first column, using 'QuickSort', etc (specified by Alg).

if nargin<3, Alg=str2func('QuickSort'); if nargin<2, Trait='real part'; end, end
switch Trait, case 'absolute value', D=[abs(D) D]; case 'real part', D=[real(D) D]; end
n=length(D); if nargout>1, [D,index]=Alg(D,0,n); else, D=Alg(D,0,n); end, D=D(:,2:end);
end % function SortComplex
```

The principle question that arises is which marker (and corresponding record) should be selected for the pivot. It might seem initially that the median marker value over the entire list should be used, thus cutting the

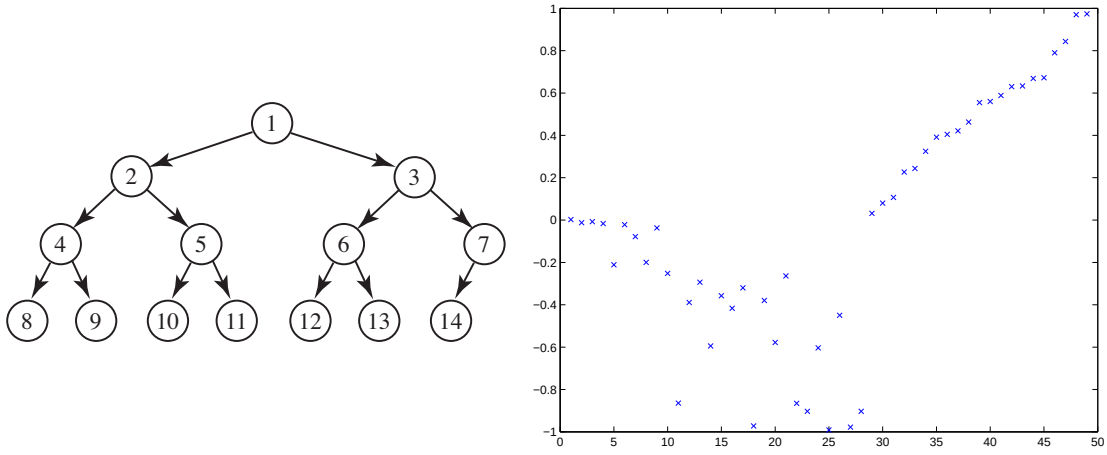


Figure 7.6: Graphical illustration of the Heap sort. (a) A **complete binary tree** structure with two children per parent and 14 records. This complete binary tree is referred to as a **binary heap** if the marker associated with each **parent** is greater than or equal to that of each of its **children**. (b) Execution of the heap sort: First, assign a tree structure to the entire list and **heapify** this tree. Then, pull off the root element to the right, **sift** to return the part that remains to heap order, and repeat until finished.

list in half after each block insertion sort. However, the median marker is prohibitively expensive to determine. A common choice is thus simply to select the pivot from the list at random; this choice turns out to not be nearly as bad as one might first think, as, half of the time, this choice is in the center half of the sorted list. A slightly better choice, referred to as the **median of three**, is to select the pivot as the median of three markers (typically, those from the records on the left, middle, and right of the list). This is the choice that, for all but the most pathological of cases, works best, and is implemented in Algorithm 7.5 and visualized in Figure 7.5.

7.1.6 Heap sort

A **binary tree** (see Figure 7.6a) is a data structure in which a single record, called the **root**, is the single **parent** of at most two **child** records. Each of these child records, in turn, may be the single parent of at most two additional child records, etc., so all records of the tree trace back to the root as their common ancestor. A **binary heap** is a binary tree that is both **complete**, meaning that all levels of the tree are fully filled except possibly the last (and, if the last level is not fully filled, it is filled in from left to right), and **ordered**, meaning that the marker associated with each parent record is greater than or equal to that of each of its child records.

Since it is a complete binary tree, it is easy to assign a binary heap structure to a list of records without resorting to the use of pointers: as indicated in Figure 7.6a, simply call record 1 the root, call records 2 and 3 its children, call records 4-7 its grandchildren, etc.

The process of swapping records in a binary tree to convert it into a binary heap is referred to as **heapifying**. Consider first the problem of heapifying a binary tree which is already almost a binary heap except for a single record which is **out of heap order**, meaning that it is characterized by a marker which is less than that of at least one of its child records. [Let's name this out-of-heap-order record A; for sake of illustration, imagine first that record A is the root.] This binary tree can be converted into a binary heap by a process known as **sifting**, which proceeds as follows:

- First, swap the out-of-heap-order record A with its child record characterized by the largest marker. After this swap, record A (in its new position) is the only record that might still be out of heap order.
- If, after this swap, record A is still out of heap order, then repeat from the first step, else exit.

Algorithm 7.7: Implementation of the heap sort.

View Test

```
function [D,index]=HeapSort(D,v,n)
% Reorder a matrix D based on the elements in its first column using a heap sort.
if nargout==2, D=[D, [1:n]']; end, for a=floor(n/2):-1:1, D=Sift(D,a,n,v); end   % Heapify
for b=n:-1:2, D([1 b],:)=D([b 1],:);          % Peel off max record & put grandchild at root
  D=Sift(D,1,b-1,v);                          % Sift (re-heapify) & repeat
end, if nargout==2, index=round(D(:,end)); D=D(:,1:end-1); end
end % function HeapSort
%%%
```

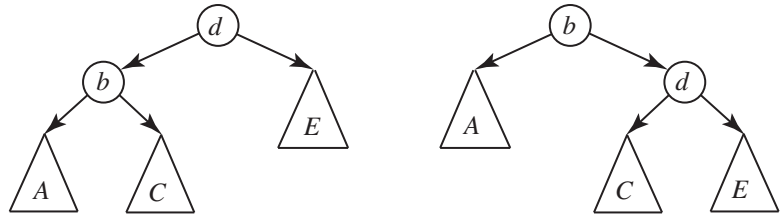


Figure 7.7: Two equivalent BSTs, where $\{b,d\}$ are individual records and $\{A,C,E\}$ are subtrees with a number of generations of records, with $A \leq b \leq C \leq d \leq E$ (where, e.g., $A \leq b$ is taken to mean that the markers in all records of subtree A are less than the marker of record b). Replacing the BST at left with the BST at right is called a **right rotation**, and improves the balance in the tree if the number of generations in subtree A is greater than the number of generations in subtree E; replacing the BST at right with the BST at left is called a **left rotation**, and improves the balance if the number of generations in subtree E is greater than that in subtree A. The connections within $\{A,C,E\}$ are unaffected by the rotation.

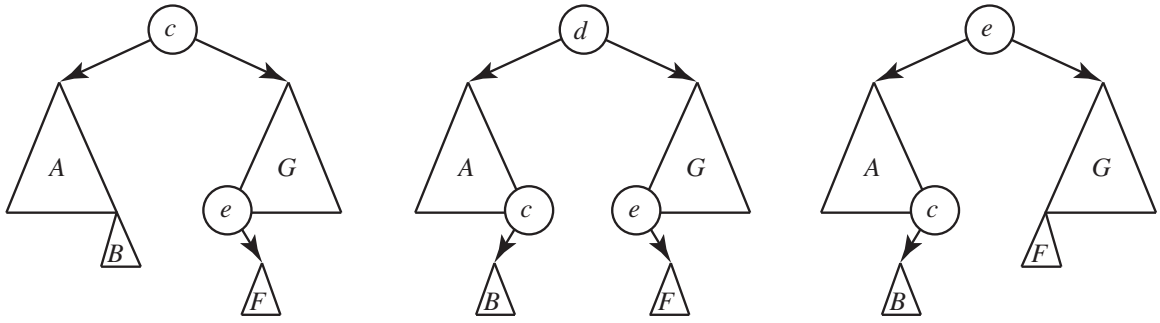


Figure 7.8: Two possible actions after deleting record d from the BST indicated in the center schematic (with $A \leq B \leq c \leq d \leq e \leq F \leq G$): **promote the in-order predecessor record** c (resulting in the BST at left), or **promote the in-order successor record** e (resulting in the BST at right).

- If the marker associated with record n is *smaller* than that of m, then:
  - (a) If the smaller child of record m is 0, set it to record n and exit; otherwise,
  - (b) compare the marker of record n to that of the smaller child of record m in an identical fashion.
- On the other hand, if the marker associated with record n is *larger or equal* to that of m, then:
  - (c) If the larger child of record m is 0, set it to record n and exit; otherwise,
  - (d) compare the marker of record n to that of the larger child of record m in an identical fashion.

To **sort an entire list** of unsorted records, take the first record and assign it as the root of the BST (with no children), then take each of the remaining records in turn and insert it in the BST via the above procedure.

Constructing a BST in such a fashion can lead to an **unbalanced** tree in which some branches are much longer than others. Indeed, if the initial list of data happens to already be in ascending or descending order, sorting a list via the approach described above requires $\sim n^2/2$ comparisons, and the resulting tree has a single branch of length n, which rather defeats the entire purpose of setting up a tree structure. Such unbalanced trees can be partially **balanced** from time to time by applying an appropriate sequence of **tree rotations**: that is,

- replacing the BST in Figure 7.7a with the BST in Figure 7.7b (referred to here as a **right rotation**), or
- replacing the BST in Figure 7.7b with the BST in Figure 7.7a (referred to here as a **left rotation**).

Note that the starting BST in either case may in fact be a subtree of a larger BST. To apply a right rotation:

- assign the root of subtree C as the smaller child of record d,

- assign record d as the larger child of record b, and
- assign record b as the root of this BST;

a left rotation is analogous. Of course, when making each of these assignments, one must ensure that the pointer keeping track of the corresponding parent of the moved record is also updated appropriately.

If a particular record in a tree has a larger child, then its **in-order successor record** is the left-most record of the subtree associated with this larger child; to find it, starting from the larger child of the original record, simply trace down to its smaller child, then to its smaller child, etc., until you reach a record without a smaller child. If a particular record in a tree does not have a larger child, then its in-order successor record is the closest-related ancestor record for which the original record is part of the subtree associated with the ancestor's smaller child; to find it, trace up through the ancestors of the original record rather than down through its descendants. The **in-order predecessor record** is found analogously. For example, in Figure 7.8b, record e is the in-order successor of record d, which is the in-order successor of record c.

To **enumerate** the (sorted) records of a BST, simply start from its smallest record (that is, trace down from the root to its smaller child, then to its smaller child, etc., until you reach a record without a smaller child), then step through the in-order successor records until you reach the end of the list.

To **delete** a record (identified here as record d) from a BST,

- If record d has no children, simply remove it from the tree, setting the corresponding child record number in the affected parent to zero.
- If record d has a single child, remove it from the tree, assigning the parent of record d as the adoptive parent of record d's child.
- If record d has two children, as illustrated in Figure 7.8, there are two options:
  (a) replace record d with its in-order successor, record e, and replace record e with the subtree, F, associated with record e's larger child (if any), or
  (b) replace record d with its in-order predecessor, record c, and replace record c with the subtree, B, associated with record c's smaller child (if any).

  To choose between these two options, simply select the one that results in the more balanced tree.

The strength of the BST approach is not in the sorting of a list from scratch (a problem for which various algorithms described earlier in §7.1 are significantly faster), but in the frequent accessing, inserting, and/or deleting of individual records in the tree, all of which can be accomplished extremely quickly if the tree is reasonably well balanced, as in such a case the root is about halfway through the sorted list of markers, its two children are about 1/4 and 3/4 of the way through the list, etc.

Exemplary applications that leverage the particular strengths of the BST approach include online dictionaries and algorithms for indexing and searching the web. Note that, when searching an extremely large BST for a particular record, those records which are higher up in the BST are found significantly faster than those records which are deeper down in the BST. Thus, instead of applying rotations solely to approximately balance a BST, as discussed previously, we may preferentially use such rotations to percolate up those records which statistically get accessed most frequently in such searches to higher positions within the BST, thus allowing these most frequently accessed records to be found especially quickly.

Implementations of a few key BST operations (initialize, insert, rotate, enumerate, successor) are given in Algorithm 7.8; others are provided in the *NRC*.

7.1.8 Sorting networks

In the **sorting algorithms** discussed above, subsequent comparisons were performed based on the results of previous comparisons. In applications in which sorting is one of the primary activities of interest, it is beneficial to implement sorting algorithms in a massively parallel framework; in some applications, special purpose hardware may even be developed to sort records. In such applications, specialized algorithms known

Algorithm 7.8: Implementation of various operations on a binary search tree (BST).

View

```
function [D,r]=BSTinitialize(D)
% Initialize a BST based on a list of records D with markers in the first column.
[n,m]=size(D); D=[D zeros(n,5)]; r=1;
for i=2:n, [D,r]=BSTinsert(D,i,r); if mod(i,1)==0, BSTplot(D,r), pause(.01), end, end
end % function BSTinitialize
```

View

```
function [D,r]=BSTinsert(D,n,r)
% Insert record n into a BST in D with root r, balancing the affected ancestors as needed.
flag=1; m=r; while flag, if D(n,1)<D(m,1) % Find appropriate open child slot & place record
  if D(m,end-3)==0, D(m,end-3)=n; D(n,end-2)=-m; flag=0; else, m=D(m,end-3); end
else
  if D(m,end-1)==0, D(m,end-1)=n; D(n,end-2)=+m; flag=0; else, m=D(m,end-1); end
end, end
flag=1; while m>0 & flag                    % Scan through ancestors of inserted record,
  gold=D(m,end); [D,m]=BSTrotateLR(D,m);    % rotate if helpful to keep balanced,
  a=D(m,end-3); if a>0, Dag=D(a,end); else, Dag=-1; end  % and update generation count.
  c=D(m,end-1); if c>0, Dcg=D(c,end); else, Dcg=-1; end, g=max(Dag+1,Dcg+1);
  D(m,end)=g; n=m; m=abs(D(n,end-2));        % Exit loop if generation count at this level
  if g==gold, flag=0; end                   % is unchanged by the insertion.
end
if m==0; r=n; end      % If scanned all the way back to root, the root might have changed.
end % function BSTinsert
```

View

```
function [D,q]=BSTrotateLR(D,q)
% Apply a left or right rotation at record q if such a rotation helps to balance the BST.
p=D(q,end-3); r=D(q,end-1);
if p>0, Dpg=D(p,end); pp=D(p,end-3); if pp>0, Dppg=D(pp,end); else, Dppg=-1; end
else, Dpg=-1; Dppg=-2; end
if r>0, Drg=D(r,end); rr=D(r,end-1); if rr>0, Drrg=D(rr,end); else, Drrg=-1; end
else, Drg=-1; Drrg=-2; end
if Dppg>Drg, [D,q]=BSTrotateR(D,q); elseif Drrg>Dpg, [D,q]=BSTrotateL(D,q); end
end % function BSTrotateLR
```

View

```
function [D,d]=BSTrotateL(D,b)
% Apply a left rotation to a BST at record b (see Figure 7.7). (BSTrotateR is similar.)
a=D(b,end-3); d=D(b,end-1); c=D(d,end-3); e=D(d,end-1); p=D(b,end-2);
if a>0, Dag=D(a,end);                  else, Dag=-1; end
if c>0, Dcg=D(c,end); D(c,end-2)=+b;   else, Dcg=-1; end
if e>0, Deg=D(e,end);                  else, Deg=-1; end
D(b,end-1)=c; D(b,end-2)=-d; D(b,end)=max(Dag+1,Dcg+1);
D(d,end-3)=b; D(d,end-2)=p;  D(d,end)=max(D(b,end)+1,Deg+1);
s=sign(p); p=abs(p); if p>0, D(p,end-2+s)=d; end
while p>0, a=D(p,end-3); if a>0, Dag=D(a,end); else, Dag=-1; end
           c=D(p,end-1); if c>0, Dcg=D(c,end); else, Dcg=-1; end, g=max(Dag+1,Dcg+1);
           if g<D(p,end), D(p,end)=g; p=abs(D(p,end-2)); else, p=0; end, end
end % function BSTrotateL
```

View

```
function index=BSTenumerate(D,r)
% Enumerate the records of a BST from smallest to largest.
n=0; m=r; while D(m,end-3)>0, m=D(m,end-3); end
while m>0, n=n+1; index(n)=m; m=BSTsuccessor(D,m); end
end % function BSTenumerate
```

View

```
function [s]=BSTsuccessor(D,r)
% Find the in-order successor record, s, of a given record r of a BST.
s=D(r,end-1); if s>0, while D(s,end-3)>0, s=D(s,end-3); end
else, s=D(r,end-2); while s>0, s=D(s,end-2); end, s=abs(s);
end % function BSTsuccessor
```

View

Algorithm 7.9: Implementations of the bitonic sort and the odd/even merge sort.

```
function [D,index]=BitonicSort(D,v,n)
% Reorder a matrix D based on the n=2^s elements in its first column using a bitonic sort.
s=log2(n); if nargout==2, D=[D, [1:n]']; end
for stage=1:s, N=2^stage; Nsets=n/N;
  for level=stage:-1:1, Nl=2^level; Ngroups=N/Nl;
    for set=0:Nsets-1, t=(-1)^set; for group=0:Ngroups-1, for i=1:Nl/2
      % Determine each pair of elements to be compared, and compare them.
      a=i+N*set+Nl*group; b=a+Nl/2; if t*D(a,1)>t*D(b,1), D([a b],:)=D([b a],:); end
    end, end, end
  end
end
if nargout==2, index=round(D(:,end)); D=D(:,1:end-1); end
end % function BitonicSort
```

View

```
function [D,index]=OddEvenMergeSort(D,v,a,n)
% Reorder D based on the n=2^s elements in its first column using an odd/even merge sort.
if nargin==3, n=a; a=1; end, if nargout==2, D=[D, [1:n]']; end
if n>1, m=round(n/2);
  D=OddEvenMergeSort(D,v,a,m);
  D=OddEvenMergeSort(D,v,a+m,m);
  D=OddEvenMerge(D,a,n,1);
end
if nargout==2, index=round(D(:,end)); D=D(:,1:end-1); end
end % function OddEvenMergeSort
%%%
```

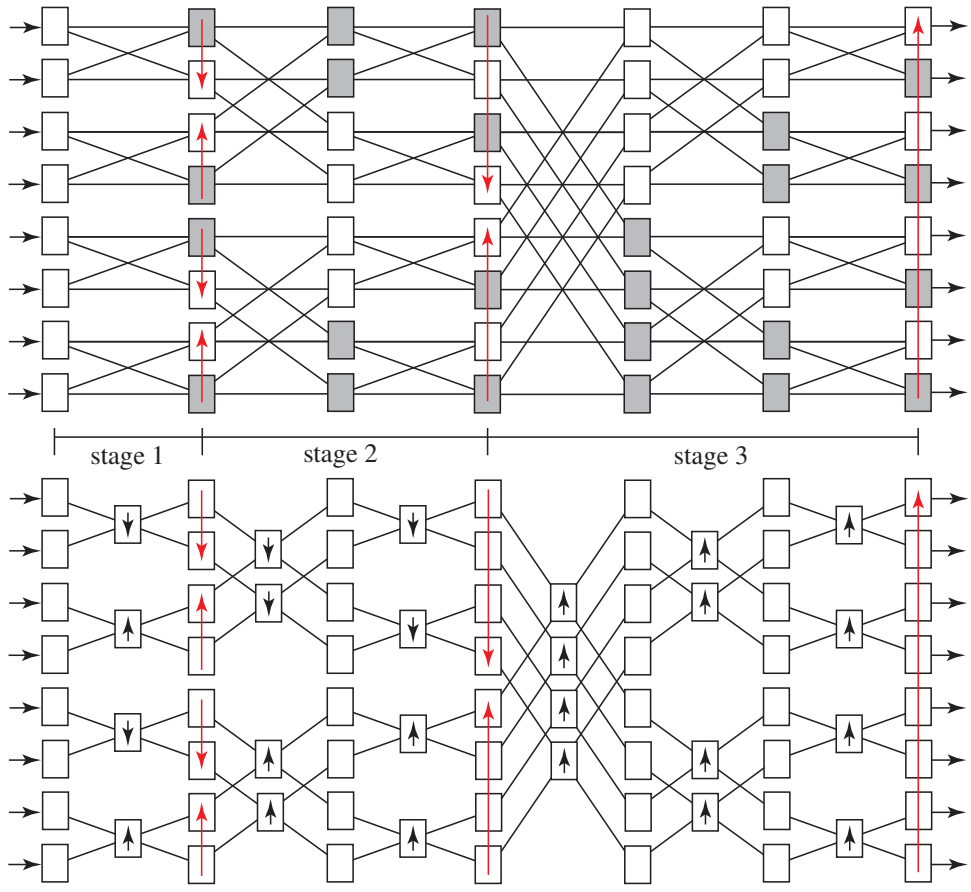


Figure 7.9: Two equivalent implementations of the bitonic sort applied to 8 records. (top) Each filled (open) square outputs the smaller (larger) value of its two inputs onto both outputs. (bottom) Comparisons are now done only in the blocks with the short arrows, with the large and small values on the two inputs sorted onto the two outputs as indicated by the direction of the arrows. In both representations, the long arrows indicate the subset of the data that is sorted immediately after that step.

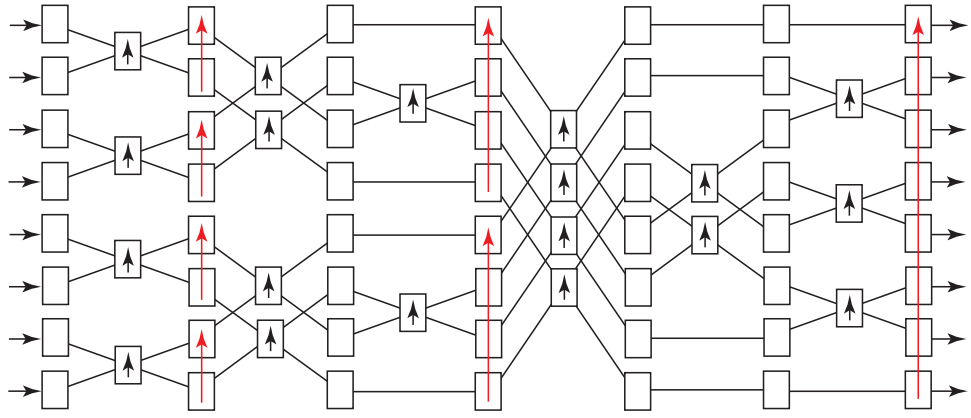


Figure 7.10: The odd-even merge sort applied to 8 records (cf. Figure 7.9b).

in Figure 7.10 that the odd/even merge sorting network is a slight departure from the assembly of butterfly graphs used in the bitonic sorting network (Figure 7.9). During the first stage of sorting, the input is sorted one pair at a time, each into increasing order. During the second stage of sorting, each pair of monotonoic groups of 2 records are merged into monotonic groups of 2^2 records. During the third stage of sorting, each pair of monotonoic groups of 2^2 records are merged into monotonic groups of 2^3 records, etc., until the entire set of records is sorted, as illustrated in Figure 7.10. Implementation is given in Algorithm 7.9.

7.2 Quantifying the distance between strings

Algorithm 7.10: Computation of the **Optimum String Alignment** distance between two strings.

View
Test

```
function dist=DistanceOSA(s,t,verbose)
% Compute the Optimal String Alignment distance between two strings, with equal cost for
% Deletion (D), Insertion (I), Substitution (S), and Exchange of adjacent symbols (E).
sl=length(s); tl=length(t); if sl==0 | tl==0, dist=sl+tl; return, end, d=zeros(sl+1,tl+1);
for i=0:sl, d(i+1,1)=i; end, for j=0:tl, d(1,j+1)=j; end   % Initialize pure D and pure I.
for j=2:tl+1, for i=2:sl+1
  d(i,j)=min([d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+(s(i-1)~=t(j-1))]); % Cost of D, I, or S.
  if i>2 && j>2 && s(i-2)==t(j-1) && s(i-1)==t(j-2)
    d(i,j) = min([ d(i,j), d(i-2,j-2)+1 ]);                          % Cost of E.
  end
end, end, dist=d(sl+1,tl+1); if nargin>2, d, end
end % function DistanceOSA
```

Algorithm 7.11: Computation of the weighted **Damerau-Levenshtein** distance between two strings.

View
Test

```
function dist=DistanceDL(a,b,W,verbose)
% Compute the weighted Damerau-Levenshtein distance between two strings; {W.D,W.I,W.S,W.E}
% are the weights on Deletion, Insertion, Substitution, and Exchange (of adjacent symbols
% only), all taken as 1 by default. Implements Algorithm S of Lowrance & Wagner (1975).
Al=length(a); Bl=length(b); if Al==0 | Bl==0, dist=Al+Bl; return, end
if nargin<3, W.D=1; W.I=1; W.S=1; W.C=1; end, INF=Al+Bl+1; H=INF*ones(Al+2,Bl+2);
for i=0:Al, H(i+2,2)=i*W.D; end % Initialize pure deletions and insertions.
for j=0:Bl, H(2,j+2)=j*W.I; end % (Note that H indices are incremented by 2 wrt LW75.)
Alphabet=Unique([a b]);          % Alphabet contains all symbols used in this problem.
for i=1:Al, A(i)=StrFind(Alphabet,a(i)); end  % Convert characters in a and b to integers.
for i=1:Bl, B(i)=StrFind(Alphabet,b(i)); end, DA(1:length(Alphabet))=0;
for i=1:Al, DB=0; for j=1:Bl, i1=DA(B(j)); j1=DB;
  % When the code gets here, DA(c) holds either 0 or the largest index I, with 1<I<i, such
  % that a(I)=c for each member c of Alphabet; in particular, i1 holds the largest index
  % I, with 1<I<i, such that a(I)=b(j). Similarly, j1 holds either 0 or the largest
  % index J, with 1<J<j, such that b(J)=a(i). Knowledge of i1 & j1 enables "Operation *":
  %  1) start from the first i1 symbols of a, modified to match the first j1 symbols of b;
  %  2) Delete those elements corresponding to the elements between i1 and i in a;
  %  3) Exchange the element corresponding to i1 in a with the (new) element to its right;
  %  4) Insert the appropriate symbols to match the elements between j1 and j in b.
  if A(i)==B(j), d=0; DB=j; else, d=W.S; end
  H(i+2,j+2)=min([ H(i+1,j+2)+W.D, H(i+2,j+1)+W.I, H(i+1,j+1)+d, ...  % Simple D, I, or S.
      H(i1+1,j1+1)+(i-i1-1)*W.D+W.E+(j-j1-1)*W.I]);  % <- Operation *, as described above.
end, DA(A(i))=i; end, dist=H(Al+2,Bl+2);  if nargin>3, H, end
end % function DistanceDL
%%%
```

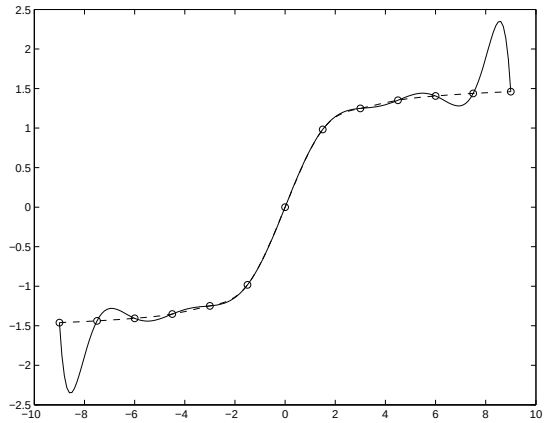


Figure 7.11: The interpolation problem [cf. the data fitting problem in Figure 4.5]: adjust a smooth curve with the specified smoothness to touch (exactly) $n+1$ points ($\circ$); the two solutions to this problem illustrated are (———) the Lagrange interpolant and (– – – –) the cubic spline interpolant (with parabolic run-out). Note that the Lagrange interpolant often gives a spurious result when the number of datapoints is large.

7.3 Interpolation over a single variable

As distinct from the data fitting framework (§4.7.2), the **interpolation** framework aspires to draw an "appropriately smooth" curve which passes exactly through a set of available datapoints, as illustrated in Figure 7.11. This problem description is subject to a significant degree of interpretation; only a few such interpretations will be discussed here. The interpolation framework is extended to the multivariate case in §§7.4-7.5.

An interpolating curve is useful when developing differentiation and integration strategies, as discussed in §§8-9, as well as when simply estimating the value of a function between known values, the need for which, in the multivariable setting, often arises in the development of **computer-generated imagery** (**CGI**).

Note specifically that the process of interpolation passes a curve *exactly* through each datapoint. This is sometimes what is desired. However, if the data is from an experiment and has any appreciable uncertainty associated with it, then it is preferred to take many measurements and use a least-squares technique to fit a low-order curve in the general vicinity of several datapoints, as discussed in the data fitting framework described in §4.7.2. This technique minimizes a weighted sum of the square distance from each datapoint to this curve without forcing the curve to pass through each datapoint individually, and generally produces a much smoother curve (and a more physically-meaningful result) when the available data is noisy.

7.3.1 Linear spline interpolation

Linear spline interpolation amounts to nothing more than the child's game of **Connect the Dots** (using straight line segments or **splines** between each pair of points). Implementation (Algorithm 7.12) is straightforward, and provides a reference solution against which improved interpolation schemes may be compared.

7.3.2 Lagrange interpolation

Suppose we have a set of $n+1$ datapoints $\{x_i, y_i\}$. The process of Lagrange interpolation fits an n'th degree polynomial (that is, a polynomial with $n+1$ degrees of freedom) exactly through this data. There are two ways of accomplishing this: solve a system of $n+1$ simultaneous equations for the $n+1$ coefficients of this polynomial, or construct the polynomial directly in factored form.

Algorithm 7.12: Linear spline interpolation.

View
Test

```
function [f]=LinearSpline(x,xd,fd)
% Perform linear interpolation based on the {xd,fd} and evaluate at the points in x.
n=length(xd); m=length(x); i=1; for j=1:m
  for i=i:n-1, if xd(i+1) > x(j), break, end, end  % Find the i such that xd(i)<=x<=xd(i+1)
  f(j)=(fd(i+1)*(x(j)-xd(i)) + fd(i)*(xd(i+1)-x(j)))/(xd(i+1)-xd(i));
end
end % function LinearSpline
```

Solving $n+1$ simultaneous equations for the $n+1$ coefficients

Consider the polynomial

$$P(x) = a_o + a_1 x + a_2 x^2 + \ldots + a_n x^n.$$

At each point x_i, the polynomial has the value y_i; that is,

$$y_i = P(x_i) = a_o + a_1 x_i + a_2 x_i^2 + \ldots + a_n x_i^n \qquad \text{for} \qquad i = 0,1,2,\ldots,n.$$

In matrix form, we may write this system as

$$\underbrace{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}}_{V} \underbrace{\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix}}_{\mathbf{a}} = \underbrace{\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}}_{\mathbf{y}}. \tag{7.1}$$

This system is of the form $V\mathbf{a} = \mathbf{y}$, where V is commonly referred to as **Vandermonde's matrix**, and may be solved for the vector $\mathbf{a}$ containing the coefficients a_i of the desired polynomial. Unfortunately, Vandermonde's matrix is usually quite poorly conditioned, and thus this technique of finding an interpolating polynomial is unreliable at best.

Constructing the polynomial directly

Consider the n'th degree polynomial given by the factored expression

$$L_\kappa(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{\kappa-1})(x-x_{\kappa+1})\cdots(x-x_n)}{(x_\kappa-x_0)(x_\kappa-x_1)\cdots(x_\kappa-x_{\kappa-1})(x_\kappa-x_{\kappa+1})\cdots(x_\kappa-x_n)} = \prod_{\substack{i=0 \\ i\neq\kappa}}^{n} \frac{x-x_i}{x_\kappa-x_i}. \tag{7.2a}$$

Note that, by construction,

$$L_\kappa(x_i) = \delta_{i\kappa} = \begin{cases} 1 & i = \kappa, \\ 0 & i \neq \kappa. \end{cases}$$

Scaling this result, the polynomial $y_\kappa L_\kappa(x)$ (no summation implied) passes through zero at every datapoint $x = x_i$ except at $x = x_\kappa$, where it has the value y_κ. Finally, a linear combination of $n+1$ of these polynomials

$$P(x) = \sum_{\kappa=0}^{n} y_\kappa L_\kappa(x) \tag{7.2b}$$

provides an n'th degree polynomial which exactly passes through *all* of the datapoints, by construction. To verify, note that $P(x_\iota) = \sum_{\kappa=0}^{n} y_\kappa \delta_{\iota\kappa} = y_\iota$ as required. Implementation of this constructive technique to determine the interpolating polynomial is given in Algorithm 7.13.

Algorithm 7.13: Lagrange interpolation.

View
Test

```
function [f]=Lagrange(x,xd,fd)
% Perform Lagrange interpolation based on the {xd,fd} and evaluate at the points in x.
n=length(xd); m=length(x); for j=1:m, f(j)=0; for k=1:n
  L=1; for i=1:n, if i~=k, L=L*(x(j)-xd(i))/(xd(k)-xd(i)); end, end, f(j)=f(j)+fd(k)*L;
end, end
end % function Lagrange
```

Unfortunately, if the number of datapoints is large, high-order polynomials sometimes meander significantly between the datapoints even if the data appears to be fairly regular, as shown in Figure 7.11. Thus, Lagrange interpolation should be thought of as dangerous for anything more than a few datapoints and avoided in favor of other techniques, such as the cubic spline interpolation technique discussed below.

7.3.3 Cubic spline interpolation

Instead of forcing a high-order polynomial through the entire dataset, we may instead construct a continuous, smooth, piecewise cubic function through the data. We will construct this function to be smooth in the sense of having continuous first and second derivatives at each datapoint. These conditions, together with the appropriate conditions at each end, uniquely determine a piecewise cubic function through the data which is usually reasonably smooth; we will call this function the **cubic spline interpolant**.

Defining the interpolant in this manner is akin to deforming a single **spline**, or a thin piece of wood or metal, to pass over all of the datapoints plotted on a large block of wood and marked with thin nails. The elasticity equation governing the deformation f of such a spline is

$$f'''' = G, \tag{7.3a}$$

where G is a force localized near each nail which is sufficient to pass the spline through the data. As G is nonzero only in the immediate vicinity of each nail, such a spline takes an approximately piecewise cubic shape between the datapoints. Thus, *between the datapoints*, we have:

$$f'''' = 0, \quad f''' = C_1, \quad f'' = C_1 x + C_2, \quad f' = \frac{C_1}{2}x^2 + C_2 x + C_3, \quad \text{and} \tag{7.3b}$$

$$f = \frac{C_1}{6}x^3 + \frac{C_2}{2}x^2 + C_3 x + C_4. \tag{7.3c}$$

Constructing the cubic spline interpolant

Let $f_i(x)$ denote the cubic in the interval $x_i \le x \le x_{i+1}$ and let $f(x)$ denote the collection of all the cubics for the entire range $x_0 \le x \le x_n$. As noted above, f_i'' varies linearly with x between each datapoint. *At* each datapoint, we would like to piece these cubics together as smoothly as possible, thereby mimicking the physical situation in which the force G localized on the spline near each nail is as smooth as possible; in fact, we have enough flexibilty to impose the following:

(a) continuity of the function f, *i.e.*, $\quad f_{i-1}(x_i) = f_i(x_i) \;= f(x_i) \;= y_i$,

(b) continuity of the first derivative f', *i.e.*, $\quad f'_{i-1}(x_i) = f'_i(x_i) \;= f'(x_i)$, $\quad$ and

(c) continuity of the second derivative f'', *i.e.*, $\quad f''_{i-1}(x_i) = f''_i(x_i) = f''(x_i)$.

We now describe a procedure to determine an f which satisfies conditions (a) and (c) by *construction*, in a manner analogous to the construction of the Lagrange interpolant in §7.3.2, and which satisfies condition (b) by setting up and solving the appropriate system of equations for the value of f'' at each datapoint x_i.

To begin the constructive procedure for determining f, note that on each interval $x_i \le x \le x_{i+1}$ for $i = 0,1,\ldots,n-1$, we may write a linear equation for $f_i''(x)$ as a function of its value at the endpoints, $f''(x_i)$ and $f''(x_{i+1})$, which are (as yet) undetermined. The following form (which is linear in x) fits the bill by construction:

$$f_i''(x) = f''(x_i)\frac{x-x_{i+1}}{x_i-x_{i+1}} + f''(x_{i+1})\frac{x-x_i}{x_{i+1}-x_i}. \tag{7.4}$$

Note that this first degree polynomial is in fact just a Lagrange interpolation of the two datapoints $\{x_i, f''(x_i)\}$ and $\{x_{i+1}, f''(x_{i+1})\}$ [see (7.2), for $n=1$]. By construction, condition (c) is satisfied. Integrating this equation twice and defining $\Delta_i = x_{i+1} - x_i$, it follows that

$$\begin{aligned} f_i'(x) &= -\frac{f''(x_i)}{2}\frac{(x_{i+1}-x)^2}{\Delta_i} + \frac{f''(x_{i+1})}{2}\frac{(x-x_i)^2}{\Delta_i} + C_1, \\ f_i(x) &= \frac{f''(x_i)}{6}\frac{(x_{i+1}-x)^3}{\Delta_i} + \frac{f''(x_{i+1})}{6}\frac{(x-x_i)^3}{\Delta_i} + C_1 x + C_2. \end{aligned}$$

The undetermined constants of integration are obtained by matching the end conditions

$$f_i(x_i) = y_i \qquad \text{and} \qquad f_i(x_{i+1}) = y_{i+1}.$$

A convenient way of constructing the linear and constant terms in the expression for $f_i(x)$ in such a way that the desired end conditions are met is by writing $f_i(x)$ in the form

$$\begin{aligned} f_i(x) = \frac{f''(x_i)}{6}\left(\frac{(x_{i+1}-x)^3}{\Delta_i} - \Delta_i(x_{i+1}-x)\right) + \frac{f''(x_{i+1})}{6}\left(\frac{(x-x_i)^3}{\Delta_i} - \Delta_i(x-x_i)\right) \\ + y_i\frac{(x_{i+1}-x)}{\Delta_i} + y_{i+1}\frac{(x-x_i)}{\Delta_i}, \qquad \text{where} \qquad x_i \le x \le x_{i+1}. \end{aligned} \tag{7.5}$$

By construction, condition (a) is satisfied. Finally, an expression for $f_i'(x)$ may now be found by differentiating this expression for $f_i(x)$, which gives

$$f_i'(x) = \frac{f''(x_i)}{6}\left(-3\frac{(x_{i+1}-x)^2}{\Delta_i} + \Delta_i\right) + \frac{f''(x_{i+1})}{6}\left(3\frac{(x-x_i)^2}{\Delta_i} - \Delta_i\right) + \frac{y_{i+1}}{\Delta_i} - \frac{y_i}{\Delta_i}.$$

The second derivative of f at each node, $f''(x_i)$, is still undetermined. A system of equations from which the $f''(x_i)$ may be found is obtained by imposing condition (b), which is achieved by setting

$$f_i'(x_i) = f_{i-1}'(x_i) \qquad \text{for} \qquad i = 1,2,\ldots,n-1.$$

Substituting appropriately from the above expression for $f_i'(x)$, noting that $\Delta_i = x_{i+1} - x_i$, leads to

$$\frac{\Delta_{i-1}}{6}f''(x_{i-1}) + \frac{\Delta_{i-1}+\Delta_i}{3}f''(x_i) + \frac{\Delta_i}{6}f''(x_{i+1}) = \frac{y_{i+1}-y_i}{\Delta_i} - \frac{y_i - y_{i-1}}{\Delta_{i-1}} \tag{7.6}$$

for $i = 1,2,\ldots,n-1$. This is a diagonally-dominant tridiagonal system of $n-1$ equations for the $n+1$ unknowns $f''(x_0)$, $f''(x_1)$, …, $f''(x_n)$. We find the two remaining equations by prescribing conditions on the interpolating function at each end. We will consider three types of end conditions:

- parabolic run-out: $f''(x_0) = f''(x_1)$ and $f''(x_n) = f''(x_{n-1})$;
- free run-out (also known as natural splines): $f''(x_0) = 0$ and $f''(x_n) = 0$; or
- periodic end conditions: $f''(x_0) = f''(x_{n-1})$ and $f''(x_1) = f''(x_n)$.

Equation (7.6) may be taken together with the appropriate choice of end conditions (depending upon the problem at hand) to give $n+1$ equations for the $n+1$ unknowns $f''(x_i)$. This set of equations is then solved for the $f''(x_i)$, which thereby ensures that condition (b) is satisfied. Once this system is solved for the $f''(x_i)$, the cubic spline interpolant follows immediately from (7.5).

Note that, when (7.6) is taken together with parabolic or free run-out at the ends, a tridiagonal system results which can be solved efficiently with the Thomas algorithm. When (7.6) is taken together periodic end conditions, a tridiagonal circulant system $A\mathbf{x}=\mathbf{b}$ results with $a_{1,1}=0$ (and, thus, Algorithm 2.10, which implements Gaussian elimination without pivoting for Circulant matrices, would fail; Exercise 7.1 considers the required modifications to the Circulant algorithm so that it can be applied to this system). A code which solves these systems with any of the above three end conditions is given in Algorithm 7.14; running this code facilitates the use of Algorithm 7.15 to determine the cubic spline interpolant at any set of points $\mathbf{x}$.

Applying periodic end conditions to develop a spline for a system that is not well approximated as periodic can leads to significant non-physical meanderings of the interpolant near the ends of the domain; thus, periodic end conditions should be reserved for systems which are actually periodic. On the other hand, parabolic run-out extends a parabolic curve between x_0 and x_1, and free run-out tapers the curvature of the interpolant down to zero near the endpoints; both of these choices usually generate reasonably smooth interpolants.

Tension splines

For certain interpolation problems, cubic splines aren't adequately smooth. In such problems, it is helpful to use **tension splines**, which are cubic splines with the mechanical equivalent of a bit of tension added to straighten out the curvature between the datapoints. As the tension gets large in this approach, the interpolant approaches a piecewise linear function. Tensioned splines obey the differential equation [cf. (7.3a)]:

$$f'''' - \sigma^2 f'' = G$$

where σ is the tension of the spline. This leads to the following relationships between the datapoints [cf. (7.3b)]:

$$[f''-\sigma^2 f]''=0, \qquad [f''-\sigma^2 f]'=C_1, \qquad [f''-\sigma^2 f]=C_1 x+C_2.$$

Solving the ODE on the right leads to an equation of the form [cf. (7.3c)]

$$f=-\sigma^{-2}(C_1 x+C_2)+C_3 e^{-\sigma x}+C_4 e^{\sigma x}.$$

Proceeding with a constructive process to satisfy condition (a) analogous to that used previously, we assemble the linear and constant terms of $f''-\sigma^2 f$ such that [cf. (7.4)]

$$\left[f_i''(x)-\sigma^2 f_i(x)\right]=\left[f_i''(x_i)-\sigma^2 y_i\right]\frac{x-x_{i+1}}{x_i-x_{i+1}}+\left[f_i''(x_{i+1})-\sigma^2 y_{i+1}\right]\frac{x-x_i}{x_{i+1}-x_i}.$$

Similarly, we assemble the exponential terms in the solution of this ODE for f in a constructive manner such that condition (c) is satisfied. Rewriting the exponentials as sinh functions, the desired solution may be written [cf. (7.5)]

$$\begin{aligned} f_i(x)=-\sigma^{-2}\Big\{ &\left[f''(x_i)-\sigma^2 y_i\right]\frac{x_{i+1}-x}{\Delta_i}+\left[f''(x_{i+1})-\sigma^2 y_{i+1}\right]\frac{x-x_i}{\Delta_i}-f''(x_i)\frac{\sinh\sigma(x_{i+1}-x)}{\sinh\sigma\Delta_i} \\ &-f''(x_{i+1})\frac{\sinh\sigma(x-x_i)}{\sinh\sigma\Delta_i}\Big\} \qquad \text{where} \qquad x_i\le x\le x_{i+1}. \end{aligned} \tag{7.7}$$

Differentiating once and appling condition (b) leads to the tridiagonal system [cf. (7.6)]

$$\begin{aligned} &\left(\frac{1}{\Delta_{i-1}}-\frac{\sigma}{\sinh\sigma\Delta_{i-1}}\right)\frac{f''(x_{i-1})}{\sigma^2}-\left(\frac{1}{\Delta_{i-1}}-\frac{\sigma\cosh\sigma\Delta_{i-1}}{\sinh\sigma\Delta_{i-1}}+\frac{1}{\Delta_i}-\frac{\sigma\cosh\sigma\Delta_i}{\sinh\sigma\Delta_i}\right)\frac{f''(x_i)}{\sigma^2} \\ &\quad+\left(\frac{1}{\Delta_i}-\frac{\sigma}{\sinh\sigma\Delta_i}\right)\frac{f''(x_{i+1})}{\sigma^2}=\frac{y_{i+1}-y_i}{\Delta_i}-\frac{y_i-y_{i-1}}{\Delta_{i-1}}. \end{aligned} \tag{7.8}$$

Algorithm 7.14: Cubic spline interpolation (setup).

View

```
function [fpp,h]=CubicSplineSetup(xd,fd,end_conditions)
% Determine the intervals h and the curvature f'' for constructing the cubic spline
% interpolant of the datapoints {xd,fd}, assuming this data is ordered as ascending in x.
n=length(xd); h(1:n-1)=xd(2:n)-xd(1:n-1);      % Calculate the h(i) = x(i+1)-x(i)
for i=2:n-1       % Now, set up and solve the tridiagonal system for g at each data point.
  a(i)=h(i-1)/6; b(i)=(h(i-1)+h(i))/3; c(i)=h(i)/6;
  g(i,1)=(fd(i+1)-fd(i))/h(i)-(fd(i)-fd(i-1))/h(i-1);
end
switch end_conditions
  case {1,'parabolic'}
    b(1)=1;  c(1)=-1;  g(1,1)=0;
    b(n)=1;  a(n)=-1;  g(n,1)=0;          [fpp]=Thomas(a,b,c,g,n);
  case {2,'natural'}
    b(1)=1;  c(1)=0;   g(1,1)=0;
    b(n)=1;  a(n)=0;   g(n,1)=0;          [fpp]=Thomas(a,b,c,g,n);
  case {3,'periodic'}
    a(1)=-1; b(1)=0; c(1)=1; g(1,1)=0;
    a(n)=-1; b(n)=0; c(n)=1; g(n,1)=0;  A=TriDiag(a,b,c), fpp=GaussCP(A,g,n);
end
end % function CubicSplineSetup
```

Algorithm 7.15: Cubic spline interpolation (evaluation).

View
Test

```
function [f,fp]=CubicSpline(x,xd,fd,fpp,h)
% Perform cubic spline interpolation based on the {xd,fd} and evaluate at the points in x.
% Note: the initialization data {fpp,h} must be computed first using CubicSplineSetup.
n=length(xd); m=length(x); i=1; for j=1:m
  for i=i:n-1, if xd(i+1) > x(j), break, end, end % Find the i such that xd(i)<=x<=xd(i+1)
  f(j)=fpp(i)/6 *((xd(i+1)-x(j))^3/h(i)-h(i)*(xd(i+1)-x(j))) +... % Compute the cubic
     fpp(i+1)/6 *((x(j) - xd(i))^3/h(i)-h(i)*(x(j) - xd(i))) +... % spline approximation
     (fd(i)*(xd(i+1)-x(j)) + fd(i+1)*(x(j)-xd(i))) / h(i);       % of the function,
  if nargout==2,
     fp(j)=fpp(i)/6*(-3*(xd(i+1)-x(j))^2/h(i)+h(i)) +...          % and (if requested) its
         fpp(i+1)/6*( 3*(x(j) - xd(i))^2/h(i)-h(i)) + (fd(i+1)-fd(i))/h(i);  % derivative.
end, end
end % function CubicSpline
```

The tridiagonal system (7.8) can be set up and solved exactly as was done with (7.6), even though the coefficients have a slightly more complicated form. The tensioned-spline interpolant is then given by (7.7).

B-splines

We may easily express the cubic spline (or tension spline) interpolant in a form similar to our construction of the Lagrange interpolant, that is,

$$f(x) = \sum_{\kappa=0}^{n} y_\kappa b_\kappa(x),$$

where the **basis functions** $b_\kappa(x)$ are spline interpolations of Kronecker delta functions such that $b_\kappa(x_i) = \delta_{i\kappa}$, as discussed in §7.3.2 for the functions $L_\kappa(x)$. The basis functions so constructed are found to have **localized support** (in other words, $b_\kappa(x) \to 0$ for large $|x - x_\kappa|$).

By relaxing some of the continuity constraints, we may confine each of the basis functions to have **compact support** (in other words, we can set $b_\kappa(x) = 0$ exactly for $|x - x_\kappa| > R$ for some R). With such functions, it is easier both to compute the interpolations themselves and to project the interpolated function onto a different grid of points.

Algorithm 7.16: Bilinear interpolation of data defined on a 2D Cartesian grid.

View
Test

```
function [f]=BilinearSpline(x,y,xd,yd,fd)
% Perform bilinear interpolation based on {xd,yd,fd} and evaluate on the grid {x,y}.
ndx=length(xd); ndy=length(yd); nx=length(x); ny=length(y); i=1;
for ii=1:nx; j=1;
  for i=i:ndx-1,   if xd(i+1)>x(ii), break, end, end  % Find i s.t. xd(i)<=x(ii)<=xd(i+1)
  for jj=1:ny
    for j=j:ndy-1, if yd(j+1)>y(jj), break, end, end  % Find j s.t. yd(j)<=y(jj)<=yd(j+1)
    d10=(xd(i+1)-x(ii))/(xd(i+1)-xd(i)); d11=1-d10;   % Compute the distance across cell,
    d20=(yd(j+1)-y(jj))/(yd(j+1)-yd(j)); d21=1-d20;   % then compute the interpolant.
    f(ii,jj)=fd(i,j)*d10*d20+fd(i+1,j)*d11*d20+fd(i,j+1)*d10*d21+fd(i+1,j+1)*d11*d21;
  end
end
end % function BilinearSpline
```

7.4 Multivariate interpolation of structured data

The interpolation strategies described above are well suited for 1D problems, and can be extended fairly easily to higher dimensions on structured n-dimensional grids. Below we describe two such extensions.

Multilinear interpolation

The idea of 1D linear spline interpolation (see §7.3.1) extends immediately to the **multilinear interpolation** of data defined on an n-dimensional Cartesian grid (that is, function values $f_{i_1,i_2,\ldots,i_n} = f(x_{1_{i_1}}, x_{2_{i_2}}, \ldots, x_{n_{i_n}})$ where $i_1 = 1,\ldots,N_1,\ i_2 = 1,\ldots,N_2$, etc.) as follows:

- Determine the grid cell that new interpolating point $\mathbf{x}$ lies in: that is, find the i_1 through i_n such that $x_{k_{i_k}} \le x_k \le x_{k_{(i_k+1)}}$ for $1 \le k \le n$.
- Determine the fraction of the distance that the new point $\mathbf{x}$ is across this cell in each direction: that is, compute $\eta_{k,0} = (x_{k_{(i_k+1)}} - x_k)/(x_{k_{(i_k+1)}} - x_{k_{i_k}})$ and $\eta_{k,1} = 1 - \eta_{k,0}$ for $1 \le k \le n$.
- Linearly interpolate in each direction independently by setting the interpolant $\tilde{f}(x)$ such that

$$\tilde{f}(x) = \sum_{d_1=0}^{1} \sum_{d_2=0}^{1} \cdots \sum_{d_n=0}^{1} f_{i_1+d_1,i_2+d_2,\ldots,i_n+d_n}\, \eta_{1,d_1}\, \eta_{2,d_2} \cdots \eta_{n,d_n}.$$

Implementation for $n = 2$ (referred to as **bilinear interpolation**) is given in Algorithm 7.16; see Figure 7.12a for typical results. Implementation in the n-dimensional case is considered in Exercise 7.2.

Multicubic interpolation

The idea of cubic spline interpolation (see §7.3.3) may be extended in a couple of different ways to data defined on an n-dimensional Cartesian grid.

An accurate and simple approach is to do cubic spline interpolation in each dimension, one at a time:

- First, interpolate onto the specified value of x_1 for each value of x_2 through x_n on the grid (that is, for $i_2 = 1,\ldots,N_2,\ i_3 = 1,\ldots,N_3,\ i_4 = 1,\ldots,N_4,$ etc.).
- Then, working only with those function values interpolated onto the specified value of x_1, interpolate onto the specified value of x_2 for each value of x_3 through x_n on the grid (that is, for $i_3 = 1,\ldots,N_3$, $i_4 = 1,\ldots,N_4$, etc.).
- Continue in an analogous fashion through the remaining dimensions, one at a time.

Recall that, in the one-dimensional case described in §7.3.3, the computationally expensive part of setting up the cubic spline interpolant could be computed once during the initialization step, then used for interpolating onto any specified point x. Unfortunately, in the multidimensional approach described above, this is no longer the case, as the interpolations performed in the x_j direction, for $j = 2, \ldots, n$, depend on the data that results from the interpolations performed in the x_1 to x_{j-1} directions. This approach is thus too expensive to be practically useful when interpolating onto a large number of gridpoints.

An inexpensive alternative[3] for extending cubic spline interpolation to n-dimensional grids follows:

(i) First, during an initialization step, approximate all first and cross derivatives of f at each gridpoint where the function f is initially specified. Note that these numerical approximations may be computed by successive cubic spline interpolations along the gridlines, evaluated at the gridpoints.

(ii) Then, as in the multilinear interpolation approach described in §7.4, determine (for each new interpolation point) which grid cell that the new interpolation point $\mathbf{x}$ lies in, and the fraction of the distance that the new point $\mathbf{x}$ is across this cell in each coordinate direction.

(iii) Finally, construct a function which is cubic in each coordinate variable and matches the first and cross derivative information computed in step i at each of the corners of the cell identified in step ii.

This idea is best made concrete by example. In the case of $n = 2$ (**bicubic interpolation**), we first use cubic spline interpolation along each of the gridlines to compute $\{f_x, f_y, f_{xy}\}$ at each of the gridpoints where the function values f are initially specified. Then, as in bilinear interpolation, we determine which grid cell that the new interpolating point $\mathbf{x}$ lies in, and the fraction of the distance that the new point $\mathbf{x}$ is across this cell in each direction (denoted here x and y). Finally, the interpolant on the cell is defined by

$$f(x,y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j, \tag{7.9}$$

where the a_{ij} for $i = 0, \ldots, 3$ and $j = 0, \ldots, 3$ are selected to match the values of $\{f, f_x, f_y, f_{xy}\}$ at each of the 4 corners of the cell (w.l.o.g, taken here to be the unit square, with corners denoted $\{\mathbf{x}_{00}, \mathbf{x}_{01}, \mathbf{x}_{10}, \mathbf{x}_{11}\}$) that contains the new interpolation point; this results in a linear 16×16 problem of the form $A\mathbf{x} = \mathbf{b}$ where

$$\begin{pmatrix}
1 & & & & & & & & & & & & & & & \\
1 & 1 & 1 & 1 & & & & & & & & & & & & \\
1 & & & & 1 & & & & 1 & & & & 1 & & & \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 & 1 & & & & & & & & & & & & & & \\
 & 1 & 2 & 3 & & & & & & & & & & & & \\
 & 1 & & & & 1 & & & & 1 & & & & 1 & & \\
 & 1 & 2 & 3 & & 1 & 2 & 3 & & 1 & 2 & 3 & & 1 & 2 & 3 \\
 & & & & 1 & & & & & & & & & & & \\
 & & & & 1 & 1 & 1 & 1 & & & & & & & & \\
 & & & & 1 & & & & 2 & & & & 3 & & & \\
 & & & & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\
 & & & & & 1 & & & & & & & & & & \\
 & & & & & 1 & 2 & 3 & & & & & & & & \\
 & & & & & 1 & & & & 2 & & & & 3 & & \\
 & & & & & 1 & 2 & 3 & & 2 & 4 & 6 & & 3 & 6 & 9
\end{pmatrix}
\begin{pmatrix} a_{00} \\ a_{10} \\ a_{20} \\ a_{30} \\ a_{01} \\ a_{11} \\ a_{21} \\ a_{31} \\ a_{02} \\ a_{12} \\ a_{22} \\ a_{32} \\ a_{03} \\ a_{13} \\ a_{23} \\ a_{33} \end{pmatrix}
=
\begin{pmatrix} f|_{\mathbf{x}_{00}} \\ f|_{\mathbf{x}_{10}} \\ f|_{\mathbf{x}_{01}} \\ f|_{\mathbf{x}_{11}} \\ f_x|_{\mathbf{x}_{00}} \\ f_x|_{\mathbf{x}_{10}} \\ f_x|_{\mathbf{x}_{01}} \\ f_x|_{\mathbf{x}_{11}} \\ f_y|_{\mathbf{x}_{00}} \\ f_y|_{\mathbf{x}_{10}} \\ f_y|_{\mathbf{x}_{01}} \\ f_y|_{\mathbf{x}_{11}} \\ f_{xy}|_{\mathbf{x}_{00}} \\ f_{xy}|_{\mathbf{x}_{10}} \\ f_{xy}|_{\mathbf{x}_{01}} \\ f_{xy}|_{\mathbf{x}_{11}} \end{pmatrix}.$$

Note that A^{-1} is particularly simple, with all integer entries, and is thus entered directly in the implementation of bicubic interpolation in Algorithms 7.17-7.18. See Figure 7.12b for typical results.

[3] Recall (from the introduction to §7.3) that the interpolation problem itself is an approximate problem subject to a significant degree of interpretation; it may thus be argued that approximate solution to a problem of this class is good enough, and one should not code up an unduly expensive scheme in order to solve an approximate problem of this class "exactly".

Algorithm 7.17: Bicubic spline interpolation (setup).

View

```
function [fx,fy,fxy,Ainv]=BicubicSplineSetup(xd,yd,fd,ec)
% Determine the derivatives {fx,fy,fxy} and Ainv for constructing the bicubic interpolant
% of the uniform grid of datapoints {xd,yd,fd}.
Nx=length(xd); Ny=length(yd);
for j=1:Ny                                                    % Compute the necessary derivatives
  [g,h]=CubicSplineSetup(xd,fd(:,j),ec); [t,fx(:,j)]=CubicSpline(xd,xd,fd(:,j),g,h);
end
fx=fx*h(1); % (scale so later interpolations can be based on integer grid spacing)
for i=1:Nx
  [g,h]=CubicSplineSetup(yd,fd(i,:),ec); [t,fy(i,:)]=CubicSpline(yd,yd,fd(i,:),g,h);
  [g,h]=CubicSplineSetup(yd,fx(i,:),ec); [t,fxy(i,:)]=CubicSpline(yd,yd,fx(i,:),g,h);
end
fy=fy*h(1); fxy=fxy*h(1);
Ainv=[1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0; % Set up Ainv
      0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0;
     -3  3  0  0 -2 -1  0  0  0  0  0  0  0  0  0  0;
      2 -2  0  0  1  1  0  0  0  0  0  0  0  0  0  0;
      0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0;
      0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0;
      0  0  0  0  0  0  0  0 -3  3  0  0 -2 -1  0  0;
      0  0  0  0  0  0  0  0  2 -2  0  0  1  1  0  0;
     -3  0  3  0  0  0  0  0 -2  0 -1  0  0  0  0  0;
      0  0  0  0 -3  0  3  0  0  0  0  0 -2  0 -1  0;
      9 -9 -9  9  6  3 -6 -3  6 -6  3 -3  4  2  2  1;
     -6  6  6 -6 -3 -3  3  3 -4  4 -2  2 -2 -2 -1 -1;
      2  0 -2  0  0  0  0  0  1  0  1  0  0  0  0  0;
      0  0  0  0  2  0 -2  0  0  0  0  0  1  0  1  0;
     -6  6  6 -6 -4 -2  4  2 -3  3 -3  3 -2 -1 -2 -1;
      4 -4 -4  4  2  2 -2 -2  2 -2  2 -2  1  1  1  1];
end % function CubicSplineSetup
```

Algorithm 7.18: Bicubic spline interpolation (evaluation).

View
Test

```
function [f]=BicubicSpline(x,y,xd,yd,fd,fx,fy,fxy,Ainv)
% Given the grid of datapoints {xd,yd,fd}, as well as the derivatives {fx,fy,fxy} on this
% same grid and Ainv (as computed by BicubicSplineSetup), determine the bicubic spline
ndx=length(xd); ndy=length(yd); nx=length(x); ny=length(y); i=1;
for ii=1:nx; j=1;
  for i=i:ndx-1,   if xd(i+1)>x(ii), break, end, end % Find i s.t. xd(i)<=x(ii)<=xd(i+1)
  for jj=1:ny
    for j=j:ndy-1, if yd(j+1)>y(jj), break, end, end % Find j s.t. yd(j)<=y(jj)<=yd(j+1)
    x1=(x(ii)-xd(i))/(xd(i+1)-xd(i)); y1=(y(jj)-yd(j))/(yd(j+1)-yd(j));
    b=[fd(i,j);  fd(i+1,j);  fd(i,j+1);  fd(i+1,j+1); ...
       fx(i,j);  fx(i+1,j);  fx(i,j+1);  fx(i+1,j+1); ...
       fy(i,j);  fy(i+1,j);  fy(i,j+1);  fy(i+1,j+1); ...
       fxy(i,j); fxy(i+1,j); fxy(i,j+1); fxy(i+1,j+1)];
    a=Ainv*b;
    f(ii,jj)=0; for I=0:3, for J=0:3, f(ii,jj)=f(ii,jj)+a(1+I+4*J)*x1^I*y1^J; end, end
  end
end
end % function BicubicSpline
```

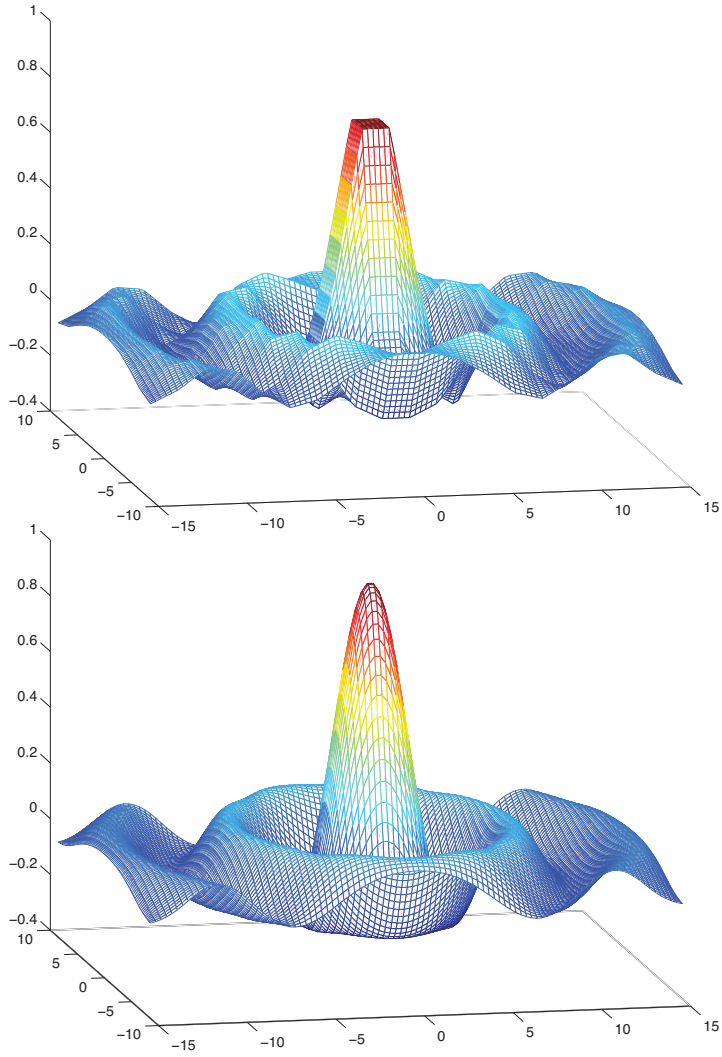


Figure 7.12: (top) Bilinear interpolation and (bottom) bicubic interpolation of a 10×20 grid of data from the function $\mathrm{sinc}(r) \triangleq \sin(r)/r$ where $r = \sqrt{x^2 + y^2}$; the bilinear case reveals some noticeable artifacts of the interpolation, whereas the bicubic interpolant is visually almost indistinguishable from the original function.

Similarly, in the case of $n = 3$ (**tricubic interpolation**), we use cubic spline interpolation along each gridline to compute $\{f_x, f_y, f_z, f_{xy}, f_{yz}, f_{xz}, f_{xyz}\}$ at each of the gridpoints where the values of f are initially prescribed. We then determine which grid cell that new interpolating point $\mathbf{x}$ lies in, and the fraction of the distance that the point $\mathbf{x}$ is across this cell in each direction. Finally, the interpolant on the cell is defined by

$$f(x,y,z) = \sum_{i=0}^{3} \sum_{j=0}^{3} \sum_{k=0}^{3} a_{ijk} x^i y^j z^j, \tag{7.10}$$

where the a_{ijk} are selected to match the values of $\{f, f_x, f_y, f_z, f_{xy}, f_{yz}, f_{xz}, f_{xyz}\}$ at each of the 8 corners of the cell (w.l.o.g, taken to be a unit cube) that contains the new interpolation point; this results in 64 linear equations for 64 unknowns which may easily be solved. Implementation is considered in Exercise 7.3.

Algorithm 7.19: Interpolation via inverse distance.

View
Test

```
function [fn]=InvDistanceInterp(xn,c,f,p,R)
% Given the data {c,f}, compute the inverse distance interpolant fn at a new point xn.
N=length(f); C=0; fn=0;
for i=1:N, d=norm(xn-c(:,i),2); if d<R, C=C+1/d^p; fn=fn+f(i)/d^p; end, end, fn=fn/C;
end % function InvDistanceInterp
```

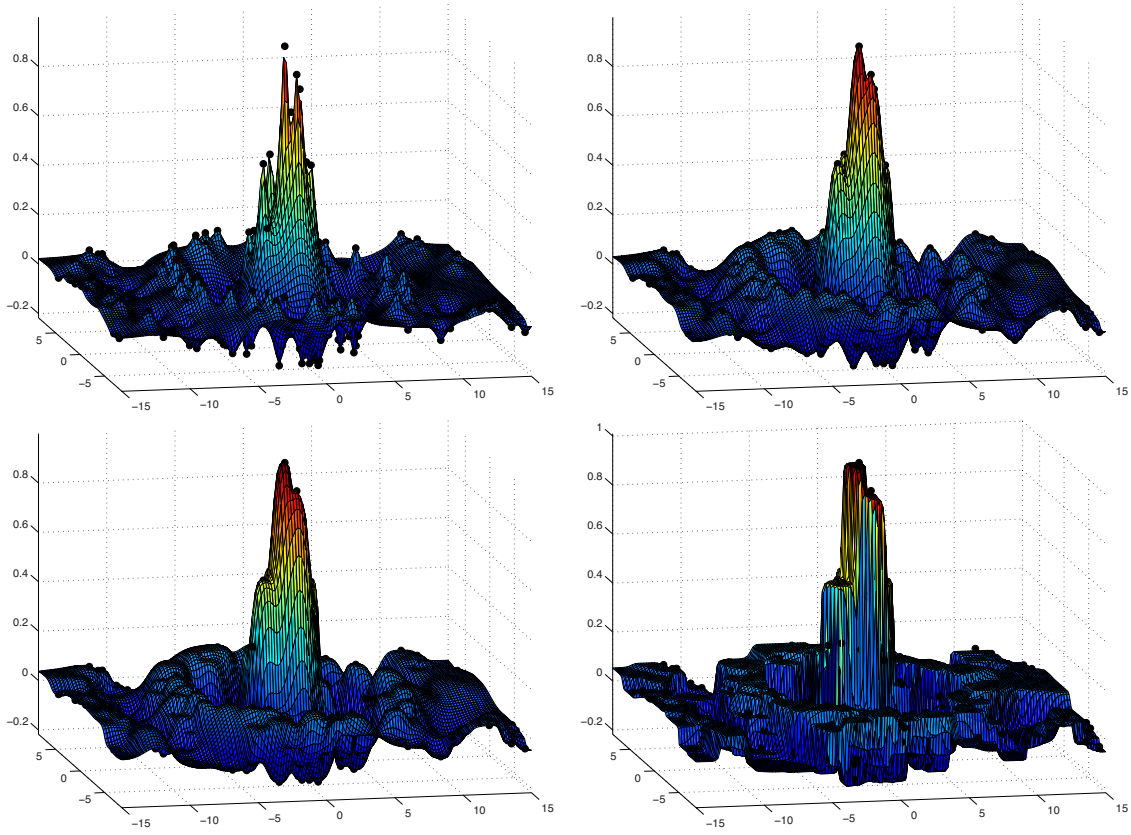


Figure 7.13: Interpolation via inverse distance of 200 points with (a) $p = 2$, (b) $p = 3$, (c) $p = 4$, (d) $p = 20$.

7.5 Multivariate interpolation of unstructured data

The extension of interpolation strategies to unstructured data (that is, for data not lying on a regular grid) requires a bit more effort than the case of structured data considered above; we will thus consider three different approaches to this problem.

7.5.1 Interpolation via inverse distance

The simplest approach for approximating the function value f at location $\mathbf{x}$ based on N known function values f_i at various locations $\mathbf{c}_i$, for $i = 1,\ldots,N$, is the **inverse distance** interpolation formulae given by

$$f(\mathbf{x}) = \frac{1}{C}\sum_{\substack{i=1\\ d_i\le R}}^{N} f_i/d_i^p \quad \text{where} \quad C = \sum_{\substack{i=1\\ d_i\le R}}^{N} 1/d_i^p \quad \text{and} \quad d_i = \|\mathbf{x}-\mathbf{c}^i\|_2,$$

where $1 \leq p < \infty$ is some power and the sum includes all known function values within some prespecified distance R of the point in question, $\mathbf{x}$. Implementation is given in Algorithm 7.19, and typical results are illustrated in Figure 7.13. Note that the minima and maxima of this interpolating function coincide with datapoints representing the largest and smallest function values in the dataset. For small p (e.g., $p = 2$), the interpolant looks like a tent propped up, and pushed down, at the various datapoints; for increasing values of p (e.g., $p = 3$, $p = 4$), the interpolant gains stronger "shoulders" near each datapoint; for $p \to \infty$, the interpolant takes the known function value at the nearest datapoint, and thus leads to a **piecewise constant** function over the **Voronoi cell** associated with each datapoint. For finite p in the limit that $R \to \infty$, the interpolation function is continuous; for reduced values of R, the interpolating function, though sometimes approximating the original function a bit more accurately, is discontinuous.

7.5.2 Polyharmonic spline interpolation

In many interpolation problems, such as that illustrated in Figure 7.13, the simple inverse distance formula for interpolation, discussed in §7.5.1, fails to give a sufficiently accurate result for any value of p. An effective alternative approach is given by the **polyharmonic spline**, a special case of which (in 2D and with $k = 2$), known as a **thin plate spline**, corresponds to the mechanical modeling of a 2D spline that is bent in order to make it touch the specified unstructured data points. This interpolation formula takes the form[4]

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i \phi(r) + \mathbf{v}^T \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} \qquad \text{where} \quad r = \|\mathbf{x} - \mathbf{c}^i\|_2, \tag{7.11a}$$

$$\phi(r) = \begin{cases} r^k & \text{for } k \text{ odd} \\ r^k \ln(r) & \text{for } k \text{ even,} \end{cases} \tag{7.11b}$$

and where the weights w_i and v_i are selected such that: (a) $f(\mathbf{c}_i) = f_i$ in the above equation for the N available data points $\{\mathbf{c}_i, f_i\}$, (b) the sum of the weights, $\sum_i w_i$, is zero, and (c) in each of the n coordinate directions, $j = 1, \ldots, n$, the weighted sum of the center locations, $\sum_i w_i c_{ji}$, is also zero. These three sets of conditions on the weights may be enforced by solving the $(N+1+n) \times (N+1+n)$ linear system

$$\begin{bmatrix} A & V^T \\ V & 0 \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ 0 \end{bmatrix} \qquad \text{where} \quad A_{ij} = \phi(\|\mathbf{c}_j - \mathbf{c}^i\|_2), \quad V = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \mathbf{c}^1 & \mathbf{c}^2 & \cdots & c^N \end{bmatrix}. \tag{7.12}$$

Implementation of (7.12) is given in Algorithm 7.20 in order to determine the $\mathbf{w}$ and $\mathbf{v}$; implementation of (7.11a) is then given in Algorithm 7.21 to determine the interpolant at any given point $\mathbf{x}$. Typical results are illustrated in Figure 7.14; note that increasing values of k are usually found to give a smoother interpolant on the interior of the portion of the domain covered by the data, but higher irregularities near the edges of this portion of the domain. Also, smaller values of k (e.g., 2 or 3) are often found to be more accurate when a relatively small number of function evaluations are available, with larger values of k (e.g., 6 to 8) being more accurate when more function evaluations are available.

It is worth noting that the **polyharmonic spline** basis functions $\phi(r)$ given in (7.11b), used in the interpolation formula given in (7.11a) and plotted in Figure 7.15, are special cases of what are commonly referred to as **radial basis functions** (**RBF**s), as they depend on the Euclidian distance $r = \|\mathbf{x} - \mathbf{c}^i\|_2$ of the new point $\mathbf{x}$ from the centers $\mathbf{c}_i$ only. RBFs come in two essential types: those which decay with radius, such as the **Gaussian RBF** $\phi(r) = e^{-(\varepsilon r)^2}$ as well as the **inverse distance RBF** $\phi(r) = 1/r^p$ used in the interpolation strategy presented in §7.5.1, and those which eventually grow with radius, such as the **polyharmonic spline RBF** defined in (7.11b) and used in the interpolation strategy presented in §7.5.2; the former are essentially "local" in nature, whereas the latter are "global" in effect, and thus their weights must be determined via a solve over the entire set of datapoints [see (7.12)].

[4] In the case with k even, the equivalent formula $\phi(r) = r^{k-1} \ln(r^r)$ is better behaved numerically for $r < 1$. Note that various other possible formulae for $\phi(r)$, such as $\phi(r) = r^2$, are found to be ill-behaved in this setting, and are thus not considered further here.

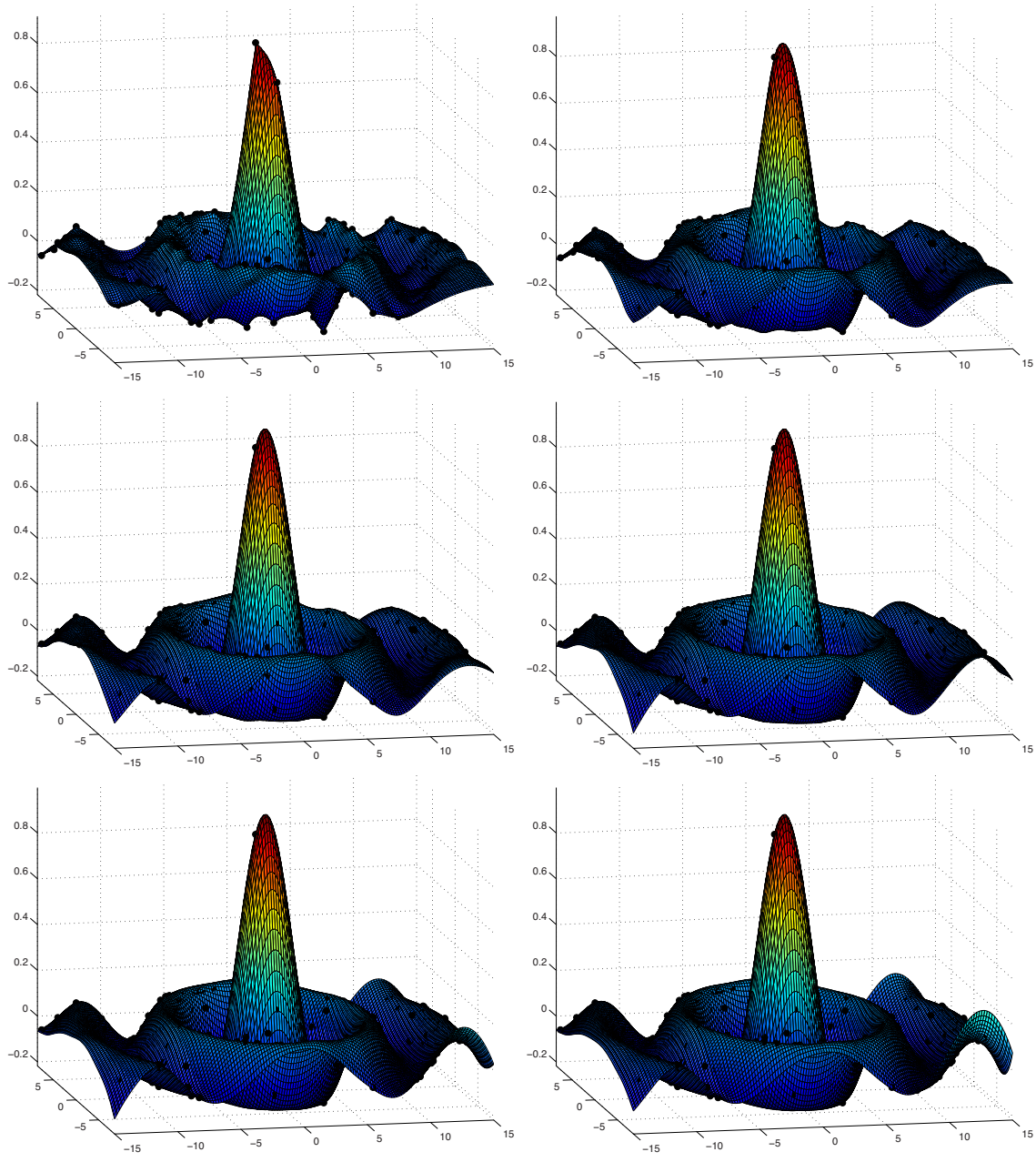


Figure 7.14: Polyharmonic spline interpolation of 200 points with (a) $k = 1$ through (f) $k = 6$. Over the domain illustrated, the maximum error is $\{0.176, 0.123, 0.119, 0.131, 0.186, 0.259\}$ and the rms error is $\{0.0376, 0.0220, 0.0157, 0.0130, 0.0142, 0.0171\}$, respectively, in the six cases considered.

Algorithm 7.20: Polyharmonic spline interpolation (setup).

View

```
function [w,v]=PolyharmonicSplineSetup(c,y,k)
% Given the centers c, the value of the function at these centers, y, and the order of the
% radial basis functions, k, calculate the weights {w,v} of the polyharmonic spline.
[n,N]=size(c); A=zeros(N,N); V=[ones(1,N); c]; % N=number of points, n=dimension of system
if mod(k,2)==1
  for i=1:N, for j=1:N
    r=norm(c(:,i)-c(:,j)); A(i,j)=r^k;
  end, end
else
  for i=1:N, for j=1:N
    r=norm(c(:,i)-c(:,j)); if r>1, A(i,j)=r^k*log(r); else, A(i,j)=r^(k-1)*log(r^r); end
  end, end
end
x=GaussPP([A V'; V zeros(n+1,n+1)],[y'; zeros(n+1,1)],N+n+1); w=x(1:N); v=x(N+1:N+n+1);
end % function PolyharmonicSplineSetup
```

Algorithm 7.21: Polyharmonic spline interpolation (evaluation).

View
Test

```
function [f]=PolyharmonicSpline(x,c,v,w,k)
% Given the centers c, the order of the radial basis functions k, and the weights {v,w} of
% the polyharmonic splines (as computed by PolyharmonicSplineSetup), compute the
% polyharmonic spline interpolant f at the point x.
N=size(c,2); f=v'*[1; x];
if mod(k,2)==1, for i=1:N
  r=norm(x-c(:,i)); f=f+w(i)*r^k;
end, else, for i=1:N
  r=norm(x-c(:,i)); if r>=1, f=f+w(i)*r^k*log(r); else, f=f+w(i)*r^(k-1)*log(r^r); end
end, end
end % function PolyharmonicSpline
```

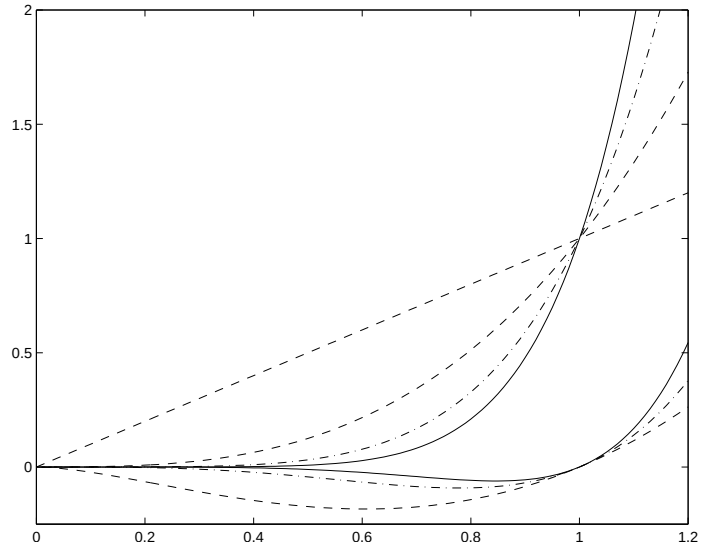


Figure 7.15: **Polyharmonic spline radial basis functions** [see (7.11b)] for (dashed) $k = 1$, $k = 2$, $k = 3$; (dot-dashed) $k = 4$, $k = 5$; (solid) $k = 6$, $k = 7$. Though they eventually grow with radius, these non-local RBFs are found to be effective in the polyharmonic spline interpolation strategy presented in §7.5.2.

7.5.3 Kriging interpolation†

The problem of interpolation fundamentally builds on some hypothesis that models the function behavior in order to "connect the dots" between known function values. The most common such model is a mechanical one, modeling the shape of a thin piece of wood or **spline** that is bent to touch all the datapoints; this mechanical model leads directly to the algorithm known as **cubic spline interpolation** in the single variable case (§7.3.3), as **multicubic interpolation** in the multivariate structured-data case (§7.4), and as **polyharmonic** or **thin-plate interpolation** in the multivariate unstructured-data case (§7.5.2).

A perhaps equally valid hypothesis, which forms the foundation for the **kriging** interpolation strategy (Krige 1951; Matheron 1963; Jones 2001; Rasmussen & Williams 2006), is to *model the underlying function as a realization, with maximum likelihood, of some stochastic process*. The stochastic model used in this approach is selected to be general enough to model a broad range of functions reasonably well, yet simple enough to be fairly inexpensive to tune appropriately based on the measured data. There are many such stochastic models which one can select (see §6); the simple stochastic model considered here leads to the easy-to-use interpolation strategy commonly referred to as **ordinary kriging**.

Notation of statistical description

To begin, consider n points $\{\mathbf{x}^1,\dots,\mathbf{x}^n\}$, at which the function will ultimately be evaluated, and model the function's value at these n points with the random vector

$$\mathbf{f} = \begin{pmatrix} f(\mathbf{x}^1) \\ \vdots \\ f(\mathbf{x}^n) \end{pmatrix} = \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix}.$$

To proceed further, we need a clear statistical framework to describe this random vector. The cumulative distribution function (CDF), denoted $d_{\mathbf{f}}(\underline{\mathbf{f}})$, and the corresponding probability density function (PDF), denoted $p_{\mathbf{f}}(\mathbf{f}')$, of the random vector $\mathbf{f}$ are defined as in §6.1. In particular, the mean $\bar{\mathbf{f}}$ and covariance $P_{\mathbf{f}}$ of the random vector $\mathbf{f}$ are defined as

$$\bar{\mathbf{f}} \triangleq \mathscr{E}\{\mathbf{f}\} = \int_{\mathbb{R}^n} \mathbf{f}'\, p_{\mathbf{f}}(\mathbf{f}')\, d\mathbf{f}', \qquad P_{\mathbf{f}} \triangleq \mathscr{E}\{(\mathbf{f}-\bar{\mathbf{f}})(\mathbf{f}-\bar{\mathbf{f}})^T\} = \int_{\mathbb{R}^n} (\mathbf{f}'-\bar{\mathbf{f}})(\mathbf{f}'-\bar{\mathbf{f}})^T\, p_{\mathbf{f}}(\mathbf{f}')\, d\mathbf{f}'.$$

Statistical modeling assumptions of the ordinary kriging model

The PDF of the random vector $\mathbf{f} = \mathbf{f}_{n\times 1}$ in this analysis is modeled as Gaussian (see §6.2.1), and is thus restricted to the generic form

$$p_{\mathbf{f}}(\mathbf{f}') = \frac{1}{(2\pi)^{n/2}|P_{\mathbf{f}}|^{1/2}} \exp \frac{-(\mathbf{f}'-\bar{\mathbf{f}})^T P_{\mathbf{f}}^{-1} (\mathbf{f}'-\bar{\mathbf{f}})}{2}, \tag{7.13a}$$

where the covariance $P_{\mathbf{f}}$ is modeled as a constant σ^2, referred to as the variance, times a correlation matrix R whose $\{i,j\}$'th component r_{ij} is given by a model of the correlation of the random function f between points $\mathbf{x}^i$ and $\mathbf{x}^j$, where this correlation model $r(\cdot,\cdot)$ itself decays exponentially with the distance between points $\mathbf{x}^i$ and $\mathbf{x}^j$; that is,

$$P_{\mathbf{f}} \triangleq \sigma^2 R, \quad \text{where} \quad r_{ij} \triangleq r(\mathbf{x}^i,\mathbf{x}^j) \quad \text{and} \quad r(\mathbf{x},\mathbf{y}) \triangleq \prod_{\ell=1}^{n} \exp\Big(-\theta_\ell |x_\ell - y_\ell|^{p_\ell}\Big) \tag{7.13b}$$

for some yet-to-be-determined constants σ^2, $\theta_\ell > 0$, and $0 < p_\ell \le 2$ for $\ell = 1,\dots,n$. The mean $\bar{\mathbf{f}}$ in the Gaussian model (7.13a) is itself modeled as uniform over all of its components:

$$\bar{\mathbf{f}} \triangleq \mu \mathbf{1} \tag{7.13c}$$

for some yet-to-be-determined constant μ. There is extensive debate in the literature (see, e.g., Isaaks & Srivastava 1989; Rasmussen & Williams 2006) on the details of the statistical modeling assumptions one should use in a kriging model of this sort. It is straightforward to extend the present discussion to incorporate less restrictive kriging models; the ordinary kriging model is discussed here primarily due to its simplicity.

Adjusting the coefficients of the model based on the data

If the vector of observed function values is

$$\mathbf{f}^o = \begin{pmatrix} f_1^o \\ \vdots \\ f_N^o \end{pmatrix},$$

then the PDF corresponding to this observation in the statistical model proposed in (7.13) can be written as

$$p_{\mathbf{f}}(\mathbf{f}^o) = \frac{1}{(2\pi)^{n/2}(\sigma^2)^{n/2}|R|^{1/2}} \exp \frac{-(\mathbf{f}^o - \mu\mathbf{1})^T R^{-1}(\mathbf{f}^o - \mu\mathbf{1})}{2\sigma^2}. \tag{7.14}$$

The process of kriging modeling boils down to selecting the parameters σ^2, θ_ℓ, p_ℓ, and μ in the statistical model proposed in (7.13) to maximize the PDF evaluated for the function values actually observed, $\mathbf{f} = \mathbf{f}^o$, as given in (7.14).

Maximizing $p_{\mathbf{f}}(\mathbf{f}^o)$ is equivalent to minimizing the negative of its log. Thus, for simplicity, consider

$$J = -\log[p_{\mathbf{f}}(\mathbf{f}^o)] = \frac{n}{2}\log(2\pi) + \frac{n}{2}\log(\sigma^2) + \frac{1}{2}\log(|R|) + \frac{(\mathbf{f}^o - \mu\mathbf{1})^T R^{-1}(\mathbf{f}^o - \mu\mathbf{1})}{2\sigma^2}. \tag{7.15}$$

Setting the derivatives of J with respect to μ and σ^2 equal to zero and solving, the optimal values of μ and σ^2 are determined immediately:

$$\mu = \frac{\mathbf{1}^T R^{-1}\mathbf{f}^o}{\mathbf{1}^T R^{-1}\mathbf{1}}, \qquad \sigma^2 = \frac{(\mathbf{f}^o - \mu\mathbf{1})^T R^{-1}(\mathbf{f}^o - \mu\mathbf{1})}{n}. \tag{7.16}$$

With these optimal values of μ and σ^2 applied, noting that the last term in (7.15) is now constant, what remains to be done is to minimize

$$J_1 = \frac{n}{2}\log(\sigma^2) + \frac{1}{2}\log(|R|) \tag{7.17}$$

with respect to the remaining free parameters[5] θ_ℓ and p_ℓ, where σ^2 is given as a function of R in (7.16) and R, in turn, is given as a function of the free parameters θ_ℓ and p_ℓ in (7.13b). This minimization must, in general, be performed numerically. However, the function J_1 is smooth in the parameters θ_ℓ and p_ℓ, so this optimization may be performed efficiently with a standard gradient-based algorithm, such as the nonquadratic conjugate gradient algorithm (see §16), where the gradient itself, for simplicity, may easily be determined via a the complex step derivative approach (see §8.3.3 and §8.3.5).

Note that, after each new function evaluation, the kriging parameters adjust only slightly, and thus the previously-converged values of these parameters form an excellent initial guess for this gradient-based optimization algorithm. Note also that, while performing this optimization, the determinant of the correlation matrix occasionally approaches machine zero. To avoid the numerical error that taking the log of zero would otherwise induce, a small $[O(10^{-6})]$ term may be added to the diagonal elements of R. By so doing, the kriging predictor does not quite have the value of the sampled data at each sampled point; however, it remains fairly close, and the algorithm is made numerically robust [Booker *et al*, 1999].

[5] To simplify this optimization, p_ℓ may be specified by the user instead of being determined via optimization; this is especially appropriate to do when the number of function evaluations N is relatively small, and thus there is not yet enough data to determine both the θ_ℓ and p_ℓ uniquely. If this approach is followed, $p_\ell = 1$ or 2 are natural choices; the case with $p_\ell = 1$ is referred to as an Ornstein-Uhlenbeck process, whereas the case with $p_\ell = 2$ is infinitely differentiable everywhere.

Using the tuned statistical model to predict the function value at new locations

Once the parameters of the stochastic model have been tuned as descred above, the tuned kriging model facilitates the computationally inexpensive prediction of the function value at any new location $\overline{\mathbf{x}}$. To perform this prediction, consider now the $n+1$ points $\{\mathbf{x}^1,\ldots,\mathbf{x}^n,\overline{\mathbf{x}}\}$, and model the function's value at these $n+1$ points with the vector

$$\overline{\mathbf{f}} = \begin{pmatrix} \mathbf{f} \\ f(\overline{\mathbf{x}}) \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \overline{f} \end{pmatrix},$$

where $\mathbf{f}$ is the $n\times 1$ random vector considered previously and $\overline{f}$ is the random scalar modeling the function at the new point. Analogous statistical assumptions as laid out in (7.13) are again applied, with the correlation matrix now written as

$$\overline{R} = \begin{bmatrix} R & \overline{\mathbf{r}} \\ \overline{\mathbf{r}}^T & 1 \end{bmatrix}, \qquad P_{\overline{\mathbf{f}}} \triangleq \sigma^2 \overline{R}, \tag{7.18}$$

where R is the $n\times n$ correlation matrix considered previously and, consistent with this definition, the vector $\overline{\mathbf{r}}$ is constructed with components

$$\overline{r}_i = r(\mathbf{x}^i,\overline{\mathbf{x}}), \quad \text{where} \quad r(\mathbf{x},\mathbf{y}) \triangleq \prod_{\ell=1}^{n} \exp\Big(-\theta_\ell |x_\ell - y_\ell|^{p_\ell}\Big).$$

Following Jones (2001), note by the matrix inversion lemma (Fact 4.2) that $\overline{R}^{-1}$ may be written

$$\overline{R}^{-1} = \begin{bmatrix} R & \overline{\mathbf{r}} \\ \overline{\mathbf{r}}^T & 1 \end{bmatrix}^{-1} = \begin{bmatrix} R^{-1} + R^{-1}\overline{\mathbf{r}}(1-\overline{\mathbf{r}}^T R^{-1}\overline{\mathbf{r}})^{-1}\overline{\mathbf{r}}^T R^{-1} & -R^{-1}\overline{\mathbf{r}}(1-\overline{\mathbf{r}}^T R^{-1}\overline{\mathbf{r}})^{-1} \\ -(1-\overline{\mathbf{r}}^T R^{-1}\overline{\mathbf{r}})^{-1}\overline{\mathbf{r}}^T R^{-1} & (1-\overline{\mathbf{r}}^T R^{-1}\overline{\mathbf{r}})^{-1} \end{bmatrix}. \tag{7.19}$$

Keeping the paramter values σ^2, θ_ℓ, p_ℓ, and μ as tuned previously, we now examine the variation of the PDF in the remaining unknown random variable, $\overline{f}$. Substituting (7.18) and (7.19) into a PDF of the form (7.13a), we may write

$$\begin{aligned} p_{\overline{\mathbf{f}}}(\overline{\mathbf{f}}') &= C_1 \cdot \exp \frac{-(\overline{\mathbf{f}}'-\mu\mathbf{1})^T \overline{R}^{-1}(\overline{\mathbf{f}}'-\mu\mathbf{1})}{2\sigma^2} = C_1 \cdot \exp \frac{-\begin{bmatrix}\mathbf{f}'-\mu\mathbf{1}\\ \overline{f}'-\mu\end{bmatrix}^T \overline{R}^{-1}\begin{bmatrix}\mathbf{f}'-\mu\mathbf{1}\\ \overline{f}'-\mu\end{bmatrix}}{2\sigma^2} = \ldots \\ &= C_2 \cdot \exp \frac{-[\overline{f}'-\hat{f}]^T[\overline{f}'-\hat{f}]}{2s^2}, \end{aligned} \tag{7.20}$$

where, with a minor amount of algebraic rearrangement, the mean and variance of this scalar Gaussian distribution modeling the random scalar $\overline{f}$ work out to be[6]

$$\hat{f}(\overline{\mathbf{x}}) = \mathscr{E}\{f(\overline{\mathbf{x}})\} = \mathscr{E}\{\overline{f}\} = \mu + \mathbf{r}^T R^{-1}(\mathbf{f}^o - \mu\mathbf{1}), \tag{7.21a}$$

$$s^2(\overline{\mathbf{x}}) = \mathscr{E}\{[f(\overline{\mathbf{x}}) - \hat{f}]^2\} = \mathscr{E}\{[\overline{f} - \hat{f}]^2\} = \sigma^2(1 - \mathbf{r}^T R^{-1}\mathbf{r}). \tag{7.21b}$$

Equation (7.21) are the final formulae for the kriging predictor, $\hat{f}(\overline{\mathbf{x}})$, and its associated uncertainty, $s^2(\overline{\mathbf{x}})$.

[6] An alternative interpretation of this process models the constant μ itself as a stochastic variable rather than as a constant. Following this line of reasoning ultimately gives the same formula for the predictor $\hat{f}(\overline{\mathbf{x}})$ as given in (7.21a), and a slightly modified formula for its associated uncertainty,

$$s^2(\overline{\mathbf{x}}) = \sigma^2\Big(1 - \mathbf{r}^T R^{-1}\mathbf{r} + \frac{(1-\mathbf{r}^T R^{-1}\mathbf{r})^2}{\mathbf{1}^T R^{-1}\mathbf{1}}\Big). \tag{7.21b'}$$

Which formula [(7.21b) or (7.21b')] is used in the present model is ultimately a matter of fairly little consequence; we thus prefer the form given in (7.21b) due to its computational simplicity.

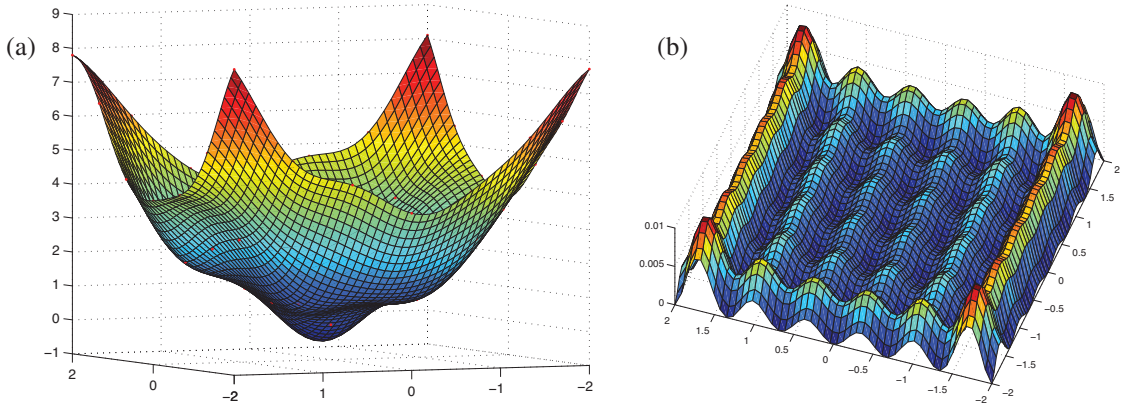


Figure 7.16: (a) The kriging predictor, $\hat{f}(\mathbf{x})$, and (b) its associated uncertainty, $s^2(\mathbf{x})$, for a perturbed quadratic bowl sampled on a square grid of 7×7 points.

When applied numerically to a representative test problem, the kriging predictor function, which we denote $\hat{f}(\overline{\mathbf{x}})$, interpolates [that is, it goes through every observed function value at points $\overline{\mathbf{x}} = \mathbf{x}^1$ to $\overline{\mathbf{x}} = \mathbf{x}^N$], while the uncertainty function, denoted $s^2(\overline{\mathbf{x}})$, is zero at each sampled point, and resembles a Gaussian "bump" between these sampled points, as seen in Figure 7.16. Once the parameters of the statistical model have been determined, as described in §7.5.3, the formula (7.21a)-(7.21b) for the kriging predictor $\hat{f}(\overline{\mathbf{x}})$ and its corresponding uncertainty $s^2(\overline{\mathbf{x}})$ at any test point $\overline{\mathbf{x}}$ is differentiable and computationally inexpensive[7].

7.6 Compression

7.6.1 Dataset compression based on the SVD (a.k.a. POD, PCA,...)†

The SVD provides a rigorous tool for identifying the most energetic recurrent features in a set of several realizations of a random field, $\mathbf{r}(t_j)$ for $j = 1, 2, \ldots, n$, aligned as the columns of a matrix A. This fact has been rediscovered several times throughout the last century, and thus slight variations of this idea go by a hodgepodge of alternative names and associated **three letter acronyms** (**TLA**s), including **proper orthogonal decomposition** (**POD**; the TLA we adopt below), **principal components analysis** (**PCA**), **common factor analysis** (**CFA**), **empirical orthogonal functions** (**EOF**) analysis, **Karhunen-Loève decomposition**, etc.

In particular, when applied to such a matrix A, the SVD provides a set of orthogonal **basis vectors** $\mathbf{u}^i$ for representation of the dataset such that, when this modal representation is truncated at any order, the resulting reduced set of modes is optimal in terms of representating the energy of the given realizations of the random field in the dataset. The decomposition may be formulated as a sequence of variational problems such that

$$\mathbf{u}^1 = \arg\max_{\mathbf{u}} \sum_{j=1}^{n} \frac{|(\mathbf{r}(t_j), \mathbf{u})|^2}{(\mathbf{u}, \mathbf{u})}, \quad \ldots$$

$$\mathbf{u}^i = \arg\max_{\mathbf{u}} \sum_{j=1}^{n} \frac{|(\mathbf{r}(t_j), \mathbf{u})|^2}{(\mathbf{u}, \mathbf{u})} \quad \text{with} \quad \mathbf{u}^i \perp \text{span}\{\mathbf{u}^1, \ldots \mathbf{u}^{i-1}\},$$

where $(\mathbf{r}, \mathbf{u}) = \mathbf{r}^H \mathbf{u}$ is the inner product. Defining $A = A_{m \times n}$ as the matrix containing the j'th realization of

[7] For efficiency, R^{-1} should be saved between function evaluations and reused for every new computation of $\hat{f}$ and s^2.

the random field $\mathbf{r}(t_j)$ as its j'th column,

$$A = \begin{pmatrix} r_1(t_1) & \dots & r_1(t_n) \\ r_2(t_1) & \dots & r_2(t_n) \\ \vdots & \ddots & \vdots \\ r_m(t_1) & \dots & r_m(t_n) \end{pmatrix}, \tag{7.22}$$

these variational problems may be written more simply as

$$\mathbf{u}^i = \arg\max_{\mathbf{u}} \frac{\|A^H\mathbf{u}\|^2}{\|\mathbf{u}\|^2} \text{ with } \mathbf{u}^i \perp \text{span}\{\mathbf{u}^1, \dots \mathbf{u}^{i-1}\}.$$

Note that this is exactly the problem defining the matrix 2-norm, as discussed in §1.3.2. As discussed further there, the vector $\mathbf{u}^1$ solving the first of these variational problems is thus given by the eigenvector corresponding to the maximum eigenvalue of the matrix (AA^H), which is simply the left singular vector $\mathbf{u}^1$ corresponding to the maximum singular value of the matrix A in the singular value decomposition $A = U\Sigma V^H$. As the eigenvectors of (Hermitian) matrix (AA^H) are orthogonal, it follows by the same logic that the vector $\mathbf{u}^2$ solving the second variational problem is given by the eigenvector corresponding to the next largest eigenvalue of the matrix (AA^H) (that is, the second left singular vector of A), etc.

As discussed in §4.4.7, there are a number of ways of computing an SVD. When the number of gridpoints is less than the number of realizations, $m \leq n$, one may use a **direct method** of calculating the leading left singular vectors of A (i.e., the leading columns of U) by simply computing the eigenvectors $\mathbf{u}^k$ corresponding to the largest eigenvalues of the $m \times m$ matrix AA^H, noting that $(AA^H) = U\Lambda U^H$.

When the number of gridpoints is greater than the number of realizations, $m > n$, one may instead use an indirect method (a.k.a. **snapshot method**) of calculating the left singular vectors of A by first computing the leading eigenvectors $\mathbf{v}^k$ of the $n \times n$ matrix (A^HA), noting that $(A^HA) = V\Lambda V^H$, then constructing $\mathbf{u}^k = (1/\sqrt{\lambda_k})A\mathbf{v}^k$, as described in Step 3 of method (i) for constructing the SVD in §4.4.7. Note that the columns of V^H represent the time-varying coefficients that relate how the several realizations of $\mathbf{r}(t)$ are decomposed in terms of the POD modes $\mathbf{u}^k$, that is,

$$\mathbf{r}(t_j) = \sum_k \mathbf{u}^k \sigma_k \overline{v}_{jk}.$$

In the POD implementation given in Algorithm 7.22, we use method (iii) of §4.4.7 (based on successive bidiagonalization of A), which is the preferred way of constructing the SVD.

The **POD eigenvalues** are defined as the squares of the singular values σ_k of the matrix A (that is, they are defined as the diagonal elements λ_k of the matrix Λ). The ratio of two POD eigenvalues reveals the ratio of the energy of the dataset represented by the corresponding two POD modes; thus, comparison of the POD eigenvalues (see, e.g., Figure 7.17c) gives a quantitivave indication of how many of the leading POD modes should be retained in order to be able to represent the vectors in the dataset with a desired degree of precision (in terms of their energy). When the dataset produced is the result of some physical process, there is often a substantial drop after one of the first few POD modes, providing a natural point to truncate the POD basis.

Many datasets are **statistically homogeneous** in one or more spatial coördinates, meaning that the statistics (mean, standard deviation, spatial correlation, etc.) of such datasets are independent of the corresponding spatial coördinate. In such datasets (see, e.g., Figure 7.17b), the POD modes in these coördinate directions come out simply as pairs sinusoidal functions offset from one another by 90°. This makes physical sense, as any other basis would represent variations of the data in certain spatial locations more accurately than in other spatial locations. That is, a basis built from sine/cosine pairs has the unique property that an arbitrary spatial shift of the basis pairs by any distance represents any data vector with an identical degree of fidelity as the original, unshifted basis. On the other hand, in the spatial coördinates for which the system is *not* statistically homogeneous, the POD modes extracted by the SVD are sometimes of certain physical interest.

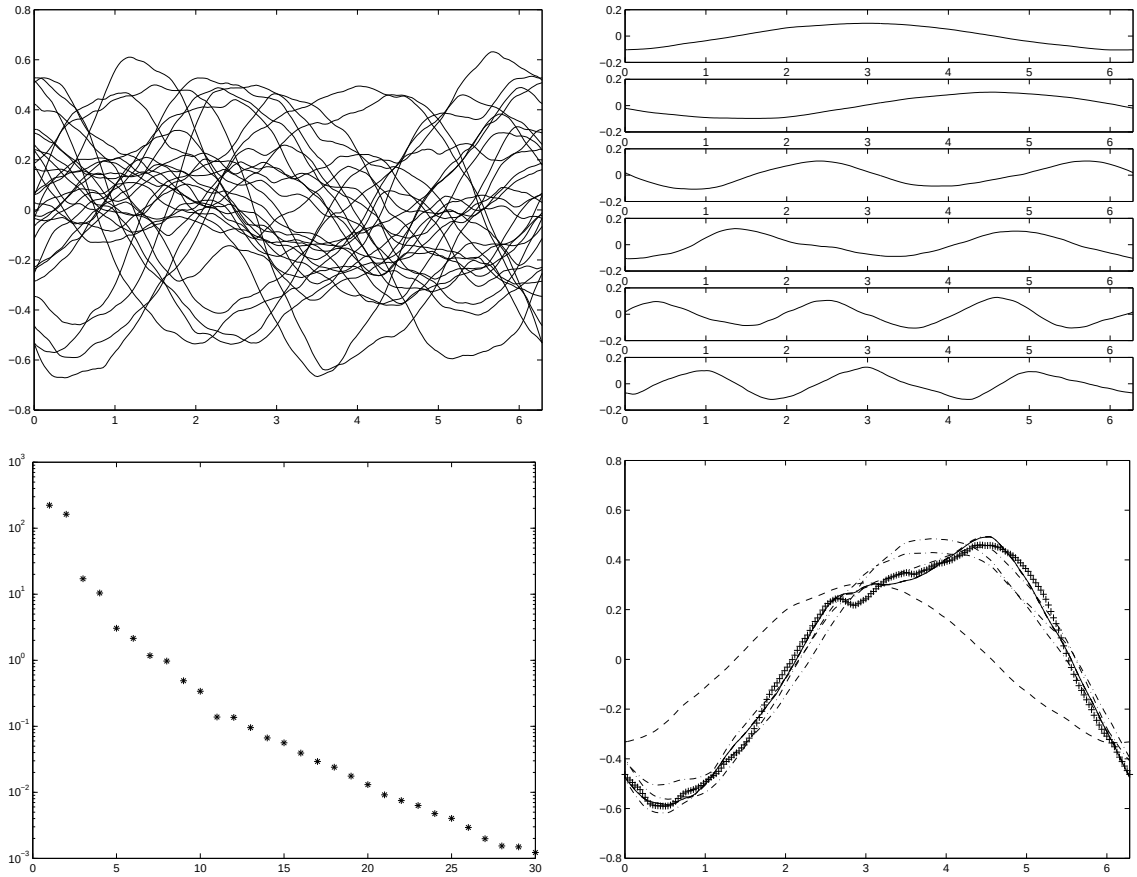


Figure 7.17: Verification of the POD algorithm on a test problem with (a) a training set consisting of 30 randomly-generated vectors, each with 200 elements, random coefficients, $1/k^2$ spectra, and periodic BCs. The resulting POD modes are shown in (b); note that, in this statistically homogeneous problem with periodic BCs, the POD modes emerge as pairs (shifted about 90 degrees from each other) of essentially sinusoidal waves of increasing wavenumber. The POD eigenvalues are shown in (c), illustrating that most of the energy in the dataset is represented by linear combinations of the first few modes. Subfigure (d) illustrates (+) a new random vector with the same spectra and boundary conditions as the training set, and its reconstruction with (dashed) one, (dot-dashed) two, three, four, five, and (solid) six POD modes. Note the improved reconstruction as more POD modes are used.

It is straightforward to introduce weighted inner products $(\mathbf{r},\mathbf{u})_Q = \mathbf{r}^H Q\mathbf{u}$, where $Q > 0$, into the above formulation to account for alternative definitions of the energy function. To accomplish this, simply scale the random vectors in A by $Q^{1/2} > 0$ such that $\mathbf{r}(t_j) \leftarrow Q^{1/2}\mathbf{r}(t_j)$, repeat the above procedure, then scale the resulting POD modes such that $\mathbf{u}^k \leftarrow Q^{-1/2}\mathbf{u}^k$.

In terms of the corresponding spatially continuous functions, the POD modes $u^k(x)$ solve an integral eigenvalue problem whose kernel is the two-point correlation function, as indicated in square brackets below:

$$\int_\Omega \Big[\sum_{j=1}^{n} r(x,t_j) r(x',t_j) \Big] u^k(x')\, d\mathbf{x}' = \lambda_k u^k(x),$$

Algorithm 7.22: Computation of the POD of a dataset.

View

```
% script <a href="matlab:PODtest">PODtest</a>
% Test the POD algorithm on a random set of periodic training data with a given spectra.
L=2*pi; m=200; n1=30; n2=3; n=n1+n2; kmax=100; x=[0:L/(m-1):L]'; p=6; A=zeros(m,n);
for j=1:n, for k=1:kmax       % Initialize n vectors (n1 for training, n2 for testing)
  c=rand-0.5; d=rand-0.5;     % of length m with 1/k^2 spectra and periodic BCs.
  for i=1:m,  A(i,j)=A(i,j)+(1/k^2)*c*sin(k*x(i))+(1/k^2)*d*cos(k*x(i));  end
end, end
[U,S,V]=SVD(A(:,1:n1));       % Calculate the POD (note: the POD is simply an SVD)
figure(1), for j=1:n1, plot(x,A(:,j)), axis([0 L -0.8 0.8]), hold on, end   % Plot data
figure(2), for k=1:p, subplot(p,1,k), plot(x,U(:,k)), axis([0 L -0.2 0.2]), end
figure(3), semilogy(diag(S).^2,'*') % Plot p leading POD modes and compare POD eigenvalues
% Now, reconstruct the n2 new "test" vectors with an increasing number of POD modes.
% Observe the improved representation of these vectors as additional modes are included.
for j=1:n2;
   figure(3+j), clf,  plot(x,A(:,j),'r+'), hold on, r=zeros(m,1); for k=1:p
     r=r+U(:,k)*S(k,k)*V(j,k); if k==1, plot(x,r,'k--'); else, plot(x,r,'-.'), end
   end, plot(x,r,'k-'), axis([0 L -0.8 0.8])
end
% end script PODtest
```

whereas the vectors v^k in the spatially continuous setting solve a (regular) eigenvalue problem given by

$$\sum_{j=1}^{n}\left[\int_\Omega r(x,t_i)r(x,t_j)\,dx\right]v_{jk}=\lambda_k v_{ik}\quad\text{with}\quad u^k(x)=(1/\sqrt{\lambda_k})\sum_{j=1}^{n}r(x,t_j)v_{jk}.$$

These three equations are akin to the relations $(AA^H)\mathbf{u}^k=\lambda_k\mathbf{u}^k$, $(A^HA)\mathbf{v}^k=\lambda_k\mathbf{v}^k$, and $\mathbf{u}^k=(1/\sqrt{\lambda_k})A\mathbf{v}^k$ in the spatially-discrete setting.

Updating a previously-computed POD

Now consider the situation in which $\underline{n}$ data vectors are obtained and a POD is calculated, then $\overline{n}$ additional data vectors are obtained. Instead of starting from scratch and calculating the POD of the entire set of $n=\underline{n}+\overline{n}$ vectors, one may instead seek to leverage the previously computed POD when updating the POD based on the new information.

In this discussion, we will denote by $A_{m\times n}=\begin{bmatrix}\underline{A}_{m\times\underline{n}} & \overline{A}_{m\times\overline{n}}\end{bmatrix}$ the data matrix of the form given in (7.22) based on all of the available information, with $\underline{A}$ containing the old data and $\overline{A}$ containing the new data.

Following the direct approach discussed above, assuming we have already computed the eigen decomposition $(\underline{A}\underline{A}^H)=\underline{U}\underline{\Lambda}\underline{U}^H$, we have

$$AA^H=\begin{bmatrix}\underline{A} & \overline{A}\end{bmatrix}\begin{bmatrix}\underline{A} & \overline{A}\end{bmatrix}^H=\underline{A}\underline{A}^H+\overline{A}\overline{A}^H=\underline{U}\underline{\Lambda}\underline{U}^H+\overline{A}\overline{A}^H=\underline{U}(\underbrace{\underline{\Lambda}+\underline{U}^H\overline{A}\overline{A}^H\underline{U}}_{Y})\underline{U}^H.$$

Though both matrices are full, the problem of computing the eigen decomposition of the Hermitian matrix $Y=\underline{\Lambda}+\underline{U}^H\overline{A}\overline{A}^H\underline{U}$ is significantly easier (will converge in fewer QR iterations) than the problem of computing the eigen decomposition of the matrix (AA^H), as the matrix Y is already strongly dominated by its diagonal components. Once the eigen decomposition $Y=S\Lambda S^H$ is determined, the eigen decomposition sought may readily be determined, as

$$AA^H=(\underline{U}S)\Lambda(\underline{U}S)^H=U\Lambda U^H.$$

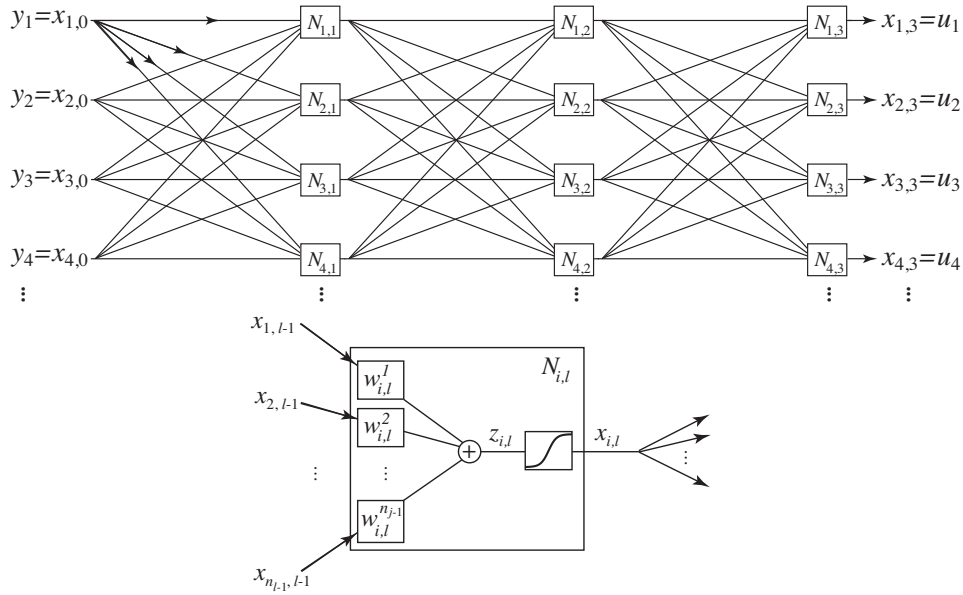


Figure 7.18: Structure and notation of a neural network with $h = 2$ hidden layers: (top) overall network; (bottom) individual node.

Following the snapshot approach discussed above, assuming we have already computed the eigen decomposition $(\underline{A}^H\underline{A}) = \underline{V}\Lambda\underline{V}^H$, we have

$$A^HA = \begin{bmatrix}\underline{A} & \overline{A}\end{bmatrix}^H\begin{bmatrix}\underline{A} & \overline{A}\end{bmatrix} = \begin{bmatrix}\underline{A}^H\underline{A} & \underline{A}^H\overline{A}\\ \overline{A}^H\underline{A} & \overline{A}^H\overline{A}\end{bmatrix} = \begin{bmatrix}\underline{V} & 0\\ 0 & I\end{bmatrix}\underbrace{\begin{bmatrix}\Lambda & \underline{V}^H\underline{A}^H\overline{A}\\ \overline{A}^H\underline{A}\underline{V} & \overline{A}^H\overline{A}\end{bmatrix}}_{Z}\begin{bmatrix}\underline{V}^H & 0\\ 0 & I\end{bmatrix}.$$

Though neither matrix has an easily leveraged banded sparsity structure, the problem of computing the eigen decomposition of the Hermitian matrix Z is easier (will converge in fewer QR iterations) than the problem of computing the eigen decomposition of the matrix (A^HA), as the matrix Z is already strongly dominated by its diagonal components. Once the eigen decomposition $Z = S\Lambda S^H$ is determined, the eigen decomposition sought may readily be determined, as

$$A^HA = \begin{bmatrix}\underline{V} & 0\\ 0 & I\end{bmatrix}S\Lambda S^H\begin{bmatrix}\underline{V}^H & 0\\ 0 & I\end{bmatrix} = V\Lambda V^H.$$

7.6.2 Neural networks

A broadly-used technique to encode otherwise hidden patterns in a given set of input/output data is the optimization, or **training**, of the multiplicative factors in a nonlinear interconnected graph called a **neural network**, as depicted in Figure 7.18. The structure of this graph is modeled loosely after the interconnection of neurons in the human brain, and thus the training of such a network is inspired by the biological process of imprinting memories. Due to their nonlinear structure and their possibly large number of adjustable parameters, neural networks are capable of encoding a myriad of subtle relationships that might otherwise be difficult to model mathematically. Once successfully trained, a neural network can frequently recognize similar patterns in input data it has not yet been exposed to; however, as with undisciplined human brains, it is difficult if not impossible to extract the precise information that the network has supposedly learned.

In addition to the **input layer** and **output layer** ($l=0$ and $l=3$ respectively in Figure 7.18a), a neural network consists of a one or more **hidden layers** ($l=1$ and $l=2$ in Figure 7.18a). The number of nodes in each of these layers, n_l, is not necessarily the same, and the number of nodes in each hidden layer is typically larger than both the number of inputs y_i and the number of outputs u_i. In a fully connected neural network, each node $N_{i,l}$ computes a weighted sum, $z_{i,l}$, of each output of the previous layer, $x_{j,l-1}$, and runs it through a hyperbolic tangent saturation function to compute $x_{i,l}$ (see Figure 7.18); that is,

$$\text{at node } N_{i,l}\text{:}\quad x_{i,l}=\tanh(z_{i,l})\quad\text{where}\quad z_{i,l}=\sum_{j=1}^{n_{l-1}} w_{i,l}^{j}\,x_{j,l-1}. \tag{7.23}$$

To train a neural network, a set of **training records** $\{\mathbf{y}^k,\mathbf{v}^k\}$ for $k=1,\ldots,N$ is obtained which well represents the patterns to be encoded. The weights in the network are initialized as small random numbers (to break symmetries), and adjusted iteratively to minimize the misfit between the output $\mathbf{u}^k$ of the network and the corresponding output $\mathbf{v}^k$ in the training set for each set of inputs $\mathbf{y}^k$. This misfit may be written as

$$J=\frac{1}{2}\sum_{k=1}^{N}\|\mathbf{u}^k(\mathbf{y}^k)-\mathbf{v}^k\|^2=\sum_{k=1}^{N}J_k\quad\text{where}\quad J_k=\frac{1}{2}\|\mathbf{u}^k(\mathbf{y}^k)-\mathbf{v}^k\|^2,$$

and is to be minimized with respect to the weights $w_j^{i,l}$. The simplest training algorithm is to step repeatedly through the training records (often, in random order to reduce biases that might otherwise develop), adjusting the weights a small amount at each step in that direction that decreases J_k; that is,

$$w_{i,l}^{j}\leftarrow w_{i,l}^{j}-\rho\,g_{i,l}^{j,k}\quad\text{where}\quad g_{i,l}^{j,k}=\frac{\partial J_k}{\partial w_{i,l}^{j}}.$$

The determination of the gradient direction $g_{i,l}^{j,k}$ is referred to as the **backpropogation** of the error J_k, and the small multiplicative factor ρ which controls the stepsize in the above update is referred to as the **training rate** or **descent parameter**. The backpropogation algorithm, which amounts to nothing more than successive application of the chain rule for differentiation, is in fact quite simple: if there are h hidden layers in the network, then, denoting $x_{i,l}^k$ as the output of node $N_{i,l}$ of the network when the input to the network is $\mathbf{y}^k$, the gradient $g_{i,l}^{j,k}$ may be determined as follows

for $i=1$ to n_{h+1}, $\dfrac{\partial J_k}{\partial x_{i,h+1}^k}=u_i^k-v_i^k$, end [note that $x_{i,h+1}^k=u_i^k$]

for $l=h+1$ downto 1

  if $l>1$, for $j=1$ to n_{l-1}, $\dfrac{\partial J_k}{\partial x_{j,l-1}^k}=0$, end, end

  for $i=1$ to n_l, $\dfrac{\partial J_k}{\partial z_{i,l}^k}=\dfrac{\partial J_k}{\partial x_{i,l}^k}\operatorname{sech}^2\!\left(z_{i,l}^k\right)$, end [see (7.23) and (B.58)]

    for $j=1$ to n_{l-1}

      $g_{i,l}^{j,k}\triangleq\dfrac{\partial J_k}{\partial w_{i,l}^{j}}=\dfrac{\partial J_k}{\partial z_{i,l}^k}x_{j,l-1}^k$ [this is the gradient info sought]

      if $l>1$, for $j=1$ to n_{l-1}, $\dfrac{\partial J_k}{\partial x_{j,l-1}^k}\leftarrow\dfrac{\partial J_k}{\partial x_{j,l-1}^k}+\dfrac{\partial J_k}{\partial z_{i,l}^k}w_{i,l}^{j}$, end, end,

    end

end

Algorithm 7.23: Efficient implement of a neural network, including a) forward propagation of a single set of inputs through the network, b) back propagation of the output error for a single data record, c) cost computation for an entire set of data records, d) simple fixed-coefficient sequential training to minimize the cost of a set of training records, and e) a test code for a random set of training data.

View

```
function [x]=NN_ForwardPropagate(y,w,h,n)
% Given the input y and the weights w of a neural network with h hidden layers and n(k)
% nodes per layer, compute the state x of the entire network, including the output x{h+2}.
x{1}=y; for k=2:h+2, for i=1:n(k), x{k}(i,1)=tanh(w{k-1}(i,:)*x{k-1}(:)); end, end
end % function NN_ForwardPropagate
```

View

```
function [g]=NN_BackPropagate(e,x,w,h,n)
% Compute the gradient g with respect to the weights w based on the output error e=u-v in
% a neural network with state x (computed using NN_ForwardPropagate).
x{h+2}(:)=e; for k=h+1:-1:1
  x{k+1}(:)=x{k+1}(:).*(sech(w{k}(:,:)*x{k}(:))).^2;
  for i=1:n(k), g{k}(:,i)=x{k+1}(:)*x{k}(i); end          % Compute g{k} = d J_k / d w
  if k>1, for i=1:n(k), x{k}(i,1)=(x{k+1}(:))'*(w{k}(:,i)); end, end
end
end % function NN_BackPropagate
```

View

```
function [J]=NN_ComputeCost(y,u,N,w,h,n)
% Compute the mean-square error over the training set used to train a neural network.
J=0; for d=1:N;
  x=NN_ForwardPropagate(y(:,d),w,h,n); J=J+(0.5/N)*norm(x{h+2}-u(:,d));
end
end % function NN_ComputeCost
```

View

```
function [w]=NN_SequentialTrain(alpha,max_iters,y,u,N,w,h,n)
% Cycle through each training record k one at a time, and perform a fixed-coefficient step
% in the downhill direction a small amount at each iteration based on g{k} = d J_k / d w.
Jsave(1)=NN_ComputeCost(y,u,N,w,h,n);
for iter=1:max_iters
  for d=1:N
    x=NN_ForwardPropagate(y(:,d),w,h,n); g=NN_BackPropagate(x{h+2}(:)-u(:,d),x,w,h,n);
    for k=1:h+1, w{k}=w{k}-alpha*g{k}; end
  end
  Jsave(iter+1)=NN_ComputeCost(y,u,N,w,h,n); plot(Jsave); pause(0.001)
end
end % function NN_SequentialTrain
```

View

```
% script <a href="matlab:NN_Test">NN_Test</a>
% Test the convergence of the neural network algorithm based on some random training data.
clear, in=60; out=1; h=2;
n(1)=in; n(2)=100; n(3)=20; n(4)=out; for k=1:h+1; w{k}=10^(-1)*randn(n(k+1),n(k)); end
N=200; y=rand(in,N); u=rand(out,N); [w]=NN_SequentialTrain(0.1,99,y,u,N,w,h,n);
% end script NN_Test
```

A representative application of neural networks to a timeless problem that largely defies accurate first-principles mathematical modeling (predicting human affections) is considered in the example below.

Example 7.1 Renaissance Dating A new internet matchmaker venture, Renaissance Dating, aspires to serve the vivacious community of romantically-inclined individuals who frequent renaissance fairs. A questionnaire with 30 appropriately revealing questions was first given to a test group of single male and female volunteers (1000 of each) at a series of recent renaissance fairs. Each of these volunteers was then introduced to several of the other volunteers of the appropriate gender, age, and physical characteristics (per their stated orientations), and asked to rank them in terms of preference as a mate. This data was then collected in order to initialize a neural network for the agency to identify good matches within this distinct demographic.

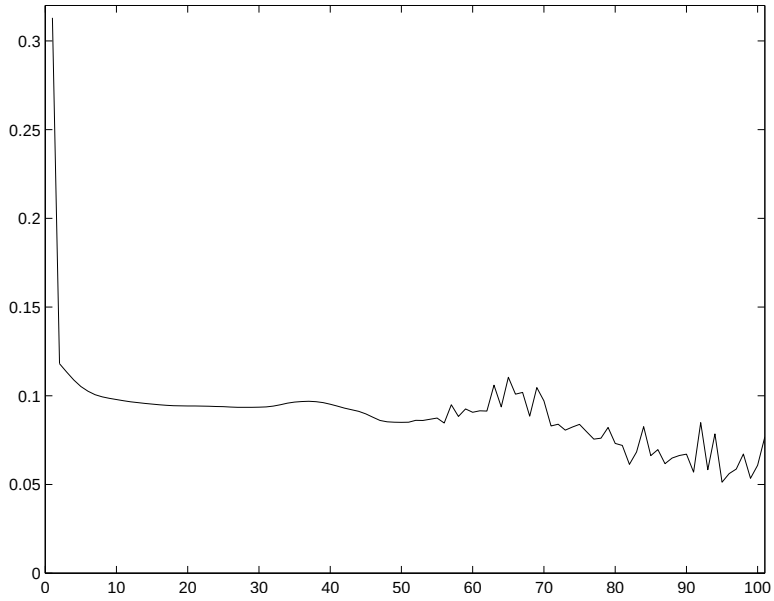


Figure 7.19: Typical cost reduction when training a neural network using simple fixed-coefficient sequential training; note that error does not reduce monotonically, and convergence is not guaranteed.

At one level, a sorting algorithm, such as one of those proposed in §7.1, may be used initially to match people of similar interests. At a deeper level, however, a neural network is valuable to identify which sets of questions tend to be most significant, as well as unexpected correlations within the data (for instance, people who like to wear chainmail might, statistically, have a mutual affinity for people who like to wear feather hats). Some such correlations might be subtle in nature and in a sense nonconvex; for instance, those with gourmet tastes might prefer partners who either really like to cook or really detest cooking but who also have gourmet tastes, but not prefer partners who are more ambivalent about both cooking and eating. A neural network is often able to distinguish such patterns if they are present in the data.

In order to process the data, the responses to the answers on the questionnaire of all subjects are normalized between 0 and 1, and the rankings that the various pairs of potential mates made of each other are also normalized between 0 and 1 and multiplied by each other to give a single pairwise ranking characterizing mutual affinity for each potential pairing. A neural network is formed which takes 60 inputs (the responses to the 30 questions by both individuals) and one output. The network is then trained using the backpropagation algorithm applied to the available training records in order to "learn", as much as possible, the conditions leading to mutual affinity as evident in the training set, with the goal of someday being able to match appropriately different individuals (not in the training set) from the same demographic. Alas, at the time this text was printed, our fledgling matchmaking agency did not have actual field data from romantically-inclined individuals in this target demographic; a representative skeleton implementation of the neural network training algorithm is thus given in Algorithm 7.23 using synthetically-generated data, which can ultimately be replaced by real data if/when Renaissance Dating incorporates.

A typical cost reduction that can be achieved in such a setting when applying fixed-coefficient sequential training using the backpropagation algorithm described above is illustrated in Figure 7.19; various ad hoc improvements of this fixed-coefficient sequential training algorithm have been proposed, such as reducing the descent parameter as convergence is approached. Note that a common technique used to precipitate convergence of the weights in a neural network, which typically do not converge otherwise, is to **prune** (that is,

cut) those connections in the network that correspond to the smallest weights as the network begins to converge; this can be accomplished simply by fixing the corresponding weights as zero and not updating them further. With some effort, an improved algorithm for optimization of the network over the entire training set, leveraging successive line minimizations to ensure uniform convergence (cf. Figure 7.19), may be developed for neural networks using the derivative-based optimization algorithms developed in §16.

Exercises

Exercise 7.1 Following closely the code `Ciculant.m` in Algorithm 2.10, write a streamlined Matlab function `Ciculant0.m`, and test script `Ciculant0Test.m`, that can be used to solve the tridiagonal ciculant system that arises when setting up the cubic spline interpolant in the case of periodic boundary conditions (see Algorithm 7.14; note that $a_{11} = 0$). Proceed by swapping the first and last rows of the A matrix to make it diagonally dominant, and handle the slightly different pattern of "fill-in" that results as the reduced form of Gaussian elimination proceeds. Modify Algorithm 7.14 in order to use your new solver, and make sure it still works.

Exercise 7.2 Extend bilinear interpolation (Algorithm 7.16) to the n-dimensional case described in §7.4.

Exercise 7.3 Extend the bicubic interpolation code implemented in Algorithms 7.17-7.18 to the 3-dimensional case. On a uniform 3D grid of data points, first use cubic spline interpolation along each of the gridlines to compute $\{f_x, f_y, f_z, f_{xy}, f_{yz}, f_{xz}, f_{xyz}\}$ at each of the gridpoints where the function values f are initially prescribed. Then set up a linear 64×64 problem of the form $A\mathbf{x} = \mathbf{b}$ to solve for the a_{ijk} coefficients in (7.10) to match the values of $\{f, f_x, f_y, f_z, f_{xy}, f_{yz}, f_{xz}, f_{xyz}\}$ at each of the 8 corners of the cell (w.l.o.g, taken to be a unit cube) that contains the new interpolation point. Is the A^{-1} matrix simple in structure (sparse with integer elements) as it was in the bicubic case?

References

Batcher, KE (1968) Sorting Networks and their Applications. *Proc. AFIPS Spring Joint Comput. Conf.*, **32**, 307-314.

Isaaks, EH, & Srivastav, RM (1989) *An Introduction to Applied Geostatistics.* Oxford.

Jones, DR (2001) A Taxonomy of Global Optimization Methods Based on Response Surfaces. *Journal of Global Optimization* **21**, 345-383.

Krige, DG (1951) *A statistical approach to some mine valuations and allied problems at the Witwatersrand.* Master's thesis of the University of Witwatersrand, South Africa.

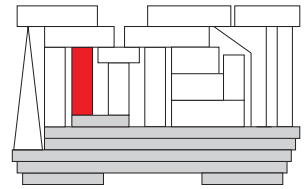
Knuth, DE (1998) *The Art of Computer Programming, Volume3: Searching and Sorting*. Addison-Wesley.

Matheron, G (1963) Principles of geostatistics. *Economic Geology* **58**, 1246-1266.

Rasmussen, CE, & Williams, CKI (2006) *Gaussian processes for machine learning.* MIT Press.

Part II
Simulation

8 Differentiation 237
9 Integration of functions 251
10 Ordinary differential equations 265
11 Partial differential equations 333
12 High performance computing 391
13 A case study in the simulation of turbulence 405



Chapter 8

Differentiation

Contents

8.1 Finite difference (FD) methods . . . 237
8.1.1 Taylor tables . . . 239
8.1.2 Padé approximations . . . 240
8.1.3 Grid stretching functions . . . 241
8.1.4 Alternative derivation of FD formulae . . . 243
8.1.4.1 Differentiating noisy data: an introduction to filtering . . . 243
8.2 Modified wavenumber analysis . . . 245
8.3 The directional derivative and the gradient . . . 247
8.3.1 Direct calculation of the directional derivative . . . 247
8.3.2 FD approximation of the directional derivative . . . 247
8.3.3 Complex-step derivative approximation of the directional derivative . . . 248
8.3.4 Direct calculation of the gradient . . . 249
8.3.5 Numerical approximation of the gradient . . . 249
Exercises . . . 250

8.1 Finite difference (FD) methods

In the simulation of physical systems, one often needs to compute the derivative of a function $f(x)$ which is known only at a discrete set of points $x_0, x_1, \ldots, x_n$, known as a **mesh** or **grid**. Effective formulae for computing such approximations, known as **finite difference** (**FD**) formulae, may be derived by combination of one or more Taylor series expansions. For example, defining $f_j = f(x_j)$, the Taylor series expansion for f at the point x_{j+1} in terms of f and its derivatives at the point x_j is given by

$$f_{j+1} = f_j + (x_{j+1} - x_j) f'_j + \frac{(x_{j+1} - x_j)^2}{2!} f''_j + \frac{(x_{j+1} - x_j)^3}{3!} f'''_j + \ldots$$

Defining $h_{j+1/2} = x_{j+1} - x_j$ and $h_j = (x_{j+1} - x_{j-1})/2$, rearrangement of the above equation leads to

$$f'_j = \frac{f_{j+1} - f_j}{h_{j+1/2}} - \frac{h_{j+1/2}}{2} f''_j + \ldots$$

Note that we indicate a **uniform** mesh by denoting its (constant) grid spacing h without a subscript, and a **nonuniform** (a.k.a. **stretched**) mesh by denoting the grid spacing with a subscript. We will assume the mesh is uniform unless indicated otherwise. For a uniform mesh, we may write the above equation as

$$f'_j = \frac{f_{j+1} - f_j}{h} + O(h),$$

where $O(h)$ denotes the contribution from all terms which have a power of h which is greater than or equal to one. Neglecting these higher-order terms for sufficiently small h, we can approximate the derivative as

$$\left(\frac{\delta f}{\delta x}\right)_j = \frac{f_{j+1} - f_j}{h}, \tag{8.1}$$

which is referred to as the **first-order forward difference approximation of the first derivative**. Note that the notation $\delta f/\delta x$ is used to denote a numerical approximation of the first derivative. The neglected term with the lowest power of h (in this case, $-h f''_j/2$) is referred to as the **leading-order error**. The exponent of h in the leading-order error is the **order of accuracy** of the method. For a sufficiently fine initial grid, if we refine the grid spacing further by a factor of 2, the truncation error of this method is also reduced by approximately a factor of 2, indicating a first-order behavior.

Similarly, by expanding f_{j-1} about the point x_j and rearranging, we obtain

$$\left(\frac{\delta f}{\delta x}\right)_j = \frac{f_j - f_{j-1}}{h}, \tag{8.2}$$

referred to as the **first-order backward difference approximation of the first derivative**. Higher-order schemes can be derived by combining Taylor series of the function f at various points near the point x_j. For example, the **second-order central difference approximation of the first derivative** can be obtained by subtracting two Taylor series:

$$\left.\begin{aligned} f_{j+1} &= f_j + h f'_j + \frac{h^2}{2} f''_j + \frac{h^3}{6} f'''_j + \dots \\ f_{j-1} &= f_j - h f'_j + \frac{h^2}{2} f''_j - \frac{h^3}{6} f'''_j + \dots \end{aligned}\right\} \quad \Rightarrow \quad f_{j+1} - f_{j-1} = 2h f'_j + \frac{h^3}{3} f'''_j + \dots,$$

leading to

$$f'_j = \frac{f_{j+1} - f_{j-1}}{2h} - \frac{h^2}{6} f'''_j + \dots \qquad \Rightarrow \qquad \left(\frac{\delta f}{\delta x}\right)_j = \frac{f_{j+1} - f_{j-1}}{2h}. \tag{8.3}$$

Similar formulae can be derived for approximating second-order derivatives. For example, by adding the above Taylor series expansions instead of subtracting them, we obtain the **second-order central difference approximation of the second derivative** given by

$$f''_j = \frac{f_{j+1} - 2f_j + f_{j-1}}{h^2} - \frac{h^2}{12} f''''_j + \dots \qquad \Rightarrow \qquad \left(\frac{\delta^2 f}{\delta x^2}\right)_j = \frac{f_{j+1} - 2f_j + f_{j-1}}{h^2}. \tag{8.4}$$

It is seen that higher accuracy can be obtained when more points are included in the FD stencil. By appropriate linear combination of four different Taylor Series expansions, the **fourth-order central difference approximation of the first derivative** may be found, which takes the form

$$\left(\frac{\delta f}{\delta x}\right)_j = \frac{f_{j-2} - 8f_{j-1} + 8f_{j+1} - f_{j+2}}{12h}. \tag{8.5}$$

The main difficulty with higher-order formulae occurs near boundaries of the domain, as they require the function values at points outside the domain. Near boundaries, one thus often resorts to lower-order formulae.

8.1.1 Taylor tables

We now present an easy-to-generalize constructive procedure for developing FD formulae on both uniform and stretched grids. Consider first the construction of a FD approximation of the first derivative on a uniform grid using only the function values f_j, f_{j+1}, and f_{j+2}. Writing

$$f'_j - \sum_{k=0}^{2} a_k f_{j+k} = \varepsilon, \tag{8.6}$$

where the a_k are the coefficients of the FD formula sought, we seek the a_k such that the error ε is as large a power of h as possible (and thus will diminish rapidly upon refinement of the grid). It is convenient to organize the Taylor series of all terms on the LHS in the above formula using a **Taylor Table**:

	f_j	f'_j	f''_j	f'''_j
f'_j	0	1	0	0
$-a_0 f_j$	$-a_0$	0	0	0
$-a_1 f_{j+1}$	$-a_1$	$-a_1 h$	$-a_1 \frac{h^2}{2}$	$-a_1 \frac{h^3}{6}$
$-a_2 f_{j+2}$	$-a_2$	$-a_2 (2h)$	$-a_2 \frac{(2h)^2}{2}$	$-a_2 \frac{(2h)^3}{6}$

The leftmost column contains all of the terms on the LHS of (8.6). The elements to the right, when multiplied by the corresponding terms at the top of each column and summed, yield the Taylor series expansion of each of the terms to the left. Summing up these terms, we get the error ε expanded in terms of powers of the grid spacing h. By appropriate choice of the available degrees of freedom $\{a_0, a_1, a_2\}$, we can set several of the coefficients of this polynomial equal to zero, thereby making ε as high a power of h as possible. For small h, this is an effective approach for minimizing this error. In the present case, we have three free coefficients, and thus, leveraging Gaussian elimination, can set the coefficients of the first three terms to zero:

$$\left.\begin{aligned} -a_0 - a_1 - a_2 &= 0 \\ 1 - a_1 h - a_2 (2h) &= 0 \\ -a_1 \frac{h^2}{2} - a_2 \frac{(2h)^2}{2} &= 0 \end{aligned}\right\} \quad \Rightarrow \quad \begin{pmatrix} -1 & -1 & -1 \\ 0 & -1 & -2 \\ 0 & -1/2 & -2 \end{pmatrix} \begin{pmatrix} a_0 h \\ a_1 h \\ a_2 h \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} -3/(2h) \\ 2/h \\ -1/(2h) \end{pmatrix}.$$

The resulting **second-order forward difference approximation of the first derivative** is

$$\left(\frac{\delta f}{\delta x}\right)_j = \frac{-3f_j + 4f_{j+1} - f_{j+2}}{2h}, \tag{8.7}$$

where the leading-order error, which may be determined by multiplying the first non-zero column sum by the term at the top of the corresponding column, is proportional to h^2:

$$\left(-a_1 \frac{h^3}{6} - a_2 \frac{(2h)^3}{6}\right) f'''_j = \frac{h^2}{3} f'''_j.$$

Algorithm 8.1: A simple code to construct and solve a Taylor Table to determine an FD expression.

View
Test

```
function [c]=TaylorTable(x,w)
% Given x locations and w= which derivative, compute the corresponding FD expression.
n=length(x); for i=1:n; for j=1:n; A(i,j)=x(j)^(i-1)/factorial(i-1); end; end
b=zeros(n,1); b(w+1)=1; c=A\b;
end % function TaylorTable
```

Extension of the above approach to achieve the highest order of accuracy possible when other stencils of points are used in (8.6), when numerical approximations of derivatives of other orders are sought, or when the grid is stretched, are all straightforward (see Algorithm 8.1 and Exercises 8.1 and 8.2).

8.1.2 Padé approximations

By including both nearby function evaluations and nearby derivative approximations on the LHS of an expression like (8.6), we can derive a banded system of equations which can easily be solved to determine a numerical approximation of the f'_j. Such approaches are referred to as **Padé approximations**. Again illustrating by example, consider the equation

$$b_{-1} f'_{j-1} + f'_j + b_1 f'_{j+1} - \sum_{k=-1}^{1} a_k f_{j+k} = \varepsilon. \tag{8.8}$$

Leveraging the Taylor series expansions

$$\begin{aligned} f_{j+1} &= f_j + h f'_j + \frac{h^2}{2} f''_j + \frac{h^3}{6} f'''_j + \frac{h^4}{24} f''''_j + \frac{h^5}{120} f'''''_j + \dots \\ f'_{j+1} &= f'_j + h f''_j + \frac{h^2}{2} f'''_j + \frac{h^3}{6} f''''_j + \frac{h^4}{24} f'''''_j + \dots, \end{aligned}$$

the corresponding Taylor table is

	f_j	f'_j	f''_j	f'''_j	f''''_j	f'''''_j
$b_{-1} f'_{j-1}$	0	b_{-1}	$b_{-1}(-h)$	$b_{-1}\frac{(-h)^2}{2}$	$b_{-1}\frac{(-h)^3}{6}$	$b_{-1}\frac{(-h)^4}{24}$
f'_j	0	1	0	0	0	0
$b_1 f'_{j+1}$	0	b_1	$b_1 h$	$b_1\frac{h^2}{2}$	$b_1\frac{h^3}{6}$	$b_1\frac{h^4}{24}$
$-a_{-1} f_{j-1}$	$-a_{-1}$	$-a_{-1}(-h)$	$-a_{-1}\frac{(-h)^2}{2}$	$-a_{-1}\frac{(-h)^3}{6}$	$-a_{-1}\frac{(-h)^4}{24}$	$-a_{-1}\frac{(-h)^5}{120}$
$-a_0 f_j$	$-a_0$	0	0	0	0	0
$-a_1 f_{j+1}$	$-a_1$	$-a_1 h$	$-a_1\frac{h^2}{2}$	$-a_1\frac{h^3}{6}$	$-a_1\frac{h^4}{24}$	$-a_1\frac{h^5}{120}$

We again use the available degrees of freedom $\{b_{-1}, b_1, a_{-1}, a_0, a_1\}$ to obtain the highest possible accuracy in (8.8). Setting the sums of the first five columns equal to zero leads to the linear system

$$\begin{pmatrix} 0 & 0 & -1 & -1 & -1 \\ 1 & 1 & h & 0 & -h \\ -h & h & -h^2/2 & 0 & -h^2/2 \\ h^2/2 & h^2/2 & h^3/6 & 0 & -h^3/6 \\ -h^3/6 & h^3/6 & -h^4/24 & 0 & -h^4/24 \end{pmatrix} \begin{pmatrix} b_{-1} \\ b_1 \\ a_{-1} \\ a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{pmatrix},$$

which is equivalent to

$$\begin{pmatrix} 0 & 0 & -1 & -1 & -1 \\ 1 & 1 & 1 & 0 & -1 \\ -1 & 1 & -1/2 & 0 & -1/2 \\ 1/2 & 1/2 & 1/6 & 0 & -1/6 \\ -1/6 & 1/6 & -1/24 & 0 & -1/24 \end{pmatrix} \begin{pmatrix} b_{-1} \\ b_1 \\ a_{-1} h \\ a_0 h \\ a_1 h \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} b_{-1} \\ b_1 \\ a_{-1} \\ a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} 1/4 \\ 1/4 \\ -3/(4h) \\ 0 \\ 3/(4h) \end{pmatrix}.$$

Thus, the system to be solved to determine the numerical approximation of the derivatives at each grid point has a typical row given by

$$\frac{1}{4}\left(\frac{\delta f}{\delta x}\right)_{j+1}+\left(\frac{\delta f}{\delta x}\right)_{j}+\frac{1}{4}\left(\frac{\delta f}{\delta x}\right)_{j-1}=\frac{3}{4h}\Big(f_{j+1}-f_{j-1}\Big), \tag{8.9}$$

and has a leading-order error of $h^4 f_j''''''/30$, and thus is fourth-order accurate. The above expression is thus referred to as the fourth-order compact Padé approximation of the first derivative. Writing out this equation for all values of j on the interior leads to the tridiagonal, diagonally-dominant system

$$\begin{pmatrix} \ddots & \ddots & \ddots & & \\ & \frac{1}{4} & 1 & \frac{1}{4} & \\ & & \ddots & \ddots & \ddots \end{pmatrix}\begin{pmatrix} \vdots \\ \left(\frac{\delta f}{\delta x}\right)_{j-1} \\ \left(\frac{\delta f}{\delta x}\right)_{j} \\ \left(\frac{\delta f}{\delta x}\right)_{j+1} \\ \vdots \end{pmatrix}=\begin{pmatrix} \vdots \\ \vdots \\ \frac{3}{4h}\Big(f_{j+1}-f_{j-1}\Big) \\ \vdots \\ \vdots \end{pmatrix}.$$

At the endpoints, a different treatment is needed; a central expression (Padé or otherwise) cannot be used at x_0, as x_{-1} is outside of the available grid of data. One often resorts to lower-order expressions at the boundaries in order to close this set of equations. The resulting system can then be solved efficiently for the numerical approximation of the derivative at all of the gridpoints using the Thomas algorithm.

Following a similar approach, the following Padé expression for the second derivative may be determined:

$$0.1\left(\frac{\delta^2 f}{\delta x^2}\right)_{j+1}+\left(\frac{\delta^2 f}{\delta x^2}\right)_{j}+0.1\left(\frac{\delta^2 f}{\delta x^2}\right)_{j-1}=\frac{1.2}{h^2}f_{j+1}-\frac{2.4}{h^2}f_j+\frac{1.2}{h^2}f_{j-1}. \tag{8.10}$$

Again, generalization to other Padé expressions is straightforward (see Exercises 8.3 and 8.4).

8.1.3 Grid stretching functions

It is often advantageous to use a **nonuniform** (a.k.a. **stretched**[1]) grid to cluster gridpoints in regions of the domain where the system under consideration requires higher resolution. If these regions are known before the calculation begins, such a nonuniform mesh is best defined using a smooth **stretching function**[2]. Any monotonic smooth function may be used to generate a nonuniform grid, thus allowing gridpoints to be clustered to varying degrees near one or both boundaries and/or near various critical points on the interior of a computational domain. To demonstrate, we present here three stretching functions appropriate for clustering gridpoints near one or both ends of a computational domain.

Figures 8.1 and 8.2a illustrate the use of the **hyperbolic tangent stretching function**

$$y_j=f(z_j)=\tanh(c\,z_j)/\tanh(c) \quad \text{where} \quad z_j=(2j-n)/n\in[-1,1] \ \text{ for } \ j=0,\ldots,n. \tag{8.11a}$$

In this example, the gridpoint $j=0$ corresponds to the lower boundary $y=-1$, and the gridpoint $j=n$ corresponds to the upper boundary $y=1$. Note that the convenient parameter c may be used with this type of grid stretching to adjust the degree to which the grid is clustered near the boundaries, with smaller values of c resulting in a more uniform mesh, as $\tan(\varepsilon)\sim\varepsilon$ for $\varepsilon\ll 1$.

[1] The generic phrase **stretched grid** is synonymous with the phrase **nonuniform grid**; in it's more narrow use, a nonuniform grid is said to be **stretched** in regions where gridpoints are sparse, and is said to be **clustered** in regions where gridpoints are dense.

[2] This approach is preferred over more ad hoc approaches for clustering gridpoints, since the proper use of a stretching function, as described here, ensures that the grid becomes **locally uniform** (that is, $h_{j-1/2}/h_{j+1/2}\to 1$ and $h_{j-1}/h_j\to 1$ for all j) as the grid is refined, a fact which usually leads to higher accuracy, and sometimes even a higher *order* of accuracy, when approximating derivatives.

Algorithm 8.2: A simple code that may be used to stretch a mesh.

View
Test

```
function x=Stretch1DMesh(x,kind,xmin,xmax,c0,c1)
% Strech a vector x distributed over [0,1] according to (8.11a) if kind='h', (8.11b) if
% kind='p', and (8.11c) if kind='c', then scale to cover the domain [xmin,xmax].
% For 'h' (hyperbolic tangent) stretching, small values of c0>0 make a more uniform mesh.
% For 'p' (polynomial) stretching, increasing c0,c1 increases clustering near xmin,xmax.
% For 'c' (cosine) stretching, we "correct" the orientation, so gridpoints increase in x.
xint=xmax-xmin; switch kind
 case 'h', x=xmin+xint*(tanh(c0*(2*x-1))/tanh(c0)+1)/2;
 case 'c', x=xmin+xint*(1-cos(pi*x))/2;
 case 'p', conc=c0+10*c1; switch conc
     case 00,x=x;                                  case 02,x=x.^3;
     case 10,x=-x.^2+2*x;                          case 12,x=-3*x.^4+4*x.^3;
     case 20,x=x.^3-3*x.^2+3*x;                    case 22,x=6*x.^5-15*x.^4+10*x.^3;
     case 30,x=-x.^4+4*x.^3-6*x.^2+4*x;            case 32,x=-10*x.^6+36*x.^5-45*x.^4+20*x.^3;
     case 01,x=x.^2;                               case 03,x=x.^4;
     case 11,x=-2*x.^3+3*x.^2;                     case 13,x=-4*x.^5+5*x.^4;
     case 21,x=3*x.^4-8*x.^3+6*x.^2;               case 23,x=10*x.^6-24*x.^5+15*x.^4;
     case 31,x=-4*x.^5+15*x.^4-20*x.^3+10*x.^2;    case 33,x=-20*x.^7+70*x.^6-84*x.^5+35*x.^4;
   end, x=xmin+xint*x;
end
end % function Stretch1DMesh
```

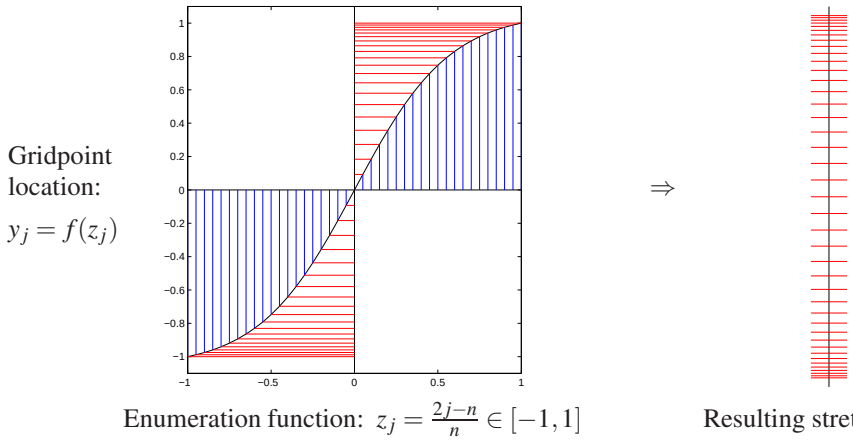


Figure 8.1: The use of the streching function (8.11a) (taking $n = 40$ and $c = 1.75$) to cluster the gridpoints, enumerated from $j = 0$ to $j = n$, near the boundaries of the computational domain $y \in [-1,1]$, .

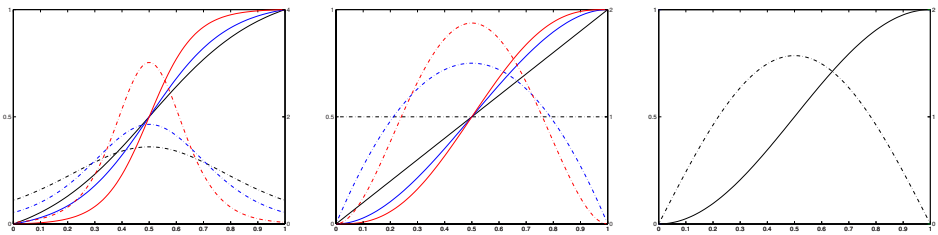


Figure 8.2: Three convenient stretching functions (solid, with scales to left) and their derivatives (dot-dashed, with scales to right): (a) the hyperbolic tangent stretching function (8.11a), plotted for (black) $c = 1.2$, (blue) $c = 1.75$, & (red) $c = 3$, (b) the polynomial stretching function (8.11b), plotted for values of $\{c_0, c_1\}$ given by (black) $\{0,0\}$, (blue) $\{1,1\}$, & (red) $\{2,2\}$, and (c) the cosine stretching function (8.11c).

The stretching function illustrated in Figure 8.2b is the **polynomial stretching function**

$$y_j = f(z_j) = a_n z_j^n + \ldots + a_1 z_j + a_0 \quad \text{where} \quad z_j = j/n \in [0,1] \;\text{ for }\; j = 0,\ldots,n, \tag{8.11b}$$

where $\{a_0, a_1, \ldots, a_n\}$ may be chosen, for example, such that $f(0) = 0$, $f(1) = 1$, the first c_0 derivatives of $f(z)$ are equal to zero at $z = 0$, and the first c_1 derivatives of $f(z)$ are equal to zero at $z = 1$, where $n = 1 + c_0 + c_1$. Larger values of c_0 & c_1 increase the degree to which gridpoints are clustered near each boundary.

The stretching function illustrated in Figure 8.2c is the **cosine stretching function**

$$y_j = f(z_j) = \cos(z_j) \quad \text{where} \quad z_j = \pi\, j/n \in [0,\pi] \;\text{ for }\; j = 0,\ldots,n. \tag{8.11c}$$

This stretching function, as formulated here, has the peculiar property that the gridpoint $j = 0$ corresponds to the *upper* boundary $y = 1$, and the gridpoint $j = n$ corresponds to the *lower* boundary $y = -1$.

All three of these stretching functions are implemented in Algorithm 8.2, and may be shifted, scaled, and/or reflected to cover any interval of interest. In practice, with the flexibility of the tunable parameter c coupled with the fact that the slope of the stretching function doesn't go all the way to zero at the boundaries (so the first gridpoint isn't unnecessarily close to the boundary), the hyperbolic tangent stretching function (8.11a) is well suited for many FD calculations that require increased resolution near the boundaries. The polynomial stretching function (8.11b) is convenient for generating orthogonal grids that may be mapped conformally using the techniques of §B.6.2 [it is used, e.g., in the creation of the locally-orthogonal O-grid around the airfoil indicated in Figure B.10b, as generated by the test code accompanying Algorithm B.9]. The cosine stretching function (8.11c) is inherent to the Chebyshev method for extending spectral methods to nonperiodic domains, as discussed in §5.13.

Stretching may also be applied in two or three directions in 2D or 3D structured grids, as illustrated in Figure 8.3 and implemented in the test code accompanying the 2D grid plotting code given in Algorithm 8.3.

8.1.4 Alternative derivation of FD formulae

Consider now the Lagrange interpolation (see §7.3.2) of the three points $\{x_{i-1}, f_{i-1}\}$, $\{x_i, f_i\}$, and $\{x_{i+1}, f_{i+1}\}$:

$$f(x) = \frac{(x-x_i)(x-x_{i+1})}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} f_{i-1} + \frac{(x-x_{i-1})(x-x_{i+1})}{(x_i-x_{i-1})(x_i-x_{i+1})} f_i + \frac{(x-x_{i-1})(x-x_i)}{(x_{i+1}-x_{i-1})(x_{i+1}-x_i)} f_{i+1}.$$

Differentiating this expression with respect to x and then evaluating at $x = x_i$ gives

$$f'(x_i) = \frac{(-h)}{(-h)(-2h)} f_{i-1} + 0 + \frac{(h)}{(2h)(h)} f_{i+1} = \frac{f_{i+1} - f_{i-1}}{2h},$$

which is the same FD approximation as obtained by the Taylor Table. In fact, following a similar approach (that is, performing a Lagrange interpolation of the available function values, differentiating an appropriate number of times, then evaluating the derivative sought at the point in question) recovers any of the FD approximations that can be developed with the Taylor Table approach. The benefits of the Taylor Table approach, of course, are that it reveals the leading-order error, and that it may be extended to Padé approximations.

8.1.4.1 Differentiating noisy data: an introduction to filtering

If the available data of the function one is trying to differentiate is corrupted by noise, then it is inappropriate to seek a highest-order polynomial fit of this data, as implied by the Taylor Table approach (see §8.1.4). Instead, it is much better to fit a p'th-order polynomial to the nearest n pieces of data, where $p \ll n$, then to evaluate the derivative of this low-order polynomial at the appropriate point. This has the effect of **filtering** out some of the noise in the individual datapoints.

Algorithm 8.3: A simple code that may be used to plot a structured 2D mesh.

View Test

```
function Plot2DMesh(z,fig,II,JJ)
% Plot a structured II by JJ mesh given by real and imaginary parts of z(1:II,1:JJ).
figure(fig); hold on;
for i=1:II, plot(real(z(i,:)),imag(z(i,:)),'b-'); end
for j=1:JJ, plot(real(z(:,j)),imag(z(:,j)),'r-'); end, hold off; axis equal; axis off
end % function Plot2DMesh
```

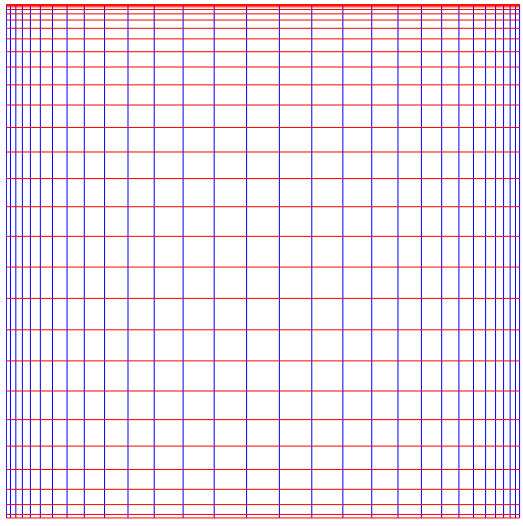


Figure 8.3: A typical 2D stretched mesh, using a hyperbolic tangent stretching function in x, with $c = 1.75$ and $n_x = 40$, and a polynomial stretching function in y, with $\{c_0, c_1\} = \{1, 2\}$ and $n_y = 40$.

To illustrate, suppose one is doing an experiment and has measured the value of $f(t)$ for several discrete values of the time t. To be specific, let's imagine that the present timestep is t_k, with a corresponding noisy measurement f_k, and that the n most recent **tap delays** of the noisy measurements, at times t_{k-1} through t_{k-n}, are given by f_{k-1} through f_{k-n}. At timestep k, we may fit a low-order polynomial

$$f^{(k)}(t) = c_0^{(k)} + c_1^{(k)} t + \ldots + c_p^{(k)} t^p \tag{8.12}$$

to the recent measurements by solving the system

$$\begin{pmatrix} 1 & t_k & t_k^2 & \ldots & t_k^p \\ 1 & t_{k-1} & t_{k-1}^2 & \ldots & t_{k-1}^p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_{k-n} & t_{k-n}^2 & \ldots & t_{k-n}^p \end{pmatrix} \begin{pmatrix} c_0^{(k)} \\ \vdots \\ c_p^{(k)} \end{pmatrix} = \begin{pmatrix} f_k \\ f_{k-1} \\ \vdots \\ f_{k-n} \end{pmatrix} \quad \Leftrightarrow \quad A\mathbf{c}^{(k)} = \mathbf{f}^{(k)}. \tag{8.13}$$

As discussed in §4.7, the least-squares solution to this system is given by $\mathbf{c}^{(k)} = A^+ \mathbf{f}^{(k)}$, where A^+ is given by the Moore-Penrose pseudoïnverse. After solution of (8.13), the low-order polynomial (8.12) and its derivatives may be evaluated to determine a filtered approximation of f and its derivatives. Note that, without loss of generality, we may shift time such that $t_k = 0$; by so doing, the filtered approximation of the $f(t_k)$ is given by $c_0^{(k)}$, the filtered approximation of $(df/dt)_{t=t_k}$ is given by $c_1^{(k)}$, etc. Further, if we assume that the timesteps are uniform, and thus $t_{k-i} = -ih$, then the matrix A does not vary from one timestep to the next. In this case, we may write, for example, $(\delta f/\delta t)_{t=t_k} = CA^+ \mathbf{f}^{(k)}$ where $C = \begin{pmatrix} 0 & 1 & 0 & \ldots \end{pmatrix}$. Note that, in this analysis, typical values of p are between 1 and 4, and typical values of n are between 5 and 20.

The method of filtering noise from an experimental dataset when computing the derivative presented in this section is heuristic. When a model of the dynamic system producing this dataset is available, a much better method is provided by the Kalman filter, as presented in §22.2.

8.2 Modified wavenumber analysis

The order of accuracy is a common indicator of the accuracy of approximate differentiation formulae. It tells, for relatively fine grids, how much further refinement of the grid improves the accuracy. For example, for a sufficiently fine grid, mesh refinement by a factor of 2 improves the accuracy of a second-order FD scheme by a factor of 4, and improves the accuracy of a fourth-order scheme by a factor of 16. Another method for quantifying the accuracy of a FD formula that yields further information is called a **modified wavenumber** analysis. To illustrate this approach, consider the complex exponential function

$$f(x) = e^{\mathrm{i}kx}. \tag{8.14}$$

[Alternatively, we can also do the derivation below with sines and cosines, but complex exponentials make the algebra a bit easier.] The exact derivative of this function is

$$f' = \mathrm{i}k e^{\mathrm{i}kx} = \mathrm{i}k f. \tag{8.15}$$

We now ask how accurately the second order central FD scheme, for example, computes the derivative of f when the x axis is discretized with a uniform mesh,

$$x_j = h \cdot j \qquad \text{where} \qquad j = 0, 1, 2, \ldots n, \qquad \text{and} \qquad h = \frac{L}{n}.$$

We first analyze the second-order central approximation of the first derivative:

$$\left.\frac{\delta f}{\delta x}\right|_j = \frac{f_{j+1} - f_{j-1}}{2h}.$$

Substituting for $f_j = e^{\mathrm{i}kx_j}$, noting that $x_{j+1} = x_j + h$ and $x_{j-1} = x_j - h$ and (B.40), we obtain

$$\left.\frac{\delta f}{\delta x}\right|_j = \frac{e^{\mathrm{i}k(x_j+h)} - e^{\mathrm{i}k(x_j-h)}}{2h} = \frac{e^{\mathrm{i}kh} - e^{-\mathrm{i}kh}}{2h} f_j = \mathrm{i}\frac{\sin(hk)}{h} f_j \triangleq \mathrm{i}k' f_j, \tag{8.16}$$

where

$$k' \triangleq \frac{\sin hk}{h} \qquad \Rightarrow \qquad hk' = \sin(hk).$$

By analogy with (8.15), k' is called the **modified wave number** for this second-order FD scheme. In an analogous manner, one can derive modified wave numbers for any FD formula. A useful measure of accuracy of the FD scheme is provided by comparing the modified wavenumber k', which appears in the numerical approximation of the derivative (8.16) of the test function (8.14), with the actual wavenumber k, which appears in the exact expression for the derivative (8.15). For small wavenumbers, the numerical approximation of the derivative on a discrete grid is fairly accurate ($k' \approx k$), but for larger wavenumbers, the numerical approximation is degraded[3]. As $k \to \pi/h$, which is known as a two-delta wave, the FD approximation of the derivative approaches zero[4].

[3] Compare and contrast this with the derivative computation using spectral methods, in which the derivative of the Fourier representation may be determined *exactly* over a range of wavenumbers from $k = -\pi/h$ to $k = pi/h$, as discussed §5.2.1.

[4] See the related issue at the Nyquist frequency in spectral methods, as discussed §5.5).

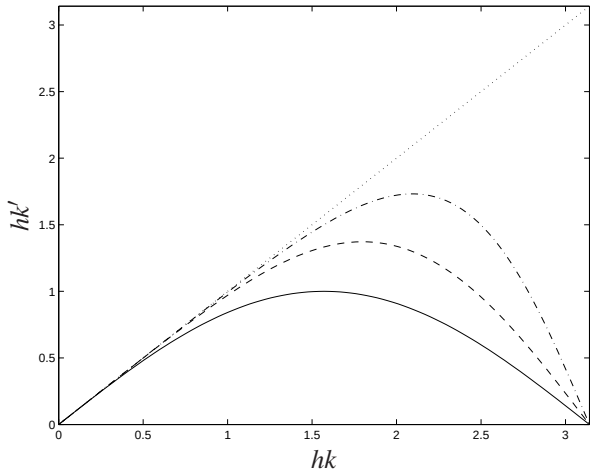


Figure 8.4: Modified wavenumber analysis of the (solid) second-order central, (dashed) fourth-order central, (dot-dashed) fourth-order Padé, and (dotted) spectral approximations of the first derivative.

We next analyze the fourth-order central approximation of the first derivative:

$$\frac{\delta f}{\delta x}\Big|_j = \frac{f_{j-2} - 8f_{j-1} + 8f_{j+1} - f_{j+2}}{12h} = \frac{2}{3h}(f_{j+1} - f_{j-1}) - \frac{1}{12h}(f_{j+2} - f_{j-2}).$$

Inserting (8.14) and manipulating as before, we obtain:

$$\frac{\delta f}{\delta x}\Big|_j = \frac{2}{3h}(e^{\mathrm{i}kh} - e^{-\mathrm{i}kh})f_j - \frac{1}{12}(e^{\mathrm{i}k2h} - e^{-\mathrm{i}k2h})f_j = \mathrm{i}\left[\frac{4}{3h}\sin(hk) - \frac{1}{6h}\sin(2hk)\right]f_j \triangleq \mathrm{i}k' f_j$$
$$\Rightarrow hk' = \frac{4}{3}\sin(hk) - \frac{1}{6}\sin(2hk).$$

Finally, we analyze the fourth-order Padé approximation of the first derivative:

$$\frac{1}{4}\left(\frac{\delta f}{\delta x}\right)_{j+1} + \left(\frac{\delta f}{\delta x}\right)_j + \frac{1}{4}\left(\frac{\delta f}{\delta x}\right)_{j-1} = \frac{3}{4h}\left(f_{j+1} - f_{j-1}\right).$$

Approximating the modified wavenumber at points x_{j+1} and x_{j-1} with their corresponding numerical approximations

$$\left(\frac{\delta f}{\delta x}\right)_{j+1} = \mathrm{i}k' e^{\mathrm{i}kx_{j+1}} = \mathrm{i}k' e^{\mathrm{i}kh} f_j \qquad \text{and} \qquad \left(\frac{\delta f}{\delta x}\right)_{j-1} = \mathrm{i}k' e^{\mathrm{i}kx_{j-1}} = \mathrm{i}k' e^{-\mathrm{i}kh} f_j$$

and inserting (8.14) and manipulating as before, we obtain

$$\mathrm{i}k'\left(\frac{1}{4}e^{\mathrm{i}kh} + 1 + \frac{1}{4}e^{-\mathrm{i}kh}\right)f_j = \frac{3}{4h}(e^{\mathrm{i}kh} - e^{-\mathrm{i}kh})f_j$$
$$\Rightarrow \quad \mathrm{i}k'\left(1 + \frac{1}{2}\cos(hk)\right)f_j = \mathrm{i}\frac{3}{2h}\sin(hk)f_j \quad \Rightarrow \quad hk' = \frac{\frac{3}{2}\sin(hk)}{1 + \frac{1}{2}\cos(hk)}.$$

Plots of hk' versus hk for the three schemes analyzed above are given in Figure 8.4. In such plots, the deviation of the curve from the diagonal line at any given value of hk reveals the error of the derivative at that

(scaled) wavenumber. Note that all of the schemes differentiate low frequencies accurately, but higher-order schemes differentiate higher frequencies more accurately and, for a given order, Padé schemes differentiate higher frequencies more accurately than standard FD schemes. Note also that, in contrast with the FD methods discussed here, spectral methods (see §5) differentiate the frequencies retained in the numerical approximation exactly, as indicated by the diagonal line in Figure 8.4.

8.3 The directional derivative and the gradient

The (scalar) **directional derivative** $d(\mathbf{u},\mathbf{p})$ of some continuous function $J(\mathbf{u})$ in the direction $\mathbf{p}$ is defined by

$$d(\mathbf{u},\mathbf{p}) \triangleq \frac{\partial J(\mathbf{u})}{\partial \mathbf{u}} \cdot \mathbf{p} = \lim_{\varepsilon \to 0^+} \frac{1}{\varepsilon} \left[J(\mathbf{u}+\varepsilon\,\mathbf{p}) - J(\mathbf{u}) \right], \tag{8.17}$$

which is simply the amount $J(\mathbf{u})$ changes when $\mathbf{u}$ is updated in the direction $\mathbf{p}$, scaled by the size of the update, in the limit that the size of the update approaches zero. Note that the calculation of $J(\mathbf{u})$ itself may, in general, involve some extensive numerical calculations, such as the time marching of an ODE (see §10).

The (vector) **gradient** $\mathbf{g}(\mathbf{u})$ of some continuous function $J(\mathbf{u})$ is given in each component i by the directional derivative in the direction of the corresponding Cartesian unit vector $\mathbf{e}^i$; that is,

$$g_i(\mathbf{u}) = d(\mathbf{u},\mathbf{e}^i) = \frac{\partial J(\mathbf{u})}{\partial \mathbf{u}} \cdot \mathbf{e}^i = \lim_{\varepsilon \to 0^+} \frac{1}{\varepsilon} \left[J(\mathbf{u}+\varepsilon\,\mathbf{e}^i) - J(\mathbf{u}) \right] \quad \Rightarrow \quad \mathbf{g}(\mathbf{u}) \triangleq \frac{\partial J(\mathbf{u})}{\partial \mathbf{u}} \triangleq \nabla J(\mathbf{u}). \tag{8.18}$$

8.3.1 Direct calculation of the directional derivative

Consider the Taylor series expansion of the function $J(\mathbf{u}+\varepsilon\,\mathbf{p})$ near $\mathbf{u}$ (that is, for some "small" ε):

$$J(\mathbf{u}+\varepsilon\,\mathbf{p}) = J(\mathbf{u}) + \varepsilon \underbrace{\frac{\partial J(\mathbf{u})}{\partial \mathbf{u}} \cdot \mathbf{p}}_{d(\mathbf{u},\mathbf{p})} + O(\varepsilon^2). \tag{8.19}$$

If the explicit form of $J(\mathbf{u})$ is known, the directional derivative $d(\mathbf{u},\mathbf{p})$ may be computed directly. As an example, if $J(\mathbf{u}) = (1/2)\mathbf{u}^T A\mathbf{u} + \mathbf{b}^T\mathbf{u} + \sin(u_1)$ where $A = A^T$ and all variables are real, then, applying the identity (B.46), we may write

$$\begin{aligned} J(\mathbf{u}+\varepsilon\,\mathbf{p}) &= (1/2)\mathbf{u}^T A\mathbf{u} + \varepsilon\mathbf{u}^T A\mathbf{p} + (\varepsilon^2/2)\mathbf{p}^T A\mathbf{p} + \mathbf{b}^T\mathbf{u} + \varepsilon\mathbf{b}^T\mathbf{p} + \sin(u_1)\cos(\varepsilon p_1) + \cos(u_1)\sin(\varepsilon p_1) \\ &= \underbrace{\left[(1/2)\mathbf{u}^T A\mathbf{u} + \mathbf{b}^T\mathbf{u} + \sin(u_1)\right]}_{J(\mathbf{u})} + \varepsilon \underbrace{\left[A\mathbf{u} + \mathbf{b} + \cos(u_1)\mathbf{e}^1\right]^T\mathbf{p}}_{d(\mathbf{u},\mathbf{p})} + O(\varepsilon^2). \end{aligned}$$

In complicated problems, in which $J(\mathbf{u})$ is related to $\mathbf{u}$ via an involved numerical simulation, it is not always easy to extract the expression for $d(\mathbf{u},\mathbf{p})$, and simple methods to approximate the directional derivative can be useful. The following two sections discuss two such approximation methods.

8.3.2 FD approximation of the directional derivative

There are a variety of ways to calculate numerically the directional derivative $d(\mathbf{u},\mathbf{p})$. The simplest is to consider the Taylor series expansion (8.19), from which a first-order FD formula for the directional derivative is easily obtained:

$$d(\mathbf{u},\mathbf{p}) = \frac{J(\mathbf{u}+\varepsilon\,\mathbf{p}) - J(\mathbf{u})}{\varepsilon} + O(\varepsilon).$$

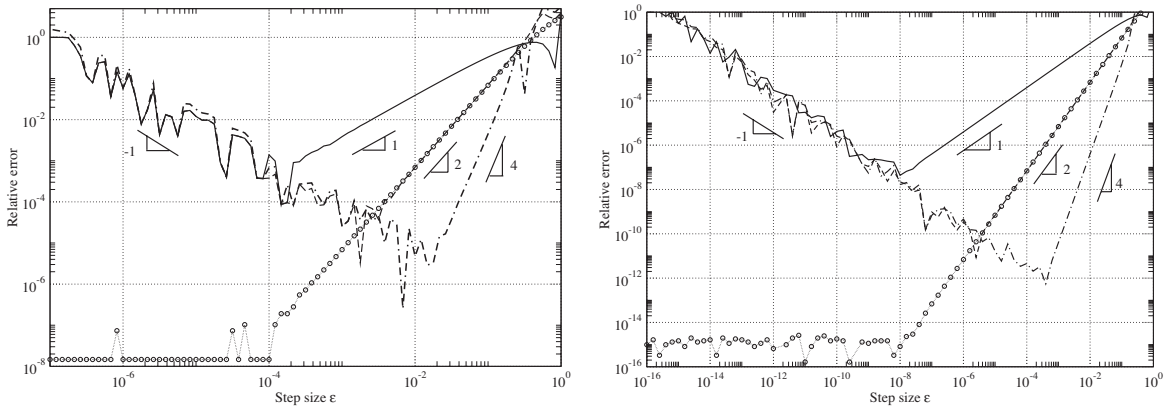


Figure 8.5: Relative error of the directional derivative of a scalar nonlinear test function given by (solid) first-order, (dashed) second-order, and (dot-dashed) fourth-order FD approaches and (circles+dots) the CSD approach using single-precision arithmetic (left) and double-precision arithmetic (right). Note that both plots have essentially the same shape; the errors on the right side of these plots are due to nonlinear effects [that is, ε not being small enough in an expression like (8.17)], where as the errors on the left side of these plots (in the FD approximations only) are due to subtractive cancellation errors.

Combining Taylor series in a manner akin to that leading to (8.3), second-order (and higher) FD formulae for the directional derivative are also easily obtained; for example,

$$d(\mathbf{u},\mathbf{p}) = \frac{J(\mathbf{u}+\varepsilon\,\mathbf{p}) - J(\mathbf{u}-\varepsilon\,\mathbf{p})}{2\varepsilon} + O(\varepsilon^2).$$

The principle drawback with such methods when using a computer with finite-precision arithmetic is the difficulty in finding the most suitable value for the step size ε, since the accuracy of the numerical approximation of $d(\mathbf{u},\mathbf{p})$ is very sensitive to this value. When ε is large, the Taylor-series truncation is not valid, and when it is small, subtractive cancellation errors dominate.

To illustrate, the derivative of a simple scalar nonlinear function $f(u) = e^{-u}/\sqrt{\tan u}$ at $u = 1$, computed using FD approximations with both single- and double-precision arithmetic and compared with the exact solution, is shown in Figure 8.5. It is seen that, for large ε, the error of the FD approximations scales with ε^n, where n is the order of truncation of the higher-order terms of the corresponding FD formulae. For small ε, the error of all three FD formula is $O(1/\varepsilon)$ due to subtractive cancellation errors—in other words, when comparing two numbers which are almost the same using finite-precision arithmetic, the relative round-off error is proportional to the inverse of the difference between the two numbers; thus, if the difference between the two numbers is decreased by an order of magnitude, the relative error with which this difference may be calculated using finite precision arithmetic is increased by an order of magnitude. Unfortunately, in problems for which the exact solution is unknown, it is not possible to make a plot like Figure 8.5, and thus it is not clear how ε should be chosen to minimize the error of the FD approximation of the directional derivative.

8.3.3 Complex-step derivative approximation of the directional derivative

The Complex-Step Derivative (CSD) approximation makes use of complex variables in order to compute the directional derivative of a real function in a more robust fashion than the FD method discussed in §8.3.2. If the complex extension $J(\mathbf{z})$ of a real-valued function $J(\mathbf{u})$ is analytic, it can be expanded with a complex

Algorithm 8.4: Compute the gradient of a multivariable function using an FD or CSD method.

View
Test

```
function [g] = ComputeGrad(x,N,Compute_f,method,e)
% Compute gradient of the the function pointed to by the function handle Compute_f by
% computing the directional derivative one element at a time using a 2nd-order FD approach
% if method="FD", or the complex step derivative approach if method="CSD".
for k=1:N, switch method
  case 'FD'
    xr=x; xr(k)=xr(k)+e; xl=x; xl(k)=xl(k)-e; g(k,1)=(Compute_f(xr)-Compute_f(xl))/(2*e);
  case 'CSD'
    xe=x; xe(k,1)=xe(k,1)+i*eps; g(k,1)=imag(Compute_f(xe))/eps;
end, end
end % function ComputeGrad
```

Taylor series. In particular, the expansion of $J(\mathbf{u}+\mathrm{i}\varepsilon\mathbf{p})$ may be written

$$J(\mathbf{u}+\mathrm{i}\varepsilon\mathbf{p}) = J(\mathbf{u}) + \mathrm{i}\varepsilon \underbrace{\frac{\partial J(\mathbf{u})}{\partial \mathbf{u}}\cdot\mathbf{p}}_{d(\mathbf{u},\mathbf{p})} - \varepsilon^2 E_1 - \mathrm{i}\varepsilon^3 E_2 + O(\varepsilon^4), \tag{8.20}$$

where E_1 and E_2 are real and are related to the higher-order derivatives of J. The directional derivative $d(\mathbf{u},\mathbf{p})$ may thus be found directly by taking the imaginary parts of the terms in this equation and rearranging:

$$d(\mathbf{u},\mathbf{p}) = \frac{\partial J(\mathbf{u})}{\partial \mathbf{u}}\cdot\mathbf{p} = \frac{1}{\varepsilon}\Im\left[J(\mathbf{u}+i\varepsilon\mathbf{p})\right] + \varepsilon^2 E_2 + H.O.T.$$

Note that the error of the approximation scales with ε^2, and *there is no subtractive cancellation error in this approximation*; that is, to leading order, the unperturbed part of the problem is represented in the real components of the variables involved, and the perturbed part of the problem is represented in the imaginary components of variables involved. As ε is reduced towards zero, the error of the approximation diminishes like ε^2 all the way to the numerical roundoff error of the finite-precision arithmetic used by the computer, as depicted in Figure 8.5. As a result, in problems for which the exact solution is unknown, one may simply select ε to be several orders of magnitude smaller than the other numerical values appearing in the problem in order to minimize the error of the CSD approximation of the directional derivative.

8.3.4 Direct calculation of the gradient

As discussed in §8.3.1, if the explicit form of $J(\mathbf{u})$ is known, the directional derivative $d(\mathbf{u},\mathbf{p})$ may be computed directly. The directional derivative comes from the term of $J(\mathbf{u}+\varepsilon\mathbf{p})$ which is linear in ε, and thus is linear in $\mathbf{p}$. We may therefore extract the gradient directly from the explicit form of $d(\mathbf{u},\mathbf{p})$. For example, if $J(\mathbf{u}) = (1/2)\mathbf{u}^T A\mathbf{u} + \mathbf{b}^T\mathbf{u} + \sin(u_1)$ where $A = A^T$ (see §8.3.1) and all variables are real, then

$$d(\mathbf{u},\mathbf{p}) = [A\mathbf{u}+\mathbf{b}+\cos(u_1)\mathbf{e}^1]\cdot\mathbf{p} = \mathbf{g}\cdot\mathbf{p} \quad \Rightarrow \quad \mathbf{g} = A\mathbf{u}+\mathbf{b}+\cos(u_1)\mathbf{e}^1.$$

Again, in complicated problems, it is not always easy to extract the expression for $d(\mathbf{u},\mathbf{p})$, and thus simple methods to approximate the gradient can be useful; the following section shows how.

8.3.5 Numerical approximation of the gradient

Consistent with its definition mentioned previously, each component g_i of the gradient vector $\mathbf{g}$ may be approximated by numerical calculation of the directional derivative in the direction of each of the Cartesian unit

vectors $\mathbf{e}^j$. To accomplish this, one could use either the FD approach described in §8.3.2 (using an appropriate intermediate value of ε, if such a value may be determined, to minimize the error of the approximation, as illustrated in Figure 8.5), or the CSD approach described in §8.3.3 (using a sufficiently small value for ε to make the error of the approximation small, as illustrated in Figure 8.5). Numerical implementations of these strategies is given in Algorithm 8.4.

Exercises

Exercise 8.1 Using a Taylor table (see §8.1.1), determine $\{a_{-1}, a_0, a_1, a_2\}$ in a four-point approximation of the second derivative on a uniform mesh. What is the order of accuracy of this approximation, and what is it's leading-order error?

Exercise 8.2 Repeat Exercise 8.1 on a nonuniform mesh.

Exercise 8.3 Using a Taylor table, determine $\{b_{-1}, b_1, a_{-2}, a_{-1}, a_0, a_1, a_2\}$ in a Padé approximation of the first derivative using five points on the RHS of the Padé formula, assuming a uniform mesh. What is the order of accuracy of this approximation, and what is it's leading-order error?

Exercise 8.4 Repeat Exercise 8.3 on a nonuniform mesh.

Exercise 8.5 Perform modified wavenumber analyses of the schemes developed in Exercises 8.1 and 8.3 (hint: note that the former is an expression for the *second* derivative; what exact expression should be compared against in this case?). Discuss.

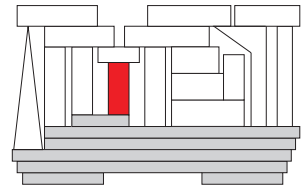
References

Cerviño, LI, & Bewley, TR (2003) On the extension of the complex-step derivative technique to pseudospectral algorithms. *J. Comp. Phys.* **187**, 544-549.

Lele, SK (1992) Compact finite difference schemes with spectral-like resolution. *J. Comp. Phys.* **103**, 16-42.

Leveque, R (2007) *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM.

Pozrikidis, C (1998) *Numerical Computation in Science & Engineering*. Oxford.



Chapter 9

Integration of functions

Contents

9.1 Basic quadrature formulae . . . 251
9.1.1 Formulae based on Lagrange interpolation . . . 251
9.1.2 Extension to several gridpoints . . . 252
9.1.3 Gauss quadrature† . . . 253
9.1.3.1 Gauss-Chebyshev quadrature† . . . 255
9.1.3.2 Gauss-Legendre quadrature† . . . 255
9.1.3.3 Gauss-Laguerre quadrature† . . . 256
9.1.3.4 Gauss-Hermite quadrature† . . . 257
9.2 Accuracy of the basic quadrature formulae . . . 257
9.3 Richardson extrapolation of trapezoidal integration . . . 258
9.4 Adaptive quadrature . . . 260
9.5 Summary: accuracy versus order of accuracy . . . 263
Exercises . . . 264

Differentiation and integration are two essential tools of calculus which we need to solve engineering problems. The previous chapter discussed methods to approximate derivatives numerically; we now turn to the problem of numerical integration. In the setting we discuss in the present chapter, in which we approximate the integral of a given function over a specified domain, this procedure is known as **numerical quadrature**.

9.1 Basic quadrature formulae

9.1.1 Formulae based on Lagrange interpolation

Consider the problem of integrating a function f on the interval $[a,c]$ when the function is evaluated only at a limited number of discrete gridpoints. One approach to approximating the integral of f is to integrate the lowest-order polynomial that passes through a specified number of function evaluations using the formulae of Lagrange interpolation (see §7.3.2).

Algorithm 9.1: The trapezoidal integration algorithm.

View Test

```
function [int,evals] = IntTrapezoidal(f,L,R,n)
% Integrate f(x) from x=L to x=R on n equal subintervals using the trapezoidal rule.
h=(R-L)/n; int=(f(L)+f(R))/2; for i=1:n-1, x=L+h*i; int=int+f(x); end, int=h*int; evals=n+1
% end IntTrapezoidal
```

For example, if the function is evaluated at the midpoint $b = (a+c)/2$, then (defining $h = c-a$) we can integrate a *constant* approximation of the function over the interval $[a,c]$, leading to the **midpoint rule**:

$$\int_a^c f(x)\,dx \approx \int_a^c \Big[f(b)\Big]dx = h\,f(b) \triangleq M(f). \tag{9.1}$$

If the function is evaluated at the two endpoints a and c, we can integrate a *linear* approximation [see (7.2)] of the function over the interval $[a,c]$, leading to the **trapezoidal rule**:

$$\int_a^c f(x)\,dx \approx \int_a^c \Big[\frac{(x-c)}{(a-c)}f(a) + \frac{(x-a)}{(c-a)}f(c)\Big]dx = h\frac{f(a)+f(c)}{2} \triangleq T(f). \tag{9.2}$$

If the function is known at all three points a, b, and c, we can integrate a *quadratic* approximation [see (7.2)] of the function over the interval $[a,c]$, leading to **Simpson's rule**:

$$\begin{aligned}\int_a^c f(x)\,dx &\approx \int_a^c \Big[\frac{(x-b)(x-c)}{(a-b)(a-c)}f(a) + \frac{(x-a)(x-c)}{(b-a)(b-c)}f(b) + \frac{(x-a)(x-b)}{(c-a)(c-b)}f(c)\Big]dx \\ &= \ldots = h\frac{f(a)+4f(b)+f(c)}{6} \triangleq S(f).\end{aligned} \tag{9.3}$$

9.1.2 Extension to several gridpoints

To compute a more accurate approximation of an integral based on several function evaluations over the domain $[L,R]$, one may simply apply one of the formulae of §9.1.1 repeatedly over several smaller subintervals. Following this approach, defining a numerical grid of points $\{x_0, x_1, \ldots, x_n\}$ distributed over the interval $[L,R]$, the intermediate gridpoints $x_{i-1/2} = (x_{i-1}+x_i)/2$, the grid spacing $h_i = (x_i - x_{i-1})$, and the function evaluations $f_i = f(x_i)$, the numerical approximation of the integral of $f(x)$ over the interval $[L,R]$ via the midpoint rule takes the form

$$\int_L^R f(x)\,dx \approx \sum_{i=1}^{n} h_i\,f_{i-1/2}, \tag{9.4}$$

numerical approximation of the integral via the trapezoidal rule takes the form

$$\int_L^R f(x)\,dx \approx \sum_{i=1}^{n} h_i\frac{f_{i-1}+f_i}{2} = \frac{h}{2}\Big[f_0 + f_n + 2\sum_{i=1}^{n-1} f_i\Big], \tag{9.5}$$

and numerical approximation of the integral via Simpson's rule takes the form

$$\int_L^R f(x)\,dx \approx \sum_{i=1}^{n} h_i\frac{f_{i-1}+4f_{i-\frac{1}{2}}+f_i}{6} = \frac{h}{6}\Big[f_0 + f_n + 4\sum_{i=1}^{n} f_{i-\frac{1}{2}} + 2\sum_{i=1}^{n-1} f_i\Big], \tag{9.6}$$

where the rightmost expressions assume a uniform grid in which the grid spacing h is constant. As an example, Algorithm 9.1 illustrates the implementation of (9.5).

9.1.3 Gauss quadrature†

If one knows that a function is well approximated by a polynomial over a given interval $[L,R]$, then, rather than following the simple approach over several subintervals $[a,b]$ as suggested in the previous section, one may instead integrate a high-order polynomial approximation of the function evaluated at several nodal points over the original interval $[L,R]$. This may be done particularly efficiently if these nodal points are chosen carefully.

Following Embree (2009), consider first the integration of the function $f(x)$ over the interval $x \in [L,R]$,

$$I(f;[L,R]) = \int_L^R f(x)\,dx,$$

and approximate this integral by evaluating the function at only two nodal points, $x_0 \in (L,R)$ and $x_1 \in (L,R)$:

$$\tilde{I}(f;[L,R]) = w_0 f(x_0) + w_1 f(x_1) \approx I(f). \tag{9.7a}$$

We seek the location of the two nodal points $\{x_0,x_1\}$ as well as the two weights $\{w_0,w_1\}$ such that this formula integrates exactly as high order a polynomial as possible. Towards this end, take $I(f;[L,R]) = \tilde{I}(f;[L,R])$ for the following four functions:

$$\begin{aligned}
f(x) = 1 \quad &\Rightarrow \quad \int_L^R f(x)\,dx = R - L = w_0 + w_1, \\
f(x) = x \quad &\Rightarrow \quad \int_L^R f(x)\,dx = (R^2 - L^2)/2 = w_0 x_0 + w_1 x_1, \\
f(x) = x^2 \quad &\Rightarrow \quad \int_L^R f(x)\,dx = (R^3 - L^3)/3 = w_0 x_0^2 + w_1 x_1^2, \\
f(x) = x^3 \quad &\Rightarrow \quad \int_L^R f(x)\,dx = (R^4 - L^4)/4 = w_0 x_0^3 + w_1 x_1^3.
\end{aligned}$$

It turns out that these four nonlinear equations may be solved for the four unknowns:

$$x_0 = (R+L)/2 - (R-L)\sqrt{3}/6, \quad x_1 = (R+L)/2 + (R-L)\sqrt{3}/6, \quad w_0 = w_1 = (R-L)/2; \tag{9.7b}$$

thus, (9.7a) with the nodal points x_i and weights w_i given in (9.7b) integrates exactly any cubic polynomial, even though the function is only evaluated at two points. For $[R,L] = [-1,1]$ and $[R,L] = [0,1]$, the resulting two point, fourth-order Gauss-Legendre quadrature formulae, **GLQ4** (see also §9.1.3.2), may be written

$$\tilde{I}(f;[-1,1]) = f\left(-\frac{\sqrt{3}}{3}\right) + f\left(\frac{\sqrt{3}}{3}\right) \tag{9.8a}$$

$$\tilde{I}(f;[0,1]) = \frac{1}{2} f\left(\frac{1}{2} - \frac{\sqrt{3}}{6}\right) + \frac{1}{2} f\left(\frac{1}{2} + \frac{\sqrt{3}}{6}\right). \tag{9.8b}$$

To extend this idea to higher-order polynomials evaluated on more gridpoints, it is convenient to make use of sets of **real orthogonal polynomials** $\phi_k(x)$, such as the Chebyshev polynomials $T_k(x)$ introduced in §5.13. Such sets of polynomials obey two important properties that we will leverage here: a **weighted orthogonality** property on an interval of interest $x \in [L,R]$, which in general may be written

$$\langle \phi_j, \phi_k \rangle = \int_L^R \phi_j(x)\,\phi_k(x)\,w(x)\,dx = 0 \quad \text{if} \quad j \neq k \tag{9.9}$$

for some appropriate weighting function $w(x)$, and a **distinct real root** property summarized as follows:

Fact 9.1 *Let $\{\phi_0(x),\phi_1(x),\ldots,\phi_n(x)\}$ be a set of real orthogonal polynomials, where $\phi_k(x)$ is a k'th-order real polynomial, and where any two of these polynomials obey the weighted orthogonality property on $[L,R]$ given in (9.9) for an appropriately-defined weighting function $w(x)$, where $w(x)>0$ for all $x\in(L,R)$, though $w(x)$ may be zero or infinity at the endpoints. Then $\phi_k(x)$ has k distinct real roots in the open interval (L,R).*

Proof: Let $\{x_1,x_2,\ldots,x_j\}$ be the set all points where the sign of $\phi_k(x)$ changes in the open interval (L,R); note that each of these distinct points is a root of $\phi_k(x)$. By the Fundamental Theorem of Algebra (Fact B.3), $\phi_k(x)$ has exactly k roots, so $j\le k$. Note that j might be less than k if some of the roots of P_k are complex, are outside the interval (L,R), or are not distinct; to prove that, in fact, $j=k$, we define a new real function $s(x)=\prod_{i=1}^{j}(x-x_i)$. Note that $s(x)$ is a j'th-order polynomial that changes sign at exactly the same points as does $\phi_k(x)$. Thus, the function $[s(x)\,\phi_k(x)\,w(x)]$ is either strictly positive or strictly negative for all $x\in[L,R]$ except the points $\{x_1,x_2,\ldots,x_j\}$ and possibly the endpoints. It follows that

$$\langle s,\phi_k\rangle=\int_L^R s(x)\,\phi_k(x)\,w(x)\,dx\neq 0. \tag{9.10}$$

If $j<k$, then $s(x)$ could be written as a linear combination of the polynomials $\{\phi_0(x),\phi_1(x),\ldots,\phi_{k-1}(x)\}$, from which it would follow immediately from (9.9) that $\langle s,\phi_k\rangle=0$, which would be a contradiction with (9.10). Thus, $j=k$; that is, $\phi_k(x)$ has k distinct roots on the open interval (L,R). □

Leveraging a set of orthogonal polynomials $\{\phi_0(x),\phi_1(x),\ldots,\phi_n(x)\}$ and its associated weighted orthogonality and distinct real root properties of over the interval of interest $[L,R]$, it is straightforward to develop a method that computes exactly the integral over $[L,R]$ of any $(2n+1)$'th-order polynomial $f(x)$ by evaluating $f(x)$ at only $n+1$ nodal points [as in the example given at the beginning of §9.1.3, in which we computed exactly the integral of a third-order polynomial by evaluating it at 2 nodal points].

To proceed, use polynomial division to factor $\phi_{n+1}(x)$ out of the $(2n+1)$'th-order polynomial $f(x)$:

$$f(x)=\phi_{n+1}(x)\,q(x)+r(x), \tag{9.11}$$

where $q(x)$ and $r(x)$ are both n'th-order polynomials derived from $f(x)$. Multiplying (9.11) by the weighting function $w(x)$, integrating over the interval $[L,R]$, noting that $q(x)$ can be written as a linear combination of the polynomials $\{\phi_0(x),\phi_1(x),\ldots,\phi_n(x)\}$, and applying (9.9) thus gives

$$I(f)=\int_L^R f(x)\,w(x)\,dx=\int_L^R \phi_{n+1}(x)\,q(x)\,w(x)\,dx+\int_L^R r(x)\,w(x)\,dx=\int_L^R r(x)\,w(x)\,dx. \tag{9.12}$$

We now approximate this integral by evaluating $f(x)$ at $n+1$ nodal points x_i in (L,R):

$$\tilde{I}(f)=\sum_{i=0}^{n} w_i f(x_i)\approx I(f). \tag{9.13a}$$

We seek the location of the nodal points $\{x_0,x_1,\ldots,x_{n_1}\}$ in (L,R) as well as the weights $\{w_0,w_1,\ldots,w_n\}$ such that the formula (9.13a) computes exactly the integral of a $(2n+1)$'th-order polynomial $f(x)$. To accomplish this, we may select the $n+1$ nodal points x_i simply as the $n+1$ distinct real roots of $\phi_{n+1}(x)$ in (L,R),

$$\{x_0,x_1,\ldots,x_n\}=\big(\text{all } x\in(L,R)\mid \phi_{n+1}(x)=0\big), \tag{9.13b}$$

in which case, leveraging (9.11) and the fact that $\phi_{n+1}(x_i)=0$, we may rewrite (9.13a) as

$$\tilde{I}(f)=\sum_{i=0}^{n} w_i f(x_i)=\sum_{i=0}^{n} w_i\,\phi_{n+1}(x_i)\,q(x_i)+\sum_{i=0}^{n} w_i\,r(x_i)=\sum_{i=0}^{n} w_i\,r(x_i).$$

To select the weights w_i, we may rewrite the n'th-order polynomial $r(x)$ in (9.11) using a Lagrange basis as in (7.2), noting that, since $\phi_{n+1}(x_i)=0$, it follows that $r(x_i)=f(x_i)$:

$$r(x)=\sum_{i=0}^{n} f(x_i)L_i(x) \quad \text{where} \quad L_i(x)=\prod_{\substack{j=0\\ j\neq i}}^{n}\frac{x-x_j}{x_i-x_j}.$$

For our $(2n+1)$'th-order polynomial $f(x)$, we may therefore rewrite (9.12) as

$$\int_L^R f(x)\,w(x)\,dx=\int_L^R r(x)\,w(x)\,dx=\int_L^R\left[\sum_{i=0}^{n} f(x_i)L_i(x)\right]w(x)\,dx=\sum_{i=0}^{n} w_i f(x_i)$$

where

$$w_i=\int_L^R L_i(x)\,w(x)\,dx \quad \text{with} \quad L_i(x)=\prod_{\substack{j=0\\ j\neq i}}^{n}\frac{x-x_j}{x_i-x_j}. \tag{9.13c}$$

Thus (9.13a), with the nodal points x_i given in (9.13b) and the weights w_i given in (9.13c), integrates a $(2n+1)$'th-order polynomial exactly.

9.1.3.1 Gauss-Chebyshev quadrature†

Recall the recursive definition in (5.55) of the Chebyshev polynomials $T_k(x)$, which obey the weighted orthogonality property (5.58) on $x\in[-1,1]$ with weighting function $w(x)=1/\sqrt{1-x^2}$. As verified immediatly using (5.56), the roots of $T_{n+1}(x)$ are given by $x_i=\cos[\pi(2i+1)/(2n+2)]$ for $i=0,\ldots,n$. The **Gauss-Chebyshev** quadrature formula on $[-1,1]$ is thus given by (9.13), substituting these values appropriately.

Note that Gauss-Chebyshev quadrature calculates an approximation of $\int_{-1}^{1} f(x)\,w(x)\,dx$, not $\int_{-1}^{1} f(x)\,dx$; if the latter is desired, the factor $1/w(x)$ must first be applied to the function being integrated.

9.1.3.2 Gauss-Legendre quadrature†

We now define recursively the **Legendre polynomials** $P_k(x)$ such that

$$P_0(x)=1, \tag{9.14a}$$
$$P_1(x)=x, \tag{9.14b}$$
$$kP_k(x)=(2k-1)xP_{k-1}(x)-(k-1)P_{k-2}(x) \qquad \text{for } k=2,3,\ldots \tag{9.14c}$$

So defined, the next several Legendre polynomials are:

$$P_2(x)=(3x^2-1)/2, \qquad P_6(x)=(231x^6-315x^4+105x^2-5)/16,$$
$$P_3(x)=(5x^3-3x)/2, \qquad P_7(x)=(429x^7-693x^5+315x^3-35x)/16,$$
$$P_4(x)=(35x^4-30x^2+3)/8, \qquad P_8(x)=(6435x^8-12012x^6+6930x^4-1260x^2+35)/128,$$
$$P_5(x)=(63x^5-70x^3+15x)/8, \qquad P_9(x)=(12155x^9-25740x^7+18018x^5-4620x^3+315x)/128.$$

The Legendre polynomials satisfy the weighted orthogonality property (9.10) on $x\in[-1,1]$ with weighting function $w(x)=1$. The roots of $P_{n+1}(x)$ must in general be found numerically. The **Gauss-Legendre** quadrature formula on $[-1,1]$ is thus given by (9.13), substituting these values appropriately.

Note that Gauss-Legendre quadrature calculates an approximation of $\int_{-1}^{1} f(x)\,dx$ directly. For $n = 1$, Gauss-Legendre quadrature integrates exactly a polynomial of order $(2n+1) = 3$, and reduces immediately to the example given in (9.8a), with the roots of $P_2(x)$ given by $x_0 = -1/\sqrt{3}$ and $x_1 = 1/\sqrt{3}$. Note also that the Gauss-Chebyshev and Gauss-Legendre quadrature nodal points x_i and weights w_i are derived above for the interval $[-1,1]$. Both strategies may easily be transformed to be used over any finite interval $[L,R]$ simply by shifting the nodal points and weights appropriately:

$$\tilde{I}(f;[L,R]) = \sum_{i=0}^{n} \hat{w}_i f(\hat{x}_i) \quad \text{where} \quad \hat{x}_i = L + (R-L)\frac{x_i+1}{2}, \quad \hat{w}_i = \frac{R-L}{2} w_i, \quad \tilde{I}(f;[-1,1]) = \sum_{i=0}^{n} w_i f(x_i),$$

as illustrated in (9.8b) for $[R,L] = [0,1]$. For $n = 2$, Gauss-Legendre quadrature integrates exactly a polynomial of order $(2n+1) = 5$, and for $[R,L] = [-1,1]$ and $[R,L] = [0,1]$, the resulting three point, sixth-order Gauss-Legendre quadrature formulae, **GLQ6**, may be written

$$\tilde{I}(f;[-1,1]) = \frac{5}{9} f\left(-\frac{\sqrt{15}}{5}\right) + \frac{8}{9} f(0) + \frac{5}{9} f\left(\frac{\sqrt{15}}{5}\right), \tag{9.15a}$$

$$\tilde{I}(f;[0,1]) = \frac{5}{18} f\left(\frac{1}{2} - \frac{\sqrt{15}}{10}\right) + \frac{8}{18} f\left(\frac{1}{2}\right) + \frac{5}{18} f\left(\frac{1}{2} + \frac{\sqrt{15}}{10}\right). \tag{9.15b}$$

For the integration of functions that are well approximated by polynomials on $x \in [L,R]$, then, we may chose between Gauss-Chebyshev quadrature and Gauss-Legendre quadrature. Both methods, evaluated at their respective $(n+1)$ gridpoints, give the exact result for a $(2n+1)$'th-order polynomial. They differ in terms of the gridpoints and weights used, with the Chebyshev-based approach clustering gridpoints closer to the endpoints. Thus, for functions that are not quite $(2n+1)$'th-order polynomials, the Chebyshev-based approach is preferred for functions which vary most quickly near the endpoints, and the Legendre-based approach is preferred for functions which vary most quickly closer to the center of the domain.

The Gauss-Chebyshev and Gauss-Legendre quadrature formulae may be also forced to include

- one endpoint (known as **Gauss-Chebyshev-Radau** and **Gauss-Legendre-Radau** quadrature, resp.), or
- both endpoints (known as **Gauss-Chebyshev-Lobatto** and **Gauss-Legendre-Lobatto** quadrature, resp.).

When using a total of $(n+1)$ interpolation points [in the former case, taking the roots of $\phi_n(x)$ in addition to one endpoint, in the latter case, taking the roots of $\phi_{n-1}(x)$ in addition to both endpoints], the former approach reduces by 1 the order of the polynomial that may be integrated exactly, whereas the latter case reduces by 2 the order of the polynomial that may be integrated exactly. These approaches are well justified for functions which are only approximately represented as polynomials, but might vary quickly very near one or both ends of the domain.

9.1.3.3 Gauss-Laguerre quadrature†

For semi-infinite domains, we now define recursively the **Laguerre polynomials** $L_k(x)$ such that

$$L_0(x) = 1, \tag{9.16a}$$

$$L_1(x) = -x + 1, \tag{9.16b}$$

$$kL_k(x) = (2k-1-x)L_{k-1}(x) - (k-1)L_{k-2}(x) \qquad \text{for } k = 2,3,\ldots \tag{9.16c}$$

So defined, the next several Laguerre polynomials are:

$$
\begin{aligned}
L_2(x) &= (x^2 - 4x + 2)/2!, \\
L_3(x) &= (-x^3 + 9x^2 - 18x + 6)/3!, \\
L_4(x) &= (x^4 - 16x^3 + 72x^2 - 96x + 24)/4!, \\
L_5(x) &= (-x^5 + 25x^4 - 200x^3 + 600x^2 - 600x + 120)/5!, \\
L_6(x) &= (x^6 - 36x^5 + 450x^4 - 2400x^3 + 5400x^2 - 4320x + 720)/6!, \\
L_7(x) &= (-x^7 + 49x^6 - 882x^5 + 7350x^4 - 29400x^3 + 52920x^2 - 35280x + 5040)/7!, \\
L_8(x) &= (x^8 - 64x^7 + 1568x^6 - 18816x^5 + 117600x^4 - 376320x^3 + 564480x^2 - 322560x + 40320)/8!.
\end{aligned}
$$

The Laguerre polynomials satisfy the weighted orthogonality property (9.10) on $x \in [0,\infty)$ with weighting function $w(x) = e^{-x}$. The roots of $L_{n+1}(x)$ must in general be found numerically. The **Gauss-Laguerre** quadrature formula on $[0,\infty)$ is thus given by (9.13), substituting these values appropriately.

9.1.3.4 Gauss-Hermite quadrature†

For infinite domains, we now define recursively the **Hermite polynomials** (a.k.a. the **probabilists' Hermite polynomials**) $H_k(x)$ such that

$$
\begin{aligned}
H_0(x) &= 1, && \text{(9.17a)} \\
H_k(x) &= xH_{k-1}(x) - H'_{k-1}(x) \qquad \text{for } k = 1,2,3,\ldots && \text{(9.17b)}
\end{aligned}
$$

So defined, the next several Hermite polynomials are:

$$
\begin{aligned}
H_1(x) &= x, & H_6(x) &= x^6 - 15x^4 + 45x^2 - 15, \\
H_2(x) &= x^2 - 1, & H_7(x) &= x^7 - 21x^5 + 105x^3 - 105x, \\
H_3(x) &= x^3 - 3x, & H_8(x) &= x^8 - 28x^6 + 210x^4 - 420x^2 + 105, \\
H_4(x) &= x^4 - 6x^2 + 3, & H_9(x) &= x^9 - 36x^7 + 378x^5 - 1260x^3 + 945x, \\
H_5(x) &= x^5 - 10x^3 + 15x, & H_{10}(x) &= x^{10} - 45x^8 + 630x^6 - 3150x^4 + 4725x^2 - 945.
\end{aligned}
$$

The Hermite polynomials satisfy the weighted orthogonality property (9.10) on $x \in (-\infty,\infty)$ with weighting function $w(x) = e^{-x^2/2}$. The roots of $H_{n+1}(x)$ must in general be found numerically. The **Gauss-Hermite** quadrature formula on $(-\infty,\infty)$ is thus given by (9.13), substituting these values appropriately.

9.2 Accuracy of the basic quadrature formulae

In order to quantify the accuracy of the three basic quadrature formulae laid out in §9.1.1, we again turn to Taylor series analysis. Replacing $f(x)$ with its Taylor series approximation about b and integrating, we obtain

$$
\begin{aligned}
\int_a^c f(x)\,dx &= \int_a^c \left[f(b) + (x-b) f'(b) + \frac{1}{2}(x-b)^2 f''(b) + \frac{1}{6}(x-b)^3 f'''(b) + \ldots \right] dx \\
&= h f(b) + \frac{1}{2}(x-b)^2 \Big|_a^c f'(b) + \frac{1}{6}(x-b)^3 \Big|_a^c f''(b) + \ldots \qquad \text{(9.18)} \\
&= h f(b) + \frac{h^3}{24} f''(b) + \frac{h^5}{1920} f''''(b) + \ldots
\end{aligned}
$$

Thus, if the integral is approximated by the midpoint rule (9.1), the leading-order error is proportional to h^3, and the approximation of the integral over this single interval is third-order accurate.

Practically speaking, the order of accuracy of a particular integration rule over a single subinterval $[a,c]$ is not of much interest. A more useful measure is the rate of convergence of the integration rule when applied over several gridpoints on a given interval $[L,R]$, as discussed in §9.1.2, as the numerical grid is refined. For example, consider the formula (9.4) on n subintervals ($n+1$ gridpoints). As $h \propto 1/n$ (the width of each subinterval is inversely proportional to the number of subintervals), the error over the *entire* interval $[L,R]$ of the numerical integration is proportional to $nh^3 = h^2$. Thus, for approximations of the integral on a given interval $[L,R]$ as the computational grid is refined, the *midpoint rule is second-order accurate*.

Consider now the Taylor series approximations of $f(a)$ and $f(c)$ about b:

$$f(a) = f(b) + \left(\frac{-h}{2}\right) f'(b) + \frac{1}{2}\left(\frac{-h}{2}\right)^2 f''(b) + \frac{1}{6}\left(\frac{-h}{2}\right)^3 f'''(b) + \dots$$
$$f(c) = f(b) + \left(\frac{h}{2}\right) f'(b) + \frac{1}{2}\left(\frac{h}{2}\right)^2 f''(b) + \frac{1}{6}\left(\frac{h}{2}\right)^3 f'''(b) + \dots$$

Combining these expressions gives

$$\frac{f(a)+f(c)}{2} = f(b) + \frac{1}{8}h^2 f''(b) + \frac{1}{384}h^4 f''''(b) + \dots$$

Solve for $f(b)$ and substituting into (9.18) yields

$$\int_a^c f(x)\,dx = h\frac{f(a)+f(c)}{2} - \frac{h^3}{12}f''(b) - \frac{h^5}{480}f''''(b) + \dots \tag{9.19}$$

As with the midpoint rule, the leading-order error of the trapezoidal rule (9.2) is proportional to h^3, and thus the trapezoidal approximation of the integral over this single subinterval $[a,c]$ is third-order accurate. Again, the most relevant measure is the rate of convergence of the integration rule (9.5) when applied over several gridpoints on a given interval $[L,R]$ as the number of gridpoints is increased; in such a setting, as with the midpoint rule, the *trapezoidal rule is second-order accurate*.

Note from (9.1)-(9.3) that $S(f) = \frac{2}{3}M(f) + \frac{1}{3}T(f)$. Adding $2/3$ times equation (9.18) plus $1/3$ times equation (9.19) gives

$$\int_a^c f(x)\,dx = h\frac{f(a)+4f(b)+f(c)}{6} - \frac{h^5}{2880}f''''(b) + \dots$$

The leading-order error of Simpson's rule (9.3) is therefore proportional to h^5, and thus the approximation of the integral over this single subinterval $[a,c]$ using this rule is fifth-order accurate. Again, the most relevant measure is the rate of convergence of the integration rule (9.6) when applied over several gridpoints on the given interval $[L,R]$ as the number of gridpoints is increased; in such a setting, *Simpson's rule is fourth-order accurate*.

9.3 Richardson extrapolation of trapezoidal integration

In the last paragraph of the previous section, it became evident that keeping track of the leading-order error of a particular numerical formula can be a useful thing to do. In fact, it was shown that Simpson's rule can be constructed simply by determining the specific linear combination of the midpoint rule and the trapezoidal rule for which the leading-order error term vanishes. We now pursue further such a constructive approach,

with a technique of successive refinements known as **Richardson extrapolation**, in order to determine even higher-order approximations of the integral on the interval $[L,R]$. This technique is based on successive linear combinations of trapezoidal approximations of the integral on a series of successively finer grids, and is commonly referred to as **Romberg integration**.

Noting that we may write

$$f''(x) = f''(b) + (x-b)f'''(b) + \frac{1}{2}(x-b)^2 f''''(b) + \ldots, \qquad f''''(x) = f''''(b) + (x-b)f'''''(b) + \ldots,$$

integrating both expressions over $[a,c]$ and rearranging gives

$$hf''(b) = \int_a^c f''(x)\,dx - \frac{h^3}{24}f''''(b) + O(h^5), \qquad hf''''(b) = \int_a^c f''''(x)\,dx + O(h^3).$$

Thus, (9.19) may be written

$$\int_a^c f(x)\,dx = h\frac{f(a)+f(c)}{2} - \frac{h^2}{12}\int_a^c f''(x)\,dx + \frac{h^4}{720}\int_a^c f''''(x)\,dx + \ldots \tag{9.20}$$

The error of the integral of the trapezoidal approximation of the integral (9.5) on the given interval $[L,R]$ may be thus be written

$$I \triangleq \int_L^R f(x)\,dx = \frac{h}{2}\left[f_0 + f_n + 2\sum_{i=1}^{n-1} f_i\right] + c_1 h^2 + c_2 h^4 + c_3 h^6 + c_4 h^8 + \ldots,$$

where

$$c_1 = -\frac{1}{12}\int_L^R f''\,dx, \qquad c_2 = \frac{1}{720}\int_L^R f''''\,dx, \qquad \text{etc.}$$

We will never need to compute the constants c_1, c_2, etc.; it is sufficient to know that they are independent of h. Let us now start with $n_1 = 2$ and $h_1 = (R-L)/n_1$ and iteratively refine the grid. Define the trapezoidal approximation of the integral on a numerical grid with $n = n_j = 2^j$ (e.g., $h = h_j = (R-L)/n_j$) as:

$$I_{j,1} = \frac{h_j}{2}\left[f_0 + f_{n_j} + 2\sum_{i=1}^{n_j-1} f_i\right]. \tag{9.21}$$

We now examine the truncation error (in terms of h_1) as the grid is refined. Note that at the first level we have

$$I_{1,1} = I - c_1 h_1^2 - c_2 h_1^4 - c_3 h_1^6 \ldots,$$

whereas at the second level we have

$$I_{2,1} = I - c_1 h_2^2 - c_2 h_2^4 - c_3 h_2^6 - \ldots = I - c_1\frac{h_1^2}{4} - c_2\frac{h_1^4}{16} - c_3\frac{h_1^6}{64} - \ldots$$

Noting that the constants c_i are the same in the two expansions, we can eliminate the error proportional to h_1^2 by taking a linear combination of $I_{1,1}$ and $I_{2,1}$ to obtain:

$$I_{2,2} = \frac{4I_{2,1} - I_{1,1}}{3} = I + \frac{1}{4}c_2 h_1^4 + \frac{5}{16}c_3 h_1^6 + \ldots$$

Note that this results in Simpson's rule, if you do the appropriate substitutions. Continuing to the third level of grid refinement, the trapezoidal approximation of the integral satisfies

$$I_{3,1} = I - c_1 h_3^2 - c_2 h_3^4 - c_3 h_3^6 - \ldots = I - c_1\frac{h_1^2}{16} - c_2\frac{h_1^4}{256} - c_3\frac{h_1^6}{4096} - \ldots$$

First, eliminate terms proportional to h_1^2 by linear combination with $I_{2,1}$:

$$I_{3,2} = \frac{4I_{3,1} - I_{2,1}}{3} = I + \frac{1}{64} c_2 h_1^4 + \frac{5}{1024} c_3 h_1^6 + \ldots$$

Then, eliminate terms proportional to h_1^4 by linear combination with $I_{2,2}$:

$$I_{3,3} = \frac{16I_{3,2} - I_{2,2}}{15} = I - \frac{1}{64} c_3 h_1^6 - \ldots$$

Note that this results in a refinement of Simpson's rule [that is, integrating a *quartic* approximation of the function over each interval $[a,c]$, following the next steps in the progression started in (9.1), (9.2), (9.3)], if you do the appropriate substitutions. This process may be repeated to provide increasingly higher-order approximations to the integral I. The structure of the refinements is:

Gridpoints	2nd-Order Approximation		4th-Order Correction		6th-Order Correction		8th-Order Correction
$n_1 = 2^1 = 2$	$I_{1,1}$						
		$\searrow$					
$n_2 = 2^2 = 4$	$I_{2,1}$	$\rightarrow$	$I_{2,2}$				
		$\searrow$		$\searrow$			
$n_3 = 2^3 = 8$	$I_{3,1}$	$\rightarrow$	$I_{3,2}$	$\rightarrow$	$I_{3,3}$		
		$\searrow$		$\searrow$		$\searrow$	
$n_4 = 2^4 = 16$	$I_{4,1}$	$\rightarrow$	$I_{4,2}$	$\rightarrow$	$I_{4,3}$	$\rightarrow$	$I_{4,4}$

The general form for the trapezoidal approximations of the integral $I_{j,1}$ (in the first column of the table above) is given in (9.21), and the general form for the correction term $I_{j,k}$ (for $j \geq 2$ and $2 \leq k \leq j$) is defined recursively as:

$$I_{j,k} = \frac{(4)^{(k-1)} I_{j,(k-1)} - I_{(j-1),(k-1)}}{(4)^{(k-1)} - 1} = I_{j,(k-1)} + \frac{I_{j,(k-1)} - I_{(j-1),(k-1)}}{4^{(k-1)} - 1}. \tag{9.22}$$

Efficient implementation of Romberg integration is illustrated in Algorithm 9.2.

9.4 Adaptive quadrature

Often, it is wasteful to use the same grid spacing h everywhere in the interval of integration $[L,R]$. Ideally, one would like to use a fine grid in the regions where the integrand varies quickly and a coarse grid where the integrand varies slowly. As we now show, **adaptive quadrature** methods automatically adjust the grid spacing in just such a manner.

Suppose we seek a numerical approximation $\tilde{I}$ of the integral I such that

$$|\tilde{I} - I| \leq \varepsilon,$$

where ε is the error tolerance provided by the user. The idea of adaptive quadrature is to spread out this error in our approximation of the integral proportionally across the subintervals spanning $[L,R]$. To demonstrate this technique, we will use Simpson's rule as the base method. First, divide the interval $[L,R]$ into several subintervals with the numerical grid $\{x_0, x_1, \ldots, x_n\}$. Evaluating the integral on a particular subinterval $[x_{i-1}, x_i]$ with Simpson's rule yields

$$S_i = \frac{h_i}{6} [f(x_i - h_i) + 4 f(x_i - \frac{h_i}{2}) + f(x_i)].$$

Algorithm 9.2: Romberg integration based on successively refined trapezoidal calculations.

View
Test

```
function [int,evals] = IntRomberg(f,L,R,toplevel)
% Integrate f(x) from x=L to x=R using Romberg integration, thus providing higher
% and higher order of accuracy as the grid is refined.
fi=[];      % note: fi stores the previous evaluations of f(x), so they may be reused.
for level=1:toplevel
  % Approximate the integral with the trapezoidal rule on 2^level subintervals
  n=2^level;  [I(level,1),fi]=IntTrapezoidalRefine(f,L,R,n,fi);
  % Perform several corrections based on I at the previous level.
  for k=2:level, I(level,k)=(4^(k-1)*I(level,k-1)-I(level-1,k-1))/(4^(k-1)-1); end
end
int=I(toplevel,toplevel); evals=n+1
end % function IntRomberg.m
%%%
```

Algorithm 9.3: Adaptive integration based Simpson's method.

View Test

```
function [int,evals] = IntAdaptive(f,a,c,epsilon,evals,fa,fb,fc)
% Integrate f(x) over the interval [a,c] using adaptive integration, taking b=(a+c)/2,
% where (fa,fb,fc) are the evalations at (a,b,c) and epsilon is the desired accuracy.
b=(a+c)/2;  d=(a+b)/2;  e=(b+c)/2;  fd=f(d);  fe=f(e);  evals=evals+2;
S1=(c-a)*(fa+4*fb+fc)/6;      % Coarse and fine approximations of integral.
S2=(c-a)*(fa+4*fd+2*fb+4*fe+fc)/12;
if abs(S2-S1)/15<=epsilon     % If accuracy of S2 on desired integral is acceptable,
    int=(16*S2-S1)/15;        % take result (further refined, as in Romberg);
else                          % otherwise, split the interval into two and refine
    [int1,evals] = IntAdaptive(f,a,b,epsilon/2,evals,fa,fd,fb);  % the estimate on
    [int2,evals] = IntAdaptive(f,b,c,epsilon/2,evals,fb,fe,fc);  % both subintervals.
    int=int1+int2;
end
end % function IntAdaptive
```

Subtracting (9.24) from (9.23), we obtain

$$S_i^{(2)} - S_i = \frac{15}{16} c h_i^5 f''''\left(x_i - \frac{h_i}{2}\right) + \ldots,$$

and substituting into the RHS of (9.24) reveals that

$$I - S_i^{(2)} \approx \frac{1}{15}(S_i^{(2)} - S_i). \tag{9.25}$$

Thus, the error in $S_i^{(2)}$ is, to leading order, $\frac{1}{15}$ of the difference between S_i and $S_i^{(2)}$. Fortunately, this difference can easily be computed. Conservatively taking the absolute value of both sides (assuming the errors on all intervals might turn out to be the same sign), if

$$\frac{1}{15}|S_i^{(2)} - S_i| \leq \frac{h_i}{R-L}\varepsilon,$$

then $S_i^{(2)}$ is sufficiently accurate for the subinterval $[x_{i-1}, x_i]$, and we may move on to the next subinterval. If this condition is not satisfied, the subinterval $[x_{i-1}, x_i]$ is subdivided further. Efficient implementation of the adaptive integration method is given in Algorithm 9.3. Note also one final tweak of the method: if the accuracy of $S_i^{(2)}$ is acceptable on interval i, then the accuracy of $S_i^{(3)} = (16S_i^{(2)} - S_i)/15$ [as motivated by solving (9.25) for I in terms of $S_i^{(2)}$ and S_i] is probably even better. Though we can not estimate the accuracy of the approximation $S_i^{(3)}$ based on known quantities (namely, $S_i^{(2)}$ and S_i), as $S_i^{(3)}$ is more accurate than both $S_i^{(2)}$ and S_i (at least, for sufficiently small h) and we can essentially determine it for free (that is, without any more function evaluations), it doesn't hurt to use it. By so doing, the estimate of the error ε used in the function call is even more conservative than it would be otherwise (that is, the result is likely much more accurate than expected based on the value of ε used in the function call); this at least doesn't hurt.

The essential idea of adaptive quadrature is thus to spread evenly the error of the numerical approximation of the integral (or, at least, an approximation of this error) over the entire interval $[L,R]$ by selective refinements of the numerical grid. As with Romberg integration, knowledge of the truncation error is leveraged to estimate the accuracy of the numerical solution without knowing the exact solution. In Romberg integration, this information is used to obtain successively *higher-order* numerical approximations of the integral. In adaptive integration, on the other hand, this information is used to *refine the grid* in the appropriate regions. In most practical problems, as illustrated in the next section, the latter is the more efficient approach.

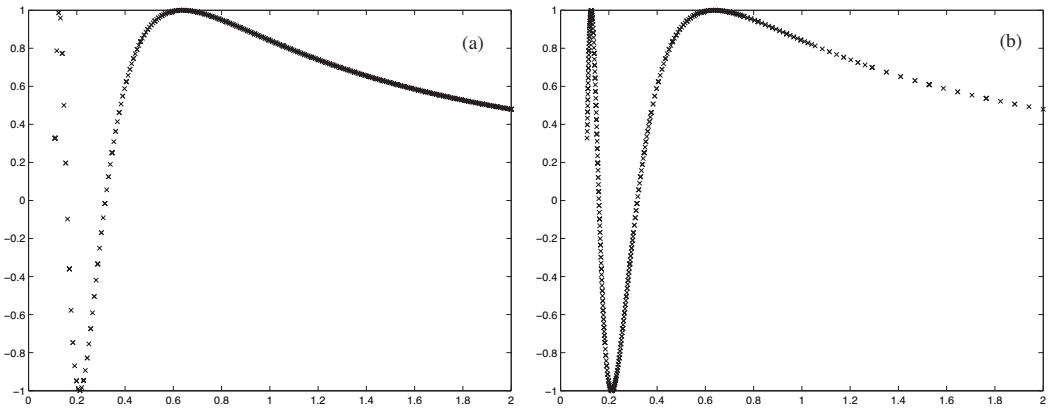


Figure 9.1: Function evaluations $f(x_i)$ calculated (a) by Algorithm 9.1 and Algorithm 9.2, and (b) by Algorithm 9.3, taking $f(x) = \sin(1/x)$ and $[L,R] = [0.1,2]$. Approximately 256 function evaluations are indicated in each case.

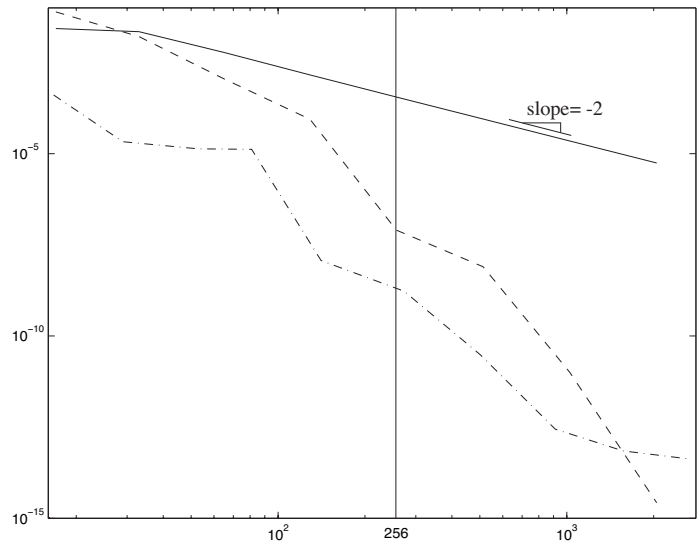


Figure 9.2: Magnitude of the error of the numerical approximation of the integral as a function of the number of function evaluations performed using the (solid) trapezoidal, (dashed) Romberg, and (dot-dashed) adaptive integration methods.

9.5 Summary: accuracy versus order of accuracy

This chapter presented four classes of algorithms for calculating the integral of a function:

- the midpoint, trapezoidal, & Simpson's quadrature formulae, derived from Lagrange interpolation over each subinterval $[a,b]$ and then applied over several subintervals to span the interval of interest;
- the Guass quadrature approach, integrating exactly a $(2n+1)$'th-order polynomial approximation of a function on $[L,R]$ by selecting the $n+1$ nodal points at which the function is to be evaluated as the roots of a polynomial selected from a class of polynomials that obey a weighted orthogonality property;
- the Richardson extrapolation procedure, building up successively higher-order approximations of the integral by combining lower-order approximations calculated on a uniform grid; and
- the adaptive integration procedure, using knowledge of leading-order error to selectively refine the numerical grid where necessary.

The Guass quadrature approach is perhaps best suited for functions that are well approximated by polynomials over the interval of interest. The other three approaches might be considered to be somewhat more general purpose, and are compared for the simple function $f(x) = \sin(1/x)$ for $[L,R] = [0.1,2]$ in Figures 9.1 and 9.2.

Note in Figure 9.1a that the function evaluations determined by the standard trapezoidal and extrapolation-based approaches are uniform in x, which leaves this particular function under-resolved near the left end and over-resolved near the right end of the interval $[L,R]$. In contrast, note in Figure 9.1b that the function evaluations are automatically selected by the adaptive approach more densely in x near the left end than they are near the right end, which leaves the function nearly uniformly resolved across the entire interval $[L,R]$. The consequence of this is shown in Figure 9.2. As expected for a simple second-order method, the slope of the error of Algorithm 9.1 as the number of function values is increased is -2 (in log-log coördinates), and thus is not competitive with the other two techniques presented. In contrast, the slope of the extrapolation-based approximation keeps getting steeper and steeper, as its order keeps increasing more and more as the grid is refined. Nonetheless, the adaptive method, which is based on a sixth-order quadrature formula applied to appropriately-selected intervals, *substantially* outperforms the extrapolation-based method on this function (and many others of a similar nature) over the bulk of the range plotted in Figure 9.2. The increasingly-high order of accuracy of the extrapolation-based approach will *eventually* make it the most accurate method for a given number of function evaluations if a very large number of function evaluations is performed and a sufficiently high numerical precision is used. However, almost all practical numerical calculations are performed on finite grids with limited computational resources, and the accuracy of the calculation that you can afford to perform (e.g., with 256 function evaluations, as marked) will likely be optimized by using an *adequate* order of accuracy and then attending to other issues, such as grid design, rather than simply maximizing the order of accuracy of the scheme used on a uniform grid (as in the extrapolation-based approach) or on a prespecified grid designed specifically for high-order polynomial functions (as in the Gauss quadrature approach). To summarize:

Guideline 9.1 *Order of accuracy is only a means to an end, and must not be viewed as the ultimate goal or sole measure of a numerical method. In the end, it is the accuracy of the calculation that you can afford to perform that is the central issue of importance in the design or selection of a numerical method.*

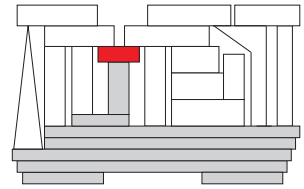
Exercises

Exercise 9.1 *Efficient implementation of Gauss-Chebyshev and Gauss-Legendre quadrature.* Write two efficient codes to implement the Gauss-Chebyshev and Gauss-Legendre quadrature approaches. Test them on the function $f(x) = \sin(1/x)$ for $[L,R] = [0.1,2]$ with varying grid resolutions, and compare your results with those plotted in Figures 9.1 and 9.2. Discuss.

References

Abramowitz, M, & Stegun, IA (1972) *Handbook of mathematical functions*, Wiley.

Embree, M (2009) *CAAM 453 Lecture Notes*.



Chapter 10

Ordinary Differential Equations

Contents

10.1 Explicit Euler (EE), implicit Euler (IE), and Crank-Nicolson (CN) . . . 267
10.2 Stability . . . 272
10.2.1 Stability regions in the complex plane of (λh) . . . 272
10.2.2 Further characterizing the stability of a time-integration scheme . . . 274
10.3 Accuracy . . . 276
10.3.1 Accuracy versus order of accuracy . . . 277
10.3.2 Classifications of error in the numerical simulation of oscillatory systems . . . 277
10.4 Non-stiff systems . . . 279
10.4.1 Explicit Runge-Kutta (ERK) methods . . . 279
10.4.1.1 Basic Runge Kutta methods: RK2 and RK4 . . . 280
10.4.1.2 Adaptive Runge-Kutta: RK4/5 . . . 287
10.4.1.3 Low-storage Runge-Kutta: RKW3, RK435, and RK548 . . . 289
10.4.2 Linear Multistep Methods (LMMs) . . . 293
10.4.2.1 Adams-Bashforth (AB) methods . . . 295
10.4.2.2 Adams-Moulton (AM) methods . . . 297
10.4.2.3 Consistency, spurious roots, zero stability, and convergence of LMMs . . . 298
10.5 Stiff systems . . . 299
10.5.1 Runge-Kutta-Chebyshev (RKC) methods† . . . 300
10.5.2 Implicit Runge-Kutta (IRK) methods . . . 303
10.5.2.1 Gauss-Legendre Runge Kutta (GLRK) methods† . . . 303
10.5.2.2 Diagonally-Implicit Runge Kutta (DIRK) methods . . . 304
10.5.3 Stiffly-stable linear multistep methods . . . 305
10.5.3.1 Backward differentiation formulae (BDF) . . . 306
10.5.3.2 Enright second derivative (ESD) methods† . . . 309
10.5.4 Implicit/Explicit (IMEX) methods . . . 309
10.6 Second-order systems . . . 313
10.6.1 Reduction to first-order form . . . 313
10.6.2 The Newmark family of methods . . . 314

10.6.3 Hamiltonian systems . . . 316
10.7 Two-point boundary value problems (TPBVPs) . . . 323
10.7.1 Shooting methods . . . 323
10.7.2 Relaxation methods . . . 324
Exercises . . . 325

Consider a scalar, first-order, possibly nonlinear ordinary differential equation (ODE) of the form[1]

$$\frac{dx(t)}{dt} = f(x(t),t) \quad \text{with} \quad x(0) \text{ specified.} \tag{10.1a}$$

We now address the **marching** of such an ODE forward in time, step by step, to determine $x(t)$ for some $t > 0$. Such marching methods approximate the solution $x(t)$ at timestep $t_{n+1} = t_n + h_n$ given the solution $x(t)$ and the function $f(x(t),t)$ at timestep t_n and, in some cases, a number of timesteps prior to t_n. For simplicity, we focus mostly on marches with constant stepsize h; generalization to nonconstant h_n is straightforward.

The exact solution of the ODE (10.1a) over a single timestep (t_n, t_{n+1}) is given by

$$x_{n+1} = x_n + \int_{t_n}^{t_{n+1}} f(x(t),t)dt, \tag{10.1b}$$

where $x_n = x(t_n)$. As the quantity being integrated, f, is itself a function of the result of the integration, x, the problem of integrating an ODE is fundamentally more difficult than the problem of numerical quadrature discussed in §9, in which the function being integrated was known in advance. The three time marching strategies we introduce in §10.1 may be related simply to the concept of integration: approximating the area in the above integral with a rectangle of height $f(x_n,t_n)$ gives the **explicit Euler** (**EE**, a.k.a. **forward Euler**, or simply "the" **Euler** method) method (10.5), approximating it with a rectangle of height $f(x_{n+1},t_{n+1})$ gives the **implicit Euler** (**IE**, a.k.a. **backward Euler**) method (10.7), and approximating it with a trapezoid with corners $\{t_n, f(x_n,t_n)\}$ and $\{t_{n+1}, f(x_{n+1},t_{n+1})\}$ gives the **Crank-Nicolson** (**CN**) method (10.9). Other approximations of the above integral, based on higher-order approximations of f, are also possible. For example, approximating f based on Lagrange interpolations of recent function evaluations lead to the **Adams-Bashforth** & **Adams-Moulton** methods (§10.4.2.1 & §10.4.2.2). Similarly, midpoint & Simpson's approximations of the integral in the exact solution of the ODE over two timesteps,

$$x_{n+1} = x_{n-1} + \int_{t_{n-1}}^{t_{n+1}} f(x(t),t)dt, \tag{10.1c}$$

lead to the **leapfrog** & **Milne** methods discussed in Exercises 10.7 & 10.8.

The exact solution to (10.1a) may also be written in terms of the Taylor series expansion

$$x(t_{n+1}) = x(t_n) + hx'(t_n) + \frac{h^2}{2!}x''(t_n) + \frac{h^3}{3!}x'''(t_n) + \frac{h^4}{4!}x''''(t_n) + \ldots \tag{10.1d}$$

where, since $f = f(x(t),t)$, it follows that

$$x' = \frac{dx}{dt} = f \tag{10.2a}$$

$$x'' = \frac{dx'}{dt} = \frac{df}{dt} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\frac{dx}{dt} = f_t + f_x f \tag{10.2b}$$

$$x''' = \frac{dx''}{dt} = \frac{d}{dt}(f_t + f_x f) = f_{tt} + 2f_{xt}f + f_x f_t + f_{xx}f^2 + f_x^2 f, \tag{10.2c}$$

$$x'''' = f_{ttt} + f_{tt}f_x + 3f_{xtt}f + 3f_{xt}f_t + 5f_{xt}f_x f + f_x^2 f_t + 3f_{xx}f_t f + 3f_{xxt}f^2 + 4f_{xx}f_x f^2 + f_x^3 f + f_{xxx}f^3. \tag{10.2d}$$

[1] An ODE system is said to be **autonomous** if the function f on the RHS depends on $x(t)$, but doesn't depend explicitly on t.

The methods we will examine extend immediately to first-order systems of ODEs of the form

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{f}(\mathbf{x}(t),t) \quad \text{with} \quad \mathbf{x}(0) = \text{ specified.} \tag{10.3a}$$

The exact solution of (10.3a) over a single timestep (t_n, t_{n+1}) may be written in integral form as

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \int_{t_n}^{t_{n+1}} \mathbf{f}(\mathbf{x}(t),t)dt \qquad \text{or} \qquad \mathbf{x}_{n+1} = \mathbf{x}_{n-1} + \int_{t_{n-1}}^{t_{n+1}} \mathbf{f}(\mathbf{x}(t),t)dt, \tag{10.3b}$$

and in Taylor-series form as

$$\mathbf{x}(t_{n+1}) = \mathbf{x}(t_n) + h\mathbf{x}'(t_n) + \frac{h^2}{2!}\mathbf{x}''(t_n) + \frac{h^3}{3!}\mathbf{x}'''(t_n) + \frac{h^4}{4!}\mathbf{x}''''(t_n) + \ldots \tag{10.3c}$$

where in this case, using summation notation for clarity,

$$x_i' = \frac{dx_i}{dt} = f_i \tag{10.4a}$$

$$x_i'' = \frac{dx_i'}{dt} = \frac{df_i}{dt} = \frac{\partial f_i}{\partial t} + \frac{\partial f_i}{\partial x_j}\frac{dx_j}{dt} = \frac{\partial f_i}{\partial t} + \frac{\partial f_i}{\partial x_j}f_j \tag{10.4b}$$

$$x_i''' = \frac{dx_i''}{dt} = \frac{d}{dt}\left(\frac{\partial f_i}{\partial t} + \frac{\partial f_i}{\partial x_j}f_j\right) = \frac{\partial^2 f_i}{\partial t^2} + 2\frac{\partial^2 f_i}{\partial x_j \partial t}f_j + \frac{\partial f_i}{\partial x_j}\frac{\partial f_j}{\partial t} + \frac{\partial^2 f_i}{\partial x_k \partial x_j}f_j f_k + \frac{\partial f_i}{\partial x_j}\frac{\partial f_j}{\partial x_k}f_k, \tag{10.4c}$$

$$\begin{aligned} x_i'''' = \frac{dx_i''}{dt} = {} & \frac{\partial^3 f_i}{\partial t^3} + \frac{\partial^2 f_j}{\partial t^2}\frac{\partial f_i}{\partial x_j} + 3\frac{\partial^3 f_i}{\partial x_j \partial t^2}f_j + 3\frac{\partial^2 f_i}{\partial x_j \partial t}\frac{\partial f_j}{\partial t} + 3\frac{\partial^2 f_i}{\partial x_j \partial t}\frac{\partial f_j}{\partial x_k}f_k + 2\frac{\partial^2 f_j}{\partial x_k \partial t}\frac{\partial f_i}{\partial x_j}f_k \\ & + \frac{\partial f_i}{\partial x_j}\frac{\partial f_j}{\partial x_k}\frac{\partial f_k}{\partial t} + 3\frac{\partial^2 f_i}{\partial x_k \partial x_j}\frac{\partial f_j}{\partial t}f_k + 3\frac{\partial^3 f_i}{\partial x_k \partial x_j \partial t}f_j f_k + 3\frac{\partial^2 f_i}{\partial x_k \partial x_j}\frac{\partial f_j}{\partial x_\ell}f_k f_\ell \\ & + \frac{\partial^2 f_j}{\partial x_k \partial x_\ell}\frac{\partial f_i}{\partial x_j}f_k f_\ell + \frac{\partial f_i}{\partial x_j}\frac{\partial f_j}{\partial x_k}\frac{\partial f_k}{\partial x_\ell}f_\ell + \frac{\partial^3 f_i}{\partial x_k \partial x_j \partial x_\ell}f_j f_k f_\ell. \end{aligned} \tag{10.4d}$$

For simplicity of the presentation, the algorithms in this chapter are *derived* in scalar form. However, for generality, the algorithms are *summarized* and *implemented* in their more general vector form. Note also that ODEs with higher-order derivatives may, via the appropriate definitions, always be reduced to first-order systems of ODEs, as illustrated by example below [in (10.11)-(10.12)]. Thus, the algorithms developed in this chapter are in fact applicable to a broad range of problems. Specific attention is paid to the time marching of second-order systems in §10.6, and our focus is generalized to two-point boundary value problems (TPBVPs) in §10.7. Note also that we refer to the independent variable in this chapter as time, t, but this is done without loss of generality, and other interpretations of the independent variable are also possible.

10.1 Explicit Euler (EE), implicit Euler (IE), and Crank-Nicolson (CN)

The time integration method obtained by retaining the first two terms on the RHS of (10.3c) is given by

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}(\mathbf{x}_n, t_n).} \tag{10.5}$$

This is referred to as the **explicit Euler** (**EE**) method. The word **explicit** is used to indicate that the function $\mathbf{f}$ is calculated only at known values of $\mathbf{x}$. Note that the EE method neglects terms which are proportional to h^2 and higher, and is thus "second-order" accurate over a single timestep. As with the problem of numerical quadrature, a more relevant measure is the accuracy achieved when marching the ODE over a given time

interval (t_0, t_0+T) as the timesteps h are made smaller. In such a setting, the number of timesteps n (each with error proportional to h^2) is given by $n = T/h$ and thus, multiplying h^2 by $n = T/h$, over a specified time interval (t_0, t_0+T), *EE is first-order accurate*. [In other words, for a given T and sufficiently small h, reducing h by another factor of 2 reduces the error by a factor of 2.] It is thus seen that, to determine the order of accuracy of a scheme over a given time interval (the **global accuracy** of the scheme), we must subtract 1 from the order of accuracy of the scheme over a single timestep (the **local accuracy** of the scheme).

We could develop a family of higher-order explicit time integration schemes simply by retaining additional terms on the RHS of (10.3c), simplifying with (10.2) or (10.4) as necessary. Such methods are sometimes called **explicit Taylor series** (**ETS**) methods. Such ETS schemes are uncommon, however, as their computational expense increases rapidly with their order (due to all of the cross derivatives required) as compared with the more efficient explicit methods we will propose below.

Note that a Taylor series expansion in time may also be written around t_{n+1}:

$$\mathbf{x}(t_n) = \mathbf{x}(t_{n+1}) - h\mathbf{x}'(t_{n+1}) + \frac{h^2}{2!}\mathbf{x}''(t_{n+1}) - \frac{h^3}{3!}\mathbf{x}'''(t_{n+1}) + \frac{h^4}{4!}\mathbf{x}''''(t_{n+1})\ldots \tag{10.6}$$

The time integration method based on the first two terms on the RHS of this Taylor series is given by

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}).} \tag{10.7}$$

This is referred to as the **implicit Euler** (**IE**) method. The word **implicit** is used to indicate that the scheme relies on the function $\mathbf{f}$ calculated at the yet-to-be-determined value of $\mathbf{x}$. If $\mathbf{f}$ is nonlinear in $\mathbf{x}$, implicit methods are difficult to use directly[2], because knowledge of $\mathbf{x}_{n+1}$ is needed (before it is computed!) to compute $\mathbf{f}$ in order to advance from $\mathbf{x}_n$ to $\mathbf{x}_{n+1}$. On the other hand, if $\mathbf{f}$ is *linear* in $\mathbf{x}$, implicit strategies such as (10.7) are easily solved for $\mathbf{x}_{n+1}$: to advance $d\mathbf{x}/dt = A\mathbf{x}$ using IE, march the equation $\mathbf{x}_{n+1} = \mathbf{x}_n + hA\mathbf{x}_{n+1} \Rightarrow (I - hA)\mathbf{x}_{n+1} = \mathbf{x}_n$ by solving for the unknown $\mathbf{x}_{n+1}$ at each timestep; this may be done quite efficiently using Gaussian elimination if A has exploitable (e.g., banded) sparsity. Note that the IE method also neglects terms which are proportional to h^2 and higher, and is thus "second-order" accurate over a single timestep. By the global versus local argument given above, over a specified time interval (t_0, t_0+T), *IE is first-order accurate*.

As in the explicit case, we could develop a family of higher-order **implicit Taylor series** (**ITS**) methods simply by retaining additional terms on the RHS of (10.6), simplifying with (10.2) or (10.4) as necessary. Indeed, the second-order method obtained by retaining three terms on the RHS of (10.6), which we may name **ITS2**, is also known as the **2nd-order Enright second derivative** (**ESD2**) method, which is considered further in §10.5.3.2. Higher-order ITS methods are uncommon, as their computational expense increases rapidly with their order as compared with the more efficient implicit methods we will propose below.

Subtracting (10.6) from (10.3c) and rearranging, noting that $\mathbf{x}''(t_{n+1}) = \mathbf{x}''(t_n) + h\mathbf{x}'''(t_n) + \ldots$, gives

$$\mathbf{x}(t_{n+1}) = \mathbf{x}(t_n) + \frac{h}{2}\left[\mathbf{x}'(t_n) + \mathbf{x}'(t_{n+1})\right] - \frac{h^3}{12}\mathbf{x}'''(t_n) + \ldots \tag{10.8}$$

The time integration method based on the first two terms of this expression is given by

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h[\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}) + \mathbf{f}(\mathbf{x}_n, t_n)]/2.} \tag{10.9}$$

This is referred to as the **Crank-Nicolson** (**CN**) or **trapezoidal** method. It neglects terms which are proportional to h^3 and higher, and is thus "third-order" accurate over a single timestep. By the global versus local argument given above, over a specified time interval (t_0, t_0+T), *CN is second-order accurate*. [In other words, for

[2] The alternative is to view an implicit marching formula such as (10.7) as a nonlinear equation to be solved iteratively at each timestep (for further discussion of such an approach, see page 281). Though such schemes are common, they introduce an added level of complexity to the time marching scheme as well as a new parameter quantifying the accuracy with which the implicit time-marching formula should be solved at each timestep. Such iterative schemes can often be avoided by the appropriate selection of a direct time marching method (or combination of methods) suited to the problem at hand, as discussed in the remainder of this chapter.

a given T and sufficiently small h, reducing h by another factor of 2 reduces the error by a factor of 4.] As with the IE method, if $\mathbf{f}$ is nonlinear in $\mathbf{x}$, the CN method is difficult to use. On the other hand, to advance the linear system $d\mathbf{x}/dt = A\mathbf{x}$ using CN, simply march $\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{h}{2}(A\mathbf{x}_n + A\mathbf{x}_{n+1}) \;\Rightarrow\; (I - \frac{h}{2}A)\mathbf{x}_{n+1} = (I + \frac{h}{2}A)\mathbf{x}_n$ by solving for the unknown $\mathbf{x}_{n+1}$ at each timestep.

Example 10.1 Numerical simulation of two model problems

The unforced continuous-time scalar **model problem** and its multivariable generalization,

$$\frac{dx}{dt} = \lambda x, \tag{10.10a}$$

$$\frac{d\mathbf{x}}{dt} = A\mathbf{x}, \tag{10.10b}$$

are useful for characterizing time marching methods. The exact solution of these systems are, respectively, $x(t) = e^{\lambda t}x(0)$ and $\mathbf{x}(t) = e^{At}\mathbf{x}(0)$ where $e^{At} \triangleq I + At + A^2t^2/2! + A^3t^3/3! + \ldots$ [note that both systems are analyzed much further in §20.1.2, which includes a detailed discussion of the matrix exponential e^{At}]. We now (in Figures 10.1 and 10.2) compare numerical approximations of these systems to their exact solutions in order to quantify the pros and cons of the EE, IE, and CN methods. The insight we gain by so doing allows us to anticipate how these methods will behave on more difficult problems for which exact solutions are not available, and foreshadows the development of the other time-marching schemes presented in this chapter.

Simulation of an exponentially-decaying system. The first problem we will consider in this example is the scalar model problem (10.10a) with $\lambda = -1$. The exact solution is $x(t) = e^{-t}x(0)$. In Figure 10.1, we demonstrate the application of the EE, IE, and CN methods to this problem. Note that EE appears to be unstable for the large values of h. Note also that all three methods are more accurate as h is refined, with CN appearing to be the most accurate.

Simulation of an undamped oscillating system. The second problem we will consider in this example is the second-order ODE for a simple mass-spring system as given by

$$\frac{d^2q}{dt^2} = -\omega^2 q \quad \text{with} \quad q(t_0) = q_0, \quad \left.\frac{dq}{dt}\right|_{t_0} = 0, \tag{10.11}$$

where $\omega = 1$. The exact solution is $q(t) = q_0\cos[\omega(t - t_0)] = (q_0/2)[e^{i\omega(t-t_0)} + e^{-i\omega(t-t_0)}]$. We may easily write this second-order ODE in the first-order **state-space form** (10.10b) by defining $x_1 = q$ and $x_2 = dq/dt$:

$$\frac{d}{dt}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \Leftrightarrow \quad \frac{d\mathbf{x}}{dt} = A\mathbf{x}, \quad \mathbf{x}(0) = \begin{pmatrix} q_0 \\ 0 \end{pmatrix} \quad \Rightarrow \quad \mathbf{x}(t) = e^{At}\mathbf{x}(0). \tag{10.12}$$

The eigenvalues of A are $\pm i\omega$. Note that, even though this is a real, physical system, these eigenvalues are imaginary; had we started with the equation for a damped oscillator, the eigenvalues would have a negative real part. Note also that, as discussed in §4.4.4, A may be diagonalized by its matrix of eigenvectors S:

$$A = S\Lambda S^{-1} \quad \text{where} \quad \Lambda = \begin{pmatrix} i\omega & 0 \\ 0 & -i\omega \end{pmatrix}.$$

Thus, we have

$$S^{-1}\left[\frac{d\mathbf{x}}{dt} = S\Lambda S^{-1}\mathbf{x}\right] \quad \Rightarrow \quad \frac{d\mathbf{z}}{dt} = \Lambda\mathbf{z} \quad \text{where} \quad \mathbf{z} \triangleq S^{-1}\mathbf{x}.$$

In terms of the components of $\mathbf{z}$, we have decoupled the dynamics of the system:

$$\begin{aligned} \frac{dz_1}{dt} &= i\omega z_1 \quad &\Rightarrow \quad z_1(t) &= e^{i\omega t}z_1(0), \\ \frac{dz_2}{dt} &= -i\omega z_2 \quad &\Rightarrow \quad z_2(t) &= e^{-i\omega t}z_2(0), \end{aligned} \qquad \text{where} \quad \mathbf{z}(0) = S^{-1}\mathbf{x}(0). \tag{10.13}$$

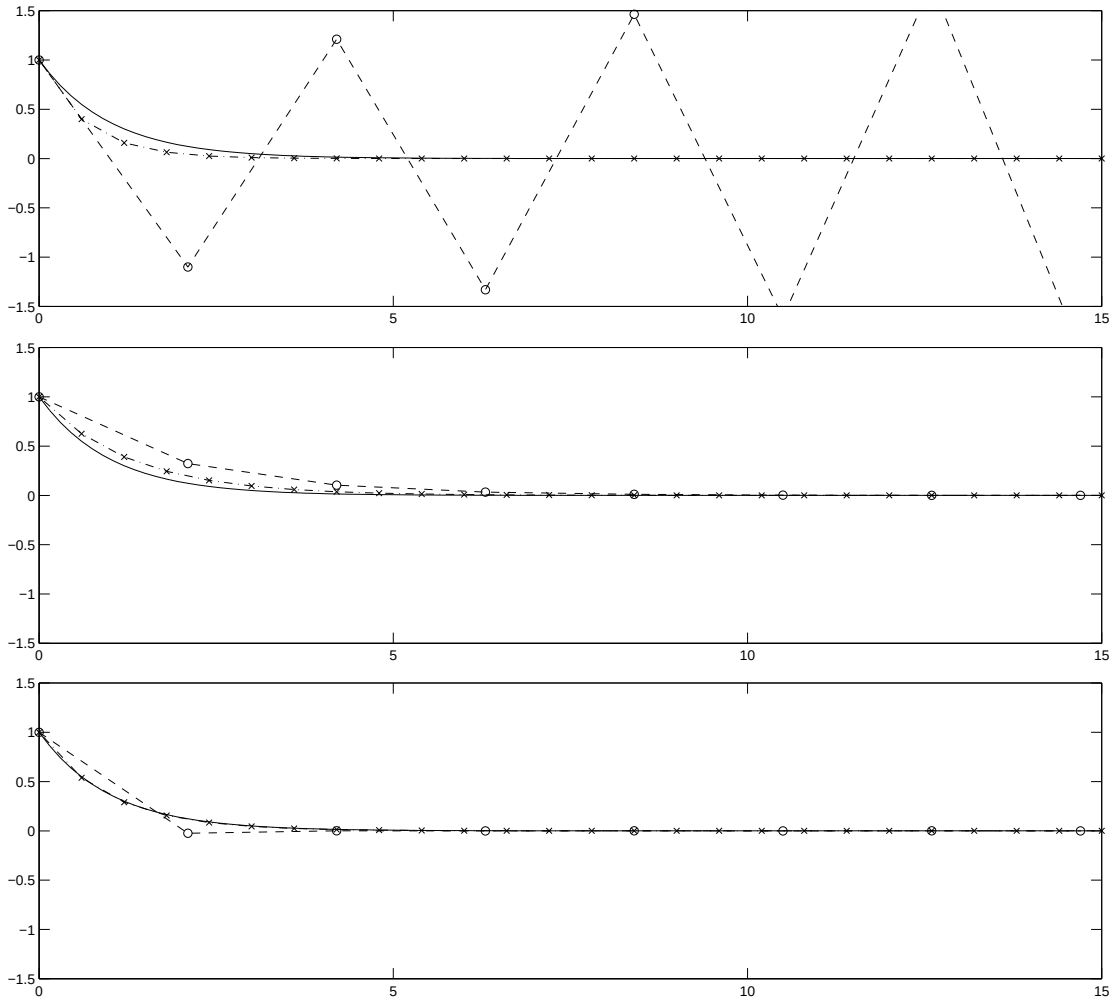


Figure 10.1: Simulation of the model problem $dx/dt = \lambda x$ with $\lambda = -1$ using EE (top), IE (middle), and CN (bottom). Symbols denote: $\circ$, $h = 2.1$; $\times$, $h = 0.6$; ——, exact solution.

Each of these scalar systems is of the same form as the scalar model problem (10.10a) with imaginary values for λ. Further, assuming exact arithmetic, numerical approximation of $\mathbf{x}(t)$ in the multivariable ODE formulation (10.12) using EE, IE, or CN is *equivalent* to numerical approximation of $\mathbf{z}(t)$ in the scalar ODE formulations (10.13) followed by inverse transforming the result[3] with $\mathbf{x}(t) = S\mathbf{z}(t)$. Thus, eigenmode decompositions of real physical linear systems motivate us to characterize the behavior of the numerical approximation of the scalar model problem (10.10a) over the entire complex plane λ.

In Figure 10.2, we show the application of the EE, IE, and CN methods to the first-order system of equations (10.12) [equivalently, to (10.13), followed by computing $\mathbf{x}(t) = S\mathbf{z}(t)$]. Note that the EE method appears to be unstable for both large and small values of h. Note also that all three methods are more accurate as h is refined, with the CN method appearing to be the most accurate.

[3] Note that, since the original multivariable ODE system (10.12) in the variable $\mathbf{x}(t)$ is real, this process of reëxpressing the dynamics into decoupled scalar *complex* ODE problems in the variables $\{z_1(t), z_2(t), \ldots\}$, marching these scalar problems to some time t, and then inverse transforming to the original coördinates $\mathbf{x}(t)$ necessarily ends up with a *real* result (that is, assuming exact arithmetic).

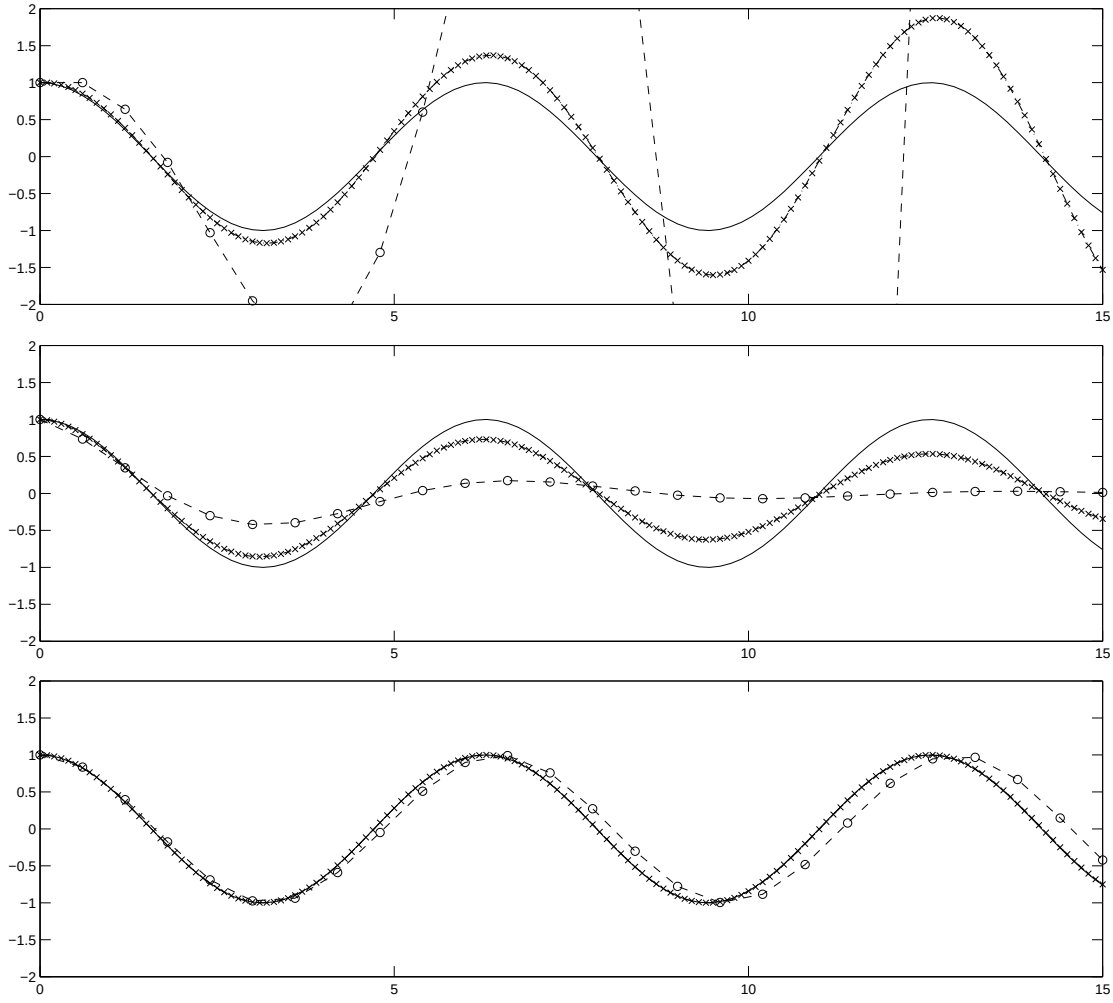


Figure 10.2: Simulation of the oscillatory system $d^2q/dt^2 = -\omega^2 q$ with $\omega = 1$ using EE (top), IE (middle), and CN (bottom). Symbols denote: $\circ$, $h = 0.6$; $\times$, $h = 0.1$; ——, exact solution.

To summarize, we see that some numerical methods for time integration of ODEs are more accurate than others, and that some numerical methods are sometimes unstable even for ODEs with stable exact solutions. In the next two sections, we develop techniques to quantify the stability and accuracy of numerical methods for time integration of ODEs by applying of these numerical methods to the scalar model problem (10.10a).

For convenience, the response(s) y_p of a **MIMO** (**multiple-input, multiple output**) linear system written in the **state-space** form

$$\mathbf{x}' = A\mathbf{x} + B\mathbf{u}, \quad \mathbf{y} = C\mathbf{x} + D\mathbf{u}, \tag{10.14}$$

to a impulse, step, or ramp on the input(s) u_m may be calculated (via the CN method) and plotted using Algorithm 10.1; systems of this form are studied in detail in §20.

Algorithm 10.1: Plot the response(s) y_p to impulse, step, or ramp input(s) u_m applied to the system (10.14).

View Test

```
function ResponseSS(A,B,C,D,w,g,Input)
% Using CN, plot the P components of the response y_1,...,y_P (in each of the P rows of
% subplots) of a MIMO LTI system.  If w=-1, the input u is specified in Input; otherwise,
% the input u is zero in all but the m'th component (in each of the M columns of subplots),
% and u_m is an impulse for w=0, step for w=1, or polynomial t^(w-1) for w>1.
% The derived type g groups together convenient plotting parameters: g.T is the interval
% over which the response is plotted, and {g.styleu,g.styley} are the linestyles used.
N=length(A); M=size(B,2); P=size(C,1); x=zeros(N,1); K=1000; h=g.T/K; t=[0:K]*h;
if w==0; u(1,:)=[2/h, zeros(1,K)]; elseif w>0, u(1,:)=t.^(w-1);
else,    M=1; for k=1:K+1; u(:,k)=Input(t(k)); end, end
for m=1:M
  Ap=eye(N)+A*h/2; if M>1; Bs=B(:,m)*h/2; Ds=D(:,m); else; Bs=B*h/2; Ds=D; end
  Am=eye(N)-A*h/2; [x(:,2),Amod]=Gauss(Am,Ap*x(:,1)+Bs*(u(:,2)+u(:,1)),N);
  for k=3:K+1, [x(:,k)]=GaussLU(Amod,Ap*x(:,k-1)+Bs*(u(:,k)+u(:,k-1)),N); end
  for p=1:P, subplot(P,M,m+(p-1)*M), hold on
     plot(t,C(p,:)*x(:,[1:K+1])+Ds(p,:)*u(:,[1:K+1]),g.styley,t,0*t,'k:'), axis tight, end
  if M*P==1, plot(t(2:end),u(1,2:end),g.styleu), end
end
end % function ResponseSS
```

10.2 Stability

For stability of a numerical time integration of an ODE, we need to ensure that, if the exact solution is bounded, the numerical solution is also bounded. For a given scalar λ in a stable scalar ODE $x' = \lambda x$, or a given set of eigenvalues $\{\lambda_1, \ldots, \lambda_n\}$ of A in a stable system of ODEs $\mathbf{x}' = A\mathbf{x} \Leftrightarrow \mathbf{z}' = \Lambda\mathbf{z}$, we often need to restrict the timestep h in order to achieve this; *for a given stable linear ODE system*, we thus say that

- a numerical scheme is **stable** when its solution is bounded for any h,
- a numerical scheme is **unstable** when its solution is unbounded for any h, and
- a numerical scheme is **conditionally stable** when its solution is bounded iff h is sufficiently small.

10.2.1 Stability regions in the complex plane of (λh)

Applying a candidate time-marching method to the model problem $x' = \lambda x$ and expressing the resulting time march as $x_{n+1} = \sigma x_n$, we can easily determine the stability of the march by evaluating the magnitude of the amplification factor σ, as illustrated by the following three examples.

Example 10.2 Stability of the EE method. Applying the EE method (10.5) to the model problem $x' = \lambda x$,

$$x_{n+1} = x_n + \lambda h x_n = (1+\lambda h) x_n \triangleq \sigma x_n \quad \Rightarrow \quad x_n = \sigma^n x_0, \quad \sigma = 1 + \lambda h. \tag{10.15}$$

For large n, writing $\lambda = \lambda_R + \mathrm{i}\lambda_I$, the numerical solution remains stable iff

$$|\sigma| \le 1 \quad \Rightarrow \quad (1+\lambda_R h)^2 + (\lambda_I h)^2 \le 1.$$

The region of the complex plane (λh) which satisfies this stability constraint is shown in Figure 10.3a. Note that this region of stability in the complex plane λh is consistent with the numerical simulations shown in Figure 10.1a and 10.2a: for real, negative λ, this numerical method is conditionally stable (*i.e.*, it is stable for sufficiently small h), whereas for imaginary λ, this numerical method is unstable for any h.

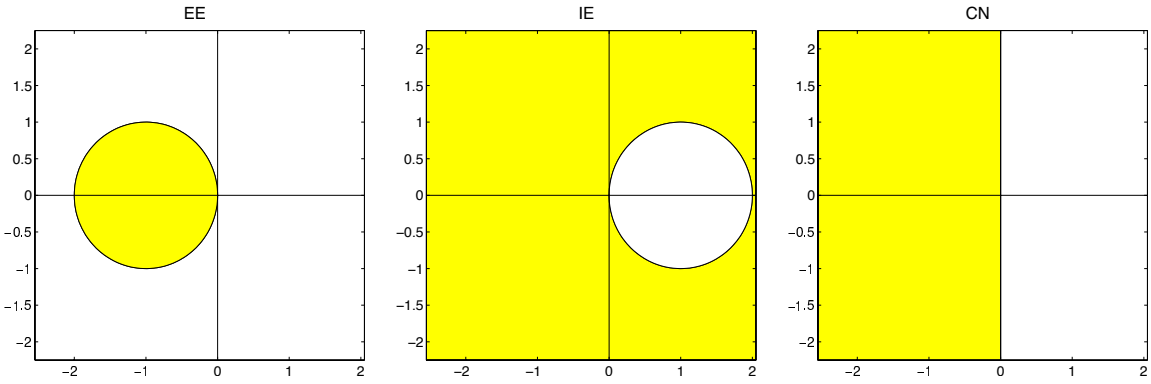


Figure 10.3: Stability regions in the complex plane λh for the numerical solution of $x' = \lambda x$ with timestep h using (left) EE, (center) IE, and (right) CN. The stable regions (i.e., the regions for which $|\sigma| \leq 1$) are shaded.

Example 10.3 Stability of the IE method. Applying the IE method (10.7) to the model problem $x' = \lambda x$,

$$x_{n+1} = x_n + \lambda h x_{n+1} \quad \Rightarrow \quad x_{n+1} = \frac{1}{1-\lambda h} x_n \triangleq \sigma x_n \quad \Rightarrow \quad x_n = \sigma^n x_0, \quad \sigma = \frac{1}{1-\lambda h}. \tag{10.16}$$

For large n, the numerical solution remains stable iff

$$|\sigma| \leq 1 \quad \Rightarrow \quad (1-\lambda_R h)^2 + (\lambda_I h)^2 \geq 1.$$

The region of the complex plane (λh) which satisfies this stability constraint is shown in Figure 10.3b. Note that this region of stability in the complex plane λh is consistent with the numerical simulations shown in Figure 10.1b and 10.2b: this method is stable for any stable ODE for any h.

Example 10.4 Stability of the CN method. Applying the CN method (10.9) to the model problem $x' = \lambda x$,

$$x_{n+1} = x_n + \frac{\lambda h}{2}(x_n + x_{n+1}) \quad \Rightarrow \quad x_{n+1} = \left(\frac{1+\frac{\lambda h}{2}}{1-\frac{\lambda h}{2}}\right) x_n \triangleq \sigma x_n \quad \Rightarrow \quad x_n = \sigma^n x_0, \quad \sigma = \frac{1+\frac{\lambda h}{2}}{1-\frac{\lambda h}{2}}. \tag{10.17}$$

For large n, the numerical solution remains stable iff

$$|\sigma| \leq 1 \quad \Rightarrow \quad \ldots \quad \Rightarrow \quad \Re(\lambda h) \leq 0.$$

The region of the complex plane (λh) which satisfies this stability constraint coincides exactly with the region of stability of the exact solution, as shown in Figure 10.3c. Note that this region of stability in the complex plane λh is consistent with the numerical simulations shown in Figure 10.1c and 10.2c, which are stable for systems with $\Re(\lambda) < 0$ and on the border of stability and instability for systems with $\Re(\lambda) = 0$.

In analyses of this sort, the relationship between (λh) and σ [see, e.g., (10.15), (10.16), (10.17)] is called the **stability polynomial**. Stability analyses of this sort are especially useful because their characterizations are precise: if a time marching method and timestep h are chosen for a linear ODE system $\mathbf{x}' = A\mathbf{x}$ such that all of the eigenvalues of A (scaled by h) are contained in the domain of stability of the method, then the numerical simulation will be stable. On the other hand, if *any* of the eigenvalues of A (scaled by h) are *not* contained in the domain of stability of the method, the simulation will generally be unstable. If the ODE

Algorithm 10.2: Code used for drawing the linear stability contours presented in §10.

View

```
% script <a href="matlab:StabContours">StabContours</a>
% Plot stability contours of several ODE marching methods in the complex plane lambda*h.
Np=201; V=[1 1.0000000001]; close all;
for k=1:49
  switch k    % Set up region in the sigma plane over which to plot the stability boundary
    case {1,2,3},                    B=[ -2.55; 2.05; -2.25; 2.25];  % EE, IE, CN
    case {4,5,6},                    B=[ -4.05; 2.05; -3.05; 3.05];  % RK2-4
   % other cases are set up similarly...
  end % switch
  LR(:,1)=[B(1):(B(2)-B(1))/(Np-1):B(2)]'; LI(:,1)=[B(3):(B(4)-B(3))/(Np-1):B(4)]';
  for j=1:Np, for i=1:Np,  L=LR(j)+sqrt(-1)*LI(i);
    switch k              % Now compute sigma over an array of points in the lambda-h plane
      case  1, sig(i,j)=abs(1+L);                                          % EE
      case  2, sig(i,j)=abs(1/(1-L));                                      % IE
      case  3, sig(i,j)=abs((1+L/2)/(1-L/2));                              % CN
      case  4, sig(i,j)=abs(1+L+L^2/2);                                    % RK2
      case  5, sig(i,j)=abs(1+L+L^2/2+L^3/6);                              % RK3
      case  6, sig(i,j)=abs(1+L+L^2/2+L^3/6+L^4/24);                       % RK4
      case  7, sig(i,j)=max(abs(roots([1 -1 0]-[0 3 -1]*L/2)));            % AB2
     % other cases are set up similarly...
    end % switch
  end, end
  figure(k), contourf(LR,LI,1./sig,V,'k-'), colormap autumn, axis('square'), hold on
  plot([B(1) B(2)],[0,0],'k-'), plot([0,0],[B(3) B(4)],'k-'),
end
% end script StabContours
```

system is nonlinear, analysis of the stability of a numerical integration method is more difficult, but may be estimated by performing linearization of the ODE about a base trajectory and evaluating the stability of the resulting linear ODE for the perturbation. Note also in the above derivations that the EE, IE, and CN methods are simple enough that their stability regions (see Figure 10.3) may be determined analytically. This is not the case for most of the time marching methods developed in the remainder of this chapter; however, it is a simple matter to determine these stability regions numerically (see Algorithm 10.2).

10.2.2 Further characterizing the stability of a time-integration scheme

For a given stable linear ODE system, we saw above that a time-integration scheme can be stable (for any h), unstable (for any h), or conditionally stable (i.e., stable for some h and unstable for others). To quantify the stability of a time-integration scheme more completely, we may look over the entire complex plane (λh): noting Figure B.2 and defining $\sigma(\infty) \triangleq \lim_{|\lambda h|\to\infty} \sigma(\lambda h)$, a time-integration scheme is said to be

- L**-stable** if its stability region contains the entire LHP and $\sigma(\infty) = 0$;
- **strongly** A**-stable** if its stability region contains the entire LHP and $|\sigma(\infty)| < 1$;
- **stiffly stable** if its stability region contains the solid shaded region in Figure 10.4b and $\sigma(\infty) = 0$;
- A**-stable** if its stability region contains the entire LHP;
- $A(\alpha)$ **stable** if its stability region contains the solid shaded region in Figure 10.4a for an angle $\alpha > 0$;
- $A(0)$ **stable** if it is $A(\alpha)$ stable for some (unspecified) $\alpha > 0$;
- A_0 **stable** if the open negative real axis is stable.

Thus:

- L-stability $\Rightarrow$ strong A-stability $\Rightarrow$ A-stability,
- L-stability $\Rightarrow$ stiff-stability $\Rightarrow$ A-stability;

- A-stability $\Rightarrow$ $A(\alpha)$-stability $\Rightarrow$ $A(0)$-stability $\Rightarrow$ A_0-stability.

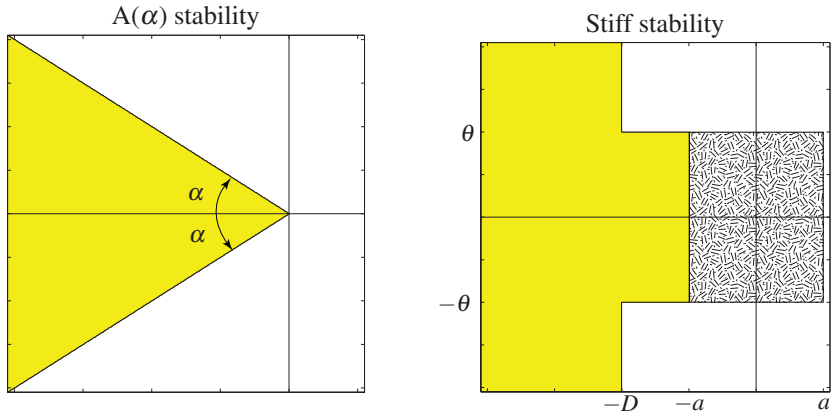


Figure 10.4: (shaded) The regions of the complex plane (λh) that must be stable for a numerical scheme to be classified as (left) $A(\alpha)$ stable, for some angle $\alpha > 0$, and (right) stiffly stable, for some $\theta > 0$, $a > 0$, and $D > a$ [note that Gear (1971) recommends $\theta \approx \pi/5$]. Note also that, for a multistep scheme (see §10.4.2) to be classified as stiffly stable, the so-called "spurious" σ roots of the multistep scheme must be smaller in magnitude than the "physical" σ root in the textured region near the origin.

Recall that a numerical time integration scheme is only accurate for (λh) sufficiently close to the origin. The reason that such careful classifications of stability are useful is that, in many problems, commonly referred to as **stiff**, we can only afford to resolve the time evolution of the slowest modes [that is, those modes for which (λh) is close to the origin]. The numerical simulation of the remaining modes, whose time evolution is unresolved by the time marching scheme, should nonetheless decay to zero if they are stable; choosing a numerical scheme with stability characteristics appropriate to the distribution of eigenvalues of the unresolved stable modes in the problem at hand helps to ensure this (for further discussion, see §10.5).

Though we focus mainly on linear stability in this text, the concept of stability may be generalized to nonlinear systems of the form (10.1a); in particular, if the RHS $f(x(t),t)$ obeys the **contractivity condition**

$$\langle f(x(t),t) - f(y(t),t), x - y \rangle \leq 0 \quad \text{for all} \quad t \geq 0 \quad \text{and for all} \quad x(t),\ y(t), \tag{10.18a}$$

for an appropriate inner product $\langle \cdot, \cdot \rangle$, then a nonlinear ODE marching scheme is said to be B **stable** if, for any h and n, it follows that

$$\|x_{n+1} - y_{n+1}\| \leq \|x_n - y_n\|, \tag{10.18b}$$

where $\{x_n, x_{n+1}\}$ denote numerical approximations of $dx/dt = f(x(t),t)$ at $\{t_n, t_{n+1}\}$, $\{y_n, y_{n+1}\}$ denote numerical approximations of $dy/dt = f(y(t),t)$ at $\{t_n, t_{n+1}\}$, and $\|z\| = \langle z, z \rangle$.

10.3 Accuracy

Revisiting the model problem $x' = \lambda x$, the exact solution (assuming $t_0 = 0$ and $h =$ constant) is

$$x(t_n) = e^{\lambda t_n} x_0 = (e^{\lambda h})^n x_0 = \left(1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \ldots\right)^n x_0.$$

On the other hand, solving the model problem with EE led to

$$x_n = (1 + \lambda h)^n x_0 \triangleq \sigma^n x_0,$$

solving the model problem with IE led to

$$x_n = \left(\frac{1}{1-\lambda h}\right)^n x_0 = \left(1+\lambda h+\lambda^2 h^2+\lambda^3 h^3+\ldots\right)^n x_0 \triangleq \sigma^n x_0,$$

and solving the model problem with CN led to

$$x_n = \left(\frac{1+\frac{\lambda h}{2}}{1-\frac{\lambda h}{2}}\right)^n x_0 = \left(1+\lambda h+\frac{\lambda^2 h^2}{2}+\frac{\lambda^3 h^3}{4}+\ldots\right)^n x_0 \triangleq \sigma^n x_0.$$

To quantify the accuracy of these three methods, we can compare the amplification factor σ in each of the numerical approximations to the exact value $e^{\lambda h}$. The leading order error of the EE and IE methods are seen to be proportional to h^2, and the leading order error of the CN method is proportional to h^3, as noted in §10.1. Thus, again, over a specified time interval (t_0, t_0+T), EE and IE are first-order accurate and CN is second-order accurate. The higher order of accuracy of the CN method implies an improved rate of convergence of this scheme to the exact solution as the timestep h is refined, as observed in Figures 10.1 and 10.2.

10.3.1 Accuracy versus order of accuracy

As with finite difference strategies for differentiation and quadrature strategies for integration (see Guideline 9.1), what you really care about when marching an ODE is the *accuracy* of the result when using the timestep that you can ultimately afford to use. The issue of *order of accuracy* is only a means to an end. Though higher-order-accurate schemes are always superior to lower-order-accurate schemes when the timestep is vanishingly small, not all schemes of a given order have the same accuracy (indeed, they can differ greatly in terms of the magnitude of the coefficient of the leading-order error), and, *for a given (reasonable) stepsize h, lower-order-accurate schemes are often more accurate than higher-order-accurate schemes*. Indeed, the goal when performing a numerical simulation is to use a timestep which is as large as you can get away with (to minimize the computation effort required to complete the simulation) while ensuring both numerical stability and that the desired degree of accuracy is obtained. Thus, selecting the most suitable time-marching method is not as simple a matter as selecting the highest-order-accurate method that you have the patience to code.

In this regard, the CN method is often the preferred choice for tightly-banded linear ODE systems: it is second-order accurate, its stability boundary coincides with that of the exact solution (see Figure 10.1c), it is computationally inexpensive per timestep, and it is relatively simple to implement (see Algorithm 10.1). On the other hand, if the ODE system is not tightly banded or is nonlinear, none of the methods explored thus far are attractive; the explicit Euler method is the only method encountered thus far that is easy to apply (that is, noniterative), but its stability and accuracy are both quite poor (see, e.g., Figures 10.1a and 10.2a). Further, even for tightly banded linear ODE systems, sometimes an order of accuracy higher than second is desired. It is for these reasons that 10.4 and 10.5 introduce several additional classes of time-marching methods.

10.3.2 Classifications of error in the numerical simulation of oscillatory systems

In the numerical simulation of purely oscillatory systems (that is, for $\mathbf{x}' = A\mathbf{x}$ where A has imaginary eigenvalues), two types of error can be distinguished, **amplitude error** and **phase error**. To characterize this, consider again the scalar case: recall that the exact solution of the oscillatory system

$$x' = \mathrm{i}\omega x \tag{10.19a}$$

over a single timestep of duration h is

$$x_{n+1} = e^{\mathrm{i}\omega h} x_n. \tag{10.19b}$$

Note that the amplitude of the exact solution is constant (i.e., multiplied by 1) from one timestep to the next, whereas the phase of the exact solution increases by an amount ωh from one timestep to the next. We now quantify the discrepencies of the three numerical methods presented thus far from this exact solution.

Example 10.5 Amplitude and phase error of EE. Application of the EE method to (10.19a) leads to

$$x_{n+1} = (1+\mathrm{i}\omega h)x_n \triangleq \sigma x_n \triangleq (|\sigma| e^{i\measuredangle\sigma})x_n \quad \text{where} \quad |\sigma|^2 = 1+(\omega h)^2, \quad \measuredangle\sigma = \operatorname{atan}(\omega h).$$

It follows from (B.81) that the **amplitude error**, AE, of EE is

$$AE \triangleq |\sigma| - 1 = \sqrt{1+(\omega h)^2} - 1 = \frac{1}{2}(\omega h)^2 + H.O.T.$$

It follows from (B.76) that the **phase error**, PE, of EE is

$$PE \triangleq \measuredangle\sigma - \omega h = \operatorname{atan}(\omega h) - \omega h = -(\omega h)^3/3 + H.O.T.$$

As readily verified in Figure 10.2a, it may thus be concluded for EE, when h is sufficiently small, that:

- as the leading-order term of AE is positive, the amplitude of the oscillations in the numerical solution will be *larger* than those in the exact solution;
- as the leading-order term of PE is negative, the phase of the oscillations in the numerical solution will be *smaller* (i.e., phase *lag*) than that of the exact solution; and
- the amplitude error, which is proportional to h^2, will dominate the phase error, proportional to h^3.

Example 10.6 Amplitude and phase error of IE. Application of the IE method to (10.19a) leads to

$$x_{n+1} = \frac{1}{1-\mathrm{i}\omega h} x_n \triangleq \sigma x_n \triangleq (|\sigma| e^{i\measuredangle\sigma})x_n \quad \text{where} \quad |\sigma|^2 = 1/[1+(\omega h)^2], \quad \measuredangle\sigma = \operatorname{atan}(\omega h).$$

It follows from (B.81) and (B.77) that the **amplitude error**, AE, of IE is

$$AE \triangleq |\sigma| - 1 = 1/\sqrt{1+(\omega h)^2} - 1 = -\frac{1}{2}(\omega h)^2 + H.O.T.$$

It follows from (B.76) that the **phase error**, PE, of IE is

$$PE \triangleq \measuredangle\sigma - \omega h = \operatorname{atan}(\omega h) - \omega h = -(\omega h)^3/3 + H.O.T.$$

As readily verified in Figure 10.2b, it may thus be concluded for IE, when h is sufficiently small, that:

- as the leading-order term of AE is negative, the amplitude of the oscillations in the numerical solution will be *smaller* than those in the exact solution;
- as the leading-order term of PE is negative, the phase of the oscillations in the numerical solution will be *smaller* (i.e., phase *lag*) than that of the exact solution; and
- the amplitude error, which is proportional to h^2, will dominate the phase error, proportional to h^3.

Example 10.7 Amplitude and phase error of CN. Application of the CN method to (10.19a) leads to

$$x_{n+1} = \frac{1+\mathrm{i}\omega h/2}{1-\mathrm{i}\omega h/2} x_n = \frac{A}{B}\frac{e^{\mathrm{i}\theta}}{e^{\mathrm{i}\alpha}} x_n \triangleq \sigma x_n \triangleq (|\sigma| e^{i\measuredangle\sigma}) x_n,$$

where $Ae^{\mathrm{i}\theta} = 1+\frac{\mathrm{i}\omega h}{2}$ and $Be^{\mathrm{i}\alpha} = 1-\frac{\mathrm{i}\omega h}{2}$, and thus $A^2 = B^2 = 1+(\frac{\omega h}{2})^2$, $\theta = \operatorname{atan}(\omega h/2)$, $\alpha = \operatorname{atan}(-\omega h/2)$. It follows that the **amplitude error**, AE, of CN is

$$AE \triangleq |\sigma| - 1 = \sqrt{A^2/B^2} - 1 = 0.$$

It follows from (B.76) that the **phase error**, PE, of CN is

$$PE \triangleq \measuredangle\sigma - \omega h = (\theta - \alpha) - \omega h = 2\left(\frac{\omega h}{2} - \frac{(\omega h)^3}{24} + \ldots\right) - \omega h = \frac{(\omega h)^3}{12} + H.O.T.$$

As readily verified in Figure 10.2c, it may thus be concluded for CN, when h is sufficiently small, that:

- as *AE* is zero, the amplitude of the oscillations in the numerical solution will be *identical* than those in the exact solution; and
- as the leading-order term of *PE* is positive, the phase of the oscillations in the numerical solution will be *larger* (i.e., phase *lead*) than that of the exact solution.

10.4 Non-stiff systems

10.4.1 Explicit Runge-Kutta (ERK) methods

An important class of explicit single-step time-marching methods, strictly called **explicit Runge-Kutta** (**ERK**) but often referred to simply as **Runge-Kutta** (**RK**) methods, may be written in the form

$$\boxed{\begin{aligned}\mathbf{f}_1 &= \mathbf{f}(\mathbf{x}_n, t_n + c_1 h)\\ \mathbf{f}_2 &= \mathbf{f}(\mathbf{x}_n + a_{2,1} h\mathbf{f}_1, t_n + c_2 h)\\ &\vdots\\ \mathbf{f}_s &= \mathbf{f}(\mathbf{x}_n + a_{s,1} h\mathbf{f}_1 + \ldots + a_{s,s-1} h\mathbf{f}_{s-1}, t_n + c_s h)\\ \mathbf{x}_{n+1} &= \mathbf{x}_n + h[b_1\mathbf{f}_1 + \ldots + b_{s-1}\mathbf{f}_{s-1} + b_s\mathbf{f}_s],\end{aligned}} \quad \Leftrightarrow \quad \begin{array}{c|cccc} c_1 & & & & \\ c_2 & a_{2,1} & & & \\ \vdots & \vdots & \ddots & & \\ c_s & a_{s,1} & \ldots & a_{s,s-1} & \\ \hline & b_1 & \ldots & b_{s-1} & b_s \end{array} \tag{10.20}$$

with $c_1 = 0$. Each evaluation of $\mathbf{f}(\mathbf{x},t)$ is referred to as a **stage** of the RK scheme; the RK scheme shown above has s stages. Note the convenient table at the right, called a **Butcher tableau**, which summarizes the coefficients in the RK scheme in the form

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \end{array}$$

where, for the moment, we restrict the coefficient matrix A to be strictly lower triangular. The constants $a_{i,j}$, c_i, and b_j in an RK scheme are selected to match as many terms as possible of the exact solution in the scalar case which, noting (10.1d)-(10.2), is given by:

$$\begin{aligned} x_{n+1} &= x_n + hx'_n + \frac{h^2}{2!}x''_n + \frac{h^3}{3!}x'''_n + \frac{h^4}{4!}x''''_n + O(h^5) \qquad (10.21)\\ &= x_n + h\Big\{f\Big\}_{(x_n,t_n)} + \frac{h^2}{2}\Big\{f_t + f_x f\Big\}_{(x_n,t_n)} + \frac{h^3}{6}\Big\{f_{tt} + f_t f_x + 2ff_{xt} + f_x^2 f + f^2 f_{xx}\Big\}_{(x_n,t_n)} + \frac{h^4}{24}\Big\{f_{ttt} + f_{tt}f_x\\ &\quad + 3f_{xtt}f + 3f_{xt}f_t + 5f_{xt}f_x f + f_x^2 f_t + 3f_{xx}f_t f + 3f_{xxt}f^2 + 4f_{xx}f_x f^2 + f_x^3 f + f_{xxx}f^3\Big\}_{(x_n,t_n)} + O(h^5) \end{aligned}$$

Conveniently, RK methods are **self starting**, as they don't require any information about the numerical approximation of the solution before time t_n to use [cf. the multistep methods of §10.4.2].

10.4.1.1 Basic Runge Kutta methods: RK2 and RK4

Consider first the family of all two-stage schemes of the form (10.20) in the scalar setting [noting that the second line below is simply a multidimensional Taylor-series expansion of $f(x,t)$ near (x_n,t_n)]:

$$\begin{aligned}
f_1 &= f(x_n, t_n + [c_1 h]), \\
f_2 &= f(x_n + \underbrace{[a_{2,1} h f_1]}_{=\Delta x}, t_n + \underbrace{[c_2 h]}_{=\Delta t}) = f(x_n,t_n) + \underbrace{[a_{2,1} h f(x_n,t_n)]}_{=\Delta x} f_x(x_n,t_n) + \underbrace{[c_2 h]}_{=\Delta t} f_t(x_n,t_n) + \dots, \\
x_{n+1} &= x_n + h[b_1 f_1 + b_2 f_2] \\
&= x_n + b_1 h f(x_n,t_n) + b_2 h [f(x_n,t_n) + a_{2,1} h f_x(x_n,t_n) f(x_n,t_n) + c_2 h f_t(x_n,t_n)] + \dots \\
&= x_n + [b_1 + b_2] h f(x_n,t_n) + [b_2 c_2] h^2 f_t(x_n,t_n) + [b_2 a_{2,1}] h^2 f_x(x_n,t_n) f(x_n,t_n) + \dots
\end{aligned} \tag{10.22}$$

Note that the extra terms (not shown) in the above expressions are exactly zero if $f(x,t)$ is linear in x and t, as it is in our model problem. The exact solution we seek to match with this scheme is given in (10.21). Matching coefficients in (10.22) and (10.21) to as high an order as possible, we require that

$$b_1 + b_2 = 1, \quad b_2 c_2 = 1/2, \quad b_2 a_{2,1} = 1/2.$$

Interpreting c as a free parameter, we satisfy the above equations if we take

$$a_{2,1} = c_2 = c, \quad b_2 = 1/(2c), \quad b_1 = 1 - 1/(2c).$$

Thus, the general form of the two-stage, second-order Runge-Kutta method (**RK2**) is

$$\boxed{\begin{aligned}
\mathbf{f}_1 &= \mathbf{f}(\mathbf{x}_n, t_n) \\
\mathbf{f}_2 &= \mathbf{f}(\mathbf{x}_n + c h \mathbf{f}_1, t_n + c h) \\
\mathbf{x}_{n+1} &= \mathbf{x}_n + h\left[\left(1 - \frac{1}{2c}\right)\mathbf{f}_1 + \frac{1}{2c}\mathbf{f}_2\right],
\end{aligned}} \quad \Leftrightarrow \quad \begin{array}{c|cc} 0 & & \\ c & c & \\ \hline & 1 - \frac{1}{2c} & \frac{1}{2c} \end{array} \tag{10.23}$$

For any c, the leading-order error [that is, for small h, the largest term of (10.21) that does not match the corresponding term in (10.22)] is proportional to h^3, and thus, over a specified time interval $(t_0, t_0 + T)$, **RK2 is second-order accurate**. For the model problem $x' = \lambda x$, this error is independent of c, and the RK2 scheme advances in (10.22) with a stability polynomial given by truncation of the Taylor series (10.21):

$$x_{n+1} = \sigma x_n \quad \text{where} \quad \sigma = 1 + \lambda h \mathbf{b}^T (I - \lambda h A)^{-1} \mathbf{1} = 1 + \lambda h + \frac{\lambda^2 h^2}{2}. \tag{10.24}$$

where the one vector, $\mathbf{1}$, is defined in §1.3. Over a large number of timesteps, this is stable iff $|\sigma| \le 1$; the resulting domain of stability of the RK2 method is illustrated in Figure 10.5a. For nonlinear ODEs, the coefficient of the leading-order error term of the RK2 method is a function of c, and selecting c somewhere in the range $c \in [1/2, 1]$ is appropriate to minimize this error. The choice $c = 1/2$, known as the **midpoint** method, has a clear geometric interpretation of approximating a central difference formula for the first derivative in the representation of the ODE on the interval $t \in [t_n, t_{n+1}]$, as shown in Figure 10.6a. The choice $c = 1$ is the most common so-called **predictor-corrector** method, and may be computed in the following order:

$$\begin{aligned}
\text{predictor}: \quad & \mathbf{x}^* = \mathbf{x}_n + h\mathbf{f}(\mathbf{x}_n, t_n) \\
\text{corrector}: \quad & \mathbf{x}_{n+1} = \mathbf{x}_n + \frac{h}{2}\left[\mathbf{f}(\mathbf{x}_n, t_n) + \mathbf{f}(\mathbf{x}^*, t_{n+1})\right].
\end{aligned}$$

The intermediate value $\mathbf{x}^*$ (which is simply an EE estimate of $\mathbf{x}_{n+1}$) is only "stepwise 2nd-order accurate". However, as shown above, calculation of the "corrector" (which looks roughly like a recalculation of $\mathbf{x}_{n+1}$

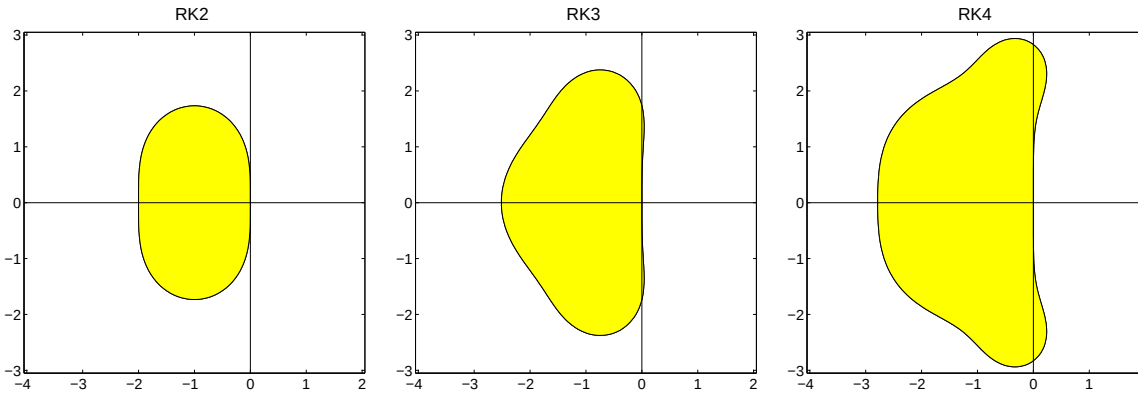


Figure 10.5: Stability regions (cf. Figure 10.3) in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using RK2 [see (10.24)], RK3 [see (10.38)], and RK4 [see (10.27)].

with CN, as shown in Figure 10.6b) results in a value for $\mathbf{x}_{n+1}$ which is "stepwise 3rd-order accurate" (and thus the scheme is globally 2nd-order accurate). Note that RK2 with $c = 1$ is also referred to as **Heun**'s method and the **modified Euler** method.

As an aside, note that if this predictor-corrector idea is taken to convergence, we arrive at the **iterative CN** method given by

$$\text{EE prediction}: \quad \mathbf{x}_0^* = \mathbf{x}_n + h\mathbf{f}(\mathbf{x}_n, t_n), \tag{10.25a}$$

$$\text{corrections}: \quad \mathbf{x}_k^* = \mathbf{x}_n + \frac{h}{2}\Big[\mathbf{f}(\mathbf{x}_n, t_n) + \mathbf{f}(\mathbf{x}_{k-1}^*, t_{n+1})\Big] \quad \text{for} \quad k = 1, 2, \ldots, \tag{10.25b}$$

which is an iterative strategy that we can march (in k) until convergence, then set $\mathbf{x}_{n+1} = \mathbf{x}_k^*$ for the numerical approximation of the solution at t_{n+1} and proceed to the next step, thereby effectively applying the CN method to a nonlinear function $\mathbf{f}$ [see also footnote 2 on page 268, and the faster-to-converge but more expensive Newton method for IRK schemes described in general in §10.5.2, taking the IRK coefficients as specified in (10.56)]. If h is sufficiently small, only a few iterations k are required to achieve convergence at each timestep.

The classical (and popular) four-stage, fourth-order Runge-Kutta method (**RK4**) is

$$\boxed{\begin{aligned} \mathbf{f}_1 &= \mathbf{f}(\mathbf{x}_n, t_n) \\ \mathbf{f}_2 &= \mathbf{f}(\mathbf{x}_n + (h/2)\mathbf{f}_1, t_{n+1/2}) \\ \mathbf{f}_3 &= \mathbf{f}(\mathbf{x}_n + (h/2)\mathbf{f}_2, t_{n+1/2}) \\ \mathbf{f}_4 &= \mathbf{f}(\mathbf{x}_n + h\mathbf{f}_3, t_{n+1}) \\ \mathbf{x}_{n+1} &= \mathbf{x}_n + h\Big[\frac{1}{6}\mathbf{f}_1 + \frac{1}{3}\mathbf{f}_2 + \frac{1}{3}\mathbf{f}_3 + \frac{1}{6}\mathbf{f}_4\Big]. \end{aligned}} \quad \Leftrightarrow \quad \begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \tag{10.26}$$

This scheme is simple, usually performs well, and is the workhorse of many ODE solvers. It also has a fairly symmetric geometric interpretation, as illustrated in Figure 10.7.

A derivation similar to that for RK2 confirms that the constants chosen in the classical RK4 scheme (10.26) indeed provide fourth-order accuracy with the stability polynomial again given by a truncated Taylor series of the exact value:

$$\sigma = 1 + \lambda h \mathbf{b}^T (I - \lambda h A)^{-1} \mathbf{1} = 1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \frac{\lambda^4 h^4}{24}. \tag{10.27}$$

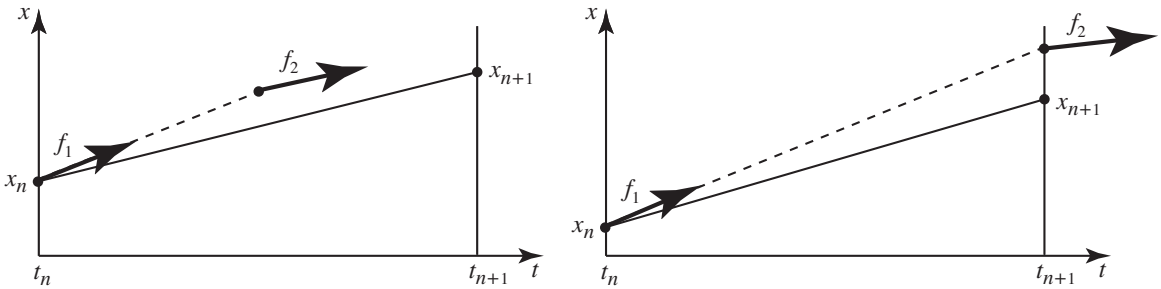


Figure 10.6: Geometric interpretation of two versions of the RK2 method: (a) the midpoint method (with $c = 1/2$), and (b) the predictor-corrector method (with $c = 1$). In the former, the initial slope (f_1) is extrapolated (dashed) to $t = t_{n+1/2}$, and the slope (f_2) recalculated there; this new slope is then used to march (solid) from t_n to t_{n+1}. In the latter, the initial slope (f_1) is extrapolated (dashed) to $t = t_{n+1}$, and the slope (f_2) recalculated there; the average of these two slopes is then used to march (solid) from t_n to t_{n+1}.

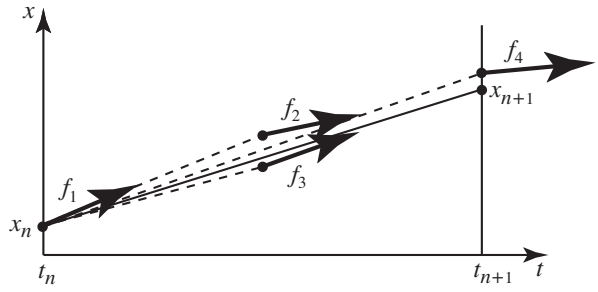


Figure 10.7: Geometric interpretation of the classical RK4 method. The interpretation of this method is analogous to the RK2 case (Figure 10.6), with the average of four evaluations $\{f_1, f_2, f_3, f_4\}$ of the slope $f(x,t)$ used to march from t_n to t_{n+1}.

Over a large number of timesteps, the method is stable iff $|\sigma| \leq 1$; the resulting domain of stability of the RK4 method is illustrated in Figure 10.5c.

Example 10.8 Simulation of the Lorenz and Rössler equations. It is in fact almost trivial to simulate low-dimensional nonlinear ODEs using an RK method, as illustrated in Figure 10.8 for the Lorenz equation

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}) \quad \text{with} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad \mathbf{f}(\mathbf{x}) = \begin{pmatrix} \sigma(x_2 - x_1) \\ -x_2 - x_1 x_3 \\ -b x_3 + x_1 x_2 - b r \end{pmatrix}, \tag{10.28}$$

and in Figure 10.9 for the Rössler equation

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}) \quad \text{with} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad \mathbf{f}(\mathbf{x}) = \begin{pmatrix} -x_2 - x_3 \\ x_1 + a x_2 \\ b + x_3(x_1 - c) \end{pmatrix}, \tag{10.29}$$

both of which were generated with the extensible test code provided in Algorithm 10.4, using the RK4 scheme as implemented in Algorithm 10.3. Note that numerical implementations of various other RK schemes are included in the *Numerical Renaissance Codebase* (hereafter abbreviated *NRC*); for brevity not all of these codes are written out here. Note also that the test code in Algorithm 10.4 is trivial to extend to other ODEs, as done in Exercise 10.1. In summary, *there is no valid justification for ever using a method as awful as EE, as you now know how to do much better with simple RK schemes!*

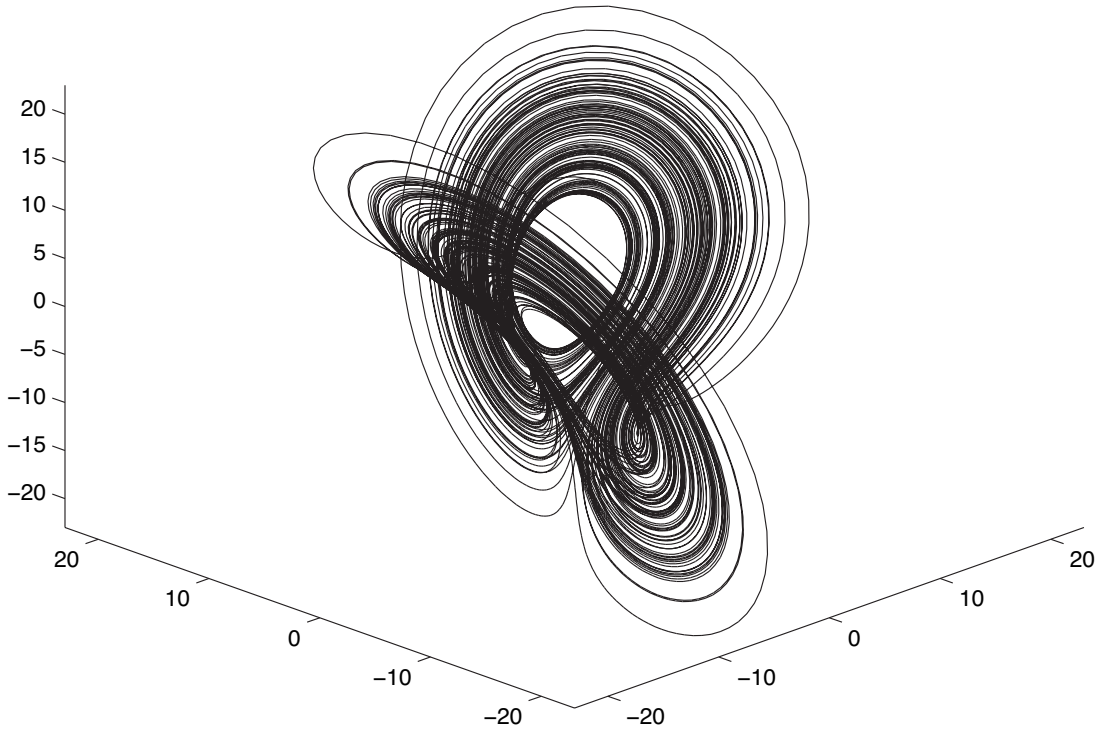


Figure 10.8: Simulation of the Lorenz equation (10.28) using RK4 with $\sigma = 4$, $b = 1$, and $r = 48$.

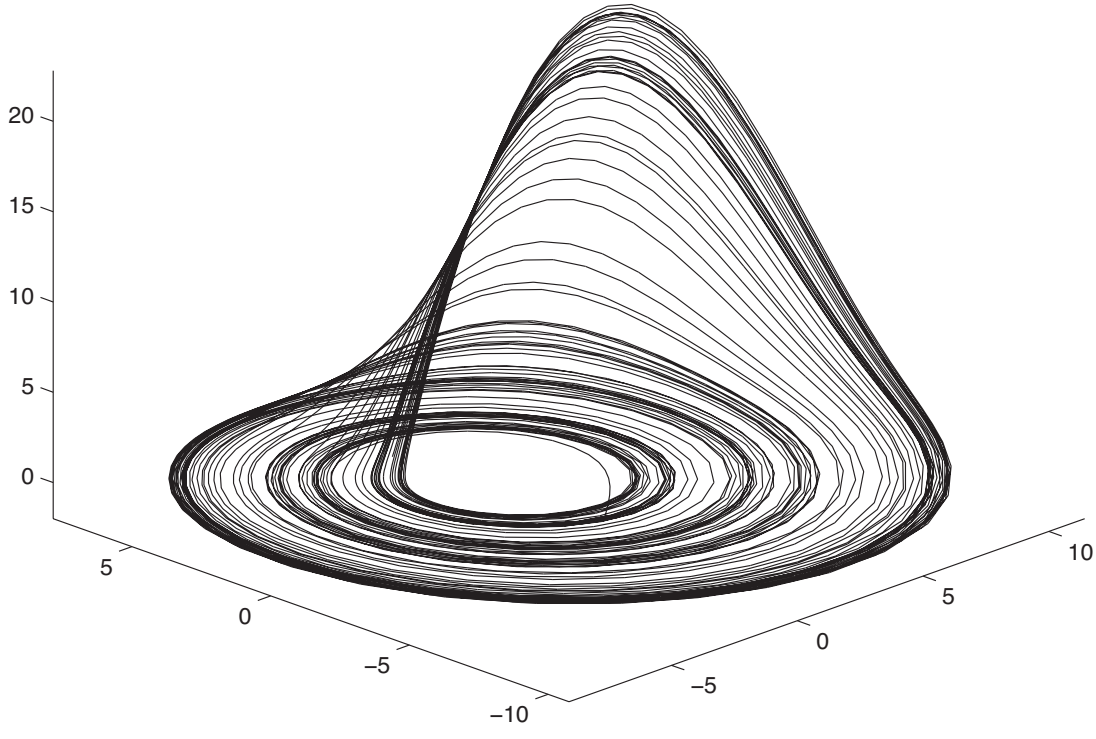


Figure 10.9: Simulation of the Rössler equation (10.29) using RK4 with $a = 0.2$, $b = 0.2$, and $c = 5.7$.

Algorithm 10.3: Simple implementation of the classical RK4 method.

View

```
function [x,t]=RK4(R,x,t,s,p,v,SimPlot)
% Simulate x'=f(x), with f implemented in R, using the classical RK4 method.
% {x,t} contains the initial {x,t} on input and the final {x,t} on output.
% The simulation parameters are s.T (time interval of simulation) and s.h (timestep).
% The function parameters p, whatever they are, are simply passed along to R.
% If v<>0, SimPlot is called at each timestep to make interactive plots.
h=s.h; if v, xold=x; end
for n=1:s.T/h;
  f1=feval(R,x,p); f2=feval(R,x+h*f1/2,p); f3=feval(R,x+h*f2/2,p); f4=feval(R,x+h*f3,p);
  x=x+h*(f1/6+(f2+f3)/3+f4/6); t=t+h;
  if v, feval(SimPlot,xold,x,t-h,t,h,h,v); xold=x; end
end
end % function RK4
```

Algorithm 10.4: Extensible codes for testing various RK methods on the Lorenz & Rossler equations.

View

```
% script <a href="matlab:RKtest">RKtest</a>
format long; while 1
  S=input('Which system (Lorenz, Rossler, or 0 to exit)? ','s');   % PROMPT FOR USER INPUT
  switch S
    case 'Lorenz',  p.sigma=4; p.b=1; p.r=48; x0=[1;1;.01];        % SET UP LORENZ SYSTEM
    case 'Rossler', p.a=.2; p.b=.2; p.c=5.7;  x0=[3;3;.1];         % SET UP ROSSLER SYSTEM
    otherwise, break
  end
  s.T=input('  Over what time interval T (try, e.g., 10)? ');
  m=input('  Which method (RK2, RK4, RK45, RKW3_2R, RK435_2R, RK435_3R, RK548_3R)? ','s');
  if m(1:3)=='RK2',s.c=input('    Value of c (1/2=midpoint, 1=predictor/corrector)? ');end
  switch m
    case {'RK45'}
      disp('    This method uses adaptive timestepping.')
      s.h       =input('    What is the initial timestep h0 (try, e.g., .01)? ');
      s.epsoverT=input('    What is the target accuracy, epsilon/T (try, e.g., .0001)? ');
    otherwise,
      disp('    This method uses uniform timestepping.')
      s.h       =input('    What is the timestep h (try, e.g., .01)? ');
  end
  v=input('  How verbose (0=fast, 1=plot attactor, 2=also plot h_n)? ');    % SET UP PLOTS
  if v>0, c=input('    Clear the plots first (y or n)? ','s'); if c=='y', close all, end
          figure(1), plot3(x0(1),x0(2),x0(3)), hold on, axis equal, view(-45,30), end
  if v>1, figure(2), plot(0,s.h), hold on, title('h_n versus t_n'), end
  [x,t]=feval(m,strcat('RHS_',S),x0,0,s,p,v,'PlotLorenzRossler')      % RUN THE SIMULATION
end, disp(' '), format short  % remember: exportfig(gcf,'Lorenzh.eps','bounds','tight');
% end script RKtest
```

View

```
function f=RHS_Lorenz(x,p)
f=[p.sigma*(x(2)-x(1)); -x(2)-x(1)*x(3); -p.b*x(3)+x(1)*x(2)-p.b*p.r];
end % function RHS_Lorenz
```

View

```
function f=RHS_Rossler(x,p)
f=[-x(2)-x(3); x(1)+p.a*x(2); p.b+x(3)*(x(1)-p.c)];
end % function RHS_Rossler
```

View

```
function PlotLorenzRossler(xo,xn,to,tn,ho,hn,v)                    % PLOTTING ROUTINE
figure(1), plot3([xo(1) xn(1)],[xo(2) xn(2)],[xo(3) xn(3)]), title(sprintf('t=%d',tn))
axis equal, if v==2, figure(2), plot([to, tn],[ho hn]), hold on, end, pause(0.00001)
end % function PlotLorenzRossler
```

Example 10.9 Simulation of the motion of an axisymmetric top. In this problem, we consider the motion of an **axisymmetric top** of mass $m = 2$ kg, moment of inertia about the axis of symmetry (about which the top spins) of $I = 0.25$ kg m^2, moments of inertia about the axes perpendicular to the axis of symmetry of $I' = 0.25$ kg m^2, and in which the distance from the center of mass to the lowest point of the top (in contact with the ground) is $L = 0.2$ m. The equations of motion governing this system, explained from first principles in §8 of Ginsberg (1998), are somewhat involved, and are summarized below without derivation.

Defining θ as the angle that the axis of the top is deflected from vertical, ψ as the angle of the (deflected) axis of the top around the vertical axis[4], and ϕ as the rotation angle of the top about its axis of symmetry, the nonlinear equations of motion of this system are

$$d^2\theta/dt^2 = (\gamma \sin\theta)/2 - (\beta_\psi - \beta_\phi \cos\theta)(\beta_\phi - \beta_\psi \cos\theta)/\sin^3\theta, \tag{10.30a}$$

$$d\psi/dt = (\beta_\psi - \beta_\phi \cos\theta)/\sin^2\theta, \tag{10.30b}$$

$$d\phi/dt = [\beta_\phi (I' \sin^2\theta + I\cos^2\theta) - \beta_\psi I \cos\theta]/(I\sin^2\theta), \tag{10.30c}$$

where $\gamma = 2mgL/I'$, and the constants β_ψ and β_ϕ defining the generalized angular momenta of the system[5], which are conserved as the system evolves, are derived from the initial conditions such that, at $t = 0$,

$$\beta_\phi = I(\dot\psi_0 \cos\theta_0 + \dot\phi_0)/I', \qquad \beta_\psi = \dot\psi_0 \sin^2\theta_0 + \beta_\phi \cos\theta_0. \tag{10.31a}$$

The top of interest in this problem is initially configured in

- slow **steady precession** at an orientation of $\theta_0 = 30^\circ$, spun up at a rate of $\dot\phi_0 = 200$ revolutions per minute about its axis of symmetry (see Figure 10.10a).

There are two possible steady precession rates consistent with these values of $\dot\phi_0$ and θ_0,

$$\dot\psi_\pm = \left[I\dot\phi_0 \pm \sqrt{I^2\dot\phi_0^2 - 2(I'-I)I'\gamma\cos\theta_0}\right]/[2(I'-I)\cos(\theta_0)]; \tag{10.31b}$$

we take $\dot\psi_0$ as the smaller of these two rates, thereby setting the constants β_ϕ and β_ψ in (10.31a).

The apex of the top is then deflected with a small impulsive force, which sends the top into a periodic precessing motion known as **nutation** (see Figures 10.10b-d); we specifically consider deflections of three different magnitudes:

- that which brings the rate of precession precisely to zero periodically, leading to the **cuspidial nutation** illustrated in Figure 10.10c,
- half this magnitude, leading to a **nutating unidirectional precession** (Figure 10.10b), and
- three times this magnitude, leading to a **looping nutation** (Figure 10.10d).

The minimum deflection when the system is in custodial motion, θ_2, is given by

$$\theta_2 = \text{acos}\,(\beta_\psi/\beta_\phi). \tag{10.32}$$

[4] Note that $\{\theta(t), \psi(t)\}$ define the orientation of the top's axis in spherical coordinates at any time t, as illustrated in Figures 10.10a-d.

[5] Defining $T = I(\dot\psi\cos\theta + \dot\phi)^2/2 + I'(\dot\psi^2\sin^2\theta + \dot\theta^2)/2$ as the kinetic energy of the motion of the system, $V = mgL\cos\theta$ as the gravitational potential energy of the system, and defining β_ϕ and β_ψ based on the (conserved) generalized angular momenta of the system, p_ϕ and p_ψ, as shown below, the total energy $E = T + V$ may be written

$$p_\phi = \frac{\partial T}{\partial\dot\phi} = I(\dot\psi\cos\theta + \dot\phi) \triangleq I'\beta_\phi, \qquad p_\psi = \frac{\partial T}{\partial\dot\psi} = I(\dot\psi\cos\theta + \dot\phi)\cos\theta + I'\dot\psi\sin^2\theta \triangleq I'\beta_\psi,$$

$$E = (I')^2\beta_\phi^2/(2I) + 0.5I'\dot\theta^2 + 0.5I'(\beta_\psi - \beta_\phi\cos\theta)^2/\sin^2\theta + mgL\cos(\theta).$$

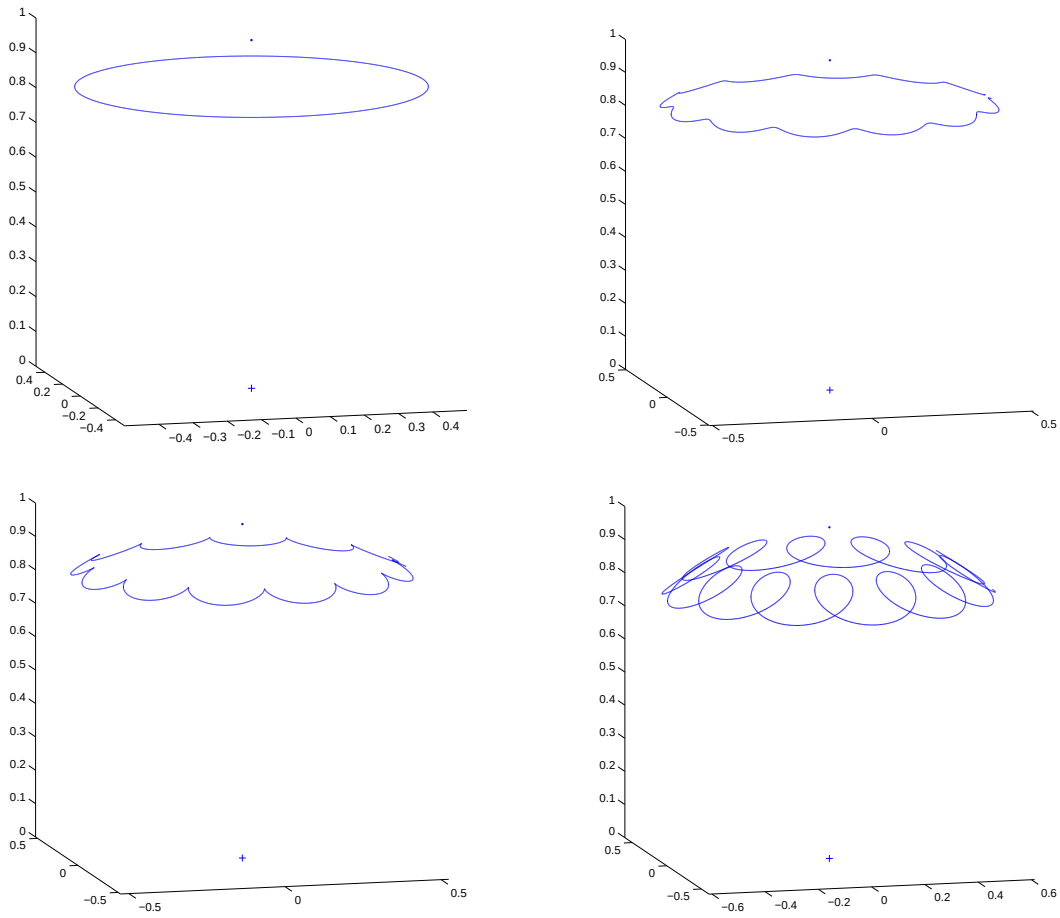


Figure 10.10: Simulation the motion of a top using RK4 in four different regimes: (a) steady precession, (b) unidirectional precession illustrating slight nutation, (c) cuspidial nutation (with the precession rate $\dot{\psi}$ periodically going to zero), and (d) looping nutation (with the precession rate $\dot{\psi}$ periodically changing sign).

The nonlinear system (10.30) may now be written in first-order form $d\mathbf{x}/dt = \mathbf{f}(\mathbf{x})$, where the state vector $\mathbf{x}$ is defined such that $\mathbf{x} = \begin{pmatrix} \theta & \dot{\theta} & \psi & \phi \end{pmatrix}^T$, and solved using RK4. For the four cases reported in Figure 10.10, the initial condition used in the simulation is $\dot{\theta}(0) = \phi(0) = \psi(0) = 0$ and $\theta(0) = \theta_0 + (\theta_2 - \theta_0)d$ where $d = 0$, 0.5, 1, and 3, respectively.

In a problem like this, in contrast with the Lorenz and Rossler systems, efficient implementation of the involved equations governing the system, as outlined on the previous page, takes dozens of lines of code (see `Top.m` and `TopTest.m` in the *NRC*), whereas the coding of the RK4 method itself is still only a couple of lines. To keep the resulting code as streamlined as possible, it is thus advised to include the (simple) RK4 time marching code directly in the main code written to solve the problem.

10.4.1.2 Adaptive Runge-Kutta: RK4/5

As shown in (10.27), the classical RK4 method introduced above has a stability polynomial given by a truncated Taylor series of the exact value. Based on this knowledge, and motivated by the algorithm for adaptive quadrature outlined in §9.4, it is straightforward to develop an algorithm for adapting the time step of an RK4 march from $t = 0$ to $t = T$ such that the error at time T is bounded by ε. [Note that we will develop this method for the scalar model problem $x' = \lambda x$, though the approach extends immediately to systems of nonlinear ODEs in a straightforward manner.]

To accomplish this, over the first timestep, march the system from the initial condition x_0 over a single timestep using the RK4 algorithm and a timestep[6] H to determine an estimate of $x(t_1)$ denoted Y. Then, march the system from the initial condition x_0 over *two* timesteps using the RK4 algorithm and a shorter timestep $h = H/2$ to determine a refined estimate of $x(t_1)$ denoted y. By comparing the stability polynomial for RK4, (10.27), with the exact solution

$$\sigma = 1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \frac{\lambda^4 h^4}{24} + \frac{\lambda^5 h^5}{120} + \frac{\lambda^6 h^6}{720} + \ldots,$$

it is seen that the error of the first estimate, Y, from the exact solution, $x(t_1)$, is

$$Y - x(t_1) = -\left[\frac{\lambda^5 H^5}{120} + \frac{\lambda^6 H^6}{720} + \ldots\right] x_0, \tag{10.33a}$$

whereas the error of the second (refined) estimate, y, is

$$y - x(t_1) = -2\left[\frac{\lambda^5 (H/2)^5}{120} + \frac{\lambda^6 (H/2)^6}{720} + \ldots\right] x_0 = -\left[\frac{\lambda^5 H^5}{16 * 120} + \frac{\lambda^6 H^6}{32 * 720} + \ldots\right] x_0. \tag{10.33b}$$

Subtracting (10.33b) from (10.33a) yields

$$Y - y \approx -\frac{15}{16} \cdot \frac{\lambda^5 H^5}{120} x_0.$$

Using this expression to reëxpress the first term on the RHS of (10.33b) leads to an estimate of the leading-order error of the locally fifth-order accurate (that is, globally fourth-order accurate) estimate of $x(t_1)$ given by y based on the calculations we have performed thus far, *without knowledge of the exact solution*:

$$|y - x(t_1)| \approx |Y - y|/15 \triangleq \delta_1. \tag{10.34a}$$

Using y as our discrete approximation of $x(t_1)$, all that remains to be done is to compare the magnitude of this error to the allowable error on this interval, $\varepsilon H/T$ (assuming, conservatively, that all errors will have the same sign), and increase or decrease the H to be used for the *next*[7] timestep appropriately. To accomplish this, noting that the error δ_1 over the first timestep, of length H_1, was dominated by a term which is proportional to H_1^5, we had over the first timestep that

$$\delta_1 = cH_1^5. \tag{10.34b}$$

Over the next timestep, we would like the leading-order error δ_2 to be a proportionate fraction of the total allowable error; that is

$$\delta_2 = cH_2^5 = \varepsilon H_2/T. \tag{10.34c}$$

[6] For the purpose of this discussion, we assume that an initial guess of a reasonable value for the initial timestep H is available; if it is not, one is easily generated via repeated application of the algorithm that follows to refine a suitable choice for H. Also, note that we use mixed case (that is, both $\{H, Y\}$ and $\{h, y\}$) in this subsection only in order to simplify the notation used.

[7] It is more conservative to *recalculate* the evolution of the system over the current timestep if H turns out to be reduced by this procedure. However, as the entire procedure is based on conservative assumptions, this is usually not necessary, especially if the appropriate values for timesteps are expected to vary slowly using this procedure, which is usually the case.

Algorithm 10.5: Straightforward implementation of the RK4/5 method with adaptive time stepping.

View

```
function [x,t]=RK45(R,x,t,s,p,v,SimPlot)
% Simulate x'=f(x), with f implemented in R, using the adaptive RK4/5 method.
% {x,t} contains the initial {x,t} on input and the final {x,t} on output.
% The simulation parameters are s.T (time interval of simulation), s.h0 (initial timestep),
% and s.epsoverT (accuracy).
% The function parameters p, whatever they are, are simply passed along to R.
% If v<>0, SimPlot is called at each timestep to make interactive plots.
H=s.h; if v, xold=x; told=t; end
while t<s.T, h=H/2;
  f1=feval(R,x,p); f2=feval(R,x+H*f1/2,p); f3=feval(R,x+H*f2/2,p); f4=feval(R,x+H*f3,p);
  X=x+H*(f1/6+(f2+f3)/3+f4/6);       % calculate X using one RK4 step with timestep H

  f1=feval(R,x,p); f2=feval(R,x+h*f1/2,p); f3=feval(R,x+h*f2/2,p); f4=feval(R,x+h*f3,p);
  x=x+h*(f1/6+(f2+f3)/3+f4/6);
  f1=feval(R,x,p); f2=feval(R,x+h*f1/2,p); f3=feval(R,x+h*f2/2,p); f4=feval(R,x+h*f3,p);
  x=x+h*(f1/6+(f2+f3)/3+f4/6);       % calculate x using two RK4 steps with timestep h=H/2;

  delta=norm(x-X,1)/15;                          % estimate error of new x and use that to
  x=(x*16-X)/15; t=t+H;                          % update old x using fifth-order formula
  H=min(H*(H*s.epsoverT/delta)^(1/4),s.T-t); % update H based on error estimate
  if v, feval(SimPlot,xold,x,told,t,2*h,H,v); xold=x; told=t; end
end
end % function RK45
```

Assuming the (unknown) coefficient c is approximately constant from the first step to the next, we may combine these two equations to eliminate c, thus determining an explicit formula for H_2:

$$H_2 = H_1\left(\frac{H_1\varepsilon}{T\delta_1}\right)^{1/4}. \tag{10.34d}$$

This is the value of H that is used at the second timestep. We proceed over the subsequent timesteps in an analogous fashion.

Note that a higher-order accurate approximation of $x(t_1)$ is available by combining (10.33a) and (10.33b) in such a way as to eliminate the fifth-order error altogether: taking $16/15$ times (10.33b) minus $1/15$ times (10.33a) results in a new approximation of $x(t_1)$, denoted x_1, such that

$$x_1 \triangleq \left[\frac{16}{15}y - \frac{1}{15}Y\right] = x(t_1) + \frac{\lambda^6 H_1^6}{30*720} + \ldots = x(t_1) + O(H_1^6).$$

The approximation x_1 defined above is locally sixth-order accurate (that is, globally fifth-order acurate); however, an estimate of the error of x_1 is not readily available. Nevertheless, since x_1 is a refined estimate of the exact value $x(t_1)$, and it is very easy to determine x_1 from y and Y, we may use x_1 instead of y as our discrete approximation of $x(t_1)$ in the above procedure. Doing such will generally only increase the accuracy of our solution, albeit an unknown amount. The resulting procedure, implemented in Algorithm 10.5, is based on a globally fifth-order scheme to march from one timestep to the next, with the timesteps h calculated based on error estimates of the RK4 steps upon which this fifth-order scheme is derived; it is thus referred to as an **RK4/5** scheme. Other adaptive RK schemes may be built up analogously; however, the simple scheme outlined above is competitive with all of them for most problems. The evolution of the timesteps H_n selected by this procedure when applied to the Lorenz problem discussed previously is plotted in Figure 10.11; note that a minimum of $H_n = 0.01$ is used for some (sensitive) steps in the simulation, though values of H_n almost an order of magnitude larger are adequate for other (less sensitive) steps in the simulation, thereby accelerating the execution of the algorithm over a given simulation window $[0,T]$.

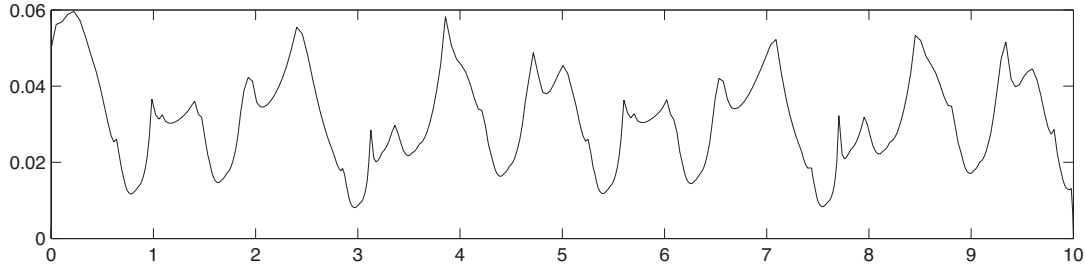


Figure 10.11: Timesteps selected by the adaptive RK4/5 scheme, implemented in Algorithm 10.5, versus time, when applied to the Lorenz system (10.28). Note that timesteps are automatically made smaller when the state moves through the most sensitive regions of the attractor (near the bottom center of Figure 10.8).

10.4.1.3 Low-storage Runge-Kutta: RKW3, RK435, and RK548

Numerical algorithms with minimal memory requirements are essential in a variety of computational grand challenge problems requiring millions or even billions of state variables. Even in smaller problems, numerical algorithms with minimal memory requirements are beneficial in order to fit a given problem into the fastest possible level of cache memory. As discussed further in §12, effective cache usage is such an important factor in the execution of numerical codes on modern CPUs that one is often willing to calculate a substantial number of extra floating point operations per timestep if one can significantly streamline memory usage. Thus, we now examine how explicit Runge-Kutta methods of the standard form (10.20), with additional constraints on the elements of the Butcher tableau, may be implemented with reduced memory usage.

In **two-register Runge-Kutta schemes** (**RK[2R]**), we restrict the elements of A in the second subdiagonal and below to equal the corresponding elements of $\mathbf{b}$, which facilitates implementation using just two registers:

$$
\begin{array}{c|cccccc}
0 & & & & & & \\
c_2 & a_{2,1} & & & & & \\
c_3 & b_1 & a_{3,2} & & & & \\
c_4 & b_1 & b_2 & a_{4,3} & & & \\
\vdots & \vdots & \vdots & \ddots & \ddots & & \\
c_s & b_1 & b_2 & \cdots & b_{s-2} & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-2} & b_{s-1} & b_s
\end{array}
\quad \Rightarrow \quad
\left\{
\begin{array}{l}
\texttt{for } i = 1:s \\
\quad \texttt{if } i = 1 \\
\quad\quad \mathbf{y} \leftarrow \mathbf{x} \\
\quad \texttt{else} \\
\quad\quad \mathbf{y} \leftarrow \mathbf{x} + (a_{i,i-1} - b_{i-1})\, h\, \mathbf{y} \\
\quad \texttt{end} \\
\quad \mathbf{y} \leftarrow \mathbf{f}(\mathbf{y}, t_n + c_i h) \\
\quad \mathbf{x} \leftarrow \mathbf{x} + b_i h \mathbf{y} \\
\texttt{end}
\end{array}
\right. \tag{10.35}
$$

In **three-register Runge-Kutta schemes** (**RK[3R]**), we restrict the elements of A in the third subdiagonal and below to equal the corresponding elements of $\mathbf{b}$, which facilitates implementation using three registers:

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
c_2 & a_{2,1} & & & & & & \\
c_3 & a_{3,1} & a_{3,2} & & & & & \\
c_4 & b_1 & a_{4,2} & a_{4,3} & & & & \\
c_5 & b_1 & b_2 & a_{5,3} & a_{5,4} & & & \\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & & \\
c_s & b_1 & b_2 & \cdots & b_{s-3} & a_{s,s-2} & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-3} & b_{s-2} & b_{s-1} & b_s
\end{array}
\quad \Rightarrow \quad
\left\{
\begin{array}{l}
\texttt{for } i = 1:s \\
\quad \texttt{if } i = 1 \\
\quad\quad \mathbf{z} \leftarrow \mathbf{x}, \;\; \mathbf{y} \leftarrow \mathbf{x} \\
\quad \texttt{else} \\
\quad\quad \mathbf{z} \leftarrow \mathbf{y} + a_{i,i-1} h \mathbf{z} \\
\quad\quad \texttt{if } i < s \\
\quad\quad\quad \mathbf{y} \leftarrow \mathbf{x} + \dfrac{a_{i+1,i-1} - b_{i-1}}{a_{i,i-1}} (\mathbf{z} - \mathbf{y}) \\
\quad\quad \texttt{end} \\
\quad \texttt{end} \\
\quad \mathbf{z} \leftarrow \mathbf{f}(\mathbf{z}, t_n + c_i h) \\
\quad \mathbf{x} \leftarrow \mathbf{x} + b_i h \mathbf{z} \\
\texttt{end}
\end{array}
\right. \tag{10.36}
$$

Verification that these reduced-storage implementations of the RK[2R] and RK[3R] classes of schemes are equivalent to the full-storage implementation given in (10.20) is straightforward (see Exersice 10.3). Note that every operation after the first (trivial) assignments in the reduced-storage implementations either updates an existing vector or overwrites the memory of an existing vector which is not used later in the timestep. Amazingly, with an increasing number of stages s, such schemes can achieve quite high orders of accuracy, even though they do not have increased memory requirements (indeed, two-register RK schemes require no more memory than does the simple and terrible EE method!). RK schemes may be limited to use 4 registers, 5 registers, etc., in an analogous fashion (see Exercise 10.4, §10.5.1), and Exercise 10.4).

It is important to point out that extra caution is required when computing a complicated vector function **in place** in computer memory [that is, in a manner which immediately overwrites the data upon which it depends, such as $\mathbf{y} \leftarrow \mathbf{f}(\mathbf{y})$]. Efficient strategies for doing such in-place computations can often be developed for large-scale computational problems of interest; however, such strategies must generally be developed on a case-by-case basis, based on careful scrutiny of the precise pattern of information flow during the computation of $\mathbf{f}(\mathbf{y})$ (for an example, see §13).

The popular third-order, three-stage RK[2R] method due to Wray (1986) and commonly referred to as **RKW3** is given by

$$\boxed{\begin{aligned} \mathbf{f}_1 &= \mathbf{f}(\mathbf{x}_n, t_n + c_1 h) \\ \mathbf{f}_2 &= \mathbf{f}(\mathbf{x}_n + a_{2,1} h \mathbf{f}_1, t_n + c_2 h) \\ \mathbf{f}_3 &= \mathbf{f}(\mathbf{x}_n + b_1 h \mathbf{f}_1 + a_{3,2} h \mathbf{f}_2, t_n + c_3 h) \\ \mathbf{x}_{n+1} &= \mathbf{x}_n + h[b_1 \mathbf{f}_1 + b_2 \mathbf{f}_2 + b_3 \mathbf{f}_3], \end{aligned}} \quad \Leftrightarrow \quad \begin{array}{c|ccc} 0 & & & \\ 8/15 & 8/15 & & \\ 2/3 & 1/4 & 5/12 & \\ \hline & 1/4 & 0 & 3/4 \end{array} \tag{10.37}$$

Note that this method is exactly of the restricted RK form depicted in (10.35), and thus may be implemented using just two storage variables, as shown in Algorithm 10.6. A derivation similar to that in §10.4.1.1 for the RK2 method confirms that the constants chosen above indeed provide third-order accuracy (see Exercise 10.2), with the stability polynomial arising when the method is applied to the model problem $x' = \lambda x$ again given by a truncated Taylor series of the exact value:

$$\sigma = 1 + \lambda h \mathbf{b}^T (I - \lambda h A)^{-1} \mathbf{1} = 1 + \lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6}. \tag{10.38}$$

Over a large number of timesteps, the method is stable iff $|\sigma| \leq 1$; the resulting domain of stability of the RKW3 method is illustrated in Figure 10.5b.

As a matter of interpretation (see Figure 10.12), it will be useful (in §10.5.4) to note here that the RKW3 scheme may also be rewritten as a march over three distinct substeps[8]

$$\begin{aligned} &\text{First RK substep:} && \begin{cases} \mathbf{f}_1 = \mathbf{f}(\mathbf{x}_n, t_n) \\ \mathbf{x}^* = \mathbf{x}_n + a_{2,1} h \mathbf{f}_1 \end{cases} \\ &\text{Second RK substep:} && \begin{cases} \mathbf{f}_2 = \mathbf{f}(\mathbf{x}^*, t_n + c_2 h) \\ \mathbf{x}^{**} = \mathbf{x}^* + a_{3,2} h \mathbf{f}_2 + \zeta_2 h \mathbf{f}_1 \end{cases} \\ &\text{Third RK substep:} && \begin{cases} \mathbf{f}_3 = \mathbf{f}(\mathbf{x}^{**}, t_n + c_3 h) \\ \mathbf{x}_{n+1} = \mathbf{x}^{**} + b_3 h \mathbf{f}_3 + \zeta_3 h \mathbf{f}_2 \end{cases} \end{aligned} \tag{10.39}$$

where $\zeta_2 = b_1 - a_{2,1} = -17/60$ and $\zeta_3 = b_2 - a_{3,2} = -5/12$.

[8] Unlike the form given in (10.35) [noting (10.37)], the form of RKw3 given in (10.39) is not a two-register scheme as written.

Algorithm 10.6: Simple implementation of the 2-register RKW3 method.

View

```
function [x,t]=RKW3_2R(R,x,t,s,p,v,SimPlot)
% Simulate x'=f(x), with f implemented in R, using the 2-register RK3 method by Wray (1986).
% {x,t} contains the initial {x,t} on input and the final {x,t} on output.
% The simulation parameters are s.T (time interval of simulation) and s.h (timestep).
% The function parameters p, whatever they are, are simply passed along to R.
% If v<>0, SimPlot is called at each timestep to make interactive plots.
a21=8/15; a32=5/12; b1=1/4; b3=3/4; h=s.h; if v, xold=x; told=t; end
for n=1:s.T/h          % Note: if v=0, entire computation is done in just 2 registers, {x,y}.
                       y=feval(R,x,p); x=x+b1*h*y;
  y=x+(a21-b1)*h*y; y=feval(R,y,p); % simplifications since b2=0
  y=x+(a32   )*h*y; y=feval(R,y,p); x=x+b3*h*y; t=t+h;
  if v, feval(SimPlot,xold,x,told,t,h,h,v); xold=x; told=t; end
end
end % function RKW3_2R
```

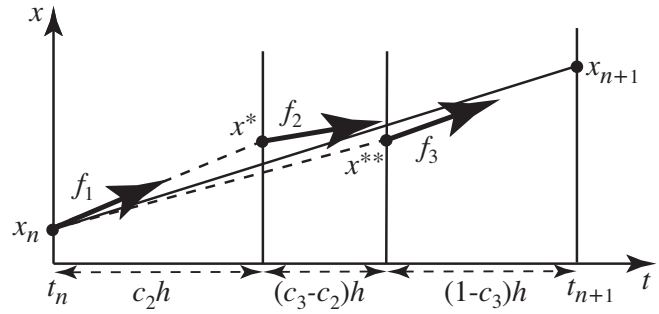


Figure 10.12: Geometric interpretation of the RKW3 method. The first interpretation of this method is analogous to the RK2 and RK4 methods (see Figures 10.6 and 10.7). A second interpretation is provided by (10.39); that is, as three distinct steps, from x_n to x^*, from x^* to x^{**}, and from x^{**} to x_{n+1}.

Kennedy, Carpenter, & Lewis (2000) optimized over the coefficients of the forms given in (10.35) and (10.36) to maximize the size of the stability region in the complex plane λh while simultaneously minimizing various combinations of metrics characterizing the errors of the schemes developed. They denote the schemes so optimized as RK$q(p)s$[rR+]X, where q is the overall order of the reduced-storage RK scheme, $p = q - 1$ is the (reduced) order of an **embedded RK scheme**[9], s is the number of stages computed at each timestep, r is the number of registers used[10], and X is a marker denoting which combination of metrics the coefficients listed were optimized for. Two of the best overall reduced storage methods so developed, referred to here as **RK435** (denoted RK4(3)5[2R+]C by KCL2000) and **RK548** (denoted RK5(4)8[3R+]C by KCL2000), are given below and implemented in the `RK435_2R.m` and `RK548_3R.m` codes provided in the *NRC*, and have stability boundaries as plotted in Figure 10.13.

[9]This **embedded RK scheme** is provided by the alternative coefficients $\hat{b}_i$ listed, applied to the same slopes $\mathbf{f}_i$ as computed by the main RK scheme. These coefficients may be used in a manner analogous to that described in (10.34a)-(10.34d) to perform adaptive timestep control, with the error of the embedded scheme estimated to be $\delta = \|\mathbf{x}_{n+1} - \hat{\mathbf{x}}_{n+1}\|$ where $\mathbf{x}_{n+1} = \mathbf{x}_n + h[b_1\mathbf{f}_1 + \ldots + b_s\mathbf{f}_s]$ and $\hat{\mathbf{x}}_{n+1} = \mathbf{x}_n + h[\hat{b}_1\mathbf{f}_1 + \ldots + \hat{b}_s\mathbf{f}_s]$, and assuming $\delta = ch^p$ [cf. (10.34a) and (10.34b)]. As in §10.4.1.2, following this approach, the timestep is calculated for the lower-order scheme, but is then applied to march the higher-order scheme, which is conservative. For implementation, see Exercise 10.5.

[10]Note that r registers are used by the scheme assuming that adaptive timestepping is *not* implemented; the number of registers used is larger if adaptive timestepping is incorporated; thus the + symbol indicated on the storage requirements specified in brackets in the name of the scheme.

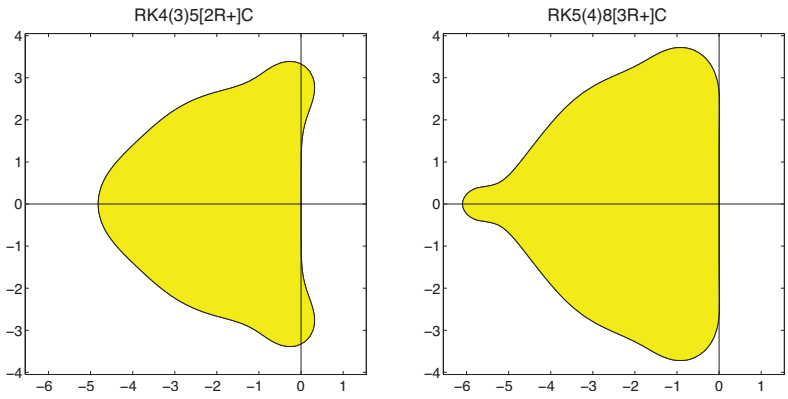


Figure 10.13: Stability regions (cf. Figures 10.3 and 10.5) in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using (left) the 2-register **RK435** scheme and (right) the 3-register **RK548** scheme.

A fourth-order, five-stage RK[2R] method, **RK435**, is given by (10.35) with $s = 5$ and

$$a_{2,1} = \tfrac{970286171893}{4311952581923},\ a_{3,2} = \tfrac{6584761158862}{12103376702013},\ a_{4,3} = \tfrac{2251764453980}{15575788980749},\ a_{5,4} = \tfrac{26877169314380}{34165994151039},$$
$$b_1 = \tfrac{1153189308089}{22510343858157},\ b_2 = \tfrac{1772645290293}{4653164025191},\ b_3 = -\tfrac{1672844663538}{4480602732383},\ b_4 = \tfrac{2114624349019}{3568978502595},\ b_5 = \tfrac{5198255086312}{14908931495163}.$$

Alternative b_i coefficients, providing an embedded third-order scheme upon which adaptive timestepping may be based, are given by

$$\hat{b}_1 = \tfrac{1016888040809}{7410784769900},\ \hat{b}_2 = \tfrac{11231460423587}{58533540763752},\ \hat{b}_3 = -\tfrac{1563879915014}{6823010717585},\ \hat{b}_4 = \tfrac{606302364029}{971179775848},\ \hat{b}_5 = \tfrac{1097981568119}{3980877426909}.$$

A fifth-order, eight-stage RK[3R] method, **RK548**, is given by (10.36) with $s = 8$ and

$$a_{2,1} = \tfrac{141236061735}{3636543850841},\ a_{3,2} = \tfrac{7367658691349}{25881828075080},\ a_{4,3} = \tfrac{6185269491390}{13597512850793},\ a_{5,4} = \tfrac{2669739616339}{18583622645114},$$
$$a_{6,5} = \tfrac{421589922673 37}{9664249073111},\ a_{7,6} = \tfrac{970532350048}{4459675494195},\ a_{8,7} = \tfrac{1415616989537}{7108576874996},\ a_{3,1} = -\tfrac{343061178215}{2523150225462},$$
$$a_{4,2} = -\tfrac{4057757969325}{18246604264081},\ a_{5,3} = \tfrac{1415180642415}{13311741862438},\ a_{6,4} = -\tfrac{93461894168145}{25333855312294},\ a_{7,5} = \tfrac{7285104933991}{14106269434317},\ a_{8,6} = -\tfrac{4825949463597}{16828400578907},$$
$$b_1 = \tfrac{514862045033}{4637360145389},\ b_2 = b_3 = b_4 = 0,\ b_5 = \tfrac{2561084526938}{7959061818733},\ b_6 = \tfrac{4857652849}{7350455163355},\ b_7 = \tfrac{1059943012790}{2822036905401},\ b_8 = \tfrac{2987336121747}{15645656703944}.$$

Alternative b_i coefficients, providing an embedded fourth-order scheme upon which adaptive timestepping may be based, are given by

$$\hat{b}_1 = \tfrac{1269299456316}{16631323494719},\ \hat{b}_2 = 0,\ \hat{b}_3 = \tfrac{2153976949307}{2236402878670 8},\ \hat{b}_4 = \tfrac{2303038467735}{18680122447354},\ \hat{b}_5 = \tfrac{7354111305649}{15643939971922},$$
$$\hat{b}_6 = \tfrac{768474111281}{10081205039574},\ \hat{b}_7 = \tfrac{3439095334143}{10786306938509},\ \hat{b}_8 = -\tfrac{3808726110015}{23644487528593}.$$

The 2-register RKW3 method described previously, as well as the 2-register RK435 and 3-register RK548 schemes described above, are good default choices for a wide variety nonstiff problems in which storage constraints are a critical issue.

Note finally that the coefficients in the two schemes listed above were found in Kennedy, Carpenter, & Lewis (2000) by numerical solution of the corresponding sets of nonlinear equations resulting in the desired order and the largest possible domain of stability while minimizing relevant error measures. Though perhaps a bit aesthetically jarring, the fact that these coefficients are not simple in form is not in itself a particular disadvantage when implementing these algorithms, as illustrated by the straightforward `RK435_2R.m` and `RK548_3R.m` codes provided in the *NRC*.

AB1 (EE)	$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}_n$
AB2	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{3}{2}h\mathbf{f}_n - \frac{1}{2}h\mathbf{f}_{n-1}$
AB3	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{23}{12}h\mathbf{f}_n - \frac{16}{12}h\mathbf{f}_{n-1} + \frac{5}{12}h\mathbf{f}_{n-2}$
AB4	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{55}{24}h\mathbf{f}_n - \frac{59}{24}h\mathbf{f}_{n-1} + \frac{37}{24}h\mathbf{f}_{n-2} - \frac{9}{24}h\mathbf{f}_{n-3}$
AB5	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{1901}{720}h\mathbf{f}_n - \frac{2774}{720}h\mathbf{f}_{n-1} + \frac{2616}{720}h\mathbf{f}_{n-2} - \frac{1274}{720}h\mathbf{f}_{n-3} + \frac{251}{720}h\mathbf{f}_{n-4}$
AB6	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{4277}{1440}h\mathbf{f}_n - \frac{7923}{1440}h\mathbf{f}_{n-1} + \frac{9982}{1440}h\mathbf{f}_{n-2} - \frac{7298}{1440}h\mathbf{f}_{n-3} + \frac{2877}{1440}h\mathbf{f}_{n-4} - \frac{475}{1440}h\mathbf{f}_{n-5}$

Table 10.1: The (explicit) Adams-Bashforth (AB) formulae.

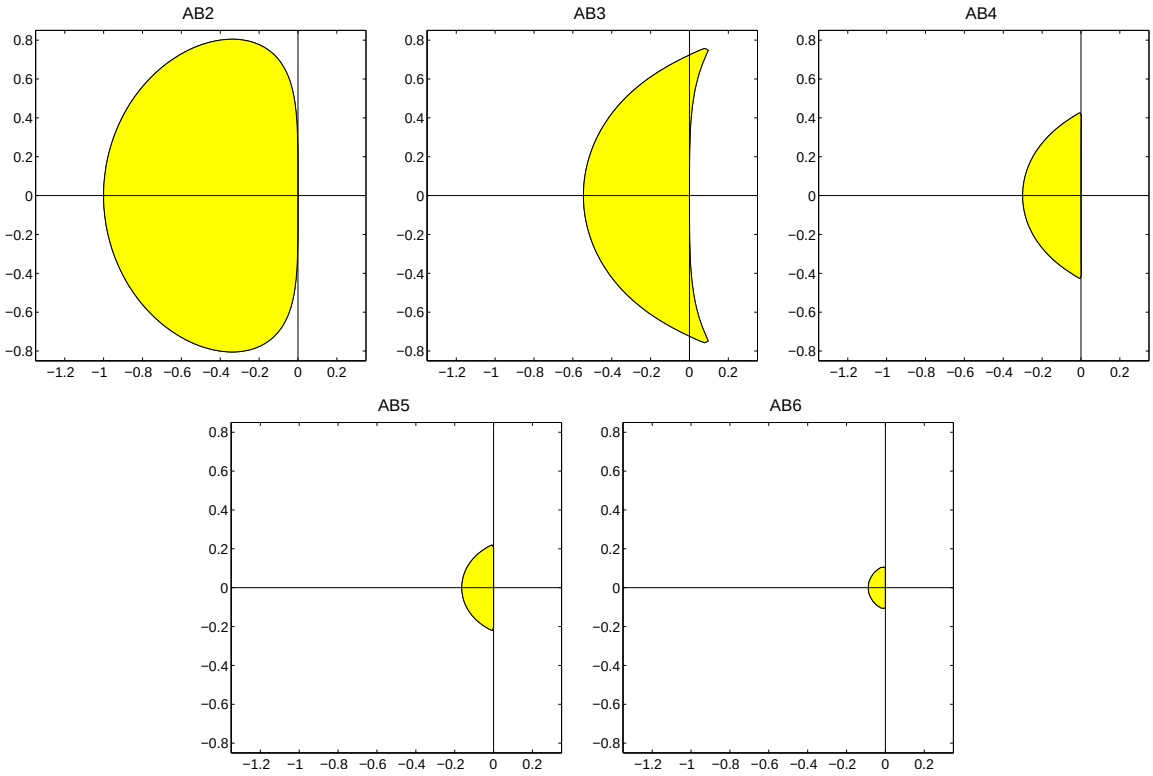


Figure 10.14: Stability regions in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using AB2, AB3, AB4, AB5, and AB6.

10.4.2 Linear Multistep Methods (LMMs)

The broad class of all **Linear Multistep Methods** (LMMs) may be written in the form

$$\boxed{\mathbf{x}_{n+1} + \sum_{i=1}^{q} \alpha_i \mathbf{x}_{n+1-i} = h \sum_{i=0}^{r} \beta_i \mathbf{f}(\mathbf{x}_{n+1-i}, t_{n+1-i}).} \tag{10.40}$$

If $\{\mathbf{x}_n, \mathbf{x}_{n-1}, \ldots\}$ are known, (10.40) may be solved for $\mathbf{x}_{n+1}$; if $\beta_0 = 0$, the resulting formula for $\mathbf{x}_{n+1}$ is explicit, otherwise it is implicit. For efficiency, recent values of $\mathbf{f}(\mathbf{x}_i, t_i)$ may be stored and reused at later timesteps.

AM1 (IE)	$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}_{n+1}$
AM2 (CN)	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{1}{2}h\mathbf{f}_{n+1} + \frac{1}{2}h\mathbf{f}_n$
AM3	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{5}{12}h\mathbf{f}_{n+1} + \frac{8}{12}h\mathbf{f}_n - \frac{1}{12}h\mathbf{f}_{n-1}$
AM4	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{9}{24}h\mathbf{f}_{n+1} + \frac{19}{24}h\mathbf{f}_n - \frac{5}{24}h\mathbf{f}_{n-1} + \frac{1}{24}h\mathbf{f}_{n-2}$
AM5	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{251}{720}h\mathbf{f}_{n+1} + \frac{646}{720}h\mathbf{f}_n - \frac{264}{720}h\mathbf{f}_{n-1} + \frac{106}{720}h\mathbf{f}_{n-2} - \frac{19}{720}h\mathbf{f}_{n-3}$
AM6	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{475}{1440}h\mathbf{f}_{n+1} + \frac{1427}{1440}h\mathbf{f}_n - \frac{798}{1440}h\mathbf{f}_{n-1} + \frac{482}{1440}h\mathbf{f}_{n-2} - \frac{173}{1440}h\mathbf{f}_{n-3} + \frac{27}{1440}h\mathbf{f}_{n-4}$

Table 10.2: The (implicit) Adams-Moulton (AM) formulae. When applied iteratively to a nonlinear ODE [see (10.25)], the corresponding AB formula of the same order (Table 10.1) is a convenient predictor to use, as it has an analogous information structure, thus using cache effectively, while providing a suitably accurate initial estimate, thus minimizing the number of iterations required for adequate convergence at each timestep.

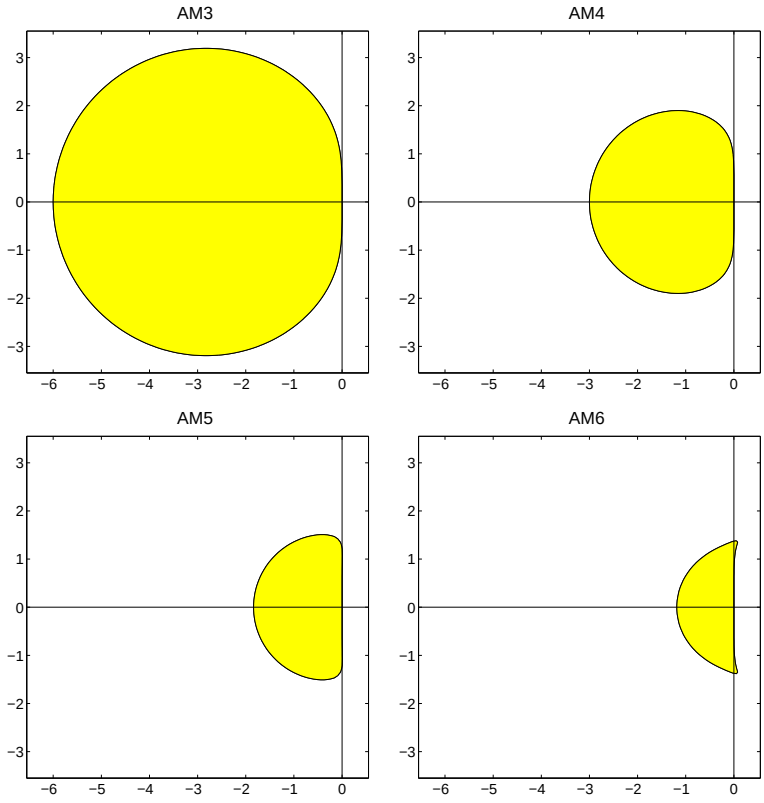


Figure 10.15: Stability regions in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using AM3, AM4, AM5, and AM6.

Roughly stated, the α_i's in (10.40) are selected such that the LHS of (10.40) approximates $c\,d\mathbf{x}/dt$ for some t, and the β_i's are selected such that the RHS approximates $c\,\mathbf{f}(\mathbf{x},t)$ at the same t, thereby approximating the ODE $d\mathbf{x}/dt = \mathbf{f}(\mathbf{x},t)$. Various special cases of (10.40) are discussed in the sections that follow.

It is a minor inconvenience that LMMs are *not* self starting (cf., e.g., the RK methods of §10.4.1); knowledge of $\mathbf{x}_0$ alone is not sufficient to determine $\mathbf{x}_1$ via (10.40) when $p = \max(q,r) > 1$. For this reason, the

Algorithm 10.7: Implementation of AB4; other AB methods, included in the *NRC*, are implemented similarly.

View

```
function [x,t,s]=AB4(R,x,t,s,p,v,SimPlot)
% Simulate x'=f(x), with f implemented in R, using the AB4 method.
% {x,t} contains the initial {x,t} on input and the final {x,t} on output.
% The simulation parameters are s.MaxTime, s.MaxSteps, s.h (timestep), and s.f,
% which contains the 3 most recent values of f on input (from a prior call to AB3/AB4),
% and the 4 most recent values of f on output (facilitating a subsequent call to AB4/AB5).
% The function parameters p, whatever they are, are simply passed along to R.
if v, xold=x; end
for n=1:min((s.MaxTime-t)/s.h,s.MaxSteps)
  s.f=[feval(R,x,p) s.f(:,1:3)];
  x=x+s.h*(55*s.f(:,1)-59*s.f(:,2)+37*s.f(:,3)-9*s.f(:,4))/24;
  t=t+s.h; if v, feval(SimPlot,xold,x,t-s.h,t,s.h,s.h,v); xold=x; end
end
end % function AB4
```

first p steps of a simulation using an LMM must be taken with an alternative method (e.g., an RK method).

For situations in which the calculation of $\mathbf{f}(\mathbf{x},t)$ is relatively slow but access to storage of the result is relatively fast, LMMs are more efficient than self-starting RK methods. In recent years, the overall execution speed of most CPUs on most (large) problems of interest has in fact become dominated by the storing and recalling of information to/from the main memory and the high-speed cache memory of the CPU and motherboard; in such situations, low-storage RK methods are often the superior choice. However, this trend might well change in the years to come as CPU architectures continue to evolve and **Graphics Processing Units** (**GPU**s) and **Application Specific Integrated Circuits** (**ASIC**s) become more widely available for high-performance computing, which might well tip the scales back in favor of LMMs for such problems.

Note also that it is straightforward to extend the Adams-Bashforth and Adams-Moulton formulae presented in the following two subsections to account for timesteps h_n that vary from one step n to the next (see, e.g., Exercise 10.12). Once this modification is made, then p'th-order and $(p-1)$'th-order multistep computations may be compared in order to perform adaptive timestepping, in a manner analogous to the adaptive timestepping suggested previously for the embedded RK methods (see Exercise 10.13).

10.4.2.1 Adams-Bashforth (AB) methods

Taking $\beta_0 = 0$, $\alpha_1 = -1$, and $q = 1$ in (10.40), and optimizing the remaining coefficients to maximize order of accuracy, gives the class of explicit methods known as **Adams-Bashforth** (**AB**) methods:

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h\sum_{i=1}^{r} \beta_i \mathbf{f}(x_{n+1-i}, t_{n+1-i}).} \tag{10.41}$$

As a particular case, taking $r = 1$ and optimizing the coefficients to maximize the order of accuracy of the resulting scheme recovers the EE method. Taking $r = 2$ gives

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\Big[\beta_1 \mathbf{f}(\mathbf{x}_n, t_n) + \beta_2 \mathbf{f}(\mathbf{x}_{n-1}, t_{n-1})\Big].$$

We now focus on this case in particular. Applying this method to the scalar model problem $dx/dt = \lambda x$ [that is, taking $f(x,t) = \lambda x$], assuming constant timestep h and a solution of the form $x_n = \sigma^n x_0$ [and thus $x_{n+1} = \sigma^{n+1} x_0$ and $x_{n-1} = \sigma^{n-1} x_0$], we find a quadratic equation for σ,

$$-\sigma^2 + (\beta_1 \lambda h + 1)\sigma + (\beta_2 \lambda h) = 0,$$

Algorithm 10.8: Implementation of iterative AM4 with an AB4 predictor; other iterative AM methods, included in the *NRC*, are implemented similarly.

View

```
function [x,t,s]=AM4iter(R,x,t,s,p,v,SimPlot)
% Simulate x'=f(x), with f implemented in R, using the iterative AM4 method with an AB4
% predictor. {x,t} contains the initial {x,t} on input and the final {x,t} on output.
% The simulation parameters are s.MaxTime, s.MaxSteps, s.MaxIters, s.h (timestep), and s.f,
% which contains the 3–4 most recent values of f on input (from a call to AM3iter/AM4iter),
% and the 4 most recent values of f on output (facilitating a call to AM4iter/AM5iter).
% The function parameters p, whatever they are, are simply passed along to R.
xold=x; for n=1:min((s.MaxTime-t)/s.h,s.MaxSteps)
  if n==1 & size(s.f,2)==3                                    % n=1: Predict with AB3
    x=xold+s.h*(23*s.f(:,1)-16*s.f(:,2)+5*s.f(:,3))/12;
  else                                                        % n>1: Predict with AB4
    x=xold+s.h*(55*s.f(:,1)-59*s.f(:,2)+37*s.f(:,3)-9*s.f(:,4))/24;
  end
  s.f(:,2:4)=s.f(:,1:3);
  for m=1:s.MaxIters, s.f(:,1)=feval(R,x,p);                  % Iteratively correct with AM4
    x=xold+s.h*(9*s.f(:,1)+19*s.f(:,2)-5*s.f(:,3)+s.f(:,4))/24;
  end, s.f(:,1)=feval(R,x,p);
  t=t+s.h; if v, feval(SimPlot,xold,x,t-s.h,t,s.h,s.h,v); end, xold=x;
end
end % function AM4iter
```

the two roots of which are given by

$$\sigma_\pm = \frac{1+\gamma \pm \sqrt{1+\varepsilon}}{2}, \quad \text{where} \quad \gamma = (\beta_1)\lambda h, \quad \varepsilon = (2\beta_1 + 4\beta_2)\lambda h + (\beta_1^2)\lambda^2 h^2.$$

By our assumed form of the solution, it follows that $x_n = \sigma_+^n x_{0,+} + \sigma_-^n x_{0,-}$. For stability of the numerical solution, we need both $|\sigma_+| \le 1$ and $|\sigma_-| \le 1$. Applying the identity (B.81), we may expand both roots in terms of powers of h. The leading-order term in the expansion in h of σ_- (referred to as a **spurious root**) is proportional to h. For small h, σ_-^n quickly decays to zero, and thus may be neglected. The leading-order terms in the expansion in h of σ_+ (referred to as the **physical root**) resemble the Taylor-series expansion of the exact solution over a single timestep:

$$\sigma_+ = 1 + \underbrace{(\beta_1+\beta_2)}_{=1}\lambda h + \underbrace{(-\beta_1\beta_2-\beta_2^2)}_{=1/2}\lambda^2 h^2 + \ldots$$

Matching coefficients with the expansion of the exact solution $\sigma = e^{\lambda h} = 1 + \lambda h + \lambda^2 h^2/2 + \ldots$, as indicated by underbraces in the above expression, we arrive at two equations for β_1 and β_2 to achieve the highest order of accuracy possible with this form:

$$\beta_1 + \beta_2 = 1, \quad -\beta_1\beta_2 - \beta_2^2 = (\beta_1+\beta_2)(-\beta_2) = 1/2.$$

It is easily verified that $\beta_1 = 3/2$ and $\beta_2 = -1/2$ satisfy these two equations. The leading-order error term of this method is proportional to h^3. Thus, over a single timestep, the scheme is "locally third-order accurate"; more significantly, over a fixed time interval $[0,T]$, the scheme is globally second-order accurate. The resulting explicit method,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\left[\frac{3}{2}\mathbf{f}(\mathbf{x}_n, t_n) - \frac{1}{2}\mathbf{f}(\mathbf{x}_{n-1}, t_{n-1})\right],$$

is thus referred to as **AB2**. The derivation of higher-order AB methods are discussed in Exercise 10.12, and are summarized in Table 10.1. Implementation is straightforward (see, e.g., Algorithm 10.7). AB methods may be viewed as multistep extensions of EE that improve the order of accuracy but degrade the domain of stability (see Figure 10.14).

10.4.2.2 Adams-Moulton (AM) methods

Taking $\beta_0 \neq 0$, $\alpha_1 = -1$, and $q = 1$ in (10.40), and optimizing the remaining coefficients to maximize order of accuracy, gives the class of implicit methods known as **Adams-Moulton** (**AM**) methods:

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h\sum_{i=0}^{r} \beta_i \mathbf{f}(\mathbf{x}_{n+1-i}, t_{n+1-i}).} \tag{10.42}$$

Taking $r = 0$ and optimizing the coefficients to maximize the accuracy of the resulting scheme recovers the IE method, whereas taking $r = 1$ and optimizing the coefficients recovers the CN method. Taking $r = 2$ gives

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\Big[\beta_0 \mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}) + \beta_1 \mathbf{f}(\mathbf{x}_n, t_n) + \beta_2 \mathbf{f}(\mathbf{x}_{n-1}, t_{n-1})\Big].$$

We now focus on this case in particular. Applying this method to the scalar model problem $dx/dt = \lambda x$ and assuming constant h and a solution of the form $x_n = \sigma^n x_0$, we find a quadratic equation for σ,

$$(\beta_0 \lambda h - 1)\sigma^2 + (\beta_1 \lambda h + 1)\sigma + (\beta_2 \lambda h) = 0,$$

the two roots of which are given by

$$\sigma_\pm = \frac{1 + \gamma \pm \sqrt{1 + \varepsilon}}{2(1 - \delta)}, \quad \text{where} \quad \gamma = (\beta_1)\lambda h, \quad \varepsilon = (2\beta_1 + 4\beta_2)\lambda h + (\beta_1^2 - 4\beta_0\beta_2)\lambda^2 h^2, \quad \delta = (\beta_0)\lambda h.$$

Applying the identities (B.81) and (B.77), we may expand both roots in terms of powers of h. By our assumed form of the solution, it follows that $x_n = \sigma_+^n x_{0,+} + \sigma_-^n x_{0,-}$. The leading-order term in the expansion in h of σ_- (a **spurious root**) is proportional to h. For small h, σ_-^n quickly decays to zero, and thus may be neglected. The leading-order terms in the expansion in h of σ_+ (the **physical root**) resemble the Taylor-series expansion of the exact solution over a single timestep:

$$\sigma_+ = 1 + \underbrace{(\beta_0 + \beta_1 + \beta_2)}_{=1}\lambda h + \underbrace{(\beta_0 + \beta_1 + \beta_2)(\beta_0 - \beta_2)}_{=1/2}\lambda^2 h^2 + \underbrace{(\beta_0^3 + \beta_0^2\beta_1 + \beta_1^2\beta_2 + \beta_0\beta_2^2 + 3\beta_1\beta_2^2 + 2\beta_2^3)}_{=1/6}\lambda^3 h^3 + \ldots.$$

Matching coefficients with the expansion of the exact solution $\sigma = e^{\lambda h} = 1 + \lambda h + \lambda^2 h^2/2 + \lambda^3 h^3/6 + \ldots$, as indicated by underbraces in the above expression, we arrive at three equations for β_0, β_1, and β_2 to achieve the highest order of accuracy possible with this form. It is easily verified that $\beta_0 = 5/12$, $\beta_1 = 8/12$, and $\beta_2 = -1/12$ satisfy these three equations. The leading-order error term of this method is proportional to h^4. Thus, over a single timestep, the scheme is "locally fourth-order accurate"; more significantly, over a fixed time interval $[0, T]$, the scheme is globally third-order accurate. The resulting method,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\Big[\frac{5}{12}\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}) + \frac{8}{12}\mathbf{f}(\mathbf{x}_n, t_n) - \frac{1}{12}\mathbf{f}(\mathbf{x}_{n-1}, t_{n-1})\Big],$$

is thus referred to as **AM3**. Higher-order AM methods are derived analogously, and are summarized in Table 10.2. Implementation in an iterative fashion [akin to that discussed in (10.25)] is straightforward (see, e.g., Algorithm 10.8); note that, when applying an AM formula to a nonlinear system, an AB formula of the same order is a natural predictor for $\mathbf{x}_{n+1}$. AM methods may be viewed as implicit multistep extensions of IE and CN that improve the order of accuracy but degrade the domain of stability (see Figure 10.15).

10.4.2.3 Consistency, spurious roots, zero stability, and convergence of LMMs

The **local error** of an LMM is the difference between the exact solution of the ODE at time t_{n+1} and the numerical result $\mathbf{x}_{n+1}$, assuming that all the previous values used by the LMM, $\{\mathbf{x}_n, \mathbf{x}_{n-1}, \ldots\}$, are exact.

An LMM is said to be **consistent** if the local error goes to zero as the timestep h approaches zero.

As illustrated by example in the previous two subsections on the AB and AM methods, the issue of consistency is a bit more delicate for LMMs than it is for single-step methods such as EE, IE, CN, and the RK methods. Recall from §10.2.1 that σ is defined as the amplification of x from one step to the next (that is, $x_{n+1} = \sigma x_n$) when a particular numerical method is applied to the model problem $x' = \lambda x$. The relationship between σ and λh for LMMs is a polynomial with multiple roots, one of which is the **physical root** which matches the Taylor-series expansion of the amplification of the exact solution, $e^{\lambda h} = 1 + \lambda h + (\lambda h)^2/2! + (\lambda h)^3/3! + \ldots$, through a given order (referred to as the **order of accuracy** of the scheme), and the others of which are **spurious roots** which must have modulus ≤ 1 (preferably, < 1) in order to not corrupt the numerical solution[11]. An LMM which is first-order accurate or better is consistent.

An LMM is said to be **zero stable** if, when applied to the model problem $x' = \lambda x$ with $\lambda = 0$ and nonzero initial conditions, the solution remains bounded. The important characterization of zero stability is essentially a special case of the characterization of stability of a time-marching method applied to the model problem $x' = \lambda x$ for arbitrary values of λ; the following fact makes this connection precise:

Fact 10.1 *An LMM, of the form given in (10.40), is zero stable if, when applied to the model problem $x' = \lambda x$ with $\lambda = 0$, all roots σ (including both the physical root and the spurious roots) lie in the unit disk (that is, $|\sigma| \leq 1$), and those roots on the unit circle (with $|\sigma| = 1$) are simple.*

An example of a candidate LMM which is consistent but *not* zero stable (and is thus unusable) is

$$\mathbf{x}_{n+1} = 2\mathbf{x}_n - \mathbf{x}_{n-1} + h\mathbf{f}(\mathbf{x}_n, t_n) - h\mathbf{f}(\mathbf{x}_{n-1}, t_{n-1}). \tag{10.43}$$

[This method was construced simply by taking the EE method (10.5) and subtracting the EE method delayed by one timestep.] Applying this method to the model problem $x' = \lambda x$, the amplification of x from one step to the next, $x_{n+1} = \sigma x_n$ (that is, $x_n = \sigma^n x_0$), is found to satisfy

$$\sigma^2 - (2 + \lambda h)\sigma + (1 + \lambda h) = 0 \quad \Rightarrow \quad \sigma_1 = 1 + \lambda h, \;\; \sigma_2 = 1.$$

Note that σ_1 is the physical root and σ_2 is the spurious root, and that $|\sigma_2| = 1$. The method is first-order accurate, and thus consistent. For the case with $\lambda = 0$, we have $\sigma_1 = \sigma_2 = 1$. In this case, taking $x_0 = 0$ and $x_1 = 1$, it follows that $x_2 = 2$, $x_3 = 3$, etc.; that is, due to the repeated roots on the unit circle, x_k grows algebraically without bound, regardless of h, and thus the LMM (10.43) is not zero stable. In contrast, the AB and AM methods given in Tables 10.1 and 10.2 above, as well as the BDF and ESD methods given in Tables 10.3 and 10.5 and the CN[ϕ] method given in (10.46), are all zero-stable LMMs.

Finally, an LMM is said to be **convergent** if the numerical approximation of the ODE using the LMM converges to the exact solution of the ODE over a given time interval $(t_0, t_0 + T)$ as the timestep h approaches zero. In addition to accuracy, convergence is the property that you ultimately care about in application. The following result (discussed further on p. 357 of Gautschi 1997) puts everything together:

Fact 10.2 (The Dahlquist Equivalence Theorem) *An LMM applied to the ODE $d\mathbf{x}(t)/dt = \mathbf{f}(\mathbf{x}(t), t)$ is* **convergent** *iff it is both* **consistent** *and* **zero stable**.

[11] Note that the numerical solution of the model problem using a p-step LMM may be written $x_n = \sigma_1^n x_{0,1} + \sigma_2^n x_{0,2} + \ldots + \sigma_p^n x_{0,p}$.

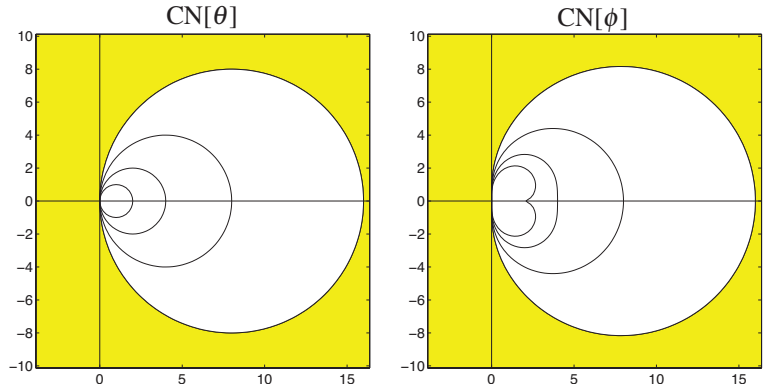


Figure 10.16: Stability regions in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using (a) the CN[θ] method [see (10.44)] for $\theta = 1/8$ (shaded), with the stability boundary also shown for $\theta = 1/4$, $\theta = 1/2$, and $\theta = 1$, the last of which corresponds to the IE method [see (10.7) and Figure 10.3b]; and (b) the CN[ϕ] method [see (10.46)] for $\phi = 1/8$ (shaded), with the stability boundary also shown for $\phi = 1/4$, $\phi = 1/2$, and $\phi = 1$, the last of which corresponds to the CN[$2h$] method [see (10.45)].

10.5 Stiff systems

As mentioned in §10.2.2, special care must be taken for ODE systems that are **stiff**, that is, linear ODEs with a broad range of eigenvalues in the LHP, or nonlinear ODEs the linearization of which on the solution trajectory of interest has a broad range of eigenvalues in the LHP. For such problems, we may sometimes simply use CN (iterative CN if the system is nonlinear), and the stability of the resulting simulation is often adequate, though the time march is only second-order in h.

Recall from the discussion in §10.2.2 that the CN scheme is A stable but not L stable. In some very stiff problems, the **far-left modes** (that is, components of the solution related to eigenvalues far out on the negative real axis) are sometimes seen to decay slowly, as these eigenvalues are, in a sense, "so far left they are right" (that is, they are near the stability boundary at the north pole in Figure B.2). In such systems, developing and using a modified CN method which is a linear combination of the CN method, which is A-stable, and a small component of some other L-stable method is sometimes a good idea to damp these far-left modes.

The (single-step) **CN[θ]** method (a.k.a. the **theta method**) blends the CN method (10.9) with a small component of the IE method (10.7), and may be written[12]

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h\big\{(1-\theta)\,[\mathbf{f}(\mathbf{x}_{n+1},t_{n+1}) + \mathbf{f}(\mathbf{x}_n,t_n)]/2 + \theta\,\mathbf{f}(\mathbf{x}_{n+1},t_{n+1})\big\}} \tag{10.44}$$

for $0 \le \theta \le 1$. The value $\theta = 0$ gives the CN method, and the value $\theta = 1$ gives the IE method; the stability boundary of the CN[θ] method for several values of θ is given in Figure 10.16a. For relatively stiff problems, $\theta \approx 1/8$ is often a suitable intermediate choice.

Unfortunately, if $\theta \neq 0$, the CN[θ] method is only first-order accurate, though the coefficient of the leading-order error term may be made small by selecting θ small. Instead of turning to IE to introduce L-stability, we now introduce a (multistep) method which we will refer to as the **CN[$2h$]** method [cf. (10.9)]:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h[3\mathbf{f}(\mathbf{x}_{n+1},t_{n+1}) + \mathbf{f}(\mathbf{x}_{n-1},t_{n-1})]/4. \tag{10.45}$$

As easily verified, this LMM is L-stable and second-order accurate, with a slightly larger coefficient on its leading-order error term than that of the CN method. By blending the CN method with a small component of

[12] Equivalently, this method is sometimes written in the form $\mathbf{x}_{n+1} = \mathbf{x}_n + h\big\{\theta\,\mathbf{f}(\mathbf{x}_{n+1},t_{n+1}) + (1-\theta)\,\mathbf{f}(\mathbf{x}_n,t_n)\big\}$ for $1/2 \le \theta \le 1$.

the CN[$2h$] method, we arrive at what we will call the **CN[ϕ]** method (a.k.a. "the" **modified CN method**)

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h\big\{(1-\phi)\,[\mathbf{f}(\mathbf{x}_{n+1},t_{n+1}) + \mathbf{f}(\mathbf{x}_n,t_n)]/2 + \phi\,[3\mathbf{f}(\mathbf{x}_{n+1},t_{n+1}) + \mathbf{f}(\mathbf{x}_{n-1},t_{n-1})]/4\big\}} \tag{10.46}$$

for $0 \le \phi \le 1$. The value $\phi = 0$ gives the CN method, and the value $\phi = 1$ gives the CN[$2h$] method; the stability boundary of the CN[ϕ] LMM for several values of ϕ is given in Figure 10.16b. For relatively stiff problems, $\phi \approx 1/8$ is a suitable intermediate choice. Note that, with $\phi > 0$, the CN[ϕ] LMM is both L-stable and second-order accurate, though its implementation requires more storage than the CN and CN[θ] methods.

The remainder of this section presents four improved classes of methods for marching stiff ODE systems:

- one based on explicit RK methods, leveraging Chebyshev polynomials, specifically designed (by sacrificing accuracy) to extend the corresponding stability region as far to the left as possible on the negative real axis in the complex plane λh,
- one based on stiffly-stable implicit RK methods (cf. §10.4.1),
- one based on stiffly-stable implicit LMMs (cf. §10.4.2), and
- one based on a suitable combination of A-stable or L-stable implicit methods (for the stiff linear terms) and easy-to-apply explicit methods (for the nonlinear terms).

10.5.1 Runge-Kutta-Chebyshev (RKC) methods†

Rather than tuning the available coefficients of an s-stage RK method (10.20) in order to achieve the highest *order* possible [with or without the constraints of a reduced-storage implementation; see (10.35) and (10.36)], we may instead constrain the coefficients of an s-stage RK method to achieve a *given* order, then tune the remaining coefficients in order to achieve the *best stability characteristics* possible. When this idea is applied to extend the stability of an s-stage RK method (see, e.g., Figure 10.5) as far to the left as possible on the negative real axis (which is generally the region of greatest concern when marching the numerical discretization of a diffusive PDE system, as discussed further in §11), it leads to what are known as Runge-Kutta-Chebyshev methods, as discussed below. For further discussion of such methods, the reader is referred to Verwer, Hundsdorfer, & Sommeijer (1990; hereafter VHS1990) and Abdulle & Medovikov (2001).

To begin, consider an explicit self-starting s-stage scheme for marching the ODE $d\mathbf{x}/dt = \mathbf{f}(\mathbf{x},t)$ in time from t_k to $t_{k+1} = t_k + h$ that is written in the form[13]

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{y}_0 + \widetilde{\mu}_1\, h\mathbf{f}_0, \\ \mathbf{y}_j &= (1-\mu_j-\nu_j)\,\mathbf{y}_0 + \nu_j\,\mathbf{y}_{j-2} + \mu_j\,\mathbf{y}_{j-1} + \widetilde{\gamma}_j\, h\mathbf{f}_0 + \widetilde{\mu}_j\, h\mathbf{f}_{j-1} \quad \text{for} \quad 2 \le j \le s, \end{aligned} \tag{10.47}$$

where $\mathbf{y}_0 = \mathbf{x}_k$, $\mathbf{y}_s = \mathbf{x}_{k+1}$, and $\mathbf{f}_j = \mathbf{f}(\mathbf{y}_j, t_k + c_j h)$. The variable $\mathbf{y}_j$ is taken to be an approximation of $\mathbf{x}(t)$ at $t = t_k + c_j h$; we thus require that $c_0 = 0$ and $c_s = 1$, and further impose an ordering condition on the substeps that $c_0 < c_1 < \ldots < c_{s-1} < c_s$. It is clear from this form (which is convenient in the discussion that follows) that each of the $\mathbf{y}_j$ depends linearly on $\mathbf{y}_0$ and the $\mathbf{f}_i$ for $0 \le i < j$; thus, this scheme is an explicit Runge Kutta method of the general form given in (10.20), where the A and $\mathbf{b}$ coefficients of the standard Butcher tableau may be expressed as simple functions of the μ_j, ν_j, $\widetilde{\gamma}_j$, and $\widetilde{\mu}_j$ coefficients, as computed in Exercise 10.9.

The polynomial that results from applying the above method to the scalar model problem $x' = \lambda x$ (that is, taking $f_j = \lambda y_j$) is a polynomial in (λh) of order j with real coefficients at each substep j; that is,

$$y_j = \sigma_j y_0 \quad \text{where} \quad \sigma_j = 1 + d_{j,1}(\lambda h) + d_{j,2}(\lambda h)^2 + \ldots + d_{j,j}(\lambda h)^j,$$

and thus $x_{k+1} = \sigma_s x_k$ where $\sigma_s(\lambda h)$, the stability polynomial of the method, is a polynomial of order s, where

$$\begin{aligned} &d_{0,1} = 0, \quad d_{1,1} = \widetilde{\mu}_1, \quad d_{j,1} = \nu_j\, d_{j-2,1} + \mu_j\, d_{j-1,1} + \widetilde{\gamma}_j + \widetilde{\mu}_j \ \text{ for } \ j > 1; \quad d_{0,2} = d_{1,2} = 0, \\ &d_{2,2} = \widetilde{\mu}_2\, d_{1,1}, \quad d_{3,2} = \widetilde{\mu}_3\, d_{2,1} + \mu_3\, d_{2,2}, \quad d_{j,2} = \widetilde{\mu}_j\, d_{j-1,1} + \mu_j\, d_{j-1,2} + \nu_j\, d_{j-2,2} \ \text{ for } \ j > 3. \end{aligned}$$

[13] Apologies for the somewhat oddly-named coefficients, which are used to be consistent with the published literature on the topic.

Comparing with the exact solution at any given substep j, for which

$$\sigma_{j,\text{exact}} = e^{c_j \lambda h} = 1 + c_j(\lambda h) + (c_j^2/2!)(\lambda h)^2 + \ldots,$$

we see that, for first-order accuracy at all substeps (including the last), we require that the c_j obey

$$c_1 = \widetilde{\mu}_1, \quad c_j = \nu_j c_{j-1} + \mu_j c_{j-1} + \widetilde{\gamma}_j + \widetilde{\mu}_j \;\text{ for }\; j > 1, \tag{10.48a}$$

and for second-order accuracy at all substeps after the first, we further require that the c_j^2 obey

$$c_2^2 = 2\widetilde{\mu}_2 c_1, \quad c_3^2 = 2\widetilde{\mu}_3 c_2 + \mu_3 c_2^2, \quad c_j^2 = 2\widetilde{\mu}_j c_{j-1} + \mu_j c_{j-1}^2 + \nu_j c_{j-2}^2 \;\text{ for }\; j > 3. \tag{10.48b}$$

To summarize, we seek to develop a family of s-stage RK schemes of the form given in (10.47), subject to either (10.48a) [for first-order accuracy at each substep], or both (10.48a) & (10.48b) [for second-order accuracy at each substep after the first]. In addition to these accuracy constraints, we would like to tune the remaining coefficients to extend the domain of stability as far to the left as possible on the negative real axis; this may be accomplished (in both the first-order and second-order cases) by first assigning σ to be a particular polynomial function of (λh), of order s, then constructing the RK scheme that corresponds to this polynomial. The polynomials that arise in this setting are Chebyshev polynomials [see §5.13] in shifted & scaled form [see (5.63)], and may be built up one substep at a time via the iteration

$$\sigma_j(\lambda h) = a_j + b_j T_j(w_0 + w_1 \lambda h) \quad \text{for} \quad 0 \le j \le s. \tag{10.49}$$

Essentially, when tuned properly, such polynomials [descriptively referred to as **equiripple** polynomials—see Figure 5.9] oscillate over the range $[-1+\varepsilon, 1-\varepsilon]$ over the largest domain possible (on the negative real axis) while matching the specified accuracy constraints; that is, while $\sigma_j(\lambda h)$ in the neighborhood of the origin has unit value, a slope of c_j, and (if imposing second-order accuracy), a curvature of c_j^2 for $j > 1$ [for further motivation on the choice of Chebyshev polynomials in this context, see van der Houwen (1977)]. As Chebyshev polynomials obey simple three-term recurrence relations [see (5.55)], they relate naturally to corresponding RK schemes of the form given in (10.47).

As suggested by VHS1990, **first-order RKC methods** satisfying (10.48a) may be developed by taking

$$w_0 = 1 + \varepsilon/s^2, \quad w_1 = T_s(w_0)/T_s'(w_0), \quad a_j = 0, \quad b_j = 1/T_j(w_0) \quad \text{for} \quad 0 \le j \le s \tag{10.50a}$$

in (10.49); for small ε, the resulting domain of stability extends to $\lambda h \approx -(2 - 4\varepsilon/3)s^2$, and the RKC scheme as formulated in (10.47) may be tuned to achieve this stability characteristic by selecting

$$\begin{gathered} \widetilde{\mu}_1 = w_1/w_0; \\ \mu_j = 2w_0 b_j/b_{j-1}, \quad \nu_j = -b_j/b_{j-2}, \quad \widetilde{\mu}_j = 2w_1 b_j/b_{j-1}, \quad \widetilde{\gamma}_j = 0 \quad \text{for} \quad 2 \le j \le s. \end{gathered} \tag{10.50b}$$

The resulting spacing of the substeps is given by $c_j = T_s(w_0)T_j'(w_0)/[T_j(w_0)T_s'(w_0)] \approx j^2/s^2$ for $0 \le j \le s$.

Similarly, **second-order RKC methods** satisfying both (10.48a) & (10.48b) may be developed by taking

$$\begin{gathered} w_0 = 1 + \varepsilon/s^2; \qquad b_j = T_j''(w_0)/[T_j'(w_0)]^2, \quad a_j = 1 - b_j T_j(w_0), \quad \text{for} \quad 2 \le j \le s; \\ w_1 = T_s'(w_0)/T_s''(w_0), \quad b_0 = b_1 = b_2, \quad a_0 = 1 - b_0, \quad a_1 = 1 - b_1 w_0 \end{gathered} \tag{10.51a}$$

in (10.49); for small ε, the resulting domain of stability extends to $\lambda h \approx -(2/3)(s^2-1)(1 - 2\varepsilon/15)$, and the RKC scheme as formulated in (10.47) may be tuned to achieve this stability characteristic by selecting

$$\begin{gathered} \widetilde{\mu}_1 = b_1/w_1; \\ \mu_j = 2w_0 b_j/b_{j-1}, \quad \nu_j = -b_j/b_{j-2}, \quad \widetilde{\mu}_j = 2w_1 b_j/b_{j-1}, \quad \widetilde{\gamma}_j = [b_{j-1}T_{j-1}(w_0) - 1]\widetilde{\mu}_j \quad \text{for} \quad 2 \le j \le s. \end{gathered} \tag{10.51b}$$

The resulting spacing of the substeps is given by $c_1 = c_2/T_2'(w_0) \approx c_2/4$, $c_j = T_s'(w_0)T_j''(w_0)/[T_j'(w_0)T_s''(w_0)] \approx (j^2-1)/(s^2-1)$ for $2 \le j \le s$.

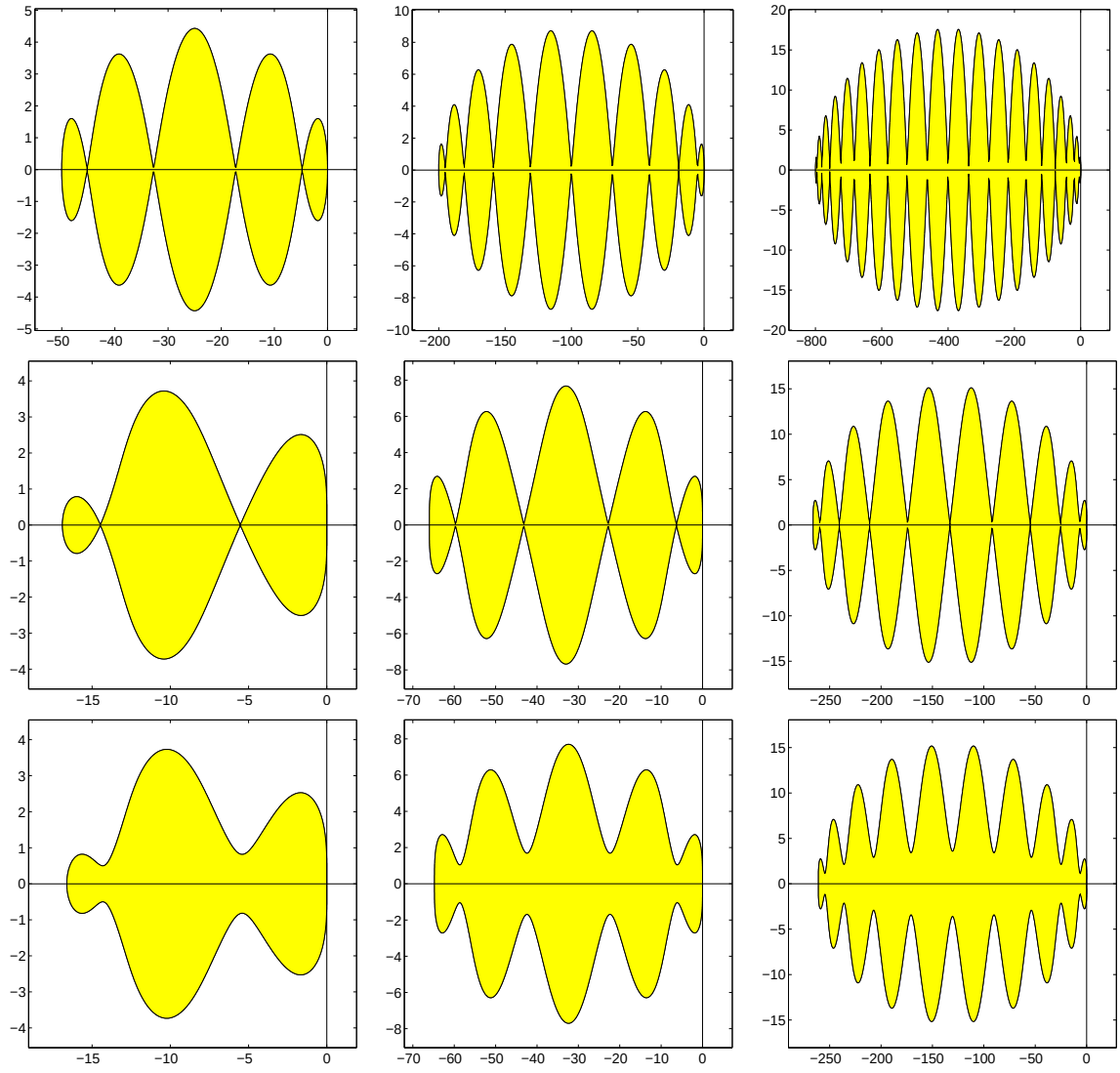


Figure 10.17: Stability regions in the complex plane λh of (top row) undamped 1st-order, with $\varepsilon = 0$, (middle row) undamped 2nd-order, with $\varepsilon = 0$, and (bottom row) damped 2nd-order, with with $\varepsilon = 0.15$, RKC methods with (left column) $s = 5$, (middle column) $s = 10$, and (right column) $s = 20$. Note horizontal scalings.

Figure 10.17 illustrates the stability contours for various incarnations of the RKC methods described above; note that 1st-order accuracy, shown for comparison purposes only, is generally not adequate for most numerical computations. The estimates cited previously of the maximum extent of the domain of stability are indeed accurate: for small ε and large s, the 1st-order schemes are stable out to about $\lambda h \approx -2s^2$, and the 2nd-order schemes are stable out to about $\lambda h \approx -2s^2/3$. This highlights the strength of RKC methods: increasing the number of stages s, while not requiring any more storage in their implementation (10.47), extends the stability of these methods on the negative real axis at a rate proportional to s^2. Note also that taking $\varepsilon = 0$ leads to several isolated points on the negative real axis where $|\sigma| = 1$, bordering on instability; increasing ε slightly, while not significantly shortening the region of stability, keeps $|\sigma|$ comfortably away from 1 until the left and right boundaries of the stability region on the negative real axis are approached.

10.5.2 Implicit Runge-Kutta (IRK) methods

Revisiting (10.20) and the Butcher tableau

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \end{array} \tag{10.52}$$

it is seen that we have so far considered explicit schemes with strictly lower triangular coefficient matrices A. If we relax this constraint, **implicit Runge Kutta** schemes may be developed with higher order and better stability for a given number of stages. Such schemes may in general be written in the form

$$\boxed{\begin{aligned} \mathbf{f}_1 &= \mathbf{f}(\mathbf{x}_n + a_{1,1}h\mathbf{f}_1 + a_{1,2}h\mathbf{f}_2 + \ldots + a_{1,s}h\mathbf{f}_s, t_n + c_1 h) \\ \mathbf{f}_2 &= \mathbf{f}(\mathbf{x}_n + a_{2,1}h\mathbf{f}_1 + a_{2,2}h\mathbf{f}_2 + \ldots + a_{2,s}h\mathbf{f}_s, t_n + c_2 h) \\ &\vdots \\ \mathbf{f}_s &= \mathbf{f}(\mathbf{x}_n + a_{s,1}h\mathbf{f}_1 + a_{s,2}h\mathbf{f}_2 + \ldots + a_{s,s}h\mathbf{f}_s, t_n + c_s h) \\ \mathbf{x}_{n+1} &= \mathbf{x}_n + h[b_1\mathbf{f}_1 + b_2\mathbf{f}_2 + \ldots + b_s\mathbf{f}_s] \end{aligned}} \quad \Rightarrow \quad \begin{array}{c|cccc} c_1 & a_{1,1} & a_{1,2} & \ldots & a_{1,s} \\ c_2 & a_{2,1} & a_{2,2} & \ldots & a_{2,s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s,1} & a_{s,2} & \ldots & a_{s,s} \\ \hline & b_1 & b_2 & \ldots & b_s \end{array} \tag{10.53}$$

The use of such implicit schemes on nonlinear problems is expensive, as it requires *iterating over the set of all s stages*, until convergence, at each timestep. Such an iteration might proceed as follows:

(1) Initialize an iteration counter $p = 0$, and estimate $\{\mathbf{f}_1^{(0)}, \ldots, \mathbf{f}_s^{(0)}\}$, at times $\{t_n + c_1 h, \ldots, t_n + c_s h\}$, using a simple explicit method.

(2) Compute the vector nonlinear functions $\mathbf{z}_1^{(p)}(\mathbf{f}_1^{(p)}, \ldots, \mathbf{f}_s^{(p)})$ through $\mathbf{z}_s^{(p)}(\mathbf{f}_1^{(p)}, \ldots, \mathbf{f}_s^{(p)})$, and the vec of the matrix formed with these s vectors as columns, $\mathbf{z}^{(p)}$ (see §1.2), as follows:

$$\begin{aligned} \mathbf{z}_1^{(p)}(\mathbf{f}_1^{(p)}, \ldots, \mathbf{f}_s^{(p)}) &= \mathbf{f}(\mathbf{x}_n + a_{1,1}h\mathbf{f}_1^{(p)} + \ldots + a_{1,s}h\mathbf{f}_s^{(p)}, t_n + c_1 h) - \mathbf{f}_1^{(p)} \\ &\vdots \\ \mathbf{z}_s^{(p)}(\mathbf{f}_1^{(p)}, \ldots, \mathbf{f}_s^{(p)}) &= \mathbf{f}(\mathbf{x}_n + a_{s,1}h\mathbf{f}_1^{(p)} + \ldots + a_{s,s}h\mathbf{f}_s^{(p)}, t_n + c_s h) - \mathbf{f}_s^{(p)} \end{aligned}$$

We will iterate, using the Newton-Raphson approach (3.4), until $\mathbf{z}_1^{(p)}$ through $\mathbf{z}_s^{(p)}$ are all close to zero.

(3) Compute the blocks $A_{ij}^{(p)}$ comprising the Jacobian matrix $A^{(p)}$,

$$A^{(p)} = \begin{bmatrix} A_{11}^{(p)} & \ldots & A_{1s}^{(p)} \\ \vdots & \ddots & \vdots \\ A_{s1}^{(p)} & \ldots & A_{ss}^{(p)} \end{bmatrix}, \qquad \text{each with elements} \qquad \left[A_{ij}^{(p)}\right]_{lm} = \partial[\mathbf{z}_i^{(p)}]_l / \partial[\mathbf{f}_j]_m.$$

(4) Defining $\mathbf{f}^{(p)} = \text{vec}\begin{bmatrix} \mathbf{f}_1^{(p)} & \ldots & \mathbf{f}_s^{(p)} \end{bmatrix}$, update $\mathbf{f}^{(p+1)} = \mathbf{f}^{(p)} + \mathbf{h}^{(n+1)}$, where $\mathbf{h}^{(p+1)}$ solves the problem $A^{(p)}\mathbf{h}^{(p+1)} = -\mathbf{z}^{(p)}$, and repeat from step (2) until convergence.

This iteration may often be significantly accelerated by calculating the Jacobian in step (3) only at the $n = 0$ iteration, then using this approximate Jacobian for all susequent iterations.

10.5.2.1 Gauss-Legendre Runge Kutta (GLRK) methods†

If A is allowed to be full in an IRK method, excellent stability and accuracy properties can be achieved with a small number of stages. The two-point, fourth-order Gauss-Legendre Runge Kutta formula, **GLRK4**, is

a generalization of the fourth-order Gauss-Legendre quadrature formula, (9.8b), to ODE systems, with the Butcher tableau

$$\begin{array}{c|cc} 1/2-\sqrt{3}/6 & 1/4 & 1/4-\sqrt{3}/6 \\ 1/2+\sqrt{3}/6 & 1/4+\sqrt{3}/6 & 1/4 \\ \hline & 1/2 & 1/2 \end{array} \tag{10.54}$$

whereas the three-point, sixth-order Gauss-Legendre Runge Kutta formula, **GLRK6**, is a generalization of the sixth-order Gauss-Legendre quadrature formula, (9.15b), to ODE systems, with the Butcher tableau

$$\begin{array}{c|ccc} 1/2-\sqrt{15}/10 & 5/36 & 2/9-\sqrt{15}/15 & 5/36-\sqrt{15}/30 \\ 1/2 & 5/36+\sqrt{15}/24 & 2/9 & 5/36-\sqrt{15}/24 \\ 1/2+\sqrt{15}/10 & 5/36+\sqrt{15}/30 & 2/9+\sqrt{15}/15 & 5/36 \\ \hline & 5/18 & 8/18 & 5/18 \end{array} \tag{10.55}$$

The GLRK4 and GLRK6 schemes are A stable with, remarkably, stability boundaries that coincide exactly with that of the exact solution (like CN). These schemes are thus, if it were not for their severe computational expense as discussed above, quite well suited to problems with purely imaginary eigenvalues.

10.5.2.2 Diagonally-Implicit Runge Kutta (DIRK) methods

Recall that ERK methods have strictly lower triangular coefficient matrices A, whereas general IRK methods have full coefficient matrices A. **Diagonally-Implicit Runge Kutta** (**DIRK**) methods partially address the severe computational expense associated with IRK methods by restricting the coefficient matrix A to be lower triangular (but not strictly lower triangular), thereby allowing the iterations for each of the $\mathbf{f}_i$ to be conducted independently. In so doing, DIRK methods still can achieve L stability, which is good, though their stability boundaries do not necessarily coincide exactly with the stability boundary of the exact solution, which is the price that is paid for eliminating the strictly upper triangular coefficients in A.

Perhaps the simplest DIRK method is the (second-order) iterative CN method presented previously, with Butcher tableau

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array} \tag{10.56}$$

As a more sophisticated example, defining γ as one of the roots of the cubic equation that comes up in the analysis,

$$6\gamma^3-18\gamma^2+9\gamma-1 \quad \Rightarrow \quad \gamma=\sqrt{2}\cos\left(\operatorname{acos}\left(\sqrt{2^3}/3\right)/3+4\pi/3\right)+1\approx 0.435866521508458,$$

and defining the additional parameters

$$a_{31}=\frac{-20\gamma^2+10\gamma-1}{4\gamma}, \qquad a_{32}=\frac{(4\gamma-1)(2\gamma-1)}{4\gamma},$$
$$b_1=\frac{24\gamma^3-36\gamma^2+12\gamma-1}{12\gamma(1-2\gamma)}, \quad b_2=\frac{12\gamma^2-6\gamma+1}{12\gamma(1-4\gamma)}, \quad b_3=\frac{6\gamma^2-6\gamma+1}{3(4\gamma-1)(2\gamma-1)},$$

a fourth-order DIRK method due to Alexander (2003) and referred to here as DIRK4 may be written

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 2\gamma & \gamma & \gamma & 0 & 0 \\ 1-2\gamma & a_{31} & a_{32} & \gamma & 0 \\ 1 & b_1 & b_2 & b_3 & \gamma \\ \hline & b_1 & b_2 & b_3 & \gamma \end{array} \tag{10.57}$$

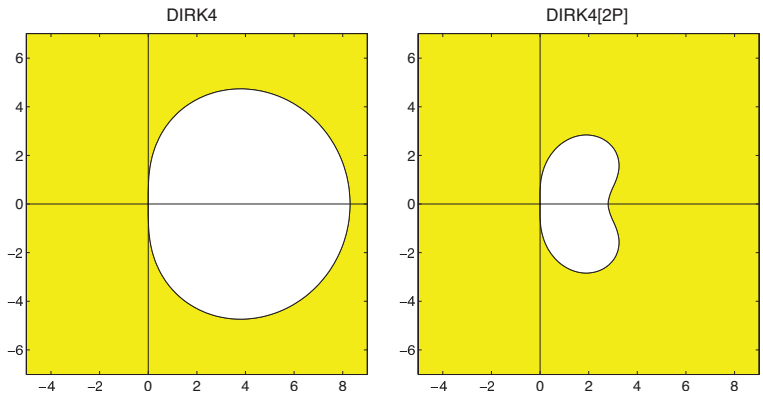


Figure 10.18: Stability regions in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using DIRK4, listed in (10.57), and DIRK4[2P], listed in (10.58).

Note that the value of $\mathbf{f}_1$ computed at the first substep of this scheme is in fact identical to the converged value of $\mathbf{f}_4$ computed at end of the previous timestep [because $A(s,:) = \mathbf{b}^T$]; this scheme is thus known as a **first-same-as-last** (**FSAL**) RK algorithm. After start-up, this DIRK scheme thus only requires the iterative computation of three stages per timestep. [Note that the DIRK interpretation of CN given in (10.56) is also an FSAL RK algorithm, with the iterative computation of one stage per timestep.]

Another class of DIRK schemes restricts the A matrix to be block lower triangular, where the $p \times p$ blocks on the main diagonal of A are themselves restricted to be diagonal. An example of such a scheme with $p = 2$, due to Jackson & Norsett (1990) and referred to here as DIRK4[2P], is

$$\begin{array}{c|cccc} 1 & 1 & 0 & 0 & 0 \\ 3/5 & 0 & 3/5 & 0 & 0 \\ 0 & 171/44 & -215/44 & 1 & 0 \\ 2/5 & -43/20 & 39/20 & 0 & 3/5 \\ \hline & 11/72 & 25/72 & 11/72 & 25/72 \end{array} \tag{10.58}$$

The advantage of this type of scheme is the manner in which it can leverage a 2-processor computer architecture. Note that $\mathbf{f}_1$ and $\mathbf{f}_2$ can be computed completely independently (via iteration) on different processors. Then, $\mathbf{f}_3$ and $\mathbf{f}_4$ can be computed completely independently (via iteration) on different processors. Finally, these four values of $\mathbf{f}_i$ may be combined to extrapolate from $\mathbf{x}_n$ to $\mathbf{x}_{n+1}$. This clever idea is extended in Exercise 10.10 to include an EE predictor at the first stage, akin to the iterative CN method in (10.56).

The stability boundaries of DIRK4 and DIRK4[2P], both of which are L-stable, are given in Figure 10.18.

10.5.3 Stiffly-stable linear multistep methods

An alternative approach to stiff systems is to use stiffly stable LMMs, two popular classes of which, **backward differentiation formulae** and **Enright second derivative** methods, are introduced below. As these methods are implicit, again, one must iterate at each timestep (in a manner similar to the iterative CN method introduced on page 281, or following a Newton-Raphson procedure as discussed on page 303) in order to apply them to nonlinear systems. Note that such iterations often require significant computational effort.

Noting the definition of zero stability in §10.4.2.3, the following two classic results, proved in Dahlquist (1956) and Dahlquist (1963), provide substantial additional insight on higher-order LMMs.

Fact 10.3 (Dahlquist's first barrier) *A zero-stable p-step LMM, with $p = \max(q,r)$ in (10.40), has order of at most $(p+1)$ if p is odd and $(p+2)$ if p is even. If the LMM is explicit, with $\beta_0 = 0$ in (10.40), then its order is at most p.*

Fact 10.4 (Dahlquist's second barrier) *All A-stable LMMs are implicit, with $\beta_0 \neq 0$ in (10.40). The highest order of an (implicit) A-stable LMM is second. Of all second-order A-stable LMMs, the CN method has the smallest error.*

That is, among other things, it is a provable fact that higher-order LMMs are not stable in the entire LHP. As a consequence, the modified stability descriptions given in §10.2.2 are sometimes useful when considering LMMs for stiff systems.

10.5.3.1 Backward differentiation formulae (BDF)

Taking $r = 0$ in (10.40) gives the class of implicit methods known as **backward differentiation formulae** (**BDF**, a.k.a. **Gear's formulae**):

$$\boxed{\mathbf{x}_{n+1} = -\sum_{i=1}^{q} \alpha_i \mathbf{x}_{n+1-i} + h\beta_0 \mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}).} \tag{10.59}$$

As a particular case, taking $q = 1$ and optimizing the coefficients to maximize the accuracy of the resulting scheme recovers the IE method. Taking $q = 2$ gives

$$\mathbf{x}_{n+1} = -\alpha_1 \mathbf{x}_n - \alpha_2 \mathbf{x}_{n-1} + h\beta_0 \mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}).$$

We now focus on this case in particular. Applying this method to the scalar model problem $dx/dt = \lambda x$ and assuming constant h and a solution of the form $x_n = \sigma^n x_0$, we find a quadratic equation for σ,

$$(1 - \beta_0 \lambda h)\sigma^2 + \alpha_1 \sigma + \alpha_2 = 0,$$

the two roots of which are given by

$$\sigma_\pm = \frac{-\alpha_1 \pm \sqrt{\gamma(1+\varepsilon)}}{2(1-\delta)}, \quad \text{where} \quad \gamma = \alpha_1^2 - 4\alpha_2, \quad \varepsilon = (4\alpha_2\beta_0/\gamma)\lambda h, \quad \delta = (\beta_0)\lambda h.$$

Applying the identities (B.81) and (B.77), we may expand both roots in terms of powers of h. By our assumed form of the solution, it follows that $x_n = \sigma_+^n x_{0,+} + \sigma_-^n x_{0,-}$. The leading-order term in the expansion in h of σ_- (a "spurious root") is proportional to h. For small h, σ_-^n quickly decays to zero, and thus may be neglected. The leading-order terms in the expansion in h of σ_+ (the "physical root") resemble the Taylor-series expansion of the exact solution over a single timestep:

$$\sigma_+ = \underbrace{\left(-\frac{\alpha_1}{2} + \frac{\sqrt{\gamma}}{2}\right)}_{=1} + \underbrace{\beta_0\left(-\frac{\alpha_1}{2} + \frac{\sqrt{\gamma}}{2}\left[1 + \frac{2\alpha_2}{\gamma}\right]\right)}_{=1}\lambda h + \underbrace{\beta_0^2\left(-\frac{\alpha_1}{2} + \frac{\sqrt{\gamma}}{2}\left[1 + \frac{2\alpha_2}{\gamma} - \frac{2\alpha_2^2}{\gamma^2}\right]\right)}_{=1/2}\lambda^2 h^2 + \ldots.$$

Matching coefficients with the exact solution $\sigma = e^{\lambda h} = 1 + \lambda h + \lambda^2 h^2/2 + \ldots$, as indicated by underbraces in the above expression, and applying the definition $\gamma = \alpha_1^2 - 4\alpha_2$, we arrive at three equations for α_1, α_2, and β_0 to achieve the highest order of accuracy possible with this form. It is easily verified that $\alpha_1 = -4/3$,

Algorithm 10.9: Implementation of iterative BDF4 with an eBDF4 predictor; other iterative BDF methods, included in the *NRC*, are implemented similarly.

View

```
function [x,t,s]=BDF4iter(R,x,t,s,p,v,SimPlot)
% Simulate x'=f(x), with f implemented in R, using the iterative BDF4 method with an eBDF4
% predictor.  t contains the initial t on input and the final t on output. x contains
% the 4 (or more) most recent values of x on input (from a call to BDF3iter/BDF4iter),
% and the 5 most recent values of x on output (facilitating a call to BDF4iter/BDF5iter).
% The simulation parameters are s.MaxTime, s.MaxSteps, s.MaxIters, s.h (timestep).
% The function parameters p, whatever they are, are simply passed along to R.
for n=1:min((s.MaxTime-t)/s.h,s.MaxSteps)
  x(:,2:5)=x(:,1:4); f=feval(R,x(:,2),p);                    % Predict with eBDF4
  x(:,1)=(-10*x(:,2)+18*x(:,3)-6*x(:,4)+x(:,5)+12*s.h*f)/3;
  for m=1:s.MaxIters, f=feval(R,x(:,1),p);                   % Iteratively correct with BDF4
    x(:,1)=(48*x(:,2)-36*x(:,3)+16*x(:,4)-3*x(:,5)+12*s.h*f)/25;
  end
  t=t+s.h; if v, feval(SimPlot,x(:,2),x(:,1),t-s.h,t,s.h,s.h,v); end
end
end % function BDF4iter
```

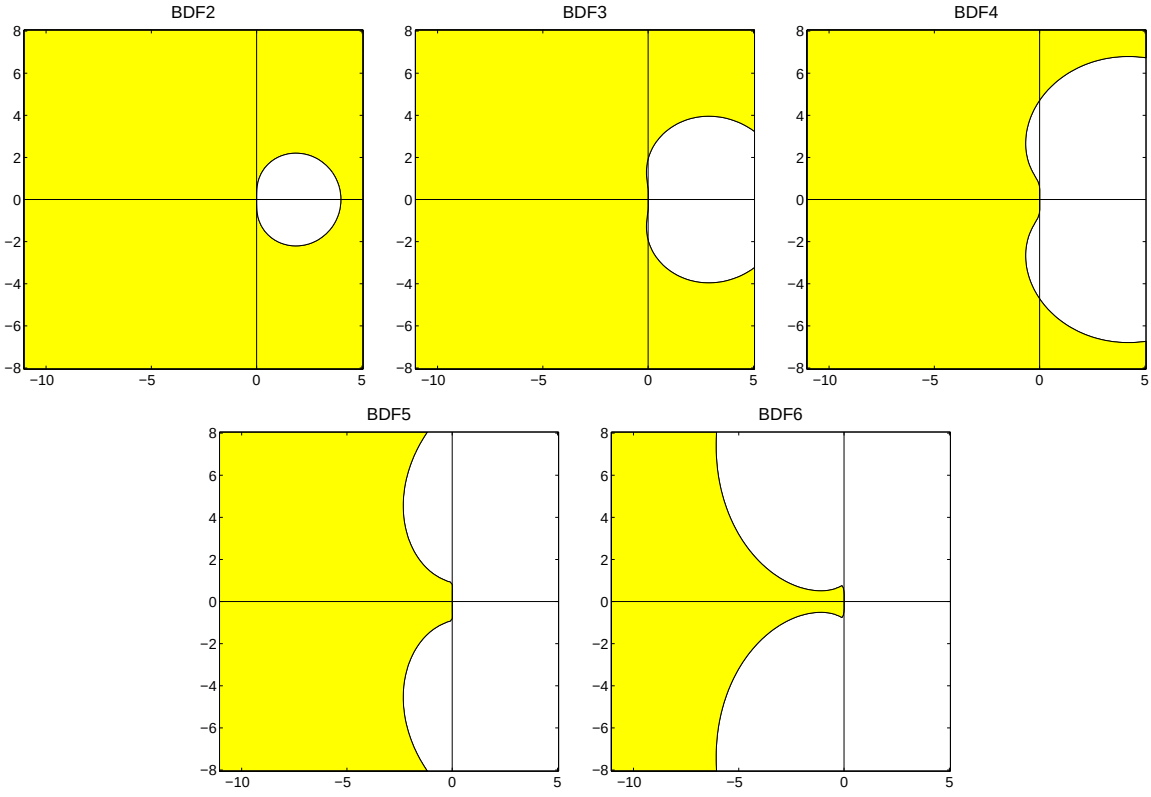


Figure 10.19: Stability regions in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using BDF2 through BDF6. In the language of §10.2.2 and Figure 10.4, BDF2 is L stable whereas BDF3-BDF6 are $A(\alpha)$ stable for $\alpha = 86.03°$, $73.35°$, $51.84°$, and $17.84°$ respectively; also, BDF2-BDF6 are stiffly stable with $D = 0$, 0.083, 0.667, 2.327, and 6.075 respectively.

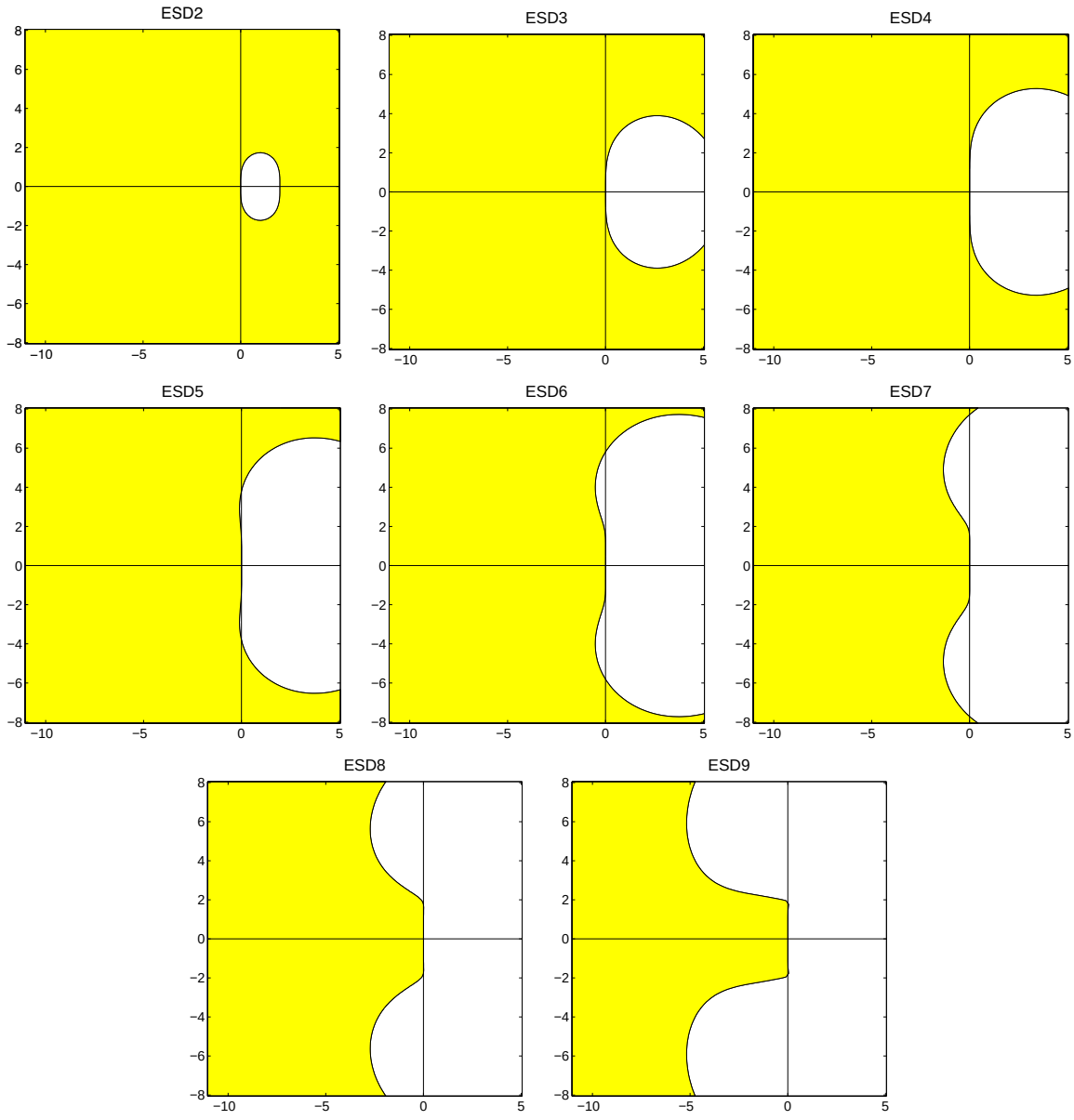


Figure 10.20: Stability regions in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using ESD2 through ESD9. In the language of §10.2.2, ESD2-ESD4 are L stable whereas ESD5-ESD9 are $A(\alpha)$ stable for $\alpha = 87.88^\circ$, 82.03°, 73.10°, 59.59°, and 37.61° respectively; also, ESD2-ESD9 are stiffly stable with $D = 0, 0, 0, 0.103, 0.526, 1.339, 2.728$, and 5.182 respectively.

$\alpha_2 = 1/3$, and $\beta_0 = 2/3$ satisfy these three equations. The leading-order error term of this method is proportional to h^3. Thus, over a single timestep, the scheme is "locally third-order accurate"; more significantly, over a fixed time interval $[0, T]$, the scheme is globally second-order accurate. The resulting method,

$$\mathbf{x}_{n+1} = \frac{4}{3}\mathbf{x}_n - \frac{1}{3}\mathbf{x}_{n-1} + h\frac{2}{3}\mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}),$$

is thus referred to as **BDF2**. Higher-order BDFs are derived analogously, and are summarized in Table 10.3. BDFs may be viewed as implicit alternatives to AM formulae that, referencing a given number of steps into the past (compare, e.g., BDF2 to AM3, or BDF3 to AM4), have a reduced order of accuracy but a greatly improved domain of stability (compare Figure 10.19a to Figure 10.15a, or Figure 10.19b to Figure 10.15b).

Recall (from the caption of Table 10.2 and Algorithm 10.7) that it is convenient, and cache efficient, to use AB methods as predictors when implementing iterative AM methods, as an (explicit) AB method may be selected that has the same information architecture and the same order of accuracy as the corresponding (implicit) AM method. Similarly when implementing iterative BDF methods, explicit methods referred to here as **eBDF** methods may be designed and implemented as predictors which have the same information architecture and the same order of accuracy as the corresponding (implicit) BDF methods; such eBDF predictors may be written in the form

$$\mathbf{x}_{n+1} = -\sum_{i=1}^{q} \alpha_i \mathbf{x}_{n+1-i} + h\beta_1 \mathbf{f}(\mathbf{x}_n, t_n). \tag{10.60}$$

Given the simplicity of the general BDF and eBDF formulae, (10.59) and (10.60), together with the simple interpretation of LMMs given in the first paragraph of §10.4.2, a convenient alternative method of determining the constants in these formulae (see Tables 10.3 and 10.4) is simply the Taylor table method described in §8.1.1, as implemented in the test code accompanying Algorithm 8.1 in the *NRC*; note that such methods are readily extensible to nonuniform grids, though they do not reveal the leading-order error.

10.5.3.2 Enright second derivative (ESD) methods†

Extending (10.40) to include a second derivative term at t_{n+1} while taking $q = 1$ gives the class of implicit methods known as **Enright second derivative** (**ESD**) methods:

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}_n + h\sum_{i=0}^{r} \beta_i \mathbf{f}(\mathbf{x}_{n+1-i}, t_{n+1-i}) - \gamma h^2 \mathbf{g}_{n+1},} \tag{10.61}$$

noting that $\mathbf{g}_{n+1} = \mathbf{x}''_{n+1}$ may be calculated as in (10.4b). The case with $r = 0$, which is second-order accurate and thus referred to as ESD2, was already encountered in §10.1, where it was identified as ITS2.

Higher-order ESD methods may be derived in an analogous fashion as the BDF methods (see, e.g., Exercise 10.11), and are summarized in Table 10.5. ESD methods may be viewed as implicit alternatives to BDF methods which, though more complex (as they involve computing $\mathbf{x}''_{n+1}$), have much smaller regions of instability in the LHP than do their BDF counterparts (compare Figure 10.20 to Figure 10.19).

10.5.4 Implicit/Explicit (IMEX) methods

Consider now a nonlinear ODE system in which the RHS can be split up into a potentially stiff part $\mathbf{f}(\mathbf{x})$ that is easy to handle with an implicit method (e.g., it might be linear with banded sparsity), and a generally nonstiff part $\mathbf{g}(\mathbf{x})$ that is not easy to handle with an implicit method (e.g., it might be nonlinear):

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}) + \mathbf{g}(\mathbf{x}). \tag{10.62}$$

To combine the excellent stability properties of an implicit method with the ease of application of an explicit method for such systems, which are in fact quite common, it is natural to propose an **Implicit/Explicit** (**IMEX**) numerical time-marching scheme which is implicit on $\mathbf{f}(\mathbf{x})$, to get the best stability possible even when the timesteps are too large to resolve all of the system dynamics, while being explicit on $\mathbf{g}(\mathbf{x})$, to facilitate fast computation.

BDF1 (IE)	$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}_{n+1}$
BDF2	$\mathbf{x}_{n+1} = \frac{4}{3}\mathbf{x}_n - \frac{1}{3}\mathbf{x}_{n-1} + \frac{2}{3}h\mathbf{f}_{n+1}$
BDF3	$\mathbf{x}_{n+1} = \frac{18}{11}\mathbf{x}_n - \frac{9}{11}\mathbf{x}_{n-1} + \frac{2}{11}\mathbf{x}_{n-2} + \frac{6}{11}h\mathbf{f}_{n+1}$
BDF4	$\mathbf{x}_{n+1} = \frac{48}{25}\mathbf{x}_n - \frac{36}{25}\mathbf{x}_{n-1} + \frac{16}{25}\mathbf{x}_{n-2} - \frac{3}{25}\mathbf{x}_{n-3} + \frac{12}{25}h\mathbf{f}_{n+1}$
BDF5	$\mathbf{x}_{n+1} = \frac{300}{137}\mathbf{x}_n - \frac{300}{137}\mathbf{x}_{n-1} + \frac{200}{137}\mathbf{x}_{n-2} - \frac{75}{137}\mathbf{x}_{n-3} + \frac{12}{137}\mathbf{x}_{n-4} + \frac{60}{137}h\mathbf{f}_{n+1}$
BDF6	$\mathbf{x}_{n+1} = \frac{360}{147}\mathbf{x}_n - \frac{450}{147}\mathbf{x}_{n-1} + \frac{400}{147}\mathbf{x}_{n-2} - \frac{225}{147}\mathbf{x}_{n-3} + \frac{72}{147}\mathbf{x}_{n-4} - \frac{10}{147}\mathbf{x}_{n-5} + \frac{60}{147}h\mathbf{f}_{n+1}$

Table 10.3: Stiffly stable backward differentiation formulae (BDF). As with the iterative AM scheme (see Table 10.2), when applied iteratively to a nonlinear ODE [see (10.25)], the corresponding eBDF formula of the same order (Table 10.4) is a convenient predictor to use, as it has an analogous information structure.

eBDF1 (EE)	$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}_n$
eBDF2 (leapfrog)	$\mathbf{x}_{n+1} = \mathbf{x}_{n-1} + 2h\mathbf{f}_n$
eBDF3	$\mathbf{x}_{n+1} = -\frac{3}{2}\mathbf{x}_n + 3\mathbf{x}_{n-1} - \frac{1}{2}\mathbf{x}_{n-2} + 3h\mathbf{f}_n$
eBDF4	$\mathbf{x}_{n+1} = -\frac{10}{3}\mathbf{x}_n + 6\mathbf{x}_{n-1} - 2\mathbf{x}_{n-2} + \frac{1}{3}\mathbf{x}_{n-3} + 4h\mathbf{f}_n$
eBDF5	$\mathbf{x}_{n+1} = -\frac{65}{12}\mathbf{x}_n + 10\mathbf{x}_{n-1} - 5\mathbf{x}_{n-2} + \frac{5}{3}\mathbf{x}_{n-3} - \frac{1}{4}\mathbf{x}_{n-4} + 5h\mathbf{f}_n$
eBDF6	$\mathbf{x}_{n+1} = -\frac{77}{10}\mathbf{x}_n + 15\mathbf{x}_{n-1} - 10\mathbf{x}_{n-2} + 5\mathbf{x}_{n-3} - \frac{3}{2}\mathbf{x}_{n-4} + \frac{1}{5}\mathbf{x}_{n-5} + 6h\mathbf{f}_n$

Table 10.4: Explicit backward differentiation formulae (eBDF).

ESD2 (ITS2)	$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}_{n+1} - \frac{1}{2}h^2\mathbf{g}_{n+1}$
ESD3	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{2}{3}h\mathbf{f}_{n+1} + \frac{1}{3}h\mathbf{f}_n - \frac{1}{6}h^2\mathbf{g}_{n+1}$
ESD4	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{29}{48}h\mathbf{f}_{n+1} + \frac{5}{12}h\mathbf{f}_n - \frac{1}{48}h\mathbf{f}_{n-1} - \frac{1}{8}h^2\mathbf{g}_{n+1}$
ESD5	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{307}{540}h\mathbf{f}_{n+1} + \frac{19}{40}h\mathbf{f}_n - \frac{1}{20}h\mathbf{f}_{n-1} + \frac{7}{1080}h\mathbf{f}_{n-2} - \frac{19}{180}h^2\mathbf{g}_{n+1}$
ESD6	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{3133}{5760}h\mathbf{f}_{n+1} + \frac{47}{90}h\mathbf{f}_n - \frac{41}{480}h\mathbf{f}_{n-1} + \frac{1}{45}h\mathbf{f}_{n-2} - \frac{17}{5760}h\mathbf{f}_{n-3} - \frac{3}{32}h^2\mathbf{g}_{n+1}$
ESD7	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{317731}{604800}h\mathbf{f}_{n+1} + \frac{2837}{5040}h\mathbf{f}_n - \frac{1271}{10080}h\mathbf{f}_{n-1} + \frac{373}{7560}h\mathbf{f}_{n-2} - \frac{529}{40320}h\mathbf{f}_{n-3} + \frac{41}{25200}h\mathbf{f}_{n-4} - \frac{863}{10080}h^2\mathbf{g}_{n+1}$
ESD8	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{247021}{483840}h\mathbf{f}_{n+1} + \frac{12079}{20160}h\mathbf{f}_n - \frac{13823}{80640}h\mathbf{f}_{n-1} + \frac{8131}{90720}h\mathbf{f}_{n-2} - \frac{5771}{161280}h\mathbf{f}_{n-3} + \frac{179}{20160}h\mathbf{f}_{n-4} - \frac{731}{725760}h\mathbf{f}_{n-5} - \frac{275}{3456}h^2\mathbf{g}_{n+1}$
ESD9	$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{1758023}{3528000}h\mathbf{f}_{n+1} + \frac{1147051}{1814400}h\mathbf{f}_n - \frac{133643}{604800}h\mathbf{f}_{n-1} + \frac{157513}{1088640}h\mathbf{f}_{n-2} - \frac{2797}{36288}h\mathbf{f}_{n-3} + \frac{86791}{3024000}h\mathbf{f}_{n-4} - \frac{35453}{5443200}h\mathbf{f}_{n-5} + \frac{8563}{12700800}h\mathbf{f}_{n-6} - \frac{33953}{453600}h^2\mathbf{g}_{n+1}$

Table 10.5: Stiffly stable Enright second derivative (ESD) formulae.

The justification for such IMEX methods is that, for sufficiently small h, any smooth nonlinear function may be linearized accurately; for h sufficiently small that this linearization is valid, superposition holds, and thus the two contributions to the increment from $\mathbf{x}_n$ to $\mathbf{x}_{n+1}$ may be computed separately and combined.

There are, of course, a variety of possible combinations of implicit and explicit schemes which one might consider; we illustrate here a few of the most common.

CNAB methods

One of the simplest class of IMEX methods, generally called **CNAB** methods, applies the (implicit) CN or CN[ϕ] method to the $\mathbf{f}(\mathbf{x})$ term and one of the (explicit) AB methods to the $\mathbf{g}(\mathbf{x})$ term. An example of a method in this class, **CN[ϕ]/AB2**, may be written

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\left(\frac{2+\phi}{4}\mathbf{f}(\mathbf{x}_{n+1}) + \frac{1-\phi}{2}\mathbf{f}(\mathbf{x}_n) + \frac{\phi}{4}\mathbf{f}(\mathbf{x}_{n-1}) + \frac{3}{2}\mathbf{g}(\mathbf{x}_n) - \frac{1}{2}\mathbf{g}(\mathbf{x}_{n-1})\right),$$

where, again, $\phi = 1/8$ is a suitable choice to ensure that the CN[ϕ] component of the scheme is L stable (if A stability is sufficient, $\phi = 0$ is preferred). Other CNAB methods may easily be written in a similar manner.

The CN/RKW3 method

An IMEX scheme which is particularly efficient with storage may be achieved by performing CN on each substep of the RKW3 scheme (10.39):

$$\begin{aligned}
&\text{First RK substep:} && \frac{\mathbf{x}^* - \mathbf{x}_n}{c_2 h} &&= \frac{1}{2}\left[\mathbf{f}(\mathbf{x}_n) + \mathbf{f}(\mathbf{x}^*)\right] && + \frac{a_{2,1}}{c_2}\mathbf{g}(\mathbf{x}_n)\\
&\text{Second RK substep:} && \frac{\mathbf{x}^{**} - \mathbf{x}^*}{(c_3 - c_2)h} &&= \frac{1}{2}\left[\mathbf{f}(\mathbf{x}^*) + \mathbf{f}(\mathbf{x}^{**})\right] && + \frac{a_{3,2}}{c_3 - c_2}\mathbf{g}(\mathbf{x}^*) + \frac{\zeta_2}{c_3 - c_2}\mathbf{g}(\mathbf{x}_n)\\
&\text{Third RK substep:} && \frac{\mathbf{x}_{n+1} - \mathbf{x}^{**}}{(1 - c_3)h} &&= \frac{1}{2}\left[\mathbf{f}(\mathbf{x}^{**}) + \mathbf{f}(\mathbf{x}_{n+1})\right] && + \frac{b_3}{1 - c_3}\mathbf{g}(\mathbf{x}^{**}) + \frac{\zeta_3}{1 - c_3}\mathbf{g}(\mathbf{x}^*).
\end{aligned}$$

Assuming first that $\mathbf{f}(\cdot) = 0$, it is seen immediately that this is equivalent to the scheme listed in (10.39), with the LHS of each expression now a numerical approximation of $d\mathbf{x}/dt$ over each RK substep (see Figure 10.12). This facilitates the accounting for the linear terms $\mathbf{f}(\mathbf{x})$ with the CN method over each RK substep. That is, assuming now that $\mathbf{g}(\cdot) = 0$, the scheme reduces to CN over each of the three substeps. Further, defining

$$\begin{gathered}
\overline{h}_1 = c_2 h, \quad \overline{h}_2 = (c_3 - c_2)h, \quad h_3 = (1 - c_3)h,\\
\overline{\beta}_1 = \frac{a_{2,1}}{c_2}, \quad \overline{\beta}_2 = \frac{a_{3,2}}{c_3 - c_2}, \quad \beta_3 = \frac{b_3}{1 - c_3}, \qquad \overline{\zeta}_1 = 0, \quad \overline{\zeta}_2 = \frac{\zeta_2}{c_3 - c_2}, \quad \zeta_3 = \frac{\zeta_3}{1 - c_3},
\end{gathered} \tag{10.63}$$

we can write the algorithm as a march over three RK substeps, each of the form

$$\frac{\mathbf{x}^{rk+1} - \mathbf{x}^{rk}}{\overline{h}_{rk}} = \frac{1}{2}[\mathbf{f}(\mathbf{x}^{rk}) + \mathbf{f}(\mathbf{x}^{rk+1})] + \overline{\beta}_{rk}\mathbf{g}(\mathbf{x}^{rk}) + \overline{\zeta}_{rk}\mathbf{g}(\mathbf{x}^{rk-1}),$$

or equivalently, if $\mathbf{f}(\mathbf{x}) = A\mathbf{x}$, as

$$\left(I - \frac{\overline{h}_{rk}}{2}A\right)\mathbf{x}^{rk+1} = \mathbf{x}^{rk} + \frac{\overline{h}_{rk}}{2}A\mathbf{x}^{rk} + \overline{h}_{rk}\overline{\beta}_{rk}\mathbf{g}(\mathbf{x}^{rk}) + \overline{h}_{rk}\overline{\zeta}_{rk}\mathbf{g}(\mathbf{x}^{rk-1}). \tag{10.64}$$

Note that, if A has tightly banded sparsity structure, this system may be solved efficiently.

IMEX Runge Kutta methods

The general idea of IMEX Runge Kutta methods is to develop a coordinated pair of DIRK and ERK methods

$$\begin{array}{c|cccc} c_1^{\rm IM} & a_{1,1}^{\rm IM} & & & \\ c_2^{\rm IM} & a_{2,1}^{\rm IM} & a_{2,2}^{\rm IM} & & \\ \vdots & \vdots & \ddots & \ddots & \\ c_S^{\rm IM} & a_{S,1}^{\rm IM} & \dots & a_{S,S-1}^{\rm IM} & a_{S,S}^{\rm IM} \\ \hline & b_1^{\rm IM} & \dots & b_{S-1}^{\rm IM} & b_S^{\rm IM} \end{array} \qquad \begin{array}{c|ccc} c_1^{\rm EX} & & & \\ c_2^{\rm EX} & a_{2,1}^{\rm EX} & & \\ \vdots & \vdots & \ddots & \\ c_S^{\rm EX} & a_{S,1}^{\rm EX} & \dots & a_{S,S-1}^{\rm EX} \\ \hline & b_1^{\rm EX} & \dots & b_{S-1}^{\rm EX} & b_S^{\rm EX} \end{array}$$

that may be used in the following sequence of numerical computations:

$$\begin{aligned} &1^{\rm IM}\text{: Compute } \mathbf{f}_1 = \mathbf{f}(\mathbf{X}_1, t_n + c_1^{\rm IM} h) \quad \text{where } \mathbf{X}_1 = \mathbf{x}_n + a_{1,1}^{\rm IM} h \mathbf{f}_1; \\ &1^{\rm EX}\text{: Compute } \mathbf{g}_1 = \mathbf{g}(\mathbf{X}_1, t_n + c_1^{\rm EX} h); \\ &2^{\rm IM}\text{: Compute } \mathbf{f}_2 = \mathbf{f}(\mathbf{X}_2, t_n + c_2^{\rm IM} h) \quad \text{where } \mathbf{X}_2 = \mathbf{x}_n + a_{2,1}^{\rm IM} h \mathbf{f}_1 + a_{2,2}^{\rm IM} h \mathbf{f}_2 + a_{2,1}^{\rm EX} h \mathbf{g}_1; \\ &2^{\rm EX}\text{: Compute } \mathbf{g}_2 = \mathbf{g}(\mathbf{X}_2, t_n + c_2^{\rm EX} h); \\ &\qquad \vdots \\ &S^{\rm IM}\text{: Compute } \mathbf{f}_S = \mathbf{f}(\mathbf{X}_S, t_n + c_S^{\rm IM} h) \quad \text{where } \begin{cases} \mathbf{X}_S = \mathbf{x}_n + a_{S,1}^{\rm IM} h \mathbf{f}_1 + a_{S,2}^{\rm IM} h \mathbf{f}_2 + \ldots + a_{S,S-1}^{\rm IM} h \mathbf{f}_{S-1} + a_{S,S}^{\rm IM} h \mathbf{f}_S \\ \qquad + a_{S,1}^{\rm EX} h \mathbf{g}_1 + a_{S,2}^{\rm EX} h \mathbf{g}_2 + \ldots + a_{S,S-1}^{\rm EX} h \mathbf{g}_{S-1}; \end{cases} \\ &S^{\rm EX}\text{: Compute } \mathbf{g}_S = \mathbf{f}(\mathbf{X}_S, t_n + c_S^{\rm EX} h); \\ &\text{March: } \mathbf{x}_{n+1} = \mathbf{x}_n + h[b_1^{\rm IM} \mathbf{f}_1 + b_2^{\rm IM} \mathbf{f}_2 + \ldots + b_S^{\rm IM} \mathbf{f}_S] + h[b_1^{\rm EX} \mathbf{g}_1 + b_2^{\rm EX} \mathbf{g}_2 + \ldots + b_S^{\rm EX} \mathbf{g}_S]. \end{aligned}$$

Note that each of the IM steps indicated above is implicit, and at the i'th stage may be completed as follows: guess $\mathbf{X}_i$, compute $\mathbf{f}_i$, recalculate $\mathbf{X}_i$ based on the computed value of $\mathbf{f}_i$, and iterate until convergence.

In contrast with the IMEX schemes presented previously, the coordinated pair of DIRK and ERK methods are developed carefully in IMEX Runge Kutta methods so that the entire timestep, including the interaction between the $\mathbf{f}(\mathbf{x})$ and $\mathbf{g}(\mathbf{x})$ terms, achieves an order of accuracy higher than first. This takes some significant care even in the case that the RHS does not explicitly depend on t: note in particular that, in the scalar case, the exact solution of $x' = f(x) + g(x)$ has many new interaction terms [cf. (10.21)]:

$$\begin{aligned} x_{n+1} &= x_n + h x_n' + \frac{h^2}{2!} x_n'' + \frac{h^3}{3!} x_n''' + O(h^4) \\ &= x_n + h\Big\{f+g\Big\}_{(x_n,t_n)} + \frac{h^2}{2}\Big\{f'f + f'g + g'f + g'g\Big\}_{(x_n,t_n)} + \frac{h^3}{6}\Big\{f''ff + 2f''fg + f''gg + g''ff \\ &\quad + 2g''fg + g''gg + f'f'f + f'g'f + g'f'f + g'g'f + f'f'g + f'g'g + g'f'g + g'g'g\Big\}_{(x_n,t_n)} + O(h^4), \end{aligned}$$

all of which must be matched properly through the desired order by the numerical scheme.

A representative third-order IMEX Runge Kutta method, due to Ascher, Ruuth, & Spiteri (1997) [see also related analysis by Pareschi & Russo (2001)], takes

$$\gamma = 0.4358665215, \quad \delta = -0.644373171, \quad \eta = 0.3966543747, \quad \mu = 0.5529291479, \tag{10.65a}$$

and defines the synchronized pair of DIRK and ERK schemes as follows:

$$\begin{array}{c|ccccc} 0 & 0 & & & & \\ \gamma & 0 & \gamma & & & \\ (1+\gamma)/2 & 0 & (1-\gamma)/2 & \gamma & & \\ 1 & 0 & 1-\delta-\gamma & \delta & \gamma & \\ \hline & 0 & 1-\delta-\gamma & \delta & \gamma & \end{array} \qquad \begin{array}{c|ccccc} 0 & & & & & \\ \gamma & \gamma & & & & \\ (1+\gamma)/2 & (1+\gamma)/2-\eta & \eta & & & \\ 1 & 1-2\mu & \mu & \mu & & \\ \hline & 0 & 1-\delta-\gamma & \delta & \gamma & \end{array} \tag{10.65b}$$

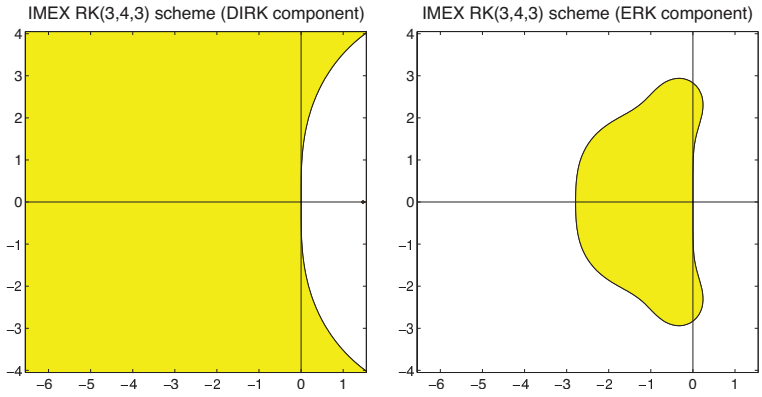


Figure 10.21: Stability regions in the complex plane λh for the numerical solution to $x' = \lambda x$ with timestep h using the (left) the DIRK component and (right) the ERK component of the IMEX RK(3,4,3) scheme given in (10.65). Note in particular that the DIRK component is L-stable.

Note that, due to the column of zeros, $\mathbf{f}_1$ is never used by the implicit part of this scheme, and thus the computation of $\mathbf{f}_1$ can be skipped. That is, this scheme actually takes 3 implicit steps and 4 explicit steps per timestep, and achieves overall third-order accuracay; it is thus classified as an **IMEX RK(3,4,3)** scheme. The stability boundaries for the explicit and implicit components of this method are plotted in Figure 10.21. Note also that, in this case, $\mathbf{c}^{\text{IM}} = \mathbf{c}^{\text{EX}}$ (that is, the implicit and explicit stage computations are synchronized in time), though this is not always the case for general IMEX RK methods.

10.6 Second-order systems

We now discuss appropriate methods for the simulation of second-order systems of the form

$$\frac{d^2\mathbf{q}}{dt^2} = \mathbf{f}\Big(\mathbf{q}, \frac{d\mathbf{q}}{dt}, t\Big), \tag{10.66a}$$

paying special attention to the linear case, written in the form

$$M\frac{d^2\mathbf{q}}{dt^2} + C\frac{d\mathbf{q}}{dt} + K\mathbf{q} = \mathbf{b}; \tag{10.66b}$$

in many problems of interest (arising, e.g., from structural vibration problems), $M > 0$, $C \geq 0$, and $K \geq 0$.

10.6.1 Reduction to first-order form

The several first-order ODE integration methods discussed earlier in this chapter are immediately applicable to (10.66) via the simple change of variables[14] $\mathbf{x}^1 = \mathbf{q}$ and $\mathbf{x}^2 = d\mathbf{q}/dt$, which allows us to rewrite (10.66a) and (10.66b) as, respectively,

$$\frac{d\mathbf{x}}{dt} = \begin{bmatrix} \mathbf{x}^2 \\ \mathbf{f}(\mathbf{x}^1, \mathbf{x}^2, t) \end{bmatrix} \quad \text{and} \quad \frac{d\mathbf{x}}{dt} = \begin{bmatrix} 0 & I \\ -M^{-1}K & -M^{-1}C \end{bmatrix}\mathbf{x} + \begin{bmatrix} 0 \\ M^{-1}\mathbf{b} \end{bmatrix} \quad \text{where} \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \end{bmatrix}. \tag{10.67}$$

[14] Noting the result of Exercise 4.3b, in the linear case it is sometimes preferred to write $\mathbf{x}^1 = M^{1/2}\mathbf{q}$ and $\mathbf{x}^2 = M^{1/2}\,d\mathbf{q}/dt$, where $M^{1/2}$ is the symmetric square root of the matrix $M > 0$, which preserves the symmetric structure of the component matrices of the block 2×2 matrix in the first-order representation of the system.

The CN, CN[θ], and CN[ϕ] methods are particularly well suited to the linear form on the right. [As an aside, note that the variable $\mathbf{q}$ is known as the **configuration** of the second-order system, whereas the variable $\mathbf{x}$ is known as the **state** of the system; initial conditions on the state $\mathbf{x}$ are sufficient to specify the subsequent time evolution of the system, whereas initial conditions on the configuration $\mathbf{q}$ alone are not.]

Application of a first-order ODE integration method to a first-order representation of a second-order problem, however, essentially neglects the special structure of problems of this class. The remainder of this section presents a few methods which exploit this structure, which presents certain numerical advantages.

10.6.2 The Newmark family of methods

We now develop a family of time integration methods particularly well suited for oscillatory linear second-order systems (10.66b) which, evaluating at t_{n+1}, may be written

$$M\mathbf{a}_{n+1} + C\mathbf{v}_{n+1} + K\mathbf{q}_{n+1} = \mathbf{b}_{n+1}. \tag{10.68}$$

Recall that the CN[θ] method, in the form given in footnote 12 on page 299, looks essentially like a truncated Taylor series expansion of $\mathbf{x}$ around the time t_n, with the last term of the truncated Taylor series blending derivative information from both the old and new timesteps. Defining $\mathbf{v} = d\mathbf{q}/dt$ and $\mathbf{a} = d^2\mathbf{q}/dt^2$, the **Newmark** family of methods starts from the following two truncated Taylor series with similarly modified last terms:

$$\mathbf{q}_{n+1} = \mathbf{q}_n + h\,\mathbf{v}_n + \frac{h^2}{2}\Big\{(1-2\beta)\,\mathbf{a}_n + 2\beta\,\mathbf{a}_{n+1}\Big\}, \tag{10.69a}$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + h\Big\{(1-\gamma)\,\mathbf{a}_n + \gamma\mathbf{a}_{n+1}\Big\}; \tag{10.69b}$$

it is found in the analysis below that parameter values of $\gamma \approx 1/2$ and $\beta \approx 1/4$, known as the **constant acceleration method**, work particularly well. Substituting (10.69a) and (10.69b) into (10.68) reveals the following:

$$D\,\mathbf{a}_{n+1} = \mathbf{b}_{n+1} - C\Big\{\mathbf{v}_n + h\,(1-\gamma)\,\mathbf{a}_n\Big\} - K\Big\{\mathbf{q}_n + h\,\mathbf{v}_n + \frac{h^2}{2}\,(1-2\beta)\,\mathbf{a}_n\Big\}, \tag{10.69c}$$

where $D = M + \gamma hC + \beta h^2 K$. At each timestep of the Newmark method, (10.69c) is first solved for $\mathbf{a}_{n+1}$, then $\mathbf{q}_{n+1}$ and $\mathbf{v}_{n+1}$ are computed using (10.69a) and (10.69b)[15]. A key advantage of the Newmark method over schemes based on reduction to first-order form is that, if M is diagonally dominant and M, C, and K have exploitable sparsity, then, given sufficiently small h, these properties are inherited by D [the only matrix that must be solved at each timestep]; this is in contrast with the matrix on the RHS in (10.67).

[15] Note that an equivalent form of the Newmark method may be developed by solving (10.69a) and (10.69b) for $\mathbf{a}_{n+1}$ and $\mathbf{v}_{n+1}$:

$$\mathbf{a}_{n+1} = \frac{1}{\beta h^2}\Big\{\mathbf{q}_{n+1} - \mathbf{q}_n - h\,\mathbf{v}_n - \frac{h^2}{2}(1-2\beta)\,\mathbf{a}_n\Big\}, \tag{10.70a}$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + h\Big\{\mathbf{a}_n + \frac{\gamma}{\beta h^2}\Big[\mathbf{q}_{n+1} - \mathbf{q}_n - h\,\mathbf{v}_n - \frac{h^2}{2}\mathbf{a}_n\Big]\Big\}; \tag{10.70b}$$

Substituting (10.70a) and (10.70b) into (10.68) then reveals the following:

$$D\,\mathbf{q}_{n+1} = \beta h^2\,\mathbf{b}_{n+1} - M\Big\{-\mathbf{q}_n - h\mathbf{v}_n - \frac{h^2}{2}(1-2\beta)\,\mathbf{a}_n\Big\} - Ch\Big\{\beta\,h\mathbf{v}_n + \beta\,h^2\mathbf{a}_n - \gamma\Big[\mathbf{q}_n + h\,\mathbf{v}_n + \frac{h^2}{2}\mathbf{a}_n\Big]\Big\}, \tag{10.70c}$$

where, again, $D = M + \gamma hC + \beta h^2 K$ [and, thus, this formulation has the same complexity as that in (10.70)]. At each timestep of this formulation of the Newmark method, (10.70c) is first solved for $\mathbf{q}_{n+1}$, then $\mathbf{a}_{n+1}$ and $\mathbf{v}_{n+1}$ are computed using (10.70a) and (10.70b).

Substituting $\mathbf{a}_n = M^{-1}(\mathbf{b}_n - C\mathbf{v}_n - K\mathbf{q}_n)$ and $\mathbf{a}_{n+1} = M^{-1}(\mathbf{b}_{n+1} - C\mathbf{v}_{n+1} - K\mathbf{q}_{n+1})$ into the RHS and defining $\hat{\beta} = 0.5 - \beta$ and $\hat{\gamma} = 1 - \gamma$, we may rewrite (10.69a)-(10.69b) as

$$\begin{bmatrix} M + \beta h^2 K & \beta h^2 C \\ \gamma h K & M + \gamma h C \end{bmatrix} \begin{bmatrix} \mathbf{q}_{n+1} \\ \mathbf{v}_{n+1} \end{bmatrix} = \begin{bmatrix} M - \hat{\beta} h^2 K & hM - \hat{\beta} h^2 C \\ -\hat{\gamma} h K & M - \hat{\gamma} h C \end{bmatrix} \begin{bmatrix} \mathbf{q}_n \\ \mathbf{v}_n \end{bmatrix} + \begin{bmatrix} \hat{\beta} h^2 \mathbf{b}_n + \beta h^2 \mathbf{b}_{n+1} \\ \hat{\gamma} h \mathbf{b}_n + \gamma h \mathbf{b}_{n+1} \end{bmatrix}.$$

Considering now the **second-order model problem** $d^2q/dt^2 - \lambda^2 q = (d/dt - \lambda)(d/dt + \lambda)q = 0$ [that is, combining the first-order scalar model problems $dq/dt = \lambda q$ and $dq/dt = -\lambda q$], we take $M = 1$, $C = 0$, $K = -\lambda^2$, and $\mathbf{b} = 0$ in (10.68) and apply Fact 1.8, reducing the above relation to

$$\begin{pmatrix} q_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 - \beta\lambda^2 h^2 & 0 \\ -\gamma\lambda^2 h & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 + \hat{\beta}\lambda^2 h^2 & h \\ \hat{\gamma}\lambda^2 h & 1 \end{pmatrix} \begin{pmatrix} q_n \\ v_n \end{pmatrix} = \begin{pmatrix} 1 - \eta/2 & -\eta/(\lambda^2 h) \\ \lambda^2 h(1 - \gamma\eta/2) & 1 - \gamma\eta \end{pmatrix} \begin{pmatrix} q_n \\ v_n \end{pmatrix} \triangleq \Sigma \begin{pmatrix} q_n \\ v_n \end{pmatrix}$$

where, applying (B.77), $\eta = -\lambda^2 h^2/(1 - \beta\lambda^2 h^2) = -\lambda^2 h^2 - \beta\lambda^4 h^4 - \beta^2\lambda^6 h^6 + \ldots$. The characteristic equation of Σ is $\sigma^2 - (2 - (\gamma + 1/2)\eta)\sigma + 1 - (\gamma - 1/2)\eta = 0$; defining $\widetilde{\gamma} = \gamma + 1/2$, the eigenvalues of Σ are

$$\begin{aligned} \sigma_\pm &= 1 - (\gamma + 1/2)\,\eta/2 \pm \sqrt{-\eta[1 - (\gamma + 1/2)^2\,\eta/4]} && \text{(10.71a)} \\ &= 1 + \widetilde{\gamma}(\lambda^2 h^2 + \beta\lambda^4 h^4 + \ldots)/2 \pm \lambda h \sqrt{(1 + \beta\lambda^2 h^2 + \beta^2\lambda^4 h^4 + \ldots)\,(1 + \widetilde{\gamma}^2\lambda^2 h^2/4 + \widetilde{\gamma}^2\beta\lambda^4 h^4/4 + \ldots)} \\ &= 1 + \widetilde{\gamma}(\lambda^2 h^2 + \beta\lambda^4 h^4 + \ldots)/2 \pm \lambda h \sqrt{1 + (\beta + \widetilde{\gamma}^2/4)\lambda^2 h^2 + (\beta^2 + \beta\widetilde{\gamma}^2/2)\lambda^4 h^4 + \ldots} \\ &= 1 \pm \lambda h + \frac{\gamma + 1/2}{2}\lambda^2 h^2 \pm \frac{\beta + (\gamma + 1/2)^2/4}{2}\lambda^3 h^3 + \frac{\gamma + 1/2}{2}\beta\lambda^4 h^4 + O(\lambda^5 h^5). && \text{(10.71b)} \end{aligned}$$

It is thus seen that:

- Taking $\gamma = 1/2$ and $\beta = 1/12$, we have $\sigma_\pm = 1 \pm \lambda h + \lambda^2 h^2/2 \pm \lambda^3 h^3/6 + \lambda^4 h^4/24 + \ldots$, and the method (known as the **Fox-Goodwin** method) is fourth-order accurate; for $\gamma = 1/2$ and $\beta \neq 1/12$, the method is second-order accurate (other common values of β include $\beta = 1/6$, known as the **linear acceleration method**, and $\beta = 1/4$, known as the **constant acceleration method**).
- If $\lambda = \mathrm{i}\omega$ and thus $\lambda^2 = -\omega^2 < 0$ & $\eta > 0$, then the eigenvalues of Σ come as a complex conjugate pair (that is, $\sigma_+ = \overline{\sigma}_-$) if

$$1 - (\gamma + 1/2)^2\,\eta/4 \geq 0 \quad \Leftrightarrow \quad (\gamma + 1/2)^2 - 4\beta \leq 4/\omega^2 h^2.$$

- If $\lambda = \mathrm{i}\omega$ and thus $\lambda^2 = -\omega^2 < 0$ & $\eta > 0$, noting (10.71a), it is further seen that
  - if $\gamma = 1/2$ and $\beta \geq 1/4$, the eigenvalues $\sigma_\pm$ come in a complex conjugate pair with $|\sigma_\pm| = 1$ for *any* ω (i.e., the time-integration method is **unconditionally stable**);
  - if $\gamma = 1/2$ and $0 < \beta < 1/4$, the eigenvalues $\sigma_\pm$ come in a complex conjugate pair with $|\sigma_\pm| = 1$ for $\omega^2 h^2 \leq 4/[(\gamma + 1/2)^2 - 4\beta]$ (i.e., the time-integration method is **conditionally stable**).

As in previous sections, the stability regions (that is, the regions for which both $|\sigma_+| \leq 1$ and $|\sigma_-| \leq 1$) are plotted in Figure 10.22 for $\beta = 1/12$, $\beta = 1/6$, and $\beta = 1/4$, and for two values of γ slightly larger than 0.5; per the last bullet point above, for $\gamma = 0.5$ exactly, the region of stability is restricted to an interval along the imaginary axis centered at the origin for $\beta < 1/4$, and to the entire imaginary axis for $\beta \geq 1/4$.

It is seen in Figure 10.22 that the Newmark family of methods is useless for exponentially damped problems (with any eigenvalues on the negative real axis). However, for purely oscillatory problems (with pure imaginary eigenvalues), we may achieve stability by taking $\gamma = 0.5$, and for problems with lightly damped complex eigenvalues, we may achieve stability by taking γ slightly larger than 0.5. In order to ensure stability for all of the oscillatory modes in a given problem (even those with large ω), we usually take $\beta = 1/4$

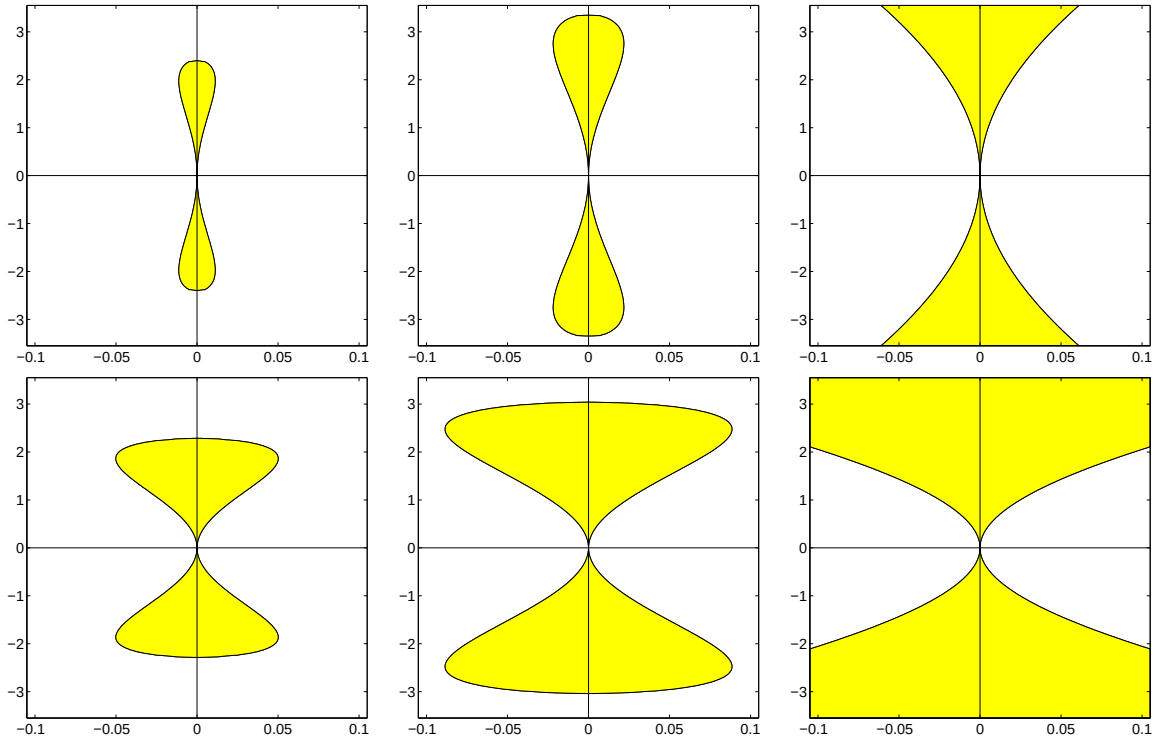


Figure 10.22: Stability regions in the complex plane λh for the numerical solution of $x'' = \lambda^2 x$ with timestep h using a Newmark method with (left) $\beta = 1/12$, (center) $\beta = 1/6$, (right) $\beta = 1/4$ and (top) $\gamma = 0.51$, (bottom) $\gamma = 0.55$. The stable regions are shaded. Note the scaling of the x axis.

in the Newmark method in order to obtain unconditional stability, thus resulting in second-order accuracy. Nonetheless, for certain finite-dimensional problems in which the oscillatory modes have have a known maximum frequency (cf. the infinite-dimensional problem simulated with this method in §11.3.2), we may take $\beta = 1/12$ (and sufficiently small h) to achieve both stability and fourth-order accuracy with this method.

10.6.3 Hamiltonian systems

Many "**conservative**" second-order systems can be written in the equivalent first-order **Hamiltonian form**

$$\left.\begin{aligned} \frac{d\mathbf{q}}{dt} &= \frac{\partial H(\mathbf{q},\mathbf{p})}{\partial \mathbf{p}} \\ \frac{d\mathbf{p}}{dt} &= -\frac{\partial H(\mathbf{q},\mathbf{p})}{\partial \mathbf{q}} \end{aligned}\right\} \quad \Leftrightarrow \quad \frac{d\mathbf{z}(t)}{dt} = \mathbf{f}[\mathbf{z}(t)] = \begin{bmatrix} \dfrac{\partial H[\mathbf{p}(t),\mathbf{q}(t)]}{\partial \mathbf{p}} \\ -\dfrac{\partial H[\mathbf{p}(t),\mathbf{q}(t)]}{\partial \mathbf{q}} \end{bmatrix} \quad \text{with} \quad \mathbf{z}(t) = \begin{bmatrix} \mathbf{q}(t) \\ \mathbf{p}(t) \end{bmatrix}, \tag{10.72}$$

where $\mathbf{q}(t)$ is referred to as a **generalized configuration vector**, $\mathbf{p}(t)$ is referred to as a **generalized momentum vector**, and the scalar $H(\mathbf{p},\mathbf{q})$ is referred to as the **Hamiltonian** of the system. Problems of this form deserve special attention, as the Hamiltonian is preserved exactly as the system evolves:

$$\frac{dH}{dt} = \frac{\partial H}{\partial \mathbf{q}} \cdot \frac{d\mathbf{q}}{dt} + \frac{\partial H}{\partial \mathbf{p}} \cdot \frac{d\mathbf{p}}{dt} = \frac{\partial H}{\partial \mathbf{q}} \cdot \frac{\partial H}{\partial \mathbf{p}} - \frac{\partial H}{\partial \mathbf{p}} \cdot \frac{\partial H}{\partial \mathbf{q}} = 0. \tag{10.73}$$

We thus seek numerical methods which share, if possible, this conservation property of the physical system.

Analysis of problems of this type is simplified significantly by introducing the **Poisson bracket**[16]

$$\{f(\mathbf{q},\mathbf{p}),g(\mathbf{q},\mathbf{p})\} \triangleq \frac{\partial f}{\partial q_i}\frac{\partial g}{\partial p_i} - \frac{\partial f}{\partial p_i}\frac{\partial g}{\partial q_i}. \tag{10.74}$$

We may then write the equations governing the evolution of the system in Hamiltonian form as

$$dq_j/dt = \partial H/\partial p_j = \{q_j,H\} \quad \text{and} \quad dp_j/dt = -\partial H/\partial q_j = \{p_j,H\}. \tag{10.75}$$

Angling for a simplified formulation leveraging the Poisson bracket, we may write

$$d\mathbf{z}/dt = D_H\mathbf{z} \quad \text{where} \quad D_H\mathbf{z} \triangleq \{\mathbf{z},H\}, \tag{10.76a}$$

where we refer to D_H as the **Poisson operator related to** H; it follows that $d^2\mathbf{z}/dt^2 = D_H(D_H\mathbf{z}) \triangleq D_H^2\mathbf{z}$, etc. The exact solution of (10.76a) over a timestep of duration h may be written using a Taylor series as

$$\mathbf{z}(t+h) = \mathbf{z}(t) + hD_H\mathbf{z}(t) + \frac{h^2}{2!}D_H^2\mathbf{z}(t) + \frac{h^3}{3!}D_H^3\mathbf{z}(t) + \ldots \triangleq e^{D_H h}\mathbf{z}(t) \tag{10.76b}$$

$$\text{where} \quad e^{D_H h} \triangleq I + hD_H + \frac{h^2}{2!}D_H^2 + \frac{h^3}{3!}D_H^3 + \ldots; \tag{10.76c}$$

note that the solution of the linear system $d\mathbf{x}/dt = A\mathbf{x}$, where A is a matrix, is written in an analogous form $\mathbf{x}(t+h) = e^{Ah}\mathbf{x}(t)$ where e^{Ah} is called the **matrix exponential**, as discussed further in §**??**. The infinite series in (10.76b) converges slowly when h is large; further, even if many steps of relatively short duration h are made, the various ODE marching techniques presented thus far in this chapter tend to introduce errors in the Hamiltonian at each timestep which accumulate over time, thus leading to an unacceptable level of violation of (10.73). We thus seek to design ODE marching techniques specifically for this class of systems.

We focus our attention on a common class of Hamiltonian systems with **separable** Hamiltonian structure,

$$H[\mathbf{p}(t),\mathbf{q}(t)] = T[\mathbf{p}(t)] + V[\mathbf{q}(t)] \quad \Rightarrow \quad d\mathbf{z}/dt = D_T\mathbf{z} + D_V\mathbf{z} \tag{10.77a}$$

$$\text{where} \quad D_H\mathbf{z} = \begin{bmatrix} \dfrac{dH[\mathbf{p}(t),\mathbf{q}(t)]}{d\mathbf{p}} \\ -\dfrac{dH[\mathbf{p}(t),\mathbf{q}(t)]}{d\mathbf{q}} \end{bmatrix}, \quad \text{and thus} \quad D_T\mathbf{z} = \begin{bmatrix} \dfrac{dT[\mathbf{p}(t)]}{d\mathbf{p}} \\ 0 \end{bmatrix}, \quad D_V\mathbf{z} = \begin{bmatrix} 0 \\ -\dfrac{dV[\mathbf{q}(t)]}{d\mathbf{q}} \end{bmatrix}. \tag{10.77b}$$

Example 10.10 Orbital Mechanics. Denote T as the **kinetic energy**, V as the **potential energy**, and H as the **total energy** of an isolated system of n bodies moving under (classical) mutual gravitational forces, where[17] the i'th body has **mass** m_i, position (a.k.a. **configuration**) $\mathbf{q}_i$, and **momentum** $\mathbf{p}_i$, and the **gravitational constant** G is 6.67428e−11 m^3/kg/sec^2. The Hamiltonian of this system is of the form (10.77a) with

$$T[\mathbf{p}(t)] = \frac{1}{2}\sum_{i=1}^{n}\mathbf{p}_i(t)\cdot\mathbf{p}_i(t)/m_i, \qquad V[\mathbf{q}(t)] = -\sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n} Gm_im_j/|\mathbf{q}_i(t)-\mathbf{q}_j(t)|.$$

[16] Though the practical utility of this construction might at first seem a bit fishy, once you trawl these waters more deeply, the lure of the simplifying nature of the Poisson bracket in this cast of problems ultimately becomes irresistible. Important properties of this noncommunative bilinear construction, all of which are straightforward to verify (see Exercise 10.14), include the following:

$$\{f,f\} = 0, \quad \{f,g\} = -\{g,f\}, \quad \{f+g,h\} = \{f,h\}+\{g,h\}, \quad \{f,g+h\} = \{f,g\}+\{f,h\},$$
$$\{fg,h\} = f\{g,h\}+\{f,h\}g, \quad \{f,gh\} = g\{f,h\}+\{f,g\}h, \quad \{f,\{g,h\}\}+\{h,\{f,g\}\}+\{g,\{h,f\}\} = 0.$$

[17] That is, the system configuration and momentum vectors, $\mathbf{p}$ and $\mathbf{q}$, are the vectors formed by assembling all of the individual configuration and momentum vectors $\mathbf{p}_i$ and $\mathbf{q}_i$ in the system.

Noting the partitioning of $\mathbf{z}$ in (10.72) and the sparsity of $D_T\mathbf{z}$ & $D_V\mathbf{z}$ in (10.77b), it follows that the D_T & D_V operators are **nilpotent of degree 2** (see §1.2.7 for the analogous definition of nilpotent *matrices*); i.e.,

$$D_T^n\mathbf{z} = D_V^n\mathbf{z} = 0 \quad \text{for} \quad n \geq 2. \tag{10.78}$$

Noting the definition of $e^{D_H h}$ in (10.76c), the following expressions are therefore exact (even for finite h):

$$e^{D_T h}\mathbf{z}(t) = \mathbf{z}(t) + hD_T\mathbf{z}(t), \qquad e^{D_V h}\mathbf{z}(t) = \mathbf{z}(t) + hD_V\mathbf{z}(t). \tag{10.79}$$

We now consider self-starting explicit time marching methods, known as **Symplectic Integrators** (**SI**s), for separable Hamiltonian systems that leverage these exact relations over p substeps and are written in the form

$$\mathbf{z}(t+h) = e^{(D_T+D_V)h}\mathbf{z}(t) = e^{D_V d_p h}e^{D_T c_p h}\cdots e^{D_V d_2 h}e^{D_T c_2 h}\cdot e^{D_V d_1 h}e^{D_T c_1 h}\mathbf{z}(t) + O(h^{p+1}) \tag{10.80}$$

where $c_1 + c_2 + \ldots c_p = 1$ and $d_1 + d_2 + \ldots d_p = 1$; note that the c_i need not equal the d_i.

With $p = 1$ in (10.80), noting (10.79), the **first-order symplectic integrator** (**SI1**, a.k.a. the **symplectic Euler method**) may be written by taking $c_1 = d_1 = 1$, and thus:

$$\mathbf{z}_* = e^{D_T h}\mathbf{z}_k = \mathbf{z}_k + hD_T\mathbf{z}_k \quad \Rightarrow \quad \mathbf{z}_* = \mathbf{z}_k + h\begin{bmatrix}\mathbf{q}'_k\\ 0\end{bmatrix} \quad \text{where} \quad \mathbf{q}'_k = \left.\frac{dT[\mathbf{p}(t)]}{d\mathbf{p}}\right|_{\mathbf{p}=\mathbf{p}_k}, \tag{10.81a}$$

$$\mathbf{z}_{k+1} = e^{D_V h}\mathbf{z}_* = \mathbf{z}_* + hD_V\mathbf{z}_* \quad \Rightarrow \quad \mathbf{z}_{k+1} = \mathbf{z}_* + h\begin{bmatrix}0\\ \mathbf{p}'_*\end{bmatrix} \quad \text{where} \quad \mathbf{p}'_* = -\left.\frac{dV[\mathbf{q}(t)]}{d\mathbf{q}}\right|_{\mathbf{q}=\mathbf{q}_*}; \tag{10.81b}$$

that is, combining (10.81a) into (10.81b),

$$\mathbf{z}_{k+1} = [\mathbf{z}_k + hD_T\mathbf{z}_k] + hD_V[\mathbf{z}_k + hD_T\mathbf{z}_k] = \mathbf{z}_k + h[D_T + D_V]\mathbf{z}_k + h^2D_VD_T\mathbf{z}_k. \tag{10.82}$$

Noting (10.78), the exact result (10.76b) for a system with a separable Hamiltonian (10.77a) is

$$\begin{aligned}\mathbf{z}(t+h) = \mathbf{z}(t) &+ h\,[D_T + D_V]\,\mathbf{z}(t) + \frac{h^2}{2!}\,[D_VD_T + D_TD_V]\,\mathbf{z}(t) + \frac{h^3}{3!}\,[D_VD_TD_V + D_TD_VD_T]\,\mathbf{z}(t)\\ &+ \frac{h^4}{4!}\,[D_VD_TD_VD_T + D_TD_VD_TD_V]\,\mathbf{z}(t) + \frac{h^5}{5!}\,[D_VD_TD_VD_TD_V + D_TD_VD_TD_VD_T]\,\mathbf{z}(t) + \ldots\end{aligned} \tag{10.83}$$

Comparing (10.82) with (10.83), it is seen that the leading-order error is proportional to h^2, and thus the SI1 scheme (10.81) is globally first-order accurate.

With $p = 2$ in (10.80), noting (10.79), a **second-order symplectic integrator** (**SI2**, a.k.a. the **Verlet method**) may be written by taking $c_1 = c_2 = 1/2$, $d_1 = 1$, $d_2 = 0$ [with the primed quantities as in (10.81)]:

$$\mathbf{z}_* = e^{D_T h/2}\mathbf{z}_k \;= \mathbf{z}_k \;+ (h/2)D_T\mathbf{z}_k \quad \Rightarrow \quad \mathbf{z}_* = \mathbf{z}_k \;+ (h/2)\begin{bmatrix}\mathbf{q}'_k\\ 0\end{bmatrix}, \tag{10.84a}$$

$$\mathbf{z}_{**} = e^{D_V h}\,\mathbf{z}_k \;= \mathbf{z}_* \;+\; h\,D_V\mathbf{z}_* \quad \Rightarrow \quad \mathbf{z}_{**} = \mathbf{z}_* \;+\; h\begin{bmatrix}0\\ \mathbf{p}'_*\end{bmatrix}, \tag{10.84b}$$

$$\mathbf{z}_{k+1} = e^{D_T h/2}\mathbf{z}_{**} = \mathbf{z}_{**} + (h/2)D_T\mathbf{z}_{**} \quad \Rightarrow \quad \mathbf{z}_{k+1} = \mathbf{z}_{**} + (h/2)\begin{bmatrix}\mathbf{q}'_{**}\\ 0\end{bmatrix}; \tag{10.84c}$$

that is, combining (10.84a) & (10.84b) into (10.84c) and applying (10.78),

$$\begin{aligned}\mathbf{z}_{k+1} &= [\mathbf{z}_* + hD_V\mathbf{z}_*] + (h/2)D_T[\mathbf{z}_* + hD_V\mathbf{z}_*] = \mathbf{z}_* + h[(1/2)D_T + D_V]\mathbf{z}_* + (h^2/2)D_TD_V\mathbf{z}_*\\ &= [\mathbf{z}_k + (h/2)D_T\mathbf{z}_k] + h[(1/2)D_T + D_V][\mathbf{z}_k + (h/2)D_T\mathbf{z}_k] + (h^2/2)D_TD_V[\mathbf{z}_k + (h/2)D_T\mathbf{z}_k]\\ &= \mathbf{z}_k + h[D_T + D_V]\mathbf{z}_k + (h^2/2)[D_VD_T + D_TD_V]\mathbf{z}_k + (h^3/4)[D_TD_VD_T]\mathbf{z}_k.\end{aligned} \tag{10.85}$$

Comparing (10.85) with (10.83), it is seen that the leading-order error is proportional to h^3, and thus the SI2 scheme (10.84) is globally second-order accurate.

Note that the last substep of one timestep and the first substep of the next in (10.84) may actually be combined (exactly) when marching over several timesteps, thus leading to a scheme of essentially the same two-substep form as (10.81) but which, when interpreted correctly, is actually second-order accurate. Stated differently, when interpreting the computed **p** values as *shifted by a half timestep* from the computed **q** values in such a scheme, an extra order of accuracy is realized. This idea is encountered again in §13.

With $p = 4$ in (10.80), noting (10.79), a **fourth-order symplectic integrator** (**SI4**, a.k.a. Ruth's method) may be written by defining $f = 2^{1/3}$ and taking

$$c_1 = c_4 = \frac{1}{2(2-f)}, \quad c_2 = c_3 = \frac{1-f}{2(2-f)}, \quad d_1 = d_3 = \frac{1}{2-f}, \quad d_2 = -\frac{f}{2-f}, \quad d_4 = 0,$$

and thus,

$$\mathbf{z}_{*1} = e^{D_T c_1 h}\mathbf{z}_k = \mathbf{z}_k + c_1 h D_T \mathbf{z}_k \quad \Rightarrow \quad \mathbf{z}_{*1} = \mathbf{z}_k + c_1 h \begin{bmatrix} \mathbf{q}'_k \\ 0 \end{bmatrix}, \tag{10.86a}$$

$$\mathbf{z}_{*2} = e^{D_V d_1 h}\mathbf{z}_{*1} = \mathbf{z}_{*1} + d_1 h D_V \mathbf{z}_{*1} \quad \Rightarrow \quad \mathbf{z}_{*2} = \mathbf{z}_{*1} + d_1 h \begin{bmatrix} 0 \\ \mathbf{p}'_{*1} \end{bmatrix}, \tag{10.86b}$$

$$\mathbf{z}_{*3} = e^{D_T c_2 h}\mathbf{z}_{*2} = \mathbf{z}_{*2} + c_2 h D_T \mathbf{z}_{*2} \quad \Rightarrow \quad \mathbf{z}_{*3} = \mathbf{z}_{*2} + c_2 h \begin{bmatrix} \mathbf{q}'_{*2} \\ 0 \end{bmatrix}, \tag{10.86c}$$

$$\mathbf{z}_{*4} = e^{D_V d_2 h}\mathbf{z}_{*3} = \mathbf{z}_{*3} + d_2 h D_V \mathbf{z}_{*3} \quad \Rightarrow \quad \mathbf{z}_{*4} = \mathbf{z}_{*3} + d_2 h \begin{bmatrix} 0 \\ \mathbf{p}'_{*3} \end{bmatrix}, \tag{10.86d}$$

$$\mathbf{z}_{*5} = e^{D_T c_3 h}\mathbf{z}_{*4} = \mathbf{z}_{*4} + c_3 h D_T \mathbf{z}_{*4} \quad \Rightarrow \quad \mathbf{z}_{*5} = \mathbf{z}_{*4} + c_3 h \begin{bmatrix} \mathbf{q}'_{*4} \\ 0 \end{bmatrix}, \tag{10.86e}$$

$$\mathbf{z}_{*6} = e^{D_V d_3 h}\mathbf{z}_{*5} = \mathbf{z}_{*5} + d_3 h D_V \mathbf{z}_{*5} \quad \Rightarrow \quad \mathbf{z}_{*6} = \mathbf{z}_{*5} + d_3 h \begin{bmatrix} 0 \\ \mathbf{p}'_{*5} \end{bmatrix}, \tag{10.86f}$$

$$\mathbf{z}_{k+1} = e^{D_T c_4 h}\mathbf{z}_{*6} = \mathbf{z}_{*6} + c_4 h D_T \mathbf{z}_{*6} \quad \Rightarrow \quad \mathbf{z}_{k+1} = \mathbf{z}_{*6} + c_4 h \begin{bmatrix} \mathbf{q}'_{*6} \\ 0 \end{bmatrix}. \tag{10.86g}$$

Combining (10.86a) through (10.86g) and comparing with (10.83) (see Exercise 10.18) confirms that the leading-order error is proportional to h^5, and thus the SI4 scheme (10.84) is globally fourth-order accurate.

As with the SI2 scheme (10.84), the last substep of one timestep and the first substep of the next in (10.86) may be combined (exactly) when marching over several timesteps, thus leading to a scheme of the same order with essentially six substeps per timestep, with the values of **q** and **p** shifted slightly in time from one another.

Symplectic integrators for symplectic operators

Recall (Fact 4.33) that a **real, symplectic matrix** M is defined via the **symplectic identity** J as

$$J^T M J = M^{-T} \quad \Leftrightarrow \quad J = M J M^T \qquad \text{where} \quad J = \begin{bmatrix} 0 & I \\ -I & 0 \end{bmatrix}, \quad J^{-1} = J^T = -J.$$

Hamiltonian systems (like the orbital mechanics problem formulated in Example 10.10 and illustrated in Figure 10.23) are special because they are **conservative** (as opposed to being **diffusive** like the heat, Burgers, and Navier-Stokes equations considered in §11); essentially, Hamiltonian systems have the same behavior marching forward in time as they do marching backward in time. One way of seeing this is writing

$$\frac{d\mathbf{z}}{dt} = D_H \mathbf{z} = \begin{bmatrix} \dfrac{\partial H}{\partial \mathbf{p}} \\ -\dfrac{\partial H}{\partial \mathbf{q}} \end{bmatrix} = J \nabla H \qquad \text{where} \quad \mathbf{z} = \begin{bmatrix} \mathbf{q} \\ \mathbf{p} \end{bmatrix}, \quad \nabla = \begin{bmatrix} \dfrac{\partial}{\partial \mathbf{q}} \\ \dfrac{\partial}{\partial \mathbf{p}} \end{bmatrix};$$

it follows that the Poisson operator related to H is **divergence free**: $\nabla^T (d\mathbf{z}/dt) = \nabla^T (D_H \mathbf{z}) = \nabla^T J \nabla H = 0$.

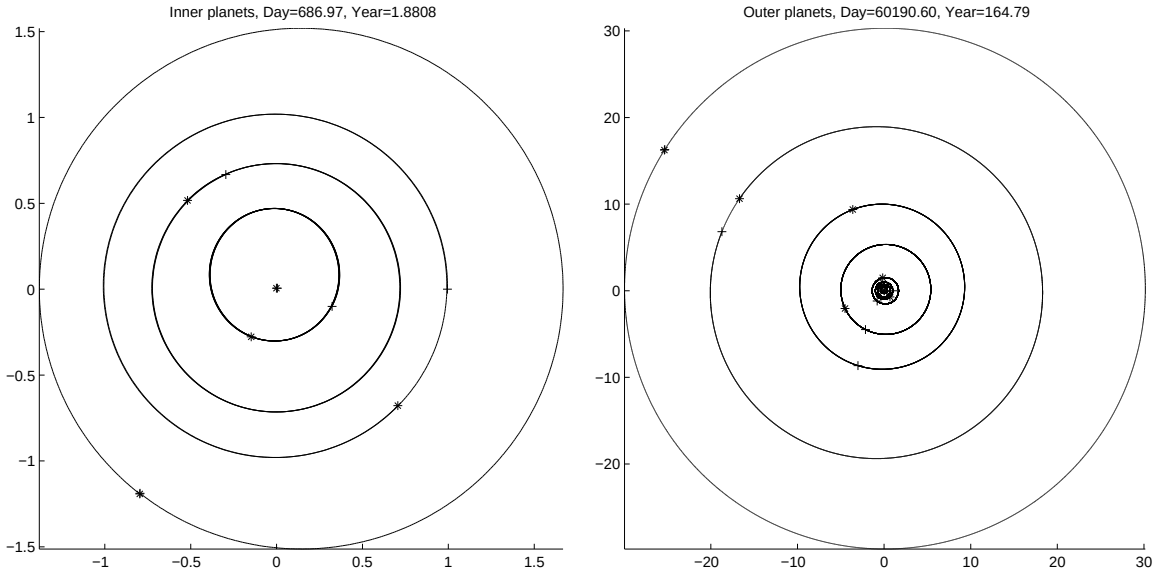


Figure 10.23: Solar system simulation, as implemented in Algorithm 10.10, using SI4 and $h \approx 1$ day. (left) Initial (+) & final (*) positions of the inner planets after 1.8808 sidereal years, after which Mars (fourth from the sun) essentially returns to its initial position. (right) Initial (+) & final (*) positions of the outer planets after 164.79 sidereal years, after which Neptune returns to its initial position.

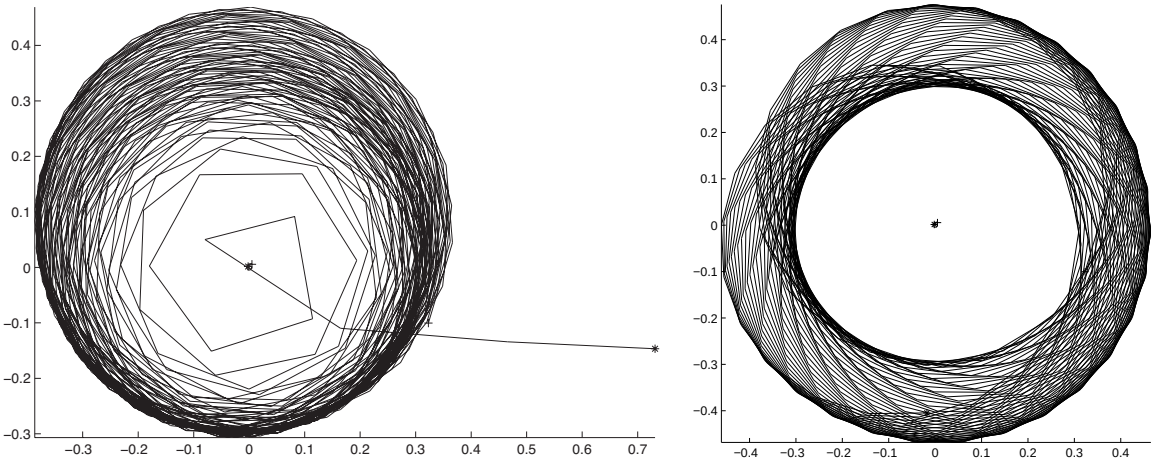


Figure 10.24: Orbit of Mercury in solar system simulations, as implemented in Algorithm 10.10, over 14 sidereal years using (left) RK4 and (right) SI4 and an unacceptably large timestep of $h \approx 10$ days. Though the initial accuracy over each timestep of the RK4 and SI4 methods are about the same, the errors in the RK4 case lead to significant changes in the orbital energy, whereas the errors in the SI4 case lead to a shift of the orbit, but not to significant changes in the orbital energy.

The SI1, SI2, and SI4 methods listed in (10.81), (10.84), and (10.86), in addition to the GLRK4 and GLRK6 methods listed in (10.54) and (10.55), share the symplectic behavior of the underlying Hamiltonian system described above. Further theoretical analysis and discussion of this interesting fact (and the fascinating and extensive literature surrounding it) is deferred to the lucid discussion in Sanz-Serna (1991).

The RK4 and SI4 methods are implemented in Algorithm 10.10 on the evolution of our solar system, as formulated in Example 10.10. The resulting orbits of the inner and outer planets, using $h \approx 1$ day, are shown in Figure 10.23. Note in particular the (correct) eccentricities of each orbit, and that the simulations match the published values of the orbital periods of Mars and Neptune to several significant figures.

Comparison of the SI4 method and the RK4 method over $T = 14$ years is given in Figure 10.24 using a timestep of $h \approx 10$ days, which is far too big for an accurate simulation, but is done here to characterize the effects of accumulating numerical errors (note that smaller timesteps lead to similar issues, they just take longer to manifest). In the simulation using RK4, Mercury's orbit *decays* until it falls in towards the sun, then *departs* the solar system altogether, while the overall energy of the orbital system increases by 1% over the period shown, then increases rapidly from there. In contrast, in the simulation using SI4, Mercury's orbit *shifts*, with initially about the same level of inaccuracy per timestep; however, these errors do not cause a significant decay or growth of Mercury's orbit, and the overall energy of the orbital system changes by a mere 0.0004% over the period shown. Though neither numerical result is perfect, the SI4 method exhibits *significantly* better stability for long-time simulations in such problems, even though it just as simple to implement as RK4, and in fact slightly cheaper per timestep to calculate.

Algorithm 10.10: SI4 and RK4 simulations of the Solar System.

View

```
function [qs,energy]=SolarSystemSimulator
% Simulate the evolution of the solar system using SI4 (or RK4).
method='SI4';                          % Try 'SI4' or 'RK4' (but check Figure 10.22 first!)
Tmax=1.8808;                           % Exact total simulation time, in sidereal years
kmax=round(Tmax*365); h=Tmax/kmax;     % Timestep (will be about 1 earth day)
% Set constants to convert distances to AU and times to sidereal years,
AU=149597870700;       % Astronomical Unit = mean orbital distance of earth from sun in m
sy=365.256363004;      % Number of days in a sidereal year
% Gravitational constant (converted from m^3/kg/s^2 to AU^3/kg/year^2)
G =6.67428e-11*(60*60*24*sy)^2/AU^3;
% Masses (kg)  [Data from Champion et al. (2010)]
M(1)=1.988920e+30;                % Sun
M(2)=3.29846e+23;                 % Mercury
M(3)=4.86854e+24;                 % Venus
M(4)=5.9736e+24 + 0.07349e+24;    % Earth + Moon
M(5)=6.41665e+23;                 % Mars
M(6)=1.899005e+27;                % Jupiter
M(7)=5.686069e+26;                % Saturn
M(8)=8.6832e+25;                  % Uranus
M(9)=1.0243e+26;                  % Neptune
% Initial positions (AU) and initial velocities (AU/day)
% [Data from Arminjon, M (2002), 00:00:00.0 on February 26, 2000.]
q=[0, 0, 0;                                                          % Sun
   -2.503321047836E-01, +1.873217481656E-01, +1.260230112145E-01;  % Mercury
   +1.747780055994E-02, -6.624210296743E-01, -2.991203277122E-01;  % Venus
   -9.091916173950E-01, +3.592925969244E-01, +1.557729610506E-01;  % Earth + Moon
   +1.203018828754E+00, +7.270712989688E-01, +3.009561427569E-01;  % Mars
   +3.733076999471E+00, +3.052424824299E+00, +1.217426663570E+00;  % Jupiter
   +6.164433062913E+00, +6.366775402981E+00, +2.364531109847E+00;  % Saturn
   +1.457964661868E+01, -1.236891078519E+01, -5.623617280033E+00;  % Uranus
   +1.695491139909E+01, -2.288713988623E+01, -9.789921035251E+00]; % Neptune
p=[0, 0, 0;                                                          % Sun
   -2.438808424736E-02, -1.850224608274E-02, -7.353811537540E-03;  % Mercury
   +2.008547034175E-02, +8.365454832702E-04, -8.947888514893E-04;  % Venus
   -7.085843239142E-03, -1.455634327653E-02, -6.310912842359E-03;  % Earth + Moon
   -7.124453943885E-03, +1.166307407692E-02, +5.542098698449E-03;  % Mars
   -5.086540617947E-03, +5.493643783389E-03, +2.478685100749E-03;  % Jupiter
   -4.426823593779E-03, +3.394060157503E-03, +1.592261423092E-03;  % Saturn
   +2.647505630327E-03, +2.487457379099E-03, +1.052000252243E-03;  % Uranus
   +2.568651772461E-03, +1.681832388267E-03, +6.245613982833E-04]; % Neptune
```

```
% Convert initial velocities (AU/day) to initial momenta (AU*kg/year)
for i=1:9, p(i,:)=M(i)*p(i,:)*sy; end
% Remove the drift of the entire system.
drift=sum(p,1)/sum(M); for i=1:9, for j=1:3, p(i,j)=p(i,j)-M(i)*drift(j); end, end
% Shift center of mass of system to the origin.
cm=[M*q(:,1) M*q(:,2) M*q(:,3)]/sum(M); for j=1:3, q(:,j)=q(:,j)-cm(j); end
% Reflect system such that [1 0 0] is normal to the plane of the ecliptic
n=0; for i=1:9, n=n+cross(q(i,:),p(i,:)); end
[s,w]=ReflectCompute(n'); [q]=Reflect(q,s,w,1,3,1,9,'R'); [p]=Reflect(p,s,w,1,3,1,9,'R');
a=q(4,2); b=q(4,3); % Rotate system such that earth is initially at 0 degrees
[c,s]=RotateCompute(a,b); [q]=Rotate(q,-c,-s,2,3,1,9,'R'); [p]=Rotate(p,-c,-s,2,3,1,9,'R');
% Set up a vector to save the simulation result, and check the initial energy
qs(:,:,1)=q; energy(1)=CheckEnergy(p,q,M,G,0,method);
% Initialize constants for SI4 time marching method of Ruth
f=2^(1/3); c(1)=1/(2*(2-f)); c(4)=c(1); c(2)=(1-f)/(2*(2-f)); c(3)=c(2);
d(1)=1/(2-f);      d(3)=d(1); d(2)=-f/(2-f);          d(4)=0;
for P=1:2, figure(P); clf; if P==1; n=5; else; n=9; end                    % Initialize plots
plot3(q(1:n,1),q(1:n,2),q(1:n,3),'k+'); hold on; view(90,0), j=1; end
for k=1:Tmax/h, t=k*h;                                  % Now perform time march using SI4 or RK4
  if method=='SI4'
    for ss=1:4, q=q+c(ss)*h*dqdt(p,M);  if ss<4, p=p+d(ss)*h*dpdt(q,M,G); end, end   % SI4
    % Note: the SI4 implementation above may be accelerated by combining the last substep
    % of each timestep with the first substep of the next, as suggested in the text.
  elseif method=='RK4'
    k1q=dqdt(p,M);                  k1p=dpdt(q,M,G);                                % RK4
    k2q=dqdt(p+(h/2)*k1p,M);        k2p=dpdt(q+(h/2)*k1q,M,G);
    k3q=dqdt(p+(h/2)*k2p,M);        k3p=dpdt(q+(h/2)*k2q,M,G);
    k4q=dqdt(p+h*k3p,M);            k4p=dpdt(q+h*k3q,M,G);
    q=q+h*(k1q/6+(k2q+k3q)/3+k4q/6); p=p+h*(k1p/6+(k2p+k3p)/3+k4p/6);
  end
  qs(:,:,k+1)=q;
  if (mod(k,5)==0 & k<=730) | mod(k,365)==0 | k==Tmax/h, l=k+1; for P=1:2, figure(P),
    if P==1, n=5; m=1; title(sprintf('Inner planets, Day=%0.2f, Year=%0.5g',t*sy,t))
    else,    n=9; m=5; title(sprintf('Outer planets, Day=%0.2f, Year=%0.5g',t*sy,t)), end
    for i=1:n
      plot3(shiftdim(qs(i,1,j:m:l)),shiftdim(qs(i,2,j:m:l)),shiftdim(qs(i,3,j:m:l)),'k-')
    end
    axis tight, axis equal, pause(0.0001);
  end, j=1; end
  if mod(k,365)==0 | k==Tmax/h, energy(end+1)=CheckEnergy(p,q,M,G,t,method); end
end
for P=1:2, figure(P), plot3(q(1:n,1),q(1:n,2),q(1:n,3),'k*'), hold off,end % Finalize plots
end % function SolarSystemSimulator
%%%
```

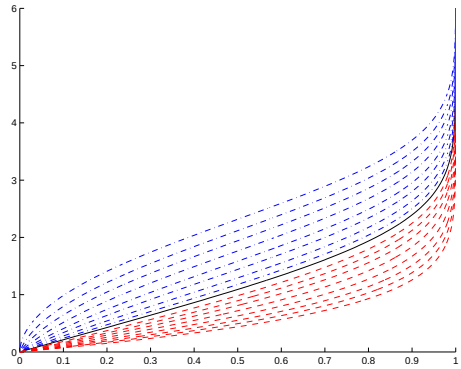


Figure 10.25: Solutions $f'(\eta)$, plotted sideways, of the Falkner-Skan equation (10.87) for various values of the nondimensionalized pressure gradient m: (dot-dashed) decelerated solutions with $m < 0$, (solid) the Blasius case with $m = 0$, (dashed) accelerated solutions with $m > 0$.

10.7 Two-point boundary value problems (TPBVPs)

The ODE problems considered thus far have all had just enough initial conditions imposed to march the ODE forward (or backward) uniquely in time (or space) from a well-defined starting point, and are thus referred to as **initial value problems** (**IVP**s). Another important class of ODE problems is set up by constraints at *both* ends of a region in space or time, and are referred to as **two-point boundary value problems** (**TPBVP**s).

A prototypical problem of this class is the **Falkner-Skan** equation for a function $f(\eta)$ modeling a **boundary layer flow**[18]. In this model, f' is proportional to the speed of the flow in the streamwise direction and η is proportional to the distance away from the surface; both f and η are appropriately nondimensionalized by the speed of the flow outside the boundary layer, U_∞, the viscosity of the fluid, ν, the streamwise distance from the leading edge, x, and the nondimensionalized streamwise pressure gradient outside the boundary layer, m. Defining the **Hartree parameter** $\beta = 2m/(m+1)$, the equation governing $f(\eta)$ is[19]

$$f''' + ff'' + \beta[1 - (f')^2] = 0 \qquad \text{with} \qquad f(0) = f'(0) = 0 \quad \text{and} \quad f'(\eta) \underset{\eta\to\infty}{\longrightarrow} 1. \tag{10.87}$$

Results for several values of m, using the numerical code developed in §10.7.1, are illustrated in Figure 10.25. Note in particular that, of the three constraints imposed, two are at $\eta = 0$ and one is as applied as $\eta \to \infty$ (in practice, the latter condition may be imposed at some "large" value of $\eta = \eta_{\max}$, such as $\eta_{\max} = 10$). We now demonstrate by example two techniques available to solve problems of this class.

10.7.1 Shooting methods

One effective method of solving two-point boundary-value problems is as follows:

- Guess the missing constraints at one of the ends of the domain under consideration, which we will call the "initial" end of the domain. Then:
- march the system defined with the additional constraints over the domain to the opposite ("terminal") end of the domain using one of the ODE marching techniques described previously in this chapter;
- refine the guessed initial constraints using a root-finding method (see §3.1); and
- repeat until the desired conditions at terminal end are matched.

[18] A boundary layer is the thin layer of fluid next to a solid surface (of, e.g., an airplane wing) that is slowed down, in the reference frame of the surface, by the effects of viscosity.

[19] Note that a special case of the Falkner-Skan equation, with the pressure gradient $m = 0$, is the **Blasius** equation $f''' + ff'' = 0$.

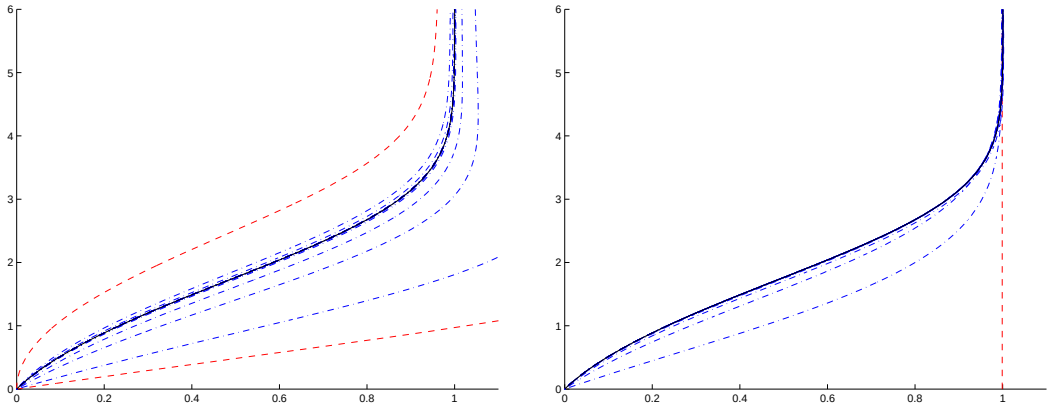


Figure 10.26: Convergence of the shooting method of §10.7.1 (left) and the relaxation method of §10.7.2 (right) for solving the Falkner-Skan equation for $m = -0.08$. Plotted are the converged solution (solid), the initial profile(s) (dashed), and the intermediate pofiles (dot-dashed).

Two simple implementations of the above algorithm on the Falkner-Skan problem described previously are illustrated in Algorithm 10.11, the first using the bisection method of §3.1.2 to refine the root, and the second using the Newton-Raphson method of §3.1.1 to refine the root. Note that the latter code incorporates the complex-step derivative method of §8.3.3 in order to compute the required derivative. The convergence of this iterative approach to the profile sought, taking $m = -0.08$, is illustrated in Figure 10.26a.

A refinement to the shooting method described above that is sometimes helpful for difficult TPBVPs is to guess the missing constraint(s) at *both* of the ends of the domain of interest, to march from both ends to some point in the center of the domain of interest, and then to iteratively refine the guessed additional constraints at both ends using an appropriate root-finding method until the two solutions match at the center point. This method is called **shooting to a fitting point**. Numerical implementation is deferred to Exercise 10.19.

10.7.2 Relaxation methods

Instead of treating just the extra initial condition(s) as the unknown(s) to be found, we may instead treat the entire solution profile, discretized on n gridpoints, as the unknowns. If the TPBVP to be solved is linear, this solution profile may be computed directly. For a nonlinear TPBVP like the Falkner-Skan problem described above, a **relaxation** procedure must be formulated and iterated until convergence from some initial guess.

To illustrate, one effective approach to the Falkner-Skan problem is to write an iteration (in k) which is linear in the unknown [denoted, for simplicity, as $f_{k+1}(\eta)$] such that, if/when the relaxation iteration converges, the converged solution solves the original nonlinear ODE. One such linearization of the ODE (10.87) may be written as

$$f'''_{k+1} + [f''_k f_{k+1} + f_k f''_{k+1}]/2 - \beta f'_k f'_{k+1} = -\beta. \tag{10.88}$$

Implementation, using second-order FD methods to discretize the derivatives, is given in Algorithm 10.12. Convergence of this iterative approach to the profile sought, taking $m = -0.08$, is illustrated in Figure 10.26b.

The relaxation approach is initialized by a guess of the solution over the entire domain, which is sometimes reasonably well known, rather than a guess of the missing initial condition(s) in the shooting approach, which is sometimes difficult to approximate. Thus, it is often easier to obtain convergence with a relaxation method. A disadvantage of the relaxation approach is that it requires storage of the entire solution profile.

At iteration k, the relaxation method (10.88) for computing the Blasius boundary layer [that is, the solution of (10.87) with $\beta = 0$] would converge in a single additional iteration if we could compute an update $\varepsilon_k(\eta)$

to the profile $f_k(\eta)$ on $0 < \eta < \eta_{\max}$ such that

$$(f_k+\varepsilon_k)'''+(f_k+\varepsilon_k)(f_k+\varepsilon_k)''=0 \quad \Leftrightarrow \quad \varepsilon_k'''+f_k\varepsilon_k''+f_k''\varepsilon_k+\varepsilon_k\varepsilon_k''=-(f_k'''+f_kf_k''). \tag{10.89a}$$

Unfortunately, this nonlinear equation can not be solved directly. Following (10.88), we instead solve

$$(f_k+\delta_k)'''+[(f_k+\delta_k)f_k''+f_k(f_k+\delta_k)'']/2=0 \quad \Leftrightarrow \quad \delta_k'''+[f_k\delta_k''+f_k''\delta_k]/2=-(f_k'''+f_kf_k''). \tag{10.89b}$$

Note that the nonlinear ODE at right in (10.89a), governing the deviation $\varepsilon_k(\eta)$ from the exact solution at iteration k, and the linear ODE at right in (10.89b), governing the update $\delta_k(\eta)$ to be made at iteration k, are both driven by $f_k'''+f_kf_k''$ on the RHS, and thus both the deviation $\varepsilon_k(\eta)$ and the update $\delta_k(\eta)$ go to zero upon convergence. The LHS of these two system differ in two important ways. First, the LHS of (10.89b) is missing the nonlinear term present in (10.89a); if ε_k is sufficiently small, then this term is small compared to the linear terms. Second, the linear operator on the LHS of (10.89b) for computing the correction $\delta_k(\eta)$ has two terms which are a factor of 2 too small; the method converges regardless. To understand its convergence, we interpret the FD discretization of (10.89b) in η as a **splitting method** [see §3.2.1, and (3.8a)-(3.8c) in particular] written in terms of a **correction** at each iteration [in this case, denoted δ_k] based on a **defect** [in this case, denoted $(f_k'''+f_kf_k'')$]. As Gauss-Seidel converges faster than Jacobi, the present relaxation scheme would converge faster if a greater fraction of the linear operator on the LHS of (10.89a) were included in the matrix operator on the LHS of the discretization of (10.89b) [denoted M in (3.8b)] at each iteration. Regardless, as seen in Figure 10.26b, the present relaxation method converges remarkably quickly given an adequate initial guess of the profile, $f_0(\eta)$; further, a rather poor initial guess is entirely sufficient in this case.

Exercises

Exercise 10.1 The simple **Lotka-Volterra Predator/Prey** model is given by

$$\mathbf{x}'=\mathbf{f}(\mathbf{x}) \quad \text{with} \quad \mathbf{x}=\begin{pmatrix}x_1\\x_2\end{pmatrix}=\begin{pmatrix}\text{prey population}\\\text{predator population}\end{pmatrix}, \quad \mathbf{f}(\mathbf{x})=\begin{pmatrix}(b-px_2)x_1\\(rx_1-d)x_2\end{pmatrix}; \tag{10.90}$$

Without predators, the prey population increases (exponentially) without bound, whereas without prey, the predator population diminishes (exponentially) to zero. The nonlinear interaction, with predators eating prey, tends to diminish the prey population and increase the predator population. Extend Algorithm 10.4 for this system and simulate using RK4/5, taking $b=p=r=d=1$, $x_1(0)=0.3$, and $x_2(0)=0.2$.

Exercise 10.2 Extending the analysis of the RK2 scheme in (10.22) and the discussion that follows, verify that the RKW3 scheme proposed in (10.37) is indeed third-order accurate. Hint: eliminate all terms proportional to h^4 as early as possible to simplify the analysis.

Exercise 10.3 Verify that the RK[2R] and RK[3R] algorithms given in (10.35) and (10.36) are equivalent to the full-storage RK implementation given in (10.20).

Exercise 10.4 Recall the low-storage (2-register & 3-register) explicit RK schemes given in (10.35) & (10.36) for arbitrary s. Devise an analogous general form for a 4-register explicit low-storage RK scheme, illustrating both the constraints on the coefficients of its Butcher tableau, and how it may be implemented using only 4 registers for arbitrary s.

Exercise 10.5 Following the adaptive timestepping algorithm implemented in Algorithm 10.5, together with the hint in Footnote 9 on Page 291, implement adaptive timestep control on the RK435 and RK548 methods developed in the text, using both the b_i and $\hat{b}_i$ coefficients listed in order to establish two different estimates of $\mathbf{x}$ at each new timestep, which may be compared.

Algorithm 10.11: Solve the Falkner-Skan problem via the shooting method of §10.7.1, following (a) the bisection approach and (b) the Newton-Raphson approach, together with the necessary supplemental codes.

View

```
% script <a href="matlab:FS_Bisection_Test">FS_Bisection_Test</a>
figure(1); clf; axis([0 1 0 6]); hold on; format compact;
m=0, f3l=0; f3u=1; yl=FSmarch(f3l,0,m); yu=FSmarch(f3u,0,m);
while abs(f3u-f3l)>6e-16   % Refine guess for f''(0) using bisection algorithm
  f3=(f3u+f3l)/2, y=FSmarch(f3,0,m);  if yl*y<0; f3u=f3; yu=y; else; f3l=f3; yl=y; end
end
f3=(f3u+f3l)/2, disp(sprintf('Error in terminal condition = %0.5g',FSmarch(f3,1,m)))
% end script FS_Bisection_Test
```

View

```
% script <a href="matlab:FS_NR_CSD_Test">FS_NR_CSD_Test</a>
figure(1); clf; axis([0 1 0 6]); hold on; ep=1e-14; epim=ep*sqrt(-1); f3=0.0035;
m=[-.09042 -.0892 -.086 -.08 -.07 -.058 -.04 -.02 0 .035 .075 .13 .225 .39 .67 1.15 2.4]';
for j=1:size(m,1);        % Loop over several interesting values of m;
  disp(sprintf('\nFalkner Skan profile for m = %0.5g',m(j)))
  f3=f3*1.125                    % This heuristic provides a good initial guess for f''(0)
  for i=1:15                     % Iteratively refine guess for f''(0)
    x=f3+epim; y=FSmarch(x,0,m(j));      % Use complex-step method to determine derivative
    f3old=f3; f3=f3-real(y)*ep/imag(y)   % Update f''(0) using Newton-Raphson
    if abs(f3-f3old)<6e-16, break; end;  % Break out of loop if converged
  end
  disp(sprintf('Error in terminal condition = %0.5g',FSmarch(f3,1,m(j)))); pause(0.001);
end
% end script FS_NR_CSD_Test
```

View

```
function y=FSmarch(x,v,m)
h=0.01; etamax=10; eta=0; f=[0; 0; x];  if v; f2save=real(f(2));  etasave=eta;  end;
for n=1:etamax/h                % March Falkner-Skan equation over [0,etamax] with RK4
  k1=RKrhs(f,m); k2=RKrhs(f+(h/2)*k1,m); k3=RKrhs(f+(h/2)*k2,m); k4=RKrhs(f+h*k3,m);
  f=f+(h/6)*k1+(h/3)*(k2+k3)+(h/6)*k4;  eta=eta+h;
  if v; f2save=[f2save; real(f(2))];  etasave=[etasave; eta]; end
end
if v; if m<0; s='b-.'; elseif m>0; s='r--'; else; s='k-'; end; plot(f2save,etasave,s); end
y=f(2)-1;
end % function FSmarch
%%%
```

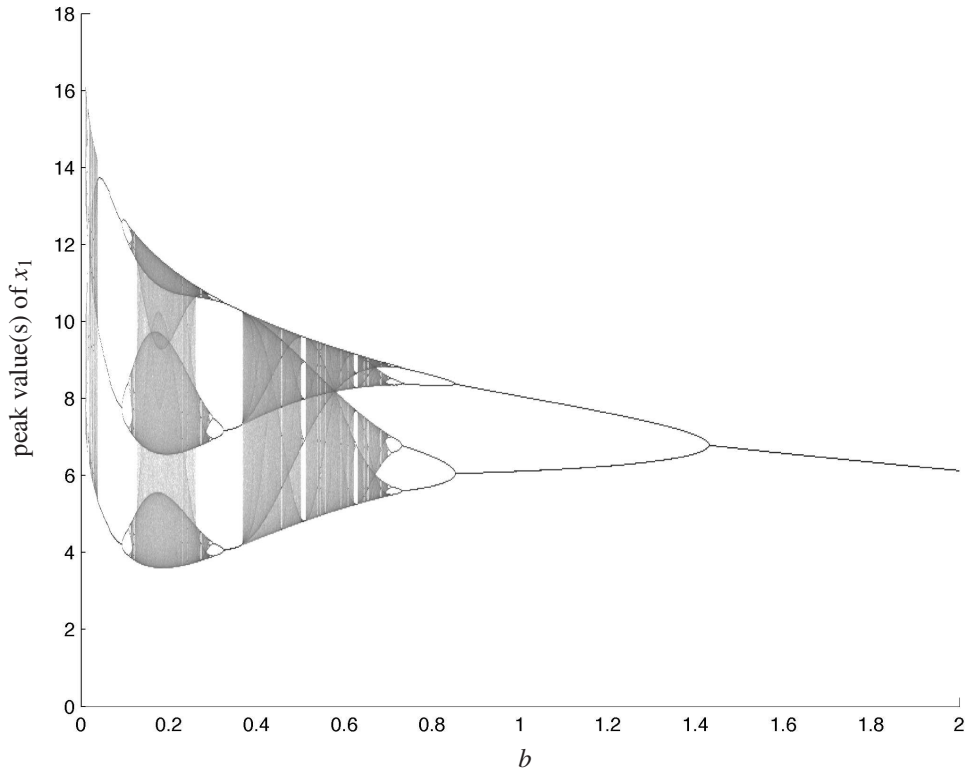


Figure 10.27: Bifurcation plot of the Rössler attractor with $a = 0.2$, $c = 5.7$, and several values of b, indicating the range of values of b for which the system is periodic and the range for which the system is chaotic.

Exercise 10.6 Modify Algorithm 10.4 to simulate the Rössler equation for $a = 0.2$, $c = 5.7$, and several values of b (ranging from $b = 0.01$ through $b = 1.5$). For each value of b in this range, simulate the Rössler equation for hundreds of cycles and keep track of the peak value of x_1 during each cycle on the attractor (that is, after the first few cycles). On a plot of x_1 versus b, put a dot corresponding to each peak value; the result is a **bifurcation plot**, as illustrated in Figure 10.27. Hint: first get the code working on a coarse grid in b and tens of cycles along the attractor for each value of b. Based on this test, determine how much refinement you can do and still have the code complete in 12 hours, and run the code overnight. The result should look something like that shown in Figure 10.27. Repeat the exercise for the peaks in x_3. For which values of b is the system periodic (and with what period), and for which values of b the system is chaotic? Discuss.

Exercise 10.7 Consider the class of (explicit) multistep **Nyström** methods

$$\mathbf{x}_{n+1} = \mathbf{x}_{n-1} + \sum_{i=1}^{r} \beta_i \mathbf{f}(\mathbf{x}_{n+1-i}, t_{n+1-i})$$

for the case with $r = 1$, which is called the **leapfrog** method, and the case $r = 2$. Repeat the analysis of the LMMs studied in §10.4.2.1 and §10.4.2.2 by applying each of these cases to the scalar model problem $dx/dt = \lambda x$, assuming constant timestep h and a solution of the form $x_n = \sigma^n x_0$, thereby determining a simple polynomial for σ. This polynomial has multiple roots, all but one of which are spurious roots that, to leading order, are proportional to h and thus, for small h, may be neglected. The expansion in h of the remaining root resembles a Taylor series expansion of the exact solution for σ; use this root to determine the constraints on the coefficients of the method necessary to achieve the highest order of accuracy possible. Modify Algorithm 10.2 to plot the stability contours of the resulting methods.

Exercise 10.8 Consider the class of (implicit) multistep **Milne-Simpson** methods

$$\mathbf{x}_{n+1} = \mathbf{x}_{n-1} + \sum_{i=0}^{r} \beta_i \mathbf{f}(\mathbf{x}_{n+1-i}, t_{n+1-i}).$$

The case with $r = 0$ recovers the implicit Euler method with a stepsize of $2h$. The case with $r = 1$ results in $\beta_0 = 0$, thus recovering the (explicit) leapfrog method derived in Exercise 10.7. Repeat Exercise 10.7 for the case with $r = 2$, known as the **Milne method**.

Exercise 10.9 Compute the A and $\mathbf{b}$ coefficients of the standard RK Butcher tableau as simple functions of the μ_j, ν_j, $\widetilde{\gamma}_j$, and $\widetilde{\mu}_j$ coefficients of the explicit RK form given (10.47). If function evaluations $\mathbf{f}_j = \mathbf{f}(\mathbf{y}_j, t_k + c_j h)$ can be computed in place in computer memory, how many registers does it take to implement the RK scheme in the form given (10.47) for large s? Is this consistent with the constraints on the coefficients of the Butcher tableau that you expect from Exercise 10.4? Discuss.

Exercise 10.10 Recall the iterative CN method written as a DIRK scheme in (10.56), and DIRK4[2P] method in (10.58). Combining the key ideas of these two methods, optimize the coefficients of a new **DIRK3[2P]** scheme with Butcher tableau

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ c_2 & a_{2,1} & a_{2,2} & 0 \\ c_3 & a_{3,1} & 0 & a_{3,3} \\ \hline & b_1 & b_2 & b_3 \end{array} \tag{10.91}$$

with $0 < c_2 < c_3 \leq 1$, to achieve third-order accuracy (if possible, take $c_2 = 1/2$ and $c_3 = 1$). Is the resulting scheme FSAL? Explain how this scheme leverages well a two-processor architecture.

Exercise 10.11 Derive the ESD3 method discussed in §10.5.3.2 and listed in Table 10.5 following a similar approach as that used for deriving the BDF2 method in §10.5.3.1.

Exercise 10.12 In the derivation of the coefficients of LMMs that lead to *quadratic* stability polynomials, the convenient quadratic formula may be used to compute the physical root σ_+ and the spurious root σ_-. For higher-order LMMs, a slightly different approach is needed.

- For the AB & AM methods, we first calculate a Lagrange interplant $\tilde{f}(t)$ [see §7.3.2] based on the available values of $\{f(x_{n+1-i}), t_{n+1-i}\}$ used by the scheme [see (10.41) & (10.42)], then integrate this interplant from t_n to t_{n+1} to approximate x_{n+1} based on x_n in accordance with (10.1b).
- For the BDF & eBDF methods, we first calculate a Lagrange interplant $\tilde{x}(t)$ based on the available values of $\{x_{n+1-i}, t_{n+1-i}\}$ used by the scheme [see (10.59) & (10.60)], then differentiate this interplant at t_{n+1} (in the BDF case) or t_n (in the eBDF case), and scale the result by the coefficient of x_{n+1}, in order to determine the coefficients in (10.59) & (10.60).

In both cases, it simplifies the algebra somewhat to focus, without loss of generality, on the case $n = 0$ in the derivation. Using these techniques, rederive the (constant-h) AB4 and AB5 schemes presented in Table 10.1, as well as the (constant-h) BDF4 and BDF5 schemes presented in Table 10.3. Then, rederive all four of these schemes, accounting for timesteps $h_n = t_{n+1} - t_n$ that vary from one timestep to the next, and verify that the (algebraically, somewhat complex) schemes so derived simplify appropriately when h is made constant.

Exercise 10.13 Recalling the RK4/5 method of §10.4.1.2 and the embedded RK algorithms of §10.4.1.3, and leveraging the variable-timestep AB formulae derived in Exercise 10.12, develop an adaptive timestepping scheme for AB5 based (conservatively) on an error estimates from AB4, implement in numerical code, test on the Lorenz problem (10.28), and compare with Algorithm 10.5 and Figure 10.11. Discuss.

Exercise 10.14 Verify the properties of the Poisson bracket 10.74 listed in Footnote 16 on Page 317.

Exercise 10.15 Recall that any (implicit or explicit) RK method may be written as in (10.53) [ERK methods have strictly lower triangular A, and DIRK methods have lower triangular A]. We now apply such an RK method to the scalar model problem $dx/dt = \lambda x$; that is, we reduce all vectors to scalars and take $f(x,t) = \lambda x$ in (10.53). We then assemble the s scalar "slope" computations over each timestep as a new vector $\mathbf{f} = \begin{pmatrix} f_1 & f_2 & \dots & f_s \end{pmatrix}^T$, and write these s equations together in vector form as $\mathbf{f} = \lambda(\mathbf{1}x_n + hA\mathbf{f})$ where $\mathbf{1}$ is a vector of ones. Solving this equation for $\mathbf{f}$ and substituting the result into the equation for x_{n+1} in (10.53), taking $\mathbf{b} = \begin{pmatrix} b_1 & b_2 & \dots & b_s \end{pmatrix}^T$ and $x_n = \sigma^n x_0$, and simplifying appropriately, verify algebraically that, for any RK method,

$$\sigma = 1 + \lambda h \mathbf{b}^T (I - \lambda h A)^{-1} \mathbf{1}. \tag{10.92}$$

By Cramer's rule (Fact 4.3), $B^{-1} = B_{\text{cof}}/|B|$, where $|B|$ is the determinant of $B = B_{n\times n}$ and

- the **minor** $M_{\alpha\beta}$ of B is the $(n-1)\times(n-1)$ matrix formed by deleting row α and column β of B,
- the **cofactor** $B_{\alpha\beta}$ is the determinant of $M_{\alpha\beta}$ with alternating sign (that is, $B_{\alpha\beta} = (-1)^{\alpha+\beta}|M_{\alpha\beta}|$), and
- the $\{i,j\}$ element of the **cofactor matrix** B_{cof} is the cofactor B_{ji}.

Based on Cramer's rule applied to $B = \ell I - A$ where $\ell = 1/(\lambda h)$, establish from (10.92) that

- σ for *any* (implicit or explicit) s-stage RK method is a rational function of (λh) [that is, σ is a polynomial in (λh), of order s, divided by another polynomial in (λh), also of order s], and
- σ for an *explicit* s-stage RK method is simply a polynomial in (λh) of order s [and, thus, if an s-stage RK method is also order s, this polynomial is just a truncation of the Taylor series expansion of the exact value for σ, given by $e^{\lambda h} = 1 + \lambda h + (\lambda h)^2/2! + (\lambda h)^3/3! + \dots$].

As a hint for the second half of this problem, note that Cramer's rule is used in a similar fashion in §20.1.5.

Exercise 10.16 We now examine the convergence of the iterative CN method (10.25) applied to the scalar problem $dx/dt = f(x)$.

a) First, perform a Taylor series expansion, in x, of $f(x)$ in the vicinity of $x = x_n$ [that is, write $f(x) =$ $f(x_n) +$ additional terms which depend on $(x - x_n)$ and the derivates of f with respect to x evaluated at $x = x_n$; explicitly write out two such additional terms].

b) Then, substitute this Taylor series into (10.25b), and subtract from the result the Taylor series expansion, in t, of the exact solution over a single time step, $x_{n+1} = x_n + hf(x_n) + h^2[f_x f]_{x=x_n}/2! + \dots$

c) Finally define $\varepsilon_k = x_k^* - x_{n+1}$, rewrite $x_{k-1}^* - x_n$ in the expression found in step b as $\varepsilon_{k-1} + (x_{n+1} - x_n)$, and substitute in the above Taylor series for $(x_{n+1} - x_n)$ where appropriate. Based on this manipulation, establish that ε_k decreases with k, and at precisely what rate, if h and ε_0 are both sufficiently small.

Exercise 10.17 Consider the following implicit RK method, dubbed the "**iterative midpoint**" method

1/2	1/2
	1

- Derive the stability polynomial of this method, noting Exercise 10.15. What is its local order of accuracy? What is its global order of accuracy? Which is more usually most relevant? Why?
- Sketch the stability region for this method. Is it A stable? Is it L stable? Explain.

Exercise 10.18 Following the discussion in the text, verify that the SI4 scheme (10.86) is globally fourth-order accurate [hint: don't forget to apply (10.78) whenever possible].

Exercise 10.19 Rewrite the Newton-Raphson-based shooting method implemented Algorithm 10.11 to incorporate the shooting to a fitting point method described at the end of §10.7.1, and test the method on the Falkner-Skan problem with $m = -0.09$ to show that it works.

Exercise 10.20 Assume f is the solution to the Falkner-Skan equation (10.87) with a given value of the Hartree parameter β. Write two codes (one using shooting, one following a relaxation approach) to solve the associated TPBVP

$$g'' + fg' = 0 \qquad \text{with} \qquad g(0) = 0 \quad \text{and} \quad g(\eta) \underset{\eta\to\infty}{\longrightarrow} 1.$$

This is called the Falkner-Skan-Cooke problem; the solution $\{f, g\}$ models a three-dimensional boundary layer, with the streamwise velocity proportional to $f'(\eta)$ and the spanwise velocity proportional to $g(\eta)$.

References

Abdulle, A, & Medovikov, AA (2001) Second order Chebyshev methods based on orthogonal polynomials, *Numer. Math.* **90** 118.

Alexander, R (2003) Design and implementation of DIRK integrators for stiff systems, *Appl. Numer. Math.* **46**, 1-17.

Arminjon, M (2002) Proper initial conditions for long-term integrations of the solar system, *Astronomy & Astrophysics* **383**, 729-737.

Ascher, UM, & Petzold, LR (1998) *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM.

Ascher, UM, Ruuth, SJ, & Spiteri, RJ (1997) Implicit-explicit Runge-Kutta methods for time dependent Partial Differential Equations, *Appl. Numer. Math.* **25**, 151-167.

Butcher, JC (2008) *Numerical methods for ordinary differential equations.* Wiley.

Champion, DJ, et al. (2010) Measuring the mass of solar system planets using pulsar timing, *The Astrophysical Journal Letters*, **720**, L201-L205.

Dahlquist, G (1963) A special stability problem for linear multistep methods, *BIT* **3**, 27-43.

Enright, WH (1974) Second Derivative Multistep Methods for Stiff Ordinary Differential Equations, *SIAM Journal on Numerical Analysis* **11**, 321-331.

Gautschi, W (1997) *Numerical Analysis: an Introduction*. Birkhäuser.

Gear, CW (1971) *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice Hall.

Hairer, E, Norsett, SP,& Wanner, G (2008) *Solving Ordinary Differential Equations I: Nonstiff problems*. Springer.

Hairer, E, & Wanner, G (2004) *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer.

van der Houwen, PJ (1977) *Construction of integration formulas for initial value problems*. North-Holland.

Jackson, KR, & Norsett, SP (1990) The potential for parallelism in Runge Kutta Methods. Part 1: RK Formulas in Standard Form. *University of Toronto Computer Science Department, Technical Report No.* 239/90.

Kennedy, CA, Carpenter, MH, & and Lewis, RM (2000) Low-storage, explicit Runge-Kutta schemes for the compressible Navier-Stokes equations, *Applied Numerical Mathematics*, **35**, 177-219.

Pareschi, L, & Russo, G (2001) Implicit-explicit Runge-Kutta schemes for stiff systems of differential equations, in: D. Trigiante (Ed.), *Recent Trends in Numerical Analysis*, Nova Science Publishers Inc., pp. 269-288.

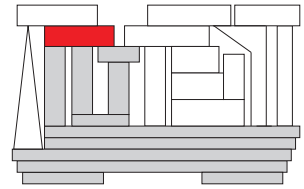
Sanz-Serna, J.M. (1991) Symplectic integrators for Hamiltonian problems: an overview, *Acta Numerica*, **1**, 243-286.

Süli, E, & Mayers, DF (2003) *Numerical Analysis: An Introduction*. Cambridge University Press.

Verwer, JG, Hundsdorfer, WH, & Sommeijer BP (1990) Convergence properties of the Runge-Kutta-Chebyshev method, *Numer. Math.* **57**, 157-178.

Wood, WL (1990) *Practical Time-stepping Schemes*. Oxford.

Wray, AA (1986) Minimal-storage time advancement schemes for spectral methods, *Technical Report, NASA Ames Research Center*.



Chapter 11

Partial differential equations

Contents

11.1 Classification, analysis, and analytic solution of PDEs . . . 334
11.1.1 The (elliptic) Laplace and Helmholtz equations . . . 336
11.1.2 The diffusion equation, and other parabolic systems . . . 337
11.1.2.1 Ill-posed problems . . . 337
11.1.2.2 Tikhonov regularization . . . 338
11.1.2.3 Inverse problems . . . 340
11.1.2.4 Convenient scaling of space, time, and the unknown(s) in a PDE . . . 340
11.1.2.5 Spectra, chaos, and the cascade of energy in chaotic nonlinear PDE systems . . . 341
11.1.3 The wave equation, and other hyperbolic systems . . . 344
11.1.3.1 Spherically-symmetric and azimuthally-symmetric solutions of the 3D wave equation . . . 344
11.1.3.2 Dispersion, and the difference between phase velocity and group velocity . . . 346
11.1.3.3 Damping . . . 347
11.1.3.4 The nonlinear 2D shallow water equation . . . 348
11.1.3.5 Jump conditions and the Rankine-Hugonoit relations . . . 352
11.1.4 Classification of more complicated PDEs . . . 354
11.1.5 Wavelike behavior and solitons . . . 355
11.1.6 The fundamental equations of computational fluid dynamics . . . 357
11.2 Numerical simulation of parabolic PDEs . . . 359
11.2.1 Time marching a spatial discretization developed via finite differences . . . 359
11.2.1.1 A typical tradeoff between flops and storage . . . 360
11.2.2 Time marching a spatial discretization developed via spectral methods . . . 361
11.2.3 Von Neumann stability analysis . . . 363
11.2.4 Consistency and the Dufort-Frankel scheme . . . 363
11.3 Numerical simulation of hyperbolic PDEs . . . 365
11.3.1 The value of high-order spatial discretization . . . 365
11.3.2 The value of a temporal discretization designed for second-order systems . . . 367
11.3.3 The value of pseudospectral methods . . . 368
11.3.4 Mixing finite difference and pseudospectral methods . . . 369

11.3.5 Godunov methods . . . 369
11.4 Numerical solution of elliptic PDEs . . . 373
11.4.1 Multigrid revisited: a Rosetta stone in Matlab, Fortran, and C . . . 373
Exercises . . . 387

We have, by now, seen how to approximate spatial derivates with both spectral (§5) and finite difference (§8) methods. Such methods may be used to approximate a **partial differential equation** (**PDE**) with either a system of algebraic equations or a system of ordinary differential equations[1], with one equation corresponding to each spatial gridpoint or Fourier mode. We have also (in §2 and §3) studied how to solve large systems of algebraic equations and (in §10) studied how to march ODEs forward in time. Thus, we already have all of the basic tools required to approximate numerically a wide variety of PDE systems: first, approximate the (infinite-dimensional) PDE of interest with a (finite-dimensional) system of algebraic equations or ODEs, then either solve the resulting system of algebraic equations or march the resulting system of ODEs in time. This chapter addresses some of the peculiar issues that arise when following such a procedure.

Before we get started, it is useful to reflect for a moment on coding philosophy. Up to now, we have been careful to keep our implementations of numerical methods as generic as possible, with application-specific functions isolated clearly in subroutines (see, e.g., Algorithm 10.3). The simulation of PDE systems is substantially more difficult, and we can usually no longer afford this luxury while achieving maximum efficiency. The codes given in this chapter are thus tuned to the specific problems to which they are applied; however, the problems selected demonstrate generic concepts typical in a range of PDE simulation problems.

11.1 Classification, analysis, and analytic solution of PDEs

The zoology of PDEs is beautifully and bewilderingly diverse. Many PDEs may be expressed as a minor variation of one of the several canonical forms introduced below. Before diving into the numerical simulation of PDEs, it is thus instructive to survey their classification and some of their significant properties.

Recalling the gradient, divergence, curl, Laplacian, and bilaplacian operators of vector calculus (§B.4), four prototypical PDEs [each in one, two, or three spatial dimensions (**1D**, **2D**, or **3D**)] which form a natural starting point for a study of PDEs are the (elliptic) **Laplace** (a.k.a. **steady heat**) equation:

$$\Delta\phi = 0, \tag{11.1}$$

the (elliptic) **Helmholtz** equation:

$$\Delta\phi + \alpha^2\phi = 0, \tag{11.2}$$

the (parabolic) **diffusion** (a.k.a. **unsteady heat**) equation:

$$\frac{\partial\phi}{\partial t} - \nu\Delta\phi = 0, \tag{11.3}$$

and the (hyperbolic) **wave** equation:

$$\frac{\partial^2\phi}{\partial t^2} - c^2\Delta\phi = 0, \tag{11.4}$$

where, in (11.3) and (11.4), t denotes either time or a "time-like" variable[2] in which the system evolves in a **causal** fashion forward in t from initial conditions (**ICs**), defined here WLOG at $t = 0$, while additionally

[1] In the case of a time-varying system, approximation of a PDE as a system of ODEs is sometimes referred to as **semi-discretization**.

[2] A well-known example of a "time-like" variable is the distance downstream (x) in thin boundary layer over a wing, which is well approximated by the parabolic-in-space equation (11.62), which is similar in structure to the parabolic-in-time equation (11.3).

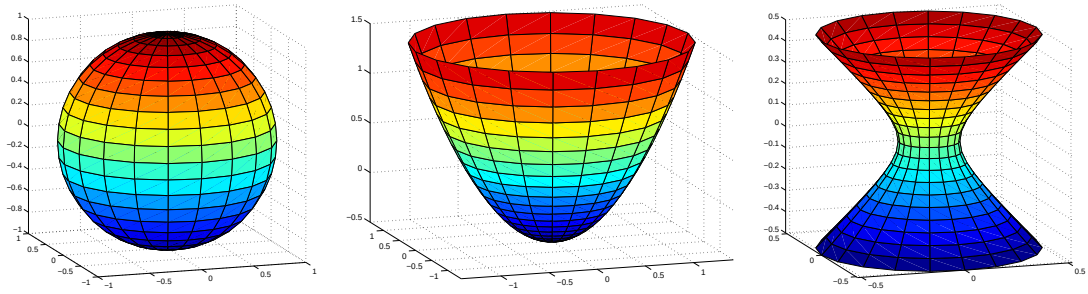


Figure 11.1: Isosurfaces of (a) the ellipse $\ell_x^2+\ell_y^2+\ell_z^2=c_0$ related to the **elliptic** PDEs (11.1) and (11.2) in 3D, (b) the parabola $s-\nu(\ell_x^2+\ell_y^2)=c_0$, related to the **parabolic** PDE (11.3) in 2D, and (c) the hyperbola $s^2-c^2(\ell_x^2+\ell_y^2)=c_0$, related to the **hyperbolic** PDE (11.4) in 2D.

satisfying the relevant boundary conditions (**BCs**) on the system. The diffusion and wave equations are thus referred to as **initial-value problems** (**IVPs**, a.k.a. **Cauchy problems**); in contrast, as discussed further below, the Laplace and Helmholtz equations are set up entirely by their boundary conditions around the domain of interest, and are thus referred to as **boundary-value problems** (**BVPs**).

Motivated by the SOV analysis method introduced in §4.3.2 and the infinite Fourier integral introduced in §5.3 (and, perhaps, the Laplace transform discussed in §17.3), many linear, constant-coefficient PDE systems on simple domains may be solved by simple superposition of separable modes of the form[3]

$$\phi=\hat{\phi}\,e^{st+\vec{\ell}\cdot\vec{x}} \quad \text{where} \quad s\triangleq-\sigma+\mathrm{i}\omega \quad \text{and} \quad \vec{\ell}\triangleq\vec{m}+\mathrm{i}\vec{k}, \tag{11.5}$$

where σ, ω, $\vec{m}$, and $\vec{k}$ are real. An equation relating the various components of $\vec{\ell}$ (and, in unsteady problems, s) to ensure the modes themselves satisfy the PDE may be obtained by inserting (11.5) into the PDE under consideration. The resulting quadratic equation is referred to as the **dispersion relation** (a.k.a. the **characteristic polynomial equation**) of the corresponding PDE, and reveals much about the nature of its solution even before the precise RHS forcing, BCs, and (in unsteady problems) ICs are applied. For the four prototypical PDEs introduced above in 3D, the corresponding dispersion relations so generated are given by

$$(11.1) \quad \Rightarrow \quad s=0, \quad \ell_x^2+\ell_y^2+\ell_z^2=0; \tag{11.6}$$

$$(11.2) \quad \Rightarrow \quad s=0, \quad \ell_x^2+\ell_y^2+\ell_z^2+\alpha^2=0; \tag{11.7}$$

$$(11.3) \quad \Rightarrow \quad s-\nu(\ell_x^2+\ell_y^2+\ell_z^2)=0; \tag{11.8}$$

$$(11.4) \quad \Rightarrow \quad s^2-c^2(\ell_x^2+\ell_y^2+\ell_z^2)=0. \tag{11.9}$$

As illustrated in Figure 11.1, representing nonzero forcing in the problems (11.1)-(11.4) by taking the right-hand sides of the above expressions as nonzero, the first two equations above are seen to be those of an ellipsoid (in particular, a sphere), the third is that of a paraboloid, and the fourth is that of a hyperboloid, thereby bestowing the names **elliptic**, **parabolic**, and **hyperbolic** to these basic types of PDEs.

[3] A common alternative formulation of this analysis defines modes with the factor $e^{\mathrm{i}(\omega t+\vec{k}\cdot\vec{x})}$, where the frequency ω and the wavenumber $\vec{k}$ are taken to be complex. Following this convention, time-varying PDEs with (complex) ω in the *lower* half plane are stable, and time-varying PDEs with (complex) ω in the *upper* half plane are unstable. We instead choose to use the convention related to the Laplace transform variable s throughout this discussion for consistency with §17.3; following this convention, time-varying PDEs with (complex) s in the *left* half plane are stable, and time-varying PDEs with (complex) s in the *right* half plane are unstable.

11.1.1 The (elliptic) Laplace and Helmholtz equations

The Laplace equation (11.1) in a 3D rectangular domain with periodic boundary conditions in x and y, with dispersion relation (11.6), may be solved via superposition of separable modes of the form (11.5) as follows:

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}=0, \quad \phi=\hat{\phi}\,e^{st+\vec{\ell}\cdot\vec{x}} \quad \Rightarrow \quad s=0, \quad \ell_x^2+\ell_y^2+\ell_z^2=0.$$

With $\vec{\ell}=\vec{m}+\mathrm{i}\vec{k}$, and choosing, for example[4], $m_x=m_y=0$, leads to $-k_x^2-k_y^2+\ell_z^2=0$ and thus

$$\left\{\begin{array}{l} m_z^{\pm}=\pm\sqrt{k_x^2+k_y^2} \\ k_z=0 \end{array}\right\} \quad \Rightarrow \quad \phi(\vec{x},t)=\sum_{p,q}(\hat{\phi}^+_{p,q}e^{m^+_{z\,p,q}z}+\hat{\phi}^-_{p,q}e^{m^-_{z\,p,q}z})e^{\mathrm{i}(k_{x_p}x+k_{y_q}y)}. \tag{11.10}$$

It is thus seen that a superposition of separable modes of the form (11.5) with $s=0$, where ℓ is selected to satisfy the dispersion relation (11.6), may be used to solve the diffusion equation (11.1) in simple domains. The Fourier coefficients $\hat{\phi}^+_{p,q}$ and $\hat{\phi}^-_{p,q}$ in this case may be found by Fourier transform of the BCs on ϕ at $z=0$ and $z=L_z$, with the spatial wavenumbers included in the series selected appropriately to fit the domain of interest (that is, as in §5.10, $k_{x_p}=2\pi/\lambda_{x_p}$ and $k_{y_q}=2\pi/\lambda_{y_q}$ where $\lambda_{x_p}=L_x/p$ and $\lambda_{y_q}=L_y/q$ for integer $\{p,q\}$). Though in principle the Fourier coefficients $\hat{\phi}^+_{p,q}$ and $\hat{\phi}^-_{p,q}$ may be found by Fourier transform of the BCs on both ϕ and $\partial\phi/\partial z$ at $z=0$, the existence of both growing and decaying exponentials in z in (11.10) renders such an approach **ill posed** in a manner similar to backwards heat equation problem discussed further in the parabolic case below, and BCs all around the domain of interest are strongly preferred.

The Helmholtz equation (11.2) in a analogous 3D domain may be solved similarly:

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}+\alpha^2\phi=0, \quad \phi=\hat{\phi}\,e^{st+\vec{\ell}\cdot\vec{x}} \quad \Rightarrow \quad s=0, \quad \ell_x^2+\ell_y^2+\ell_z^2+\alpha^2=0.$$

With $\vec{\ell}=\vec{m}+\mathrm{i}\vec{k}$, choosing $m_x=m_y=0$ leads to $-k_x^2-k_y^2+\ell_z^2=-\alpha^2$ and thus

$$\left\{\begin{array}{ll} k_z^{\pm}=\pm\sqrt{\alpha^2-k_x^2-k_y^2}, & m_z=0 \quad \text{if } k_x^2+k_y^2\le\alpha^2 \\ m_z^{\pm}=\pm\sqrt{k_x^2+k_y^2-\alpha^2}, & k_z=0 \quad \text{if } k_x^2+k_y^2>\alpha^2 \end{array}\right\} \quad \Rightarrow$$

$$\phi(\vec{x},t)=\sum_{\substack{p,q \\ k_{x_p}^2+k_{y_q}^2\le\alpha^2}}(\hat{\phi}^+_{p,q}e^{\mathrm{i}k^+_{z\,p,q}z}+\hat{\phi}^-_{p,q}e^{\mathrm{i}k^-_{z\,p,q}z})e^{\mathrm{i}(k_{x_p}x+k_{y_q}y)}+\sum_{\substack{p,q \\ k_{x_p}^2+k_{y_q}^2>\alpha^2}}(\hat{\phi}^+_{p,q}e^{m^+_{z\,p,q}z}+\hat{\phi}^-_{p,q}e^{m^-_{z\,p,q}z})e^{\mathrm{i}(k_{x_p}x+k_{y_q}y)}. \tag{11.11}$$

It is seen that the Helmholtz equation in this domain is solved with two set of modes: one set with sinusoidal oscillations in space which satisfy $k_x^2+k_y^2+k_z^2=\alpha^2$, and one set which (as for the Laplace equation) decay exponentially away from the boundaries at $z=0$ and $z=L_z$.

The inhomogeneous forms of the basic equations discussed here are also quite important. For example, the inhomogeneous version of the Laplace equation (11.1) may be identified as the (elliptic) **Poisson** equation:

$$\Delta q=f, \tag{11.12}$$

This equation was first encountered in (3.11), where its solution via splitting methods was discussed, and was encountered again in (5.45b), where an algorithm for its solution via Fourier methods was given; it will be discussed again later in this chapter, where its solution via multigrid methods will be examined in detail.

[4] It works out that this choice is appropriate for problems set up by the Fourier transform of the BCs on ϕ at $z=0$ and $z=L_z$. Other choices are also possible, and better suited for other problems; their analysis is similar.

11.1.2 The diffusion equation, and other parabolic systems

The diffusion equation (11.3) in a 3D rectangular domain with periodic boundary conditions, with dispersion relation (11.8), may be solved via superposition of separable modes of the form (11.5) as follows:

$$\frac{\partial \phi}{\partial t} - \nu\left(\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2}\right) = 0, \quad \phi = \hat{\phi}^0 e^{st+\vec{\ell}\cdot\vec{x}} \quad \Rightarrow \quad s - \nu(\ell_x^2 + \ell_y^2 + \ell_z^2) = 0.$$

With $s = -\sigma + \mathrm{i}\omega$ and $\vec{\ell} = \vec{m} + \mathrm{i}\vec{k}$, choosing $\vec{m} = 0$ leads to $s + \nu(k_x^2 + k_y^2 + k_z^2) = 0$ and thus

$$\left.\begin{aligned} \sigma &= \nu(k_x^2 + k_y^2 + k_z^2) \\ \omega &= 0 \end{aligned}\right\} \quad \Rightarrow \quad \phi(\vec{x},t) = \sum_{p,q,r} \hat{\phi}^0_{p,q,r} e^{-\sigma_{p,q,r}t + \mathrm{i}(k_{x_p}x + k_{y_q}y + k_{z_r}z)}. \tag{11.13}$$

It is thus seen that a superposition of separable modes of the form (11.5), where s and ℓ are selected to satisfy the dispersion relation (11.8), may be used to solve the diffusion equation (11.3) in simple (e.g., rectangular) domains. The Fourier coefficients $\hat{\phi}^0_{p,q,r}$ may be found by Fourier transform of the ICs on ϕ at $t = 0$, with, again, the spatial wavenumbers included in the series selected appropriately to fit the domain of interest (that is, $k_{x_p} = 2\pi p/L_x$, $k_{y_q} = 2\pi q/L_y$, and $k_{z_r} = 2\pi r/L_z$ for integer $\{p,q,r\}$). Note that, if the domain of interest is infinite in one or more directions, the corresponding sum in (11.13) is replaced by an integral (see §5.3).

11.1.2.1 Ill-posed problems

If $\nu > 0$ and thus $\sigma \geq 0$, the solution (11.13) decays in time, with the modes with large $|\vec{k}|$ decaying fastest; thus, the diffusion equation may be thought of as a **smoother**. However, if $\nu < 0$ (or, equivalently, if one attempts to march the diffusion equation with $\nu > 0$ *backward* in time), then solutions *grow* during the time march. Note that the growth rate of these solutions *increases without bound* for increasing $|\vec{k}|$. This problem is known as the **backwards heat equation** problem, and is the prototypical **ill-posed problem**[5] due to the increasingly rapid blow up of solution components at increasing wavenumbers. To state the situation plainly, it is straightforward to determine the temperature of a system for $t > 0$ from the ICs at $t = 0$, but it is difficult to approximate the high-wavenumber components of a temperature of the system for $t < 0$ with a high degree of accuracy due to the smoothing behavior of the diffusion equation in forward time.

To avoid such difficulties, one is nominally advised simply to avoid the simulation of ill-posed problems whenever possible. However, it is sometimes desired to fly in the face of this practical advice and march a diffusive system a short amount of time in the ill-posed direction; the extent to which this is possible in any given system is sometimes referred to as **quasi-reversibility**. Numerical strategies which attempt to accomplish such a march are all approximate at best due to the inevitable errors at high wavenumbers.

If a spectral representation of an ill-posed system is used during its simulation, the simplest class of approaches, known as **spectral truncation**, just set all of the Fourier coefficients of the solution higher than a certain wavenumber to zero. This may be interpreted as adjusting the growth rate σ of each mode as follows:

$$\sigma = \begin{cases} \nu|\vec{k}|^2 & |k| \leq k_{\text{cutoff}}, \\ -R & \text{otherwise (for } R \to \infty). \end{cases} \tag{11.14}$$

Another class of approaches for ill-posed problems is to add a high-order spatial derivative term (referred to as a **hyperviscosity** term), of stabilizing sign and a "small" coefficient, to the governing PDE. For example, to approximate the march of the heat equation backward in time, one may march the augmented system

$$\frac{\partial \phi}{\partial t} - \nu\Delta\phi - \varepsilon\Delta\Delta\phi = 0 \quad \text{with} \quad \varepsilon > 0. \tag{11.15}$$

[5] A **well-posed problem**, as defined by Hadamard, is one for which a solution *exists*, is *unique*, and *depends continuously on the data*; a problem which is not well posed in this sense, such as the backwards heat equation described here, is called **ill posed**.

In 1D, a modal analysis of this augmented system may be performed as follows:

$$\frac{\partial \phi}{\partial t} - \nu \frac{\partial^2 \phi}{\partial x^2} - \varepsilon \frac{\partial^4 \phi}{\partial x^4} = 0, \quad \phi = \hat{\phi}\, e^{-\sigma t + i k_x x} \quad \Rightarrow \quad \sigma = \nu k_x^2 - \varepsilon k_x^4. \tag{11.16}$$

A **trilaplacian** term (that is, $+\varepsilon\Delta\Delta\Delta\phi$) may be used for faster hyperviscosity regularization at high spatial wavenumbers; a modal analysis in 1D in this case results in

$$\frac{\partial \phi}{\partial t} - \nu \frac{\partial^2 \phi}{\partial x^2} + \varepsilon \frac{\partial^6 \phi}{\partial x^6} = 0, \quad \phi = \hat{\phi}\, e^{-\sigma t + i k_x x} \quad \Rightarrow \quad \sigma = \nu k_x^2 - \varepsilon k_x^6. \tag{11.17}$$

For a given maximum $|\vec{k}|$ of interest and suffiently small ε, the hyperviscosity term is negligible over the wavenumbers of interest; thus, the dynamics of these augmented systems mimic accurately the dynamics of the original system (11.3). However, for a given ε and sufficiently large $|\vec{k}|$, the augmented systems are dominated by the hyperviscosity term; thus, the time-accurate march of these augmented systems backward in time is stable (that is, $\sigma < 0$ for large $|\vec{k}|$, and thus these modes decay when marched backward in time).

The hyperviscosity term introduced above is often referred to as a **regularization** term (with an associated **regularization coefficient** ε), as it makes the solution **regular** (i.e., smooth) when the system is marched backward in time. There are many other ways to regularize a diffusive PDE; another class of approaches is to add an appropriate high-order **mixed time-space derivative** term:

$$\frac{\partial \phi}{\partial t} - \nu \Delta \phi - \varepsilon \Delta \frac{\partial \phi}{\partial t} = 0 \quad \text{with} \quad \varepsilon > 0. \tag{11.18}$$

In 1D, a modal analysis of this augmented system may be performed as follows:

$$\frac{\partial \phi}{\partial t} - \nu \frac{\partial^2 \phi}{\partial x^2} - \varepsilon \frac{\partial^3 \phi}{\partial x^2\, \partial t} = 0, \quad \phi = \hat{\phi}\, e^{-\sigma t + i k_x x} \quad \Rightarrow \quad \sigma = \nu \frac{k_x^2}{1 + \varepsilon k_x^2}. \tag{11.19}$$

Again, a higher spatial derivative in the added term (e.g., $+\varepsilon\Delta\Delta\partial\phi/\partial t$) may be used for faster regularization at high spatial wavenumbers; a modal analysis in 1D in this case results in

$$\frac{\partial \phi}{\partial t} - \nu \frac{\partial^2 \phi}{\partial x^2} + \varepsilon \frac{\partial^5 \phi}{\partial x^4\, \partial t} = 0, \quad \phi = \hat{\phi}\, e^{-\sigma t + i k_x x} \quad \Rightarrow \quad \sigma = \nu \frac{k_x^2}{1 + \varepsilon k_x^4}. \tag{11.20}$$

Again, for a given maximum $|\vec{k}|$ of interest and suffiently small ε, the mixed time-space derivative term is negligible over the wavenumbers of interest, and thus the dynamics of these augmented systems mimic accurately the dynamics of (11.3). For a given ε as $|\vec{k}|$ is made large, the growth rate of the augmented system (11.19) approaches a positive constant, and the growth rate of the augmented system (11.20) approaches zero.

The growth rate σ of the regularized 1D diffusion equation following the spectral truncation, hyperviscosity, and mixed time-space derivative approaches is plotted in Figure 11.2. Recall that the hyperviscosity approach drives σ negative for large $|\vec{k}|$; for short marches in time, which is all one can hope to achieve with any semblance of accuracy in an ill-posed problem, driving σ all the way to negative values is overkill; thus, *a mixed time-space derivative approach is often preferred for numerical simulations*. Note also that numerical approximations of ill-posed problems are inevitably performed at *finite resolution* and over *finite* (usually, relatively short) *periods of time*; strict analyses of stability of such simulations in the infinite-horizon PDE limit are thus, arguably, of limited relevance.

11.1.2.2 Tikhonov regularization

A perhaps less *ad hoc* class of approaches for solving ill-posed problems, known as **Tikhonov regularization**, is to formulate, and subsequently minimize, a cost function which balances the equation one is trying to solve with a generalized measure of the "energy" of the solution, thereby seeking an accurate solution of the equations with a not unreasonable "energy".

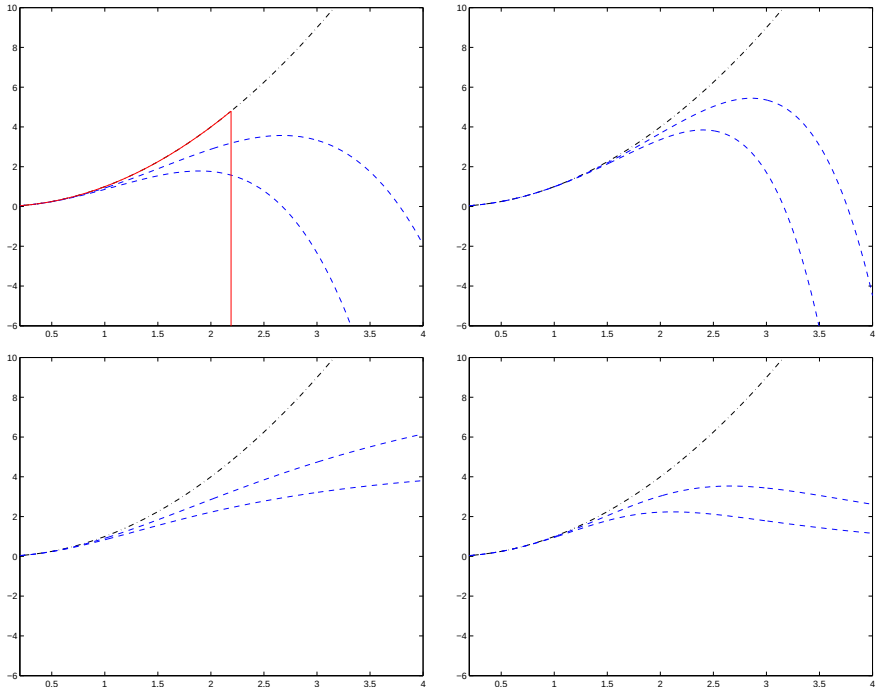


Figure 11.2: Decay rate σ of the regularized 1D diffusion equation using [top left, solid] spectral truncation and, for various values of the regularization coefficient ε, [top left, dashed] fourth-order hyperviscosity (11.16), [top right, dashed] sixth-order hyperviscosity (11.17), [bottom left, dashed] a mixed first-order-time-second-order-space derivative term (11.19), and [bottom right, dashed] a mixed first-order-time-fourth-order-space derivative term (11.20), as compared with [dot-dashed] the growth rate of the original 1D diffusion equation (11.3). Note that the effect of sixth-order hyperviscosity has a significantly sharper onset as $|k|$ is increased than does fourth-order hyperviscosity.

This approach is best introduced in an algebraic, finite-dimensional setting. Consider the problem of solving the linear ill-conditioned system $A\mathbf{x} = \mathbf{b}$ (arising, e.g., from the discretization of an ill-posed PDE). Following the Tikhonov regularization approach, with $Q > 0$ and $R > 0$, we seek to minimize the cost

$$J = \frac{1}{2}\|A\mathbf{x}-\mathbf{b}\|_Q^2 + \frac{\varepsilon}{2}\|\mathbf{x}\|_R^2 = \frac{1}{2}(A\mathbf{x}-\mathbf{b})^H Q(A\mathbf{x}-\mathbf{b}) + \frac{\varepsilon}{2}\mathbf{x}^H R\mathbf{x}. \tag{11.21}$$

Performing a perturbation analysis (that is, replacing $\mathbf{x}$ with $\mathbf{x}+\mathbf{x}'$ and J with $J+J'$ and keeping all terms which are linear in the perturbations), we may write

$$J' = [(A\mathbf{x}-\mathbf{b})^H QA + \varepsilon\mathbf{x}^H R]\mathbf{x}' = [A^H Q(A\mathbf{x}-\mathbf{b}) + \varepsilon R\mathbf{x}]^H\mathbf{x}' \triangleq [DJ/D\mathbf{x}]^H\mathbf{x}'.$$

Setting $DJ/D\mathbf{x} = 0$ results immediately in

$$(A^H QA + \varepsilon R)\mathbf{x} = A^H Q\mathbf{b} \quad \Rightarrow \quad \mathbf{x} = (A^H QA + \varepsilon R)^{-1}A^H Q\mathbf{b}. \tag{11.22a}$$

Leveraging the reduced SVD $A = \underline{U}\Sigma\underline{V}^H$ with singular values σ_i in the case with $Q = R = I$, we may write

$$\mathbf{x} = \underline{V}D\underline{U}^H\mathbf{b} \quad \text{where} \quad D = \text{diag}(\varsigma_p), \quad \varsigma_p = \frac{\sigma_p}{\sigma_p^2+\varepsilon}. \tag{11.22b}$$

For $\varepsilon = 0$, this reduces to the unregularized solution $D = \text{diag}(\varsigma_p)$ where $\varsigma_p = 1/\sigma_p$ and $\mathbf{x} = A^+\mathbf{b}$; note that ς_p increases without bound for diminishing σ_p. Setting ε as a small positive constant bounds the maximum value of ς_p (for σ_p of the same order of magnitude as ε), and actually takes $\varsigma_p \to 0$ as $\sigma_p \to 0$.

11.1.2.3 Inverse problems

Returning for a moment to the (elliptic) Poisson equation $\Delta q = f$ mentioned previously, the usual problem considered is to determine q given f. Occasionally, the **inverse problem** is of interest; that is, determining f given q. Such inverse problems are generally ill posed in a manner similar to the backwards heat equation problem discussed above; that is, small changes to q at high wavenumbers create large changes to f. Writing this inverse problem as $\Delta^{-1} f = q$, we denote a (high-resolution) discretization of the inverse Laplacian operator on the LHS as A, the discretization of the unknown f as $\mathbf{x}$, and the discretization of the data q setting up the problem as $\mathbf{b}$. Following the Tikhonov regularization approach described above, in the 1D case, the singular values σ_i of the matrix A are numerical approximations of $1/k_x^2$, and thus the D matrix given above, which can be interpreted as the amplification of the data q in the regularized solution f, is given by

$$D = \operatorname{diag}\Big(\frac{\sigma_p}{\sigma_p^2+\varepsilon}\Big) \approx \operatorname{diag}\Big(\frac{1/k_{x_p}^2}{1/k_{x_p}^4+\varepsilon}\Big) = \operatorname{diag}\Big(\frac{k_{x_p}^2}{1+\varepsilon k_{x_p}^4}\Big).$$

Note that the effect of Tikhonov regularization on such an inverse problem is reminiscent of the effect of mixed time-space regularization on the backwards heat equation [see (11.20)]; that is, the amplification of the low wavenumbers in A^{-1} is represented faithfully in $\underline{V}D\underline{U}^H$ (as $\approx k_{x_i}^2$), whereas the amplification of the high wavenumber components of the solution is muted.

11.1.2.4 Convenient scaling of space, time, and the unknown(s) in a PDE

To facilitate understanding, space, time, the parameters, and the unknowns in a PDE are often normalized such that transformed PDE appears as simple as possible. As an example, consider the equation

$$\frac{\partial u'}{\partial t'} = -c_1 u' \frac{\partial u'}{\partial x'} + c_2 \frac{\partial^2 (u')^3}{\partial x'^2} - c_3 \frac{\partial^2 u'}{\partial x'^2} - c_4 \frac{\partial^4 u'}{\partial x'^4} \quad \text{on the domain } x' \in [0, L'], \tag{11.23}$$

with $c_1 \neq 0$, $c_2 > 0$, $c_3 > 0$, $c_4 > 0$. We may rescale the primed variables such that

$$x = x' \cdot c_3^{1/2}/c_4^{1/2}, \quad u = u' \cdot c_2^{1/2}/c_3^{1/2}, \quad t = t' \cdot c_3^2/c_4, \quad L = L' \cdot c_3^{1/2}/c_4^{1/2}, \quad D = c_1 c_4^{1/2}/(c_3 c_2^{1/2}),$$

thereby reducing (11.23), WLOG, to the **1D convective Cahn-Hilliard** (**CCH**) equation

$$\frac{\partial u}{\partial t} = -D u \frac{\partial u}{\partial x} + \frac{\partial^2 (u^3)}{\partial x^2} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4} \quad \text{on the domain } x \in [0, L], \tag{11.24}$$

where D is constant. Thus, the form of the 1D CCH equation given in (11.24) is used henceforth in this work WLOG[6]. Most PDEs presented below have been rescaled similarly, mutatis mutandis.

In the 1D CCH equation above, the dynamics of the result depend on two parameters, D and L. The $D \to 0$ limit of the 1D CCH equation reduces it to the **1D Cahn-Hilliard** (**CH**) equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 (u^3)}{\partial x^2} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^4 u}{\partial x^4} \quad \text{on the domain } x \in [0, L], \tag{11.25}$$

whereas substituting $u = v/D$ into the 1D CCH equation, multiplying by D, then taking the $D \to \infty$ limit reduces it to the **1D Kuramoto-Sivashinsky** (**KS**) equation

$$\frac{\partial v}{\partial t} = -v \frac{\partial v}{\partial x} - \frac{\partial^2 v}{\partial x^2} - \frac{\partial^4 v}{\partial x^4} \quad \text{on the domain } x \in [0, L]. \tag{11.26}$$

The parameter D thus effectively balances the effect of the two nonlinear terms in the 1D CCH, with D large emphasizing the effect of the convective term $u\,\partial u/\partial x$, and D small emphasizing the effect of the cubic term

[6] That is, other than the mild restrictions on the c_i mentioned previously, which are typical in the cases of interest.

$\partial^2(u^3)/\partial x^3$. A related equation, which may be obtained from (11.23) with $c_1 \neq 0$, $c_3 < 0$, and $c_2 = c_4 = 0$, is given by the **1D Burgers'** equation

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2} \quad \text{on the domain } x \in [0, L]. \tag{11.27}$$

Denoting by $\langle f \rangle$ the average of f over the spatial domain considered, one form[7] of a related 2D model with interesting dynamics, referred to here as the **2D convective Cahn-Hilliard** model, is given by

$$\frac{\partial u}{\partial t} = D\Big[|\nabla u|^2 - \langle |\nabla u|^2 \rangle\Big] + \Delta(u^3) - \Delta u - \Delta\Delta u \quad \text{on the domain } x \in [0, L_x],\ y \in [0, L_y],\ . \tag{11.28}$$

11.1.2.5 Spectra, chaos, and the cascade of energy in chaotic nonlinear PDE systems

We now focus specifically on the dynamics of the 1D Burgers' and KS equations, (11.27) and (11.26). The dynamics of these equations may be understood by calculating their Fourier transforms: that is, applying $\frac{1}{N}\sum_{p=0}^{N-1}\big[\,\cdot\,\big]e^{-ik_{x_p}x}$ to these equations [see (5.23b)] leads to

$$\frac{d\hat{u}_p}{dt} = -\widehat{\Big[u\frac{\partial u}{\partial x}\Big]}_p - k_{x_p}^2\hat{u}_p \qquad \text{and} \qquad \frac{d\hat{u}_p}{dt} = -\widehat{\Big[u\frac{\partial u}{\partial x}\Big]}_p + (k_{x_p}^2 - k_{x_p}^4)\hat{u}_p \tag{11.29}$$

at each wavenumber k_{x_p} retained in the discretization (i.e., for $-N/2 < p < N/2$). In the case of Burgers, the second-derivative term stabilizes the calculation[8]. In the case of KS, the fourth derivative term stabilizes the calculation, whereas the second derivative term destabilizes the calculation; as shown in Figure 11.3a, the stabilizing fourth derivative term dominates for large wavenumbers, whereas the destabilizing second derivative term dominates for small wavenumbers. The nonlinear **convective** term in both the 1D Burgers' and KS equations, $-u\,\partial u/\partial x$, does not change the energy, $E \triangleq \int_0^L u^2/2\,dx$, in the Burgers and KS systems. This is seen, e.g., in the case of Burgers' equation (with either periodic or homogeneous Dirichlet BCs) by multiplying by u, integrating over Ω, and integrating by parts, which leads to:

$$\int_0^L u\Big[\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2}\Big]dx \quad \Rightarrow \quad \frac{d}{dt}\Big[\int_0^L \frac{u^2}{2}dx\Big] = -\int_0^L \Big[\frac{\partial u}{\partial x}\Big]^2 dx \le 0; \tag{11.30a}$$

note in particular that the contribution from the convective term vanishes since, via integration by parts and applying the periodic BCs on u, it follows that

$$\int_0^L u^2\frac{\partial u}{\partial x}dx = 0 - \int_0^L \frac{\partial u^2}{\partial x}u\,dx = -2\int_0^L u^2\frac{\partial u}{\partial x}dx \quad \Rightarrow \quad 3\int_0^L u^2\frac{\partial u}{\partial x}dx = 0. \tag{11.30b}$$

Thus, the convective term $-u\,\partial u/\partial x$ is said to be **energy conserving**; that is, such nonlinear terms simply **scatter energy** across the wavenumber spectrum (see Fact 5.5). The effect of the cubic term $\partial^2(u^3)/\partial x^2$ on the energy of the 1D CCH and CH systems is considered in Exercise 11.2.

To summarize, the time evolution of the Burgers' system is characterized by the *scattering* of energy across all wavenumbers due to the energy-conserving nonlinear term, and the *dissipation* of energy at the high wavenumbers due to the stable linear operator. Without supplemental excitation (from the boundary conditions or additional RHS forcing of the PDE), the Burgers' system thus eventually decays to zero.

[7] The 2D CCH model given here is one of three reviewed by Golovin & Pismen (2004); a fourth is studied in Golovin *et al.* (2001). Note that, despite the similarity of the names typically assigned to them, the "2D CCH" model (11.28) does not reduce to the "1D CCH" model (11.24) when symmetry in one spatial dimension is applied; that is, these two models represent slightly different physical phenomena, they are just attributed to early work by the same set of authors, and thus commonly go by similar names.

[8] That is, with the second derivative term only on the RHS, the Fourier-space representation of this equation is of the model form $dy/dt = \lambda y$ with $\lambda = -k_{x_p}^2 < 0$ at each wavenumber $k_{x_p} \neq 0$.

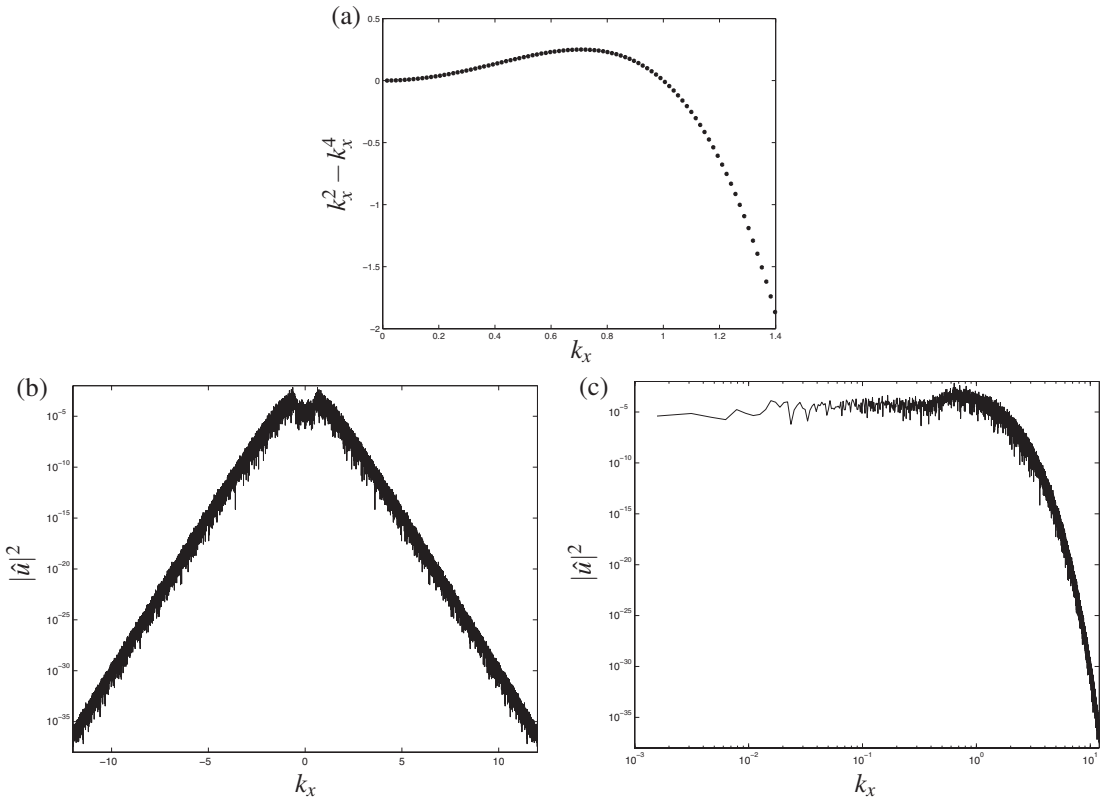


Figure 11.3: (a) Plot of the linear operator of the 1D CCH, CH, and KS equations (11.24), (11.25), and (11.26) in Fourier space at wavenumbers $k_{x_p} = 2\pi p/L_x$ for $p = 1, 2, \ldots$ and $L_x = 400$, illustrating amplification at small wavenumbers and attenuation at large wavenumbers; (b) log-linear, and (c) log-log plots of the energy spectrum of a KS system at statistical steady state with periodic BCs, with $L = 4000$ and $N = 32768$ grid points used in the numerical simulation, which implemented a CN/RKW3 IMEX temporal discretization and a pseudospectral spatial discretization (see §11.2.2 and Algorithm 11.4).

The evolution of the KS system, on the other hand, is characterized by the *excitation* of the system at the low wavenumbers due to the unstable part of the linear operator, a *scattering* of energy across all wavenumbers due to the energy-conserving nonlinear term, and the *dissipation* of energy at the high wavenumbers due to the stable part of the linear operator[9] (see Figure 11.3a). For sufficiently large L_x (to include a sufficient number of unstable wavenumbers k_m in the representation) and sufficiently large ICs, the KS system neither blows up nor settles to an equilibrium state; instead, a statistical balance is reached amongst the three RHS terms, and the system approaches a **high-dimensional chaotic attractor** [cf. the low-dimensional attractors in Figures 10.8 and 10.9] in a high-dimensional phase space representing the spatial discretization of this infinite-dimensional system. When the system is on this chaotic attractor (that is, at **statistical steady state**), the energy spectrum peaks in the low wavenumbers and diminishes at higher wavenumbers (see

[9] A catchy poem by Lewis Richardson used to describe the cascade of energy from large scales to small scales in fluid turbulence is:

Big whirls have little whirls that feed on their velocity, and little whirls have lesser whirls and so on to viscosity.

A pithy retort by Carl Gibson representing an alternative viewpoint in the same class of systems, emphasizing the simultaneous transfer of energy from small scales back to large scales due to nonlinear interactions, is:

Little whorls on vortex sheets, form and pair with more of whorls that grow by vortex forces. Slava Kolmogorov!

The former emphasizes the fact that **chickens lay eggs**, whereas the latter emphasizes the fact that **eggs become chickens**. Both true.

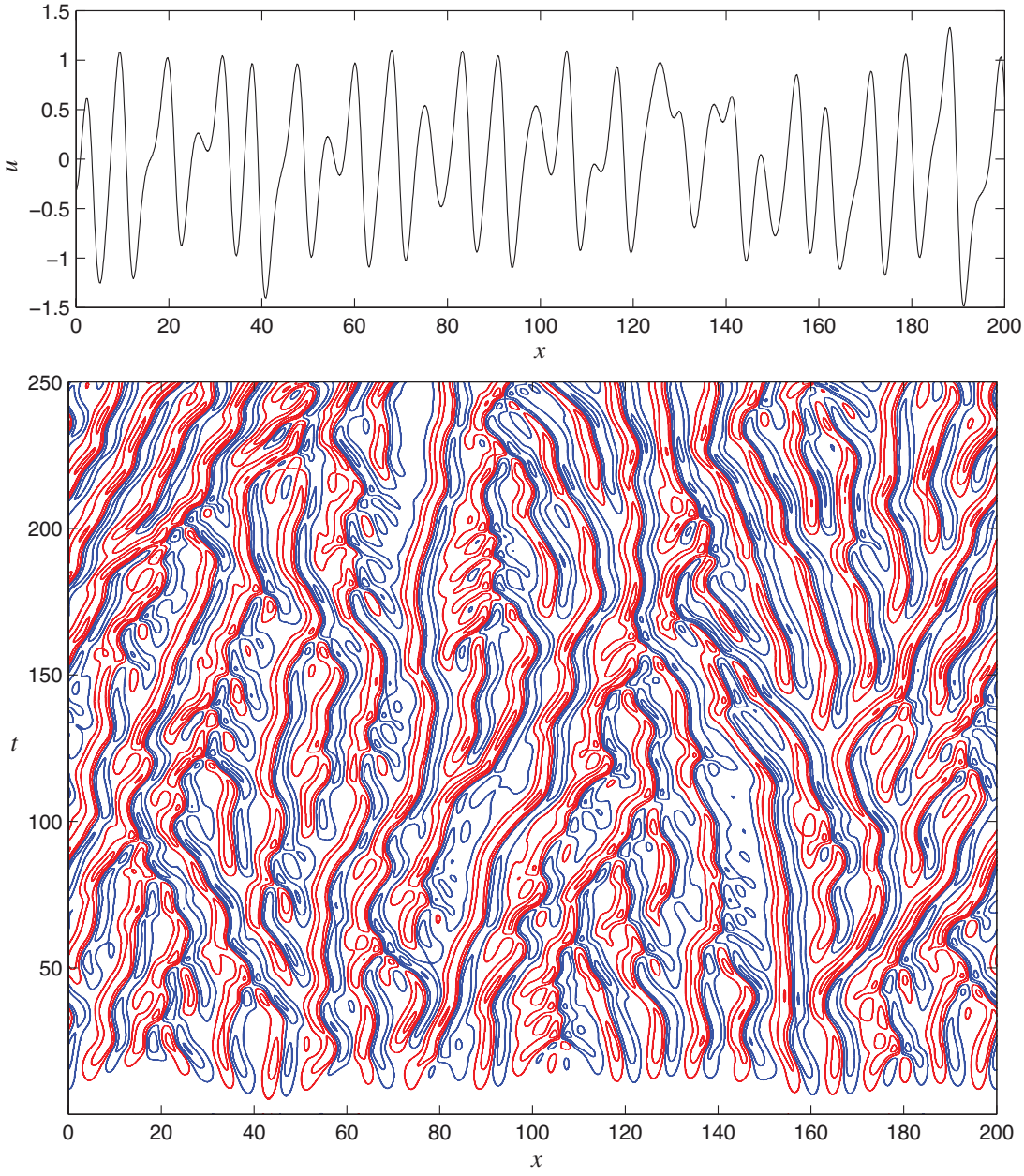


Figure 11.4: (top) A snapshot of the state $u(x,t)$ on the chaotic attractor of the KS system, and (bottom) the evolution in space/time of $u(x,t)$ with red indicating positive contours and blue indicating negative contours, illustrating **transition** from a small perturbation of u at $t = 0$ to statistical steady state for large t.

Figure 11.3b-c). The variable L_x acts like the Reynolds number in Navier-Stokes systems; for small L_x, the stabilizing (fourth-derivative) linear terms dominate at all wavenumbers in the system, which is thus stable. For increasing L_x, the destabilizing (second-derivative) linear terms dominate at an increasing number of the smaller wavenumbers. The dimension of the chaotic attractor of the KS system is too high to be visualized directly; additional visualizations are given in Figure 11.4.

11.1.3 The wave equation, and other hyperbolic systems

The wave equation (11.4) in 3D, with dispersion relation (11.9), may be solved via superposition of separable modes of the form (11.5) as follows:

$$\frac{\partial^2\phi}{\partial t^2} - c^2\Big(\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2}\Big) = 0, \quad \phi = \hat{\phi}\, e^{st+\vec{\ell}\cdot\vec{x}} \quad \Rightarrow \quad s^2 - c^2(\ell_x^2+\ell_y^2+\ell_z^2) = 0. \tag{11.31a}$$

With $s = -\sigma + \mathrm{i}\omega$ and $\vec{\ell} = \vec{m} + \mathrm{i}\vec{k}$, choosing $\vec{m} = 0$ leads to $s^2 + c^2(k_x^2+k_y^2+k_z^2) = 0$ and thus

$$\left.\begin{array}{l} \sigma = 0 \\ \omega^{\pm} = \pm c\sqrt{k_x^2+k_y^2+k_z^2} \end{array}\right\} \quad \Rightarrow \quad \phi(\vec{x},t) = \sum_{p,q,r} (\hat{\phi}^+_{p,q,r} e^{\mathrm{i}\omega^+_{p,q,r}t} + \hat{\phi}^-_{p,q,r} e^{\mathrm{i}\omega^-_{p,q,r}t}) e^{\mathrm{i}(k_{x_p}x + k_{y_q}y + k_{z_r}z)}. \tag{11.31b}$$

It is thus seen that a superposition of separable modes of the form (11.5), where s and ℓ are selected to satisfy the dispersion relation (11.9), may be used to solve the wave equation (11.4) in simple domains. The Fourier coefficients $\hat{\phi}^+_{p,q,r}$ and $\hat{\phi}^-_{p,q,r}$ may, for example, be found by Fourier transform of the ICs on ϕ and $\partial\phi/\partial t$ at $t = 0$, with the spatial wavenumbers included in the series selected appropriately to fit the domain of interest.

As anticipated by the name of the PDE which generated it, the solution (11.31b) propogates via **traveling waves** which oscillate in time. Indeed, in sharp contrast with the diffusion equation, the wave equation is as easy to propogate backward in time as it is to propogate forward in time. In the 1D case, (11.31a) reduces to

$$\frac{\partial^2\phi}{\partial t^2} - c^2\frac{\partial^2\phi}{\partial x^2} = \Big(\frac{\partial}{\partial t} - c\frac{\partial}{\partial x}\Big)\Big(\frac{\partial}{\partial t} + c\frac{\partial}{\partial x}\Big)\phi = 0, \tag{11.32a}$$

and its solution (11.31b) reduces to

$$\phi(x,t) = \sum_p (\hat{\phi}^+_p e^{\mathrm{i}k_{x_p}(x+\overline{c}t)} + \hat{\phi}^-_p e^{\mathrm{i}k_{x_p}(x-\overline{c}t)}) \triangleq \phi^+(x,t) + \phi^-(x,t) \quad \text{where} \quad \overline{c} \triangleq |\omega_p/k_{x_p}| = c. \tag{11.32b}$$

Noting that the **phase velocity** $\overline{c}$ is independent of p in this problem (that is, noting that there is no **dispersion**), the solution $\phi(x,t)$ is seen to be the superposition of two sets of modes, one $[\phi^+(x,t)]$ which **convects**, or transports, to the left at a uniform speed $\overline{c}$ and satisfies $(\partial/\partial t - \overline{c}\,\partial/\partial x)\phi^+ = 0$, and the other $[\phi^-(x,t)]$ which convects to the right at a uniform speed $\overline{c}$ and satisfies $(\partial/\partial t + \overline{c}\,\partial/\partial x)\phi^- = 0$. Thus, initial conditions formed from just one of these sets of modes will simply *convect at uniform speed without distortion*.

11.1.3.1 Spherically-symmetric and azimuthally-symmetric solutions of the 3D wave equation

Noting (B.34d) and (11.32b), the (nondispersive, undamped) **3D wave equation with spherical symmetry enforced** may be transformed and solved as follows:

$$\frac{\partial^2\phi}{\partial t^2} - c^2\Big(\frac{\partial^2\phi}{\partial r^2} + \frac{2}{r}\frac{\partial\phi}{\partial r}\Big) = 0 \quad \Leftrightarrow \quad \frac{\partial^2(r\phi)}{\partial t^2} - c^2\frac{\partial^2(r\phi)}{\partial r^2} = 0 \tag{11.33a}$$

$$\Rightarrow \quad \phi(r,t) = \frac{1}{r}\sum_p (\hat{\varphi}^+_p e^{\mathrm{i}k_{r_p}(r+ct)} + \hat{\varphi}^-_p e^{\mathrm{i}k_{r_p}(r-c_p t)}). \tag{11.33b}$$

That is, in spherical coördinates, the quantity $(r\phi)$ satisfies the 1D wave equation, and thus spherically-symmetric waves propogate in r without distortion (other than scaling by r).

In contrast, noting (B.33d), the **2D wave equation with azimuthal symmetry enforced** is given by

$$\frac{\partial^2 h}{\partial t^2} - c^2\Big(\frac{\partial^2 h}{\partial r^2} + \frac{1}{r}\frac{\partial h}{\partial r}\Big) = 0 \tag{11.34a}$$

subject to

$$\text{BCs:}\quad \begin{cases} h \text{ finite} & \text{at } r=0 \\ h=0 & \text{at } r=r_{\max}, \end{cases} \qquad \text{ICs:}\quad h=a(r) \text{ and } \frac{\partial h}{\partial t}=b(r) \quad \text{at } t=0. \tag{11.34b}$$

Note that, unlike (11.33a), (11.34a) does *not* admit a simple change of variables to reduce it to a 1D wave equation. To solve this problem analytically, we must start from scratch with a new SOV analysis. Following the analysis of §4.3.2, we again seek separable modes that satisfy the SOV ansatz

$$h^m(r,t)=R(r)\,T(t). \tag{11.35}$$

Inserting (11.35) into (11.34a), we find that

$$RT''=c^2(R''T+R'T/r) \quad \Rightarrow \quad \frac{T''}{T}=c^2\left(\frac{R''+R'/r}{R}\right)\triangleq -\omega^2 \quad \Rightarrow \quad \begin{aligned} T''&=-\omega^2 T, \\ R''+\frac{R'}{r}&+\frac{\omega^2}{c^2}R=0. \end{aligned}$$

As in §4.3.2, the ODE for $T(t)$ is solved by $T(t)=A\sin(\omega t)+B\cos(\omega t)$. Performing the change of variables $\rho=|\omega|r/c$, the ODE for R may be written and solved as follows:

$$\frac{\partial^2 R}{\partial \rho^2}+\frac{1}{\rho}\frac{\partial R}{\partial \rho}+R=0 \quad \Rightarrow \quad R=CJ_0(\rho)+DY_0(\rho)=CJ_0(\omega r/c)+DY_0(\omega r/c), \tag{11.36a}$$

where $J_0(\rho)$ and $Y_0(\rho)$ denote **Bessel functions** (of zero'th order) of the first and second kind, respectively. Due to the BC at $r=0$, it follows immediately that $D=0$. Due to the BC at $r=r_{\max}$, it follows for most ω that $C=0$ as well, and thus $h^m(r,t)=0$ for all $\{r,t\}$. However, for certain values of ω [specifically, for $|\omega|r_{\max}/c=\lambda$, where $\lambda>0$ is one of the zeros of the associated Bessel function (that is, $J_0(\lambda)=0$)], R satisfies the homogeneous boundary condition at $r=r_{\max}$ even for nonzero values of C. These special values of ω are the eigenvalues[10] of the PDE system $R''+R'/r+\omega^2R/c^2=0$ with finite R and homogeneous BCs $R=0$ at $r=r_{\max}$. The solution of (11.34) may thus be written as a superposition of separable modes,

$$\begin{aligned} h(r,t)&=\sum_p (\hat{h}_p^+ e^{\mathrm{i}\,\omega_p^+ t}+\hat{h}_p^- e^{\mathrm{i}\,\omega_p^- t})\,J_0\left(\frac{\lambda_p r}{r_{\max}}\right) \\ \text{where}\quad \omega_p^{\pm}&=\pm\frac{\lambda_p c}{r} \quad \text{and} \quad J_0(\lambda_p)=0 \quad \text{for} \quad p=1,2,\ldots, \end{aligned} \tag{11.36b}$$

where the coefficients $\hat{h}_p^+$ and $\hat{h}_p^-$ may be found via Fourier-Bessel transform (see §5.14) of the initial conditions on h and $\partial h/\partial t$ at $t=0$. In contrast with the 3D case, the solution in the 2D case can *not* be written as a linear combination of terms with $(x-ct)$ and $(x+ct)$ dependences, as the spatial dependence is Bessel.

An alternative explanation of the solution of the 2D wave equation with azimuthal symmetry enforced is provided by considering the superposition of the 3D waves of spherical symmetry emanating from a series of identical impulsive sources on a line in a 3D field, each separated by a distance Δz and of strength Δz, in the limit that $\Delta z\to 0$. Centering a cylindrical coördinate system along this line, the first (spherically symmetric) wave to reach an observer situated somewhere in the $z=0$ plane is that emanating from the source at $r=0$ and $z=0$. At later times, the signal reaching this same observer is augmented by waves, all traveling at speed c, from other sources at locations with $r=0$ and increasing values of $|z|$, which are farther away. This gives the appearance of something like a dispersive behavior (see §11.1.3.2 below) when just looking in the plane $z=0$. However, this system is *not* labelled as dispersive; the modal solution restricted to the $z=0$ plane is not even exponential. Of course, this problem is clearly hyperbolic, as the "speed of propagation of information" (that is, the maximum wave speed) is exactly c, which is finite.

[10] Recall from footnote 1 on page 75 that this is called a **Sturm-Liouville eigenvalue problem**.

11.1.3.2 Dispersion, and the difference between phase velocity and group velocity

The nondispersive wave behavior described in the introduction of §11.1.3, in which traveling waves maintain their shape as they convect, is in contrast with many other systems of physical interest, in which sinusoidal modes of different wavelengths convect at different speeds, and thus waves formed as a superposition of such modes *distort in shape significantly as they convect*. We now consider two such **dispersive** systems.

The traveling-wave behavior of the deflection ϕ of a stiff string, accounting for its finite thickness, may be modeled with the dispersive 1D wave equation

$$\frac{\partial^2\phi}{\partial t^2} - c^2\frac{\partial^2\phi}{\partial x^2} + \kappa\frac{\partial^4\phi}{\partial x^4} = 0 \quad \text{where} \quad \kappa > 0. \tag{11.37a}$$

Taking $\phi = \hat{\phi}\, e^{st+\ell_x x}$ with $\ell_x = \mathrm{i}k_x$ now leads to the dispersion relation $s^2 + c^2k_x^2 + \kappa k_x^4 = 0$, and thus

$$\left.\begin{aligned} &\sigma = 0 \\ &\omega^{\pm} = \pm\sqrt{c^2k_x^2 + \kappa k_x^4} \end{aligned}\right\} \quad \Rightarrow \quad \phi(x,t) = \sum_p (\hat{\phi}_p^+ e^{\mathrm{i}\omega_p^+ t} + \hat{\phi}_p^- e^{\mathrm{i}\omega_p^- t}) e^{\mathrm{i}k_{x_p}x}$$
$$\Leftrightarrow \quad \phi(x,t) = \sum_p (\hat{\phi}_p^+ e^{\mathrm{i}k_{x_p}(x+\overline{c}_p t)} + \hat{\phi}_p^- e^{\mathrm{i}k_{x_p}(x-\overline{c}_p t)}) \quad \text{where} \quad \overline{c}_p \triangleq |\omega_p/k_{x_p}| = \sqrt{c^2 + \kappa k_{x_p}^2}. \tag{11.37b}$$

For small $(\kappa k_{x_p}^2)$, solutions of the dispersive 1D wave equation (11.37a) are similar to those in the nondispersive case (with $\kappa = 0$), with wave speed $\overline{c}_p \approx c$. However, for increasing p with $\kappa > 0$, the wave speed $\overline{c}_p$ increases. The solution $\phi(x,t)$ is no longer simply the superposition of two sets of modes convecting to the left and right at uniform speeds. Instead, oscillations at *shorter wavelengths* $\lambda_{x_p} = 2\pi/k_{x_p}$ convect at *higher phase velocities* $\overline{c}_p$; this characteristic is referred to as "**anomalous dispersion**".

The 2D traveling-wave behavior of the height of the free surface f of a body of water of constant depth b is, in general, somewhat more complicated to derive. *Considering a single sinusoidal wave at a time*, and aligning the x direction with the direction of propagation of this wave, [that is, taking $\phi(x,t) = Ce^{\mathrm{i}k_x(x+\overline{c}t)}$ for a wave of wavelength λ_x], its evolution may be modeled with a dispersive 1D wave equation [cf. (11.37a)]

$$\frac{\partial^2 f}{\partial t^2} - \overline{c}^2\frac{\partial^2 f}{\partial x^2} = 0 \quad \text{where} \quad \overline{c} = \sqrt{\frac{g}{k_x}\left[1 + \frac{Tk_x^2}{g\rho}\right]\tanh(k_x b)}, \tag{11.38}$$

where $k_x = 2\pi/\lambda_x$, T is the surface tension per unit length, ρ is the water density, and $T/\rho = 7.4\times10^{-5}$ m^3/s^2 for an air/water interface. Defining $\lambda_m = 2\pi\sqrt{T/(g\rho)} = 0.0173$ m, if $b \gg \lambda_m$, then

$$\overline{c} \approx \begin{cases} \sqrt{Tk_x/\rho} & \text{for } \lambda_x \ll \lambda_m \quad \text{"\textbf{capillary waves}"} \\ \sqrt{(g/k_x)[1 + Tk_x^2/(g\rho)]} & \text{for } \lambda_x \approx O(\lambda_m) \\ \sqrt{g/k_x} & \text{for } \lambda_m \ll \lambda_x \ll 2\pi b \quad \text{"\textbf{deep-water waves}"} \\ \sqrt{(g/k_x)\tanh(k_x b)} = \sqrt{gb - gk_x^2b^3/3 + \ldots} & \text{for } \lambda_x \approx O(2\pi b) \quad \text{[see (B.75)]} \\ \sqrt{gb} & \text{for } 2\pi b \ll \lambda_x \quad \text{"\textbf{shallow-water waves}"; see §11.1.3.4.} \end{cases}$$

Thus, in the deep water case, oscillations at *longer wavelengths* $\lambda_{x_p} = 2\pi/k_{x_p}$ convect at *higher phase velocities* $\overline{c}_p$; this characteristic is referred to as "**normal dispersion**"; note that shallow-water waves are nondispersive, whereas capillary waves exhibit the anomalous dispersion characteristic described previously.

Defining $\overline{c}_{\max} = \max_p |\overline{c}_p|$ as the maximum wave speed in an unsteady PDE system, we may now identify the **domain of dependence**, as well as the **range of influence**, of the state of the system ϕ at any given point (x,t) in space and time; as illustrated in Figure 11.5, both regions are conical in hyperbolic systems. Changes to the system outside the domain of dependence do not affect the value of $\phi(x,t)$ at the point at center of this

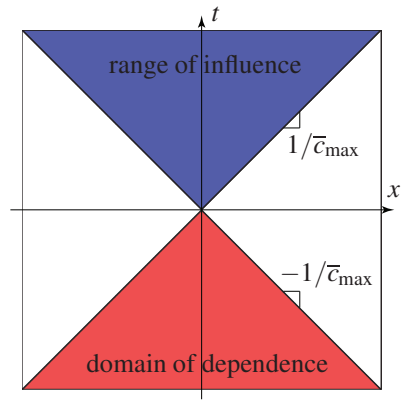


Figure 11.5: **Domain of dependence** and **range of influence** of the state ϕ of a hyperbolic system at an arbitrary point (x,t) in space and time, take here WLOG as the origin.

plot, and values of $\phi(x,t)$ outside the range of influence are not affected by changes to the system at the point at the center of this plot. In a hyperbolic system such as (11.32a), the (finite) maximum wave speed $\overline{c}_{\max}$ defines the slope of these cones in space-time. In contrast, in a parabolic system such as (11.37a), the wave speed $\overline{c}_p$ is unbounded as p increases, and thus in such systems the (red) domain of dependence in Figure 11.5 is the entire lower half plane and the (blue) range of influence is the entire upper half plane.

11.1.3.3 Damping

It is instructive at this point to visit another variation of (11.32a), namely a **damped** 1D wave equation

$$\frac{\partial^2\phi}{\partial t^2} - c^2\frac{\partial^2\phi}{\partial x^2} + b_1\frac{\partial\phi}{\partial t} = 0 \quad \text{where} \quad b_0 > 0, \tag{11.39a}$$

which accounts for the "damping" of the string due, e.g., to the wind resistance and sound generation created by its motion[11]. Analyzing as before leads to the dispersion relation $s^2 + c^2k_x^2 + b_1 s = 0$, and thus

$$s^{\pm} = -b_1/2 \pm c\sqrt{b_1^2/(4c^2) - k_x^2} \quad \Rightarrow \quad \phi(x,t) = \sum_p (\hat{\phi}_p^+ e^{s_p^+ t} + \hat{\phi}_p^- e^{s_p^- t}) e^{\mathrm{i}k_{x_p}x} = \sum_p \phi_p(x,t). \tag{11.39b}$$

For $k_{x_p} > b_1/(2c)$, we have $s_p^{\pm} = -\sigma + \mathrm{i}\omega_p^{\pm}$ with $\sigma = b_1/2 > 0$ and $\omega_p^{\pm} = \pm c\sqrt{k_{x_p}^2 - b_1^2/(4c^2)}$, and thus

$$\phi_p(x,t) = e^{-\sigma t}\left(\hat{\phi}_p^+ e^{\mathrm{i}k_{x_p}(x+\overline{c}_p t)} + \hat{\phi}_p^- e^{\mathrm{i}k_{x_p}(x-\overline{c}_p t)}\right) \quad \text{where} \quad \overline{c}_p = |\omega_p/k_{x_p}| = c\sqrt{1 - b_1^2/(4c^2k_{x_p}^2)}.$$

For $k_{x_p} \le b_1/(2c)$, we have $s_p^{\pm} = -\sigma_p^{\pm} + \mathrm{i}\omega$ with $\sigma_p^{\pm} = b_1/2 \pm c\sqrt{b_1^2/(4c^2) - k_{x_p}^2} \ge 0$ and $\omega = 0$, and thus

$$\phi_p(x,t) = \left(\hat{\phi}_p^+ e^{-\sigma_p^+ t} + \hat{\phi}_p^- e^{-\sigma_p^- t}\right) e^{\mathrm{i}k_{x_p}x}.$$

[11] Damping is what causes the volume of the sound generated by an oscillating piano wire to decay with time. An improved model for the motion of a piano wire of length L, which accounts for frequency-dependent **Kelvin-Voigt damping** due to molecular heating (b_2) in addition to both the **viscous damping** due to wind resistance and heat generation (b_1) and the **dispersion due to stiffness** (κ), is

$$\frac{\partial^2\phi}{\partial t^2} - c^2\frac{\partial^2\phi}{\partial x^2} + \kappa\frac{\partial^4\phi}{\partial x^4} + b_1\frac{\partial\phi}{\partial t} - b_2\frac{\partial^3\phi}{\partial t\,\partial^2 x} = 0;$$

for middle C, with a fundamental frequency of 261.6 Hz, we may take $\{c,\kappa,b_1,b_2,L\} = \{329.6, 1.5625, 2.2, 0.00054, 0.63\}$ (see Bensa *et al.* 2003), as considered further in Exercise 11.6.

For small b_1 (or large k_{x_p}), the modes are damped (that is, exponentially-decaying) sinusoids; note that there is both damping *and* (normal) dispersion, as the wave speed $\overline{c}_p$ reduces with increasing p. For large b_1 (or small k_{x_p}), the oscillations in time are removed completely, and the modes (referred to as **evanescent** modes) simply decay exponentially in time. The maximum wave speed $\overline{c}_{\max}$ is finite, so this (hyperbolic) system has a conical domain of dependence and range of influence similar to that depicted in Figure 11.5.

Finally, note that, in 1D, the maximum wave speed of the (hyperbolic) wave equation (11.32a) and the (hyperbolic) damped wave equation (11.39a) is c, whereas the wave speed of the (parabolic; see §11.1.4) dispersive wave equation (11.37a) is unbounded. In the limit that $c \to \infty$ with b_1/c^2 constant, the (hyperbolic) damped wave equation (11.39a) reduces to the (parabolic) diffusion equation (11.3), whereas in the limit that $\kappa \to 0$, the (parabolic) dispersive wave equation (11.37a) reduces to the (hyperbolic) wave equation (11.32a).

11.1.3.4 The nonlinear 2D shallow water equation

We now turn our attention to systems governed by the 2D **shallow water equation** (**SWE**). Denoting

- $u(x,y,t)$ & $v(x,y,t)$ as the x & y components of the velocity averaged vertically over the water column,
- $b(x,y,t)$ as the (specified) reference depth of the bottom in the body of water under consideration,
- $f(x,y,t)$ as the height of the free surface above or below its reference height of $f = 0$,
- $h(x,y,t) = f(x,y,t) + b(x,y)$ as the total height of the water column at any given $\{x,y,t\}$,
- $\omega(y) = 2\,\Omega \sin y$ (where y is the latitude and $\Omega = 7.2921 \times 10^{-5}$rad/s on Earth) as the **Coriolis frequency**, defined as twice the vertical component of the planet's angular velocity about the local vertical,
- μ as the viscous drag coefficient (which my be determined empirically),

and $g = 9.8\text{m/s}^2$, the SWE may be written over any 2D domain Ω in its **integral conservation form** as

$$\frac{d}{dt}\int_\Omega \mathbf{q}\, dA + \int_{\partial\Omega} F(\mathbf{q})\,\mathbf{n}\, ds = \int_\Omega \mathbf{r}\, dA, \tag{11.40a}$$

$$\mathbf{q} = \begin{pmatrix} h \\ hu \\ hv \end{pmatrix}, \quad F(\mathbf{q}) = \begin{pmatrix} hu & hv \\ hu^2 + gh^2/2 & huv \\ huv & hv^2 + gh^2/2 \end{pmatrix}, \quad \mathbf{r} = \begin{pmatrix} 0 \\ gh\,\partial b/\partial x + \omega hv - \mu hu \\ gh\,\partial b/\partial y - \omega hu - \mu hv \end{pmatrix}. \tag{11.40b}$$

Writing a system in its most general integral conservation form is often invaluable, as this form can handle **discontinuities** (a.k.a. **jumps** or **shocks**) that may be present in the solution, whereas PDE forms that may be derived (see below) from the integral conservation form can not be applied across such discontinuities.

Assuming $\mathbf{q}$ is continuous & differentiable in space & time, partitioning $F = \begin{bmatrix}\mathbf{f}_x & \mathbf{f}_y\end{bmatrix}$, applying Gauss's theorem (B.29) to the second term on the LHS of (11.40a), and noting that Ω is arbitrary, it follows that

$$\int\!\!\int_\Omega \frac{\partial \mathbf{q}}{\partial t}\, dx\, dy + \int\!\!\int_\Omega \left(\frac{\partial \mathbf{f}_x(\mathbf{q})}{\partial x} + \frac{\partial \mathbf{f}_y(\mathbf{q})}{\partial y}\right) ds = \int\!\!\int_\Omega \mathbf{r}\, dx\, dy \quad \Rightarrow \quad \frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{f}_x(\mathbf{q})}{\partial x} + \frac{\partial \mathbf{f}_y(\mathbf{q})}{\partial y} = \mathbf{r},$$

which may be written out in terms of the **conservative variables** $\{h, (hu), (hv)\}$ (note that $u = hu/h$ and $v = hv/h$) as the **conservative PDE form** of the SWE:

$$\frac{\partial h}{\partial t} = -\frac{\partial hu}{\partial x} - \frac{\partial hv}{\partial y}, \tag{11.41a}$$

$$\frac{\partial hu}{\partial t} = -\frac{\partial (uhu + gh^2/2)}{\partial x} - \frac{\partial vhu}{\partial y} + gh\frac{\partial b}{\partial x} + \omega hv - \mu hu, \tag{11.41b}$$

$$\frac{\partial hv}{\partial t} = -\frac{\partial uhv}{\partial x} - \frac{\partial (vhv + gh^2/2)}{\partial y} + gh\frac{\partial b}{\partial y} - \omega hu - \mu hv. \tag{11.41c}$$

This form is convenient for certain numerical implementations, as it enables (11.40a) to be enforced exactly over various subdomains in the numerical representation. Note that (11.41) may be rewritten in terms of what may be identified as the **primitive variables** $\{f,u,v\}$, thereby casting the SWE in the **primitive PDE form**

$$\frac{\partial f}{\partial t} = -u\frac{\partial (f+b)}{\partial x} - v\frac{\partial (f+b)}{\partial y} - (f+b)\Big(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\Big) - \frac{\partial b}{\partial t}, \tag{11.42a}$$

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - v\frac{\partial u}{\partial y} - g\frac{\partial f}{\partial x} + \omega v - \mu u, \tag{11.42b}$$

$$\frac{\partial v}{\partial t} = -u\frac{\partial v}{\partial x} - v\frac{\partial v}{\partial y} - g\frac{\partial f}{\partial y} - \omega u - \mu v. \tag{11.42c}$$

Note also that $b(x,y,t)$ appears explicitly on the RHS of (11.41b)-(11.41c), and appears in a different form on the RHS of (11.42a); familiarity with various equivalent formulations of a given system is often found to be useful to sharpen one's physical understanding of particular aspects of the system dynamics.

Linearizing (11.42) around zero (that is, assuming $\{f,u,v\}$ are all small, and thus products of these quantities are negligible) and assuming the Coriolis and viscous terms are also negligible, it follows [by taking $\partial/\partial x$ of b times (b) and $\partial/\partial y$ of b times (c) and substituting into $\partial/\partial t$ of (a) below] that

$$\left.\begin{array}{ll} \text{(a)} & \dfrac{\partial f}{\partial t} = -\dfrac{\partial\, u\, b}{\partial x} - \dfrac{\partial\, v\, b}{\partial y} - \dfrac{\partial b}{\partial t} \\ \text{(b)} & \dfrac{\partial u}{\partial t} = -g\dfrac{\partial f}{\partial x} \\ \text{(c)} & \dfrac{\partial v}{\partial t} = -g\dfrac{\partial f}{\partial y} \end{array}\right\} \quad \Rightarrow \quad \frac{\partial^2 f}{\partial t^2} = g\Big(\frac{\partial}{\partial x} b \frac{\partial f}{\partial x} + \frac{\partial}{\partial y} b \frac{\partial f}{\partial y}\Big) - \frac{\partial^2 b}{\partial t^2}. \tag{11.43}$$

Subject to the added assumption that the reference depth b is constant in both time & space, it follows that

$$\frac{\partial^2 f}{\partial t^2} = g\, b\Big(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\Big). \tag{11.44}$$

Defining the wave speed $c = \sqrt{g\,b}$, this linearized system is considered analytically in a rectangular domain in §4.3.3, and in an infinite domain in polar coordinates with azimuthal symmetry in §11.1.3.1; for a finite circular domain, the Fourier-Bessel representation discussed in §5.14 is well suited.

Conservation properties

As shown in §4.3.3, oscillations in a finite domain governed by the linearized SWE (11.44), where b is constant, do not decay or grow in time. Relaxing the assumption of constant b, the linearized SWE system (11.43), with $b = b(x,y)$, is still expected to be "purely oscillatory" as, apparently, a sloping bottom profile doesn't dampen energy in the system, it just modifies various wave speeds. This expectation can be made precise, even in the nonlinear setting, by performing an energy analysis directly on the nonlinear SWE. Defining the **kinetic energy of the water column per unit area** as $K = h(u^2+v^2)/2$, multiplying (11.42b) by hu and (11.42c) by hv and adding, then adding $(u^2+v^2)/2$ times (11.41a) and simplifying gives

$$\frac{\partial K}{\partial t} = -\frac{\partial uK}{\partial x} - \frac{\partial vK}{\partial y} - hu\frac{\partial gf}{\partial x} - hv\frac{\partial gf}{\partial y} - \mu K/2. \tag{11.45}$$

Defining the **potential energy of the water column per unit area** as $P = \int_{-b}^{f}(g\,z)dz = g(f^2-b^2)/2$, multiplying (11.41a) by $g\,f$ gives

$$\frac{\partial P}{\partial t} = -g\,f\frac{\partial hu}{\partial x} - g\,f\frac{\partial hv}{\partial y}. \tag{11.46}$$

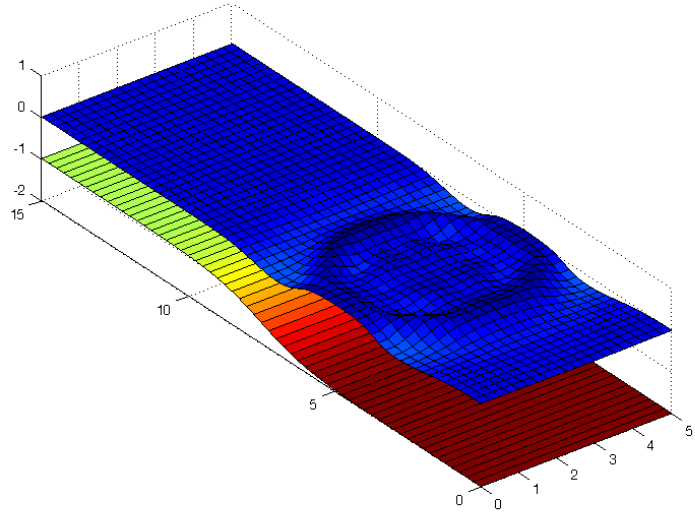


Figure 11.6: Simulation of of the wave resulting from a stone thrown into a shallow pool, as approximated by the shallow water equations (11.41) with boundary conditions as given in (11.49).

Thus, the evolution of the total energy $E = K + P$ is governed by

$$\frac{\partial E}{\partial t} = -\frac{\partial u(K+hgf)}{\partial x} - \frac{\partial v(K+hgf)}{\partial y} - \mu K/2. \tag{11.47}$$

Integrating over a domain with zero normal velocity at the boundaries, it is thus clear that, in the limit that $\mu \to 0$, energy is conserved. This result may be written back in integral conservation form as

$$\frac{d}{dt}\int\int_{\Omega} E\,dxdy + \int_{\partial\Omega} \begin{pmatrix} u(K+hgf) & v(K+hgf) \end{pmatrix} \mathbf{n}\,ds = -\int\int_{\Omega} \mu K/2\,dxdy. \tag{11.48}$$

Boundary conditions

Consider now a system governed by the nonlinear SWE (11.40) [which may, if the solution is smooth, be written in the PDE forms (11.41) or (11.42)] inside a rectangular $L_x \times L_y$ domain Ω with $\partial b/\partial x = 0$ at $x = \pm L_x/2$, and that $\partial b/\partial y = 0$ at $y = \pm L_y/2$. The boundary conditions on the system are

$$u = 0 \quad \text{and} \quad \frac{\partial h}{\partial x} = 0 \quad \text{at} \quad x = \pm L_x/2 \quad \text{for} \quad -L_y/2 \le y \le L_y/2; \tag{11.49a}$$

$$v = 0 \quad \text{and} \quad \frac{\partial h}{\partial y} = 0 \quad \text{at} \quad y = \pm L_y/2 \quad \text{for} \quad -L_x/2 \le x \le L_x/2. \tag{11.49b}$$

The BCs provided above only list *two* conditions along each boundary, though we must ultimately specify the evolution of all *three* variables $\{u, v, h\}$ at the boundaries in order for the problem to be well posed. Fortunately, the evolution of v at $x = \pm L_x/2$ may be determined directly from the $\partial v/\partial t$ component of the SWE itself combined with the BC $u = 0$; ditto for the evolution of u at $y = \pm L_y/2$. In PDE form, we have

$$\frac{\partial v}{\partial t} = -v\frac{\partial v}{\partial y} - g\frac{\partial (h-b)}{\partial y} - \mu v \quad \text{at} \quad x = \pm L_x/2 \quad \text{for} \quad -L_y/2 < y < L_y/2; \tag{11.49c}$$

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - g\frac{\partial (h-b)}{\partial x} - \mu u \quad \text{at} \quad y = \pm L_y/2 \quad \text{for} \quad -L_x/2 < x < L_x/2. \tag{11.49d}$$

The time evolution of the linearized SWE system with $b =$constant and $f = h - b$, as governed by (11.44) with BCs given by (11.49), may be solved analytically with the SOV approach (§4.3.3). Relaxing the assumption of constant b, the linearized 2D system (11.43) with $b = b(x,y)$ does not lend itself to SOV analysis[12]. However, we may still numerically simulate (11.43) [and, indeed, the nonlinear PDE forms (11.41) and (11.42)] using FD or spectral methods, as illustrated in Figure 11.6 and discussed further in §11.3.

[12]Note that the 1D form of (11.43) with $b = b(x)$ *does* lend itself to simple SOV analysis, however, with $f^m(x,t) = X(x)T(t)$ and $T''(t)/T(t) = g\,[b(x)X'(x)]'/X(x)$. The x component of this SOV analysis is a Sturm-Liouville problem (see Footnote 1 on Page 75). Several cases of this class of problems, corresponding here to specific functions $b(x)$, have well known solutions. For example, in the case of quadratic $b(x)$, the equation for $X(x)$ reduces to the **Legendre equation**, and thus $X(x)$ may be expanded with **Legendre polynomials** instead of sinusoidal functions. See Grimshaw *et al.* (2010) for extensive further discussion.

11.1.3.5 Jump conditions and the Rankine-Hugonoit relations

Hyperbolic systems like the SWE and Euler equation, in integral conservation form, admit solutions that are only **piecewise continuous**; the discontinuities in such solutions, known as **hydraulic jumps** in the case of the SWE or **shocks** in the case of Euler, are important phenomena to capture in numerical simulations.

To illustrate, consider the 2D SWE in integral conservation form, (11.40), and, at time t, select, WLOG, local coordinates in the vicinity some point on the (possibly moving) jump, $\{x_s(t), y_s(t)\}$, such that the x direction is locally normal to the jump, the y direction is locally tangential to the jump, and $u > 0$. Define a small fixed domain, $x_- < x < x_+$ and $y_1 < y < y_2$, with $x_+ - x_- = y_2 - y_1 = \delta$, which happens to be centered at $\{x_s(t), y_s(t)\}$ at time t (and, thus, $x_- < x_s(t) < x_+$ at time t). Assuming the bottom profile $b(x,y)$ is smooth within this domain and taking the limit as $\delta \to 0$ (and, thus, x_- denotes the location just upstream of the jump, and x_+ denotes the location just downstream of the jump) reduces (11.40) to

$$\frac{d}{dt}\int_{x_-}^{x_+}\int_{y_1}^{y_2} \mathbf{q}\, dxdy = -\int_{y_1}^{y_2} \mathbf{f}_x(\mathbf{q})\Big|_{x=x_-}^{x=x_+} ds, \quad \mathbf{q} = \begin{pmatrix} h \\ hu \\ hv \end{pmatrix}, \quad \mathbf{f}_x(\mathbf{q}) = \begin{pmatrix} hu \\ hu^2 + gh^2/2 \\ huv \end{pmatrix}$$

$$\Rightarrow \quad \begin{cases} \dfrac{d}{dt}\left[\displaystyle\int_{x_-}^{x_s(t)} h\,dx \;+ \int_{x_s(t)}^{x_+} h\,dx\right] = -\Big[hu\Big]_{x_-}^{x_+} & \Rightarrow \quad s = \dfrac{u_+h_+ - u_-h_-}{h_+ - h_-} \\ \dfrac{d}{dt}\left[\displaystyle\int_{x_-}^{x_s(t)} hu\,dx + \int_{x_s(t)}^{x_+} hu\,dx\right] = -\Big[hu^2 + \dfrac{gh^2}{2}\Big]_{x_-}^{x_+} & \Rightarrow \quad s = \dfrac{h_+u_+^2 - h_-u_-^2 + [gh_+^2 - gh_-^2]/2}{h_+u_+ - h_-u_-} \\ \dfrac{d}{dt}\left[\displaystyle\int_{x_-}^{x_s(t)} hv\,dx + \int_{x_s(t)}^{x_+} hv\,dx\right] = -\Big[huv\Big]_{x_-}^{x_+} & \Rightarrow \quad s = \dfrac{h_+u_+v_+ - h_-u_-v_-}{h_+v_+ - h_-v_-}. \end{cases} \tag{11.50}$$

where $h_\pm = h(x_\pm, y_s)$, $u_\pm = u(x_\pm, y_s)$, $v_\pm = v(x_\pm, y_s)$, and $s = dx_s(t)/dt$. The last three conditions at right are known as the **jump conditions** of the SWE. The third relation follows trivially from the first if $v_+ = v_-$. In the special case of a **stationary hydraulic jump** ($s = 0$), it follows that

$$h_-u_- = h_+u_+ \triangleq Q, \qquad Qu_- + gh_-^2/2 = Qu_+ + gh_+^2/2. \tag{11.51}$$

To simplify the discussion that follows, we thus shift to a reference frame moving with the hydraulic jump itself, and denote by u_- and u_+ the flow velocity normal to the jump upstream and downstream of the jump in this reference frame. Combining (11.51) to eliminate $\{u_+, u_-\}$, it follows that the height of the water column downstream of a jump may be computed in terms of the conditions upstream of the jump as follows:

$$Q^2 = g(h_+ + h_-)h_+h_-/2 \quad \Rightarrow \quad h_+ = \left(-h_- + \sqrt{h_-^2 + 8u_-^2h_-/g}\right)/2. \tag{11.52a}$$

Motivated by the flux terms in (11.47), we define the **energy flux** in the x direction as $u(K + hgf) \triangleq Qeg$, where the **specific energy** of the shallow water flow may be defined as $e = h - b + (u^2 + v^2)/(2g)$; note that the specific energy of the flow has dimensions of length, and is often referred to (especially in **open channel flows**) as the **head** of the flow. Looking at the conditions immediately upstream and downstream of a hydraulic jump and applying the condition at left in (11.52a), we may write

$$e_- = e_+ + \Delta e \quad \Rightarrow \quad h_- + u_-^2/(2g) = h_+ + u_+^2/(2g) + \Delta e \quad \Rightarrow \quad \Delta e = (h_+ - h_-)^3/(4h_+h_-), \tag{11.52b}$$

where Δe denotes the **head loss** (i.e., the loss of specific energy) associated with the jump. Defining the **Froude number** of a shallow water flow as $Fr = u/\sqrt{gh}$, it follows that $h_+ > h_-$, and thus that[13] $\Delta e > 0$, if $Fr_- > 1$ (i.e., if the flow upstream of the jump is **supercritical**), in which case $Fr_+ < 1$ (i.e., the flow downstream of the jump will be **subcritical**). If, on the other hand, $Fr_- < 1$ (that is, if a shallow water flow is initially subcritical), then a hydraulic jump will not occur, as it would require a spontaneous *gain* in the specific energy of the flow in the jump, which is unphysical.

[13] The flow loses specific energy in a hydraulic jump due to the turbulence it induces; this lost energy is ultimately dissipated as heat.

Similarly, consider the **compressible 3D Euler equation** in integral conservation form [cf. (11.40)]:

$$\frac{d}{dt}\int_\Omega \mathbf{q}\,dV + \int_{\partial\Omega} F(\mathbf{q})\,\mathbf{n}\,dA = 0, \tag{11.53a}$$

$$\mathbf{q} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{pmatrix}, \quad F(\mathbf{q}) = \begin{pmatrix} \rho u & \rho v & \rho w \\ p+\rho u^2 & \rho uv & \rho uw \\ \rho uv & p+\rho v^2 & \rho vw \\ \rho uw & \rho vw & p+\rho w^2 \\ u(E+p) & v(E+p) & w(E+p) \end{pmatrix} = \begin{bmatrix} \mathbf{f}_x & \mathbf{f}_y & \mathbf{f}_z \end{bmatrix}, \tag{11.53b}$$

where ρ is the **density** of the flow, p is the **pressure** of the flow, $E = \rho[e + (u^2+v^2+w^2)/2]$ is the **total energy per unit volume** of the flow, and, for a **calorically perfect gas**, $e = p/[\rho(\gamma-1)]$ is the **internal energy per unit mass** of the flow, where $\gamma = c_p/c_v > 1$ is the (constant) **ratio of specific heats** of the fluid[14]. At time t, select local coordinates in the vicinity of a point on the shock, $\{x_s(t), y_s(t), z_s(t)\}$, such that the x direction is normal to the shock, the y and z directions are tangential to the shock, and $u > 0$. Define a small fixed domain, $x_- < x < x_+$, $y_1 < y < y_2$, $z_1 < z < z_2$, with $x_+ - x_- = y_2 - y_1 = z_2 - z_1 = \delta$, which is centered at $\{x_s(t), y_s(t), z_s(t)\}$ at time t. Taking the limit as $\delta \to 0$ (and, thus, x_- denotes the location just upstream of the shock, and x_+ denotes the location just downstream the shock) reduces (11.53) to

$$\frac{d}{dt}\int_{x_-}^{x_+}\int_{y_1}^{y_2}\int_{z_1}^{z_2} \mathbf{q}\,dx\,dy\,dz = -\int_{y_1}^{y_2}\int_{z_1}^{z_2} \mathbf{f}_x(\mathbf{q})\Big|_{x=x_-}^{x=x_+} dy\,dz$$

$$\Rightarrow \quad \begin{cases} \dfrac{d}{dt}\left[\displaystyle\int_{x_-}^{x_s(t)} \rho\,dx + \int_{x_s(t)}^{x_+} \rho\,dx\right] = -\Big[\rho u\Big]_{x_-}^{x_+} & \Rightarrow \quad s = \dfrac{\rho_+ u_+ - \rho_- u_-}{\rho_+ - \rho_-} \\ \dfrac{d}{dt}\left[\displaystyle\int_{x_-}^{x_s(t)} \rho u\,dx + \int_{x_s(t)}^{x_+} \rho u\,dx\right] = -\Big[p+\rho u^2\Big]_{x_-}^{x_+} & \Rightarrow \quad s = \dfrac{(p_+ + \rho_+ u_+^2) - (p_- + \rho_- u_-^2)}{\rho_+ u_+ - \rho_- u_-} \\ \dfrac{d}{dt}\left[\displaystyle\int_{x_-}^{x_s(t)} \rho v\,dx + \int_{x_s(t)}^{x_+} \rho v\,dx\right] = -\Big[\rho uv\Big]_{x_-}^{x_+} & \Rightarrow \quad s = \dfrac{\rho_+ u_+ v_+ - \rho_- u_- v_-}{\rho_+ v_+ - \rho_- v_-} \\ \dfrac{d}{dt}\left[\displaystyle\int_{x_-}^{x_s(t)} \rho w\,dx + \int_{x_s(t)}^{x_+} \rho w\,dx\right] = -\Big[\rho uw\Big]_{x_-}^{x_+} & \Rightarrow \quad s = \dfrac{\rho_+ u_+ w_+ - \rho_- u_- w_-}{\rho_+ w_+ - \rho_- w_-} \\ \dfrac{d}{dt}\left[\displaystyle\int_{x_-}^{x_s(t)} E\,dx + \int_{x_s(t)}^{x_+} E\,dx\right] = -\Big[u(E+p)\Big]_{x_-}^{x_+} & \Rightarrow \quad s = \dfrac{u_+(E_+ + p_+) - u_-(E_- + p_-)}{E_+ - E_-}, \end{cases} \tag{11.54}$$

where $\rho_\pm = \rho(x_\pm, y_s, z_s)$, etc., and $s = dx_s(t)/dt$. The third and fourth relations follow trivially from the first if $v_- = v_+$ and $w_- = w_+$. In the case of a **stationary shock** ($s = 0$), it follows that

$$\rho_- u_- = \rho_+ u_+, \qquad p_- + \rho_- u_-^2 = p_+ + \rho_+ u_+^2, \qquad u_-^2/2 + e_- + p_-/\rho_- = u_+^2/2 + e_+ + p_+/\rho_+. \tag{11.55}$$

To simplify, we thus shift to a reference frame moving with the shock, and denote by u_- and u_+ the flow velocity normal to the shock upstream and downstream of the shock in this reference frame. Combining these equations leads, after some algebra, to the **Rankine-Hugonoit** relations

$$\frac{p_+}{p_-} = \frac{(\gamma+1)\rho_+ - (\gamma-1)\rho_-}{(\gamma+1)\rho_- - (\gamma-1)\rho_+}, \quad \frac{\rho_+}{\rho_-} = \frac{(\gamma+1)p_+ + (\gamma-1)p_-}{(\gamma+1)p_- + (\gamma-1)p_+}, \quad u_- u_+ = \frac{p_+ - p_-}{\rho_+ - \rho_-} = \gamma\frac{p_+ + p_-}{\rho_+ + \rho_-}. \tag{11.56}$$

For a **thermally perfect gas**, we may write $p = \rho RT$ where T is the local **temperature** of the flow; note that the **specific gas constant** $R = 287.058$ J/(kg K) for dry air. The local **speed of sound** is $a = \sqrt{\gamma RT}$, and

[14] Note that $\gamma = 1.40$ for air at standard atmospheric pressure over a wide range of fluid temperatures.

the local **entropy** of the flow, compared with some reference state r, is $s - s_r = c_v \log[(p/p_r)/(\rho/\rho_r)^\gamma]$. The **normal**[15] **Mach number** is $M_- = u_-/a_-$ upstream of the shock and $M_+ = u_+/a_+$ downstream of the shock. It follows (see the indispensable NACA Report 1135 for further such relations) that

$$M_+^2 = \frac{(\gamma-1)M_-^2+2}{2\gamma M_-^2-(\gamma-1)}, \quad \frac{p_+}{p_-} = \frac{2\gamma M_-^2-(\gamma-1)}{\gamma+1}, \quad \frac{T_+}{T_-} = \frac{a_+^2}{a_-^2} = \frac{[2\gamma M_-^2-(\gamma-1)]\,[(\gamma-1)M_-^2+2]}{(\gamma+1)^2 M_-^2},$$
$$\frac{\rho_+}{\rho_-} = \frac{u_-}{u_+} = \frac{(\gamma+1)M_-^2}{(\gamma-1)M_-^2+2}, \quad \frac{s_+-s_-}{c_v} = \log\left[\frac{2\gamma M_-^2-(\gamma-1)}{\gamma+1}\right] - \gamma\log\left[\frac{(\gamma+1)M_-^2}{(\gamma-1)M_-^2+2}\right].$$

Thus, if the flow upstream of the shock is **supersonic** ($M_- > 1$), then the flow downstream of the shock is **subsonic** ($M_+ < 1$), higher pressure ($p_+ > p_-$), higher temperature ($T_+ > T_-$), higher density ($\rho_+ > \rho_-$), lower speed ($u_+ < u_-$), and higher entropy ($s_+ > s_-$). If, on the other hand, $M_- < 1$ (that is, if the flow is initially subsonic), then a shock will not occur, as it would require a *decrease* in entropy, which is unphysical.

11.1.4 Classification of more complicated PDEs

Identification of a PDE as elliptic, parabolic, or hyperbolic is a valuable step in its characterization, as it reveals the domain of dependence of any given system state as well as the range of influence of any given input to the system. Such identification is also useful for determining the number of boundary conditions (and, in unsteady problems, initial conditions) that are necessary to solve it.

The elliptic, parabolic, and hyperbolic classifications introduced at the beginning of §11.1 are often extended to more complicated PDEs by noting the relative signs of their highest-order derivative terms. In systems with at most second-order derivatives, if a change of variables is performed such that all cross derivative terms are eliminated [note that (11.1)-(11.4) are already in this form], then a general scalar second-order system in n dimensions may be written:

$$\alpha_1 \frac{\partial^2\phi}{\partial\xi_1^2} + \alpha_2 \frac{\partial^2\phi}{\partial\xi_2^2} + \ldots + \alpha_n \frac{\partial^2\phi}{\partial\xi_n^2} = \ldots, \tag{11.57a}$$

where there are no derivatives of ϕ higher than first on the RHS (though it may be nonlinear). Considering modes of the form $\phi = \hat{\phi}\, e^{\vec{\ell}\cdot\vec{\xi}}$, the dispersion relation corresponding to this differential equation is

$$\alpha_1\ell_1^2 + \alpha_2\ell_2^2 + \ldots + \alpha_n\ell_n^2 = \ldots, \tag{11.57b}$$

where there are no powers of ℓ_i higher than one in the terms on the RHS. Replacing the RHS with a constant, as illustrated for the canonical PDEs (11.1)-(11.4) in Figure 11.1,

- if all of the α_i are positive, isosurfaces of (11.57b) produce an ellipse;
- if one of the α_i is zero and the others positive, isosurfaces of (11.57b) produce a parabola;
- if one of the α_i is negative and the others positive, isosurfaces of (11.57b) produce a hyperbola;

the corresponding PDEs are thus referred to as **elliptic**, **parabolic**, and **hyperbolic** respectively[16].

Fourth-order, sixth-order, and eighth-order PDEs are sometimes characterized as elliptic, parabolic, and hyperbolic, respectively, by the natural generalizations of the second-order case. For example, the equations

$$\ell_1^4 + \ell_2^4 + \ell_3^4 = c_0, \qquad s - \nu(\ell_1^4 + \ell_2^4 + \ell_3^4) = c_0, \qquad \text{and} \qquad s^4 - c^4(\ell_1^4 + \ell_2^4 + \ell_3^4) = c_0$$

[15] That is, the Mach number of the component of the flow normal to the shock.

[16] The other possible cases are rare in problems of physical significance. One additional category sometimes mentioned is the case when half of the α_i are positive and the other half are negative, the corresponding PDE in this case is referred to as **ultrahyperbolic**.

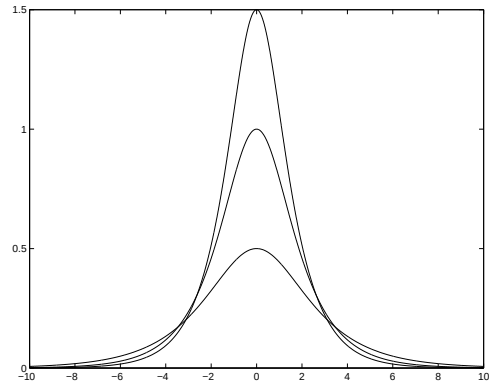


Figure 11.7: Plot of the soliton solution (11.59) to the KdV equation (11.58) at $t = 0$ for $c = 1$, 2, and 3; note that, as the soliton gradually reduces in height as it propogates in any physical implementation (due to unmodeled losses due to viscosity of the fluid and surface tension at the air/fluid interface), the width of the soliton increases slightly, and its wave speed c decreases significantly.

might be said, with a minor **abuse of notation**[17], to define, respectively, a "fourth-order ellipse", a "fourth-order parabola", and a "fourth-order hyperbola", and thus fourth-order PDEs with dispersion relations in such forms may be characterized accordingly.

Unfortunately, attempts at classification of PDEs of order higher than two as elliptic, parabolic, or hyperbolic can sometimes lead to ambiguities; for further discussion of such attempts, see Garabedian (1998).

11.1.5 Wavelike behavior and solitons

It is important to note that traveling wave behavior in the dynamics of a solution to a PDE is not sufficient to identify it as hyperbolic. This was already seen in the dispersive 1D wave equation (11.37a), for which solutions appeared essentially as traveling waves for small k_p (or small d), yet the equation was in fact parabolic (due to the fact that the wave speed $\overline{c}_p$ increases without bound with increasing k_p).

Another curious system which demonstrates traveling wave behavior, but is certainly not hyperbolic (it might be said to be "third-order parabolic"), is the **Korteweg and deVries** (**KdV**) equation

$$\frac{\partial u}{\partial t} = -6u\frac{\partial u}{\partial x} - \frac{\partial^3 u}{\partial x^3} \tag{11.58}$$

on the real line $x \in (-\infty, \infty)$. This equation describes weakly nonlinear shallow water waves. A traveling wave solution of this equation, which was first observed by John Scott Russel on the Union Canal near Edinburgh, Scotland in 1834 (and reportedly convected for well over a mile without significant change of shape—at thirty feet long and over a foot high!), is given by setting $u(x,t) = f(\xi)$ where $\xi \triangleq x - ct$ in (11.58), resulting in

$$\frac{df}{d\xi}\frac{\partial \xi}{\partial t} = -6f\frac{df}{d\xi}\frac{\partial \xi}{\partial x} - \frac{d^3 f}{d\xi^3}\left(\frac{\partial \xi}{\partial x}\right)^3 \quad \Rightarrow \quad cf' = 6ff' + f'''.$$

Integrating once, with zero boundary conditions at infinity, then multiplying by f' and integrating again yields

$$cf = 3f^2 + f'' \quad \Rightarrow \quad \frac{c}{2}f^2 = f^3 + \frac{1}{2}(f')^2 \quad \Rightarrow \quad \frac{df}{d\xi} = \sqrt{cf^2 - 2f^3}.$$

[17] As mentioned in Footnote 8 on Page 24, an **abuse of notation** is the use of a mathematical term in a setting in which it is not formally correct, but is suggestive of the correct mathematical notion.

The interesting solution of the latter equation observed by Russel, as easily verified by substitution, is

$$f(\xi) = \frac{c}{2}\,\mathrm{sech}^2\,\frac{\sqrt{c}\,\xi}{2} \quad \Rightarrow \quad u(x,t) = f(x-ct) = \frac{c}{2}\,\mathrm{sech}^2\,\frac{\sqrt{c}\,(x-ct)}{2}. \tag{11.59}$$

Noting the $(x-ct)$ dependence, it is seen that this solution, referred to as a **soliton** (see Figure 11.7), is in fact a traveling wave with a particular shape (a single "hump") and a particular height.

11.1.6 The fundamental equations of computational fluid dynamics

To motivate the development of efficient simulation tools, we now survey a variety of prototypical PDEs commonly encountered in **computational fluid dynamics** (**CFD**). Some parabolic forms include:

the incompressible Navier-Stokes equation (**NSE**):
[$\vec{u}$ = velocity, p = pressure, $\vec{\psi}$ = forcing, $\{\mu,\rho\}$ = constants]

$$\begin{cases} \dfrac{\partial \vec{u}}{\partial t} = -(\vec{u}\cdot\nabla)\vec{u} + \dfrac{\mu}{\rho}\Delta\vec{u} - \dfrac{\nabla p}{\rho} + \vec{\psi}, \\ \nabla\cdot\vec{u} = 0; \end{cases} \tag{11.60}$$

the passive scalar convection/diffusion equation:
[ϕ = scalar concentration, k = constant, ξ = forcing]

$$\frac{\partial \phi}{\partial t} = -(\vec{u}\cdot\nabla)\phi + k\Delta\phi + \xi; \tag{11.61}$$

the 2D boundary-layer equation:
[$p(x)$ = pressure gradient of external flow]

$$\begin{cases} \dfrac{\partial u}{\partial x} = \dfrac{1}{u}\Big[-v\dfrac{\partial u}{\partial y} - \dfrac{1}{\rho}\dfrac{\partial p}{\partial x} + \dfrac{\mu}{\rho}\dfrac{\partial^2 u}{\partial y^2}\Big], \\ \dfrac{\partial p}{\partial y} = 0, \quad \dfrac{\partial v}{\partial y} = -\dfrac{\partial u}{\partial x}. \end{cases} \tag{11.62}$$

Some hyperbolic forms include:

the vector wave equation:
[$\vec{u}$ = velocity, $\{\lambda,\mu\}$ = constants, $\vec{\psi}$ = forcing]

$$\frac{\partial^2 \vec{u}}{\partial t^2} = (\lambda+2\mu)\nabla(\nabla\cdot\vec{u}) - \mu\nabla\times(\nabla\times\vec{u}) + \vec{\psi}; \tag{11.63}$$

the 2D shallow water equation (**SWE**):
[$\vec{u}$ = velocity, h = water depth, b = bottom depth below mean, $(h-b)$ = free surface above/below mean, g = gravity]

$$\begin{cases} \dfrac{\partial \vec{u}}{\partial t} = -(\vec{u}\cdot\nabla)\vec{u} - g\nabla(h-b), \\ \dfrac{\partial h}{\partial t} = -\nabla\cdot(h\vec{u}). \end{cases} \tag{11.64}$$

A prototypical PDE of **mixed elliptic/hyperbolic** type is the **compressible Euler equation** for a perfect gas:

$$\frac{\partial \rho}{\partial t} = -\nabla\cdot(\rho\vec{u}), \quad \frac{\partial(\rho\vec{u})}{\partial t} = -\nabla\cdot(\rho\vec{u}\otimes\vec{u}) - \nabla p, \quad \frac{\partial p}{\partial t} = -\nabla\cdot(\rho\vec{u}) - (\gamma-1)\,p\,(\nabla\cdot\vec{u}); \tag{11.65}$$

note that, if the flow considered is steady and irrotational, (11.65) may be written in *potential form* as

$$\Big(1-\frac{u^2}{c^2}\Big)\frac{\partial^2\Phi}{\partial x^2} + \Big(1-\frac{v^2}{c^2}\Big)\frac{\partial^2\Phi}{\partial y^2} + \Big(1-\frac{w^2}{c^2}\Big)\frac{\partial^2\Phi}{\partial z^2} = \frac{2uv}{c^2}\frac{\partial^2\Phi}{\partial x\partial y} + \frac{2vw}{c^2}\frac{\partial^2\Phi}{\partial y\partial z} + \frac{2uw}{c^2}\frac{\partial^2\Phi}{\partial x\partial z}, \tag{11.66a}$$

where

$$u = \frac{\partial\Phi}{\partial x}, \quad v = \frac{\partial\Phi}{\partial y}, \quad w = \frac{\partial\Phi}{\partial z}, \quad q = \sqrt{u^2+v^2+w^2}, \tag{11.66b}$$

the local speed of sound c may be determined from **Crocco's theorem**

$$c^2 + \frac{\gamma-1}{2}q^2 = c_\infty^2 + \frac{\gamma-1}{2}q_\infty^2, \tag{11.66c}$$

γ denotes the specific heat ration ($\gamma = C_p/C_v \approx 1.40$ for air), and the subscript ∞ denotes the (known) **free-stream** reference conditions (that is, away from the region of interest); note further that (11.66) may itself be written and solved in a convenient *nondimensional conservative form*

$$\frac{\partial}{\partial x}\Big(\rho'\frac{\partial\Phi'}{\partial x}\Big) + \frac{\partial}{\partial y}\Big(\rho'\frac{\partial\Phi'}{\partial y}\Big) + \frac{\partial}{\partial z}\Big(\rho'\frac{\partial\Phi'}{\partial z}\Big) = 0, \tag{11.67a}$$

where

$$\Phi' = \frac{\Phi}{q_\infty L}, \quad \rho' = \frac{\rho}{\rho_\infty} = \left[1 + \frac{\gamma - 1}{2} M_\infty^2 \left(1 - \frac{q^2}{q_\infty^2}\right)\right]^{1/(\gamma-1)}, \tag{11.67b}$$

L is a reference length scale in the system, and $M_\infty = q_\infty / c_\infty$ is the freestream Mach number. Note in particular in (11.66a) that, if the coördinates are rotated such that $v = w = 0$, it is easily seen that the system is **locally hyperbolic** wherever the local Mach number $M = q/c > 1$, and **locally elliptic** wherever $M = q/c < 1$.

Algorithm 11.1: Simulating the 1D diffusion equation with CN in time and FD in space.

View

```
function Diffusion1D_CN_FD
% Simulate the 1D diffusion equation on 0<x<L with Dirichlet BCs
% using CN in time and 2nd-order central FD in space.
clear all; L=5; Tmax=10; N=100; dt=0.05; TimeSteps=Tmax/dt; dx=L/N;
t=0; x=(0:N)'*dx; phi=zeros(N+1,1); PlotXY(x,phi,t,0,L,-1,1);
a(2:N,1)= -dt/(2*dx^2); a(1)=0;                    % Precalculate time-stepping coefficients
b(2:N,1)=1+dt/dx^2;      b(1)=1;                   % in order to minimize the flops needed
c(2:N,1)= -dt/(2*dx^2); c(1)=0;  e=dt/(2*dx^2);    % inside the time-marching loop.
rhs=zeros(N,1); [rhs,a,b,c]=Thomas(a,b,c,rhs,N); % Determine L & U to reuse during march.
for n=1:TimeSteps                                  % Leading-order cost:
  rhs(2:N,1)=phi(2:N)+e*(phi(3:N+1)-2*phi(2:N)+phi(1:N-1));   % ~ 5N
  t=t+dt; rhs(1)=sin(t); phi(1:N)=ThomasLU(a,b,c,rhs,N);      % ~ 5N
  PlotXY(x,phi,t,0,L,-1,1);
end                                                % Total: ~ 10N per timestep
end % function Diffusion1D_CN_FD
```

11.2 Numerical simulation of parabolic PDEs

11.2.1 Time marching a spatial discretization developed via finite differences

We introduce this section with the simulation of the time evolution of $\phi(x,t)$ in the 1D diffusion equation,

$$\frac{\partial \phi}{\partial t} = \nu \Delta \phi, \tag{11.68}$$

on the domain $x \in [0, 2\pi]$ for $t \geq 0$ with BCs $\phi(x=0) = \sin(t)$ and $\phi(x=2\pi) = 0$ and ICs $\phi(t=0) = 0$. As suggested at the beginning of §11, we may approach this problem by first discretizing the PDE in space to arrive at a system of ODEs, then marching the resulting ODE system in time. We illustrate below using a second-order central FD method for the spatial discretization and the CN method for the time discretization:

1. Discretize the unknown variable $\phi(x)$ on a spatial grid with $N-1$ interior gridpoints and uniform grid spacing $\Delta x = 2\pi/N$ such that the x coördinate of the j'th gridpoint is $x_j = j(\Delta x)$ for $j = 0 \ldots N$ (note that, in Matlab, all indices must be shifted by 1, because Matlab indices must begin from 1).

2. Enforce an approximation of the PDE at the interior gridpoints by replacing the spatial derivatives on the RHS of (11.68) with the second-order central approximation of the second derivative such that

$$\frac{\partial \phi_j}{\partial t} = \frac{\phi_{j+1} - 2\phi_j + \phi_{j-1}}{(\Delta x)^2} \quad \text{for} \quad j = 1 \ldots N-1.$$

3. Discretize this ODE system in time using CN (since the ODE system is linear with tightly banded A) and apply the BCs $\phi_0 = \sin(t)$ and $\phi_N = 0$, resulting in the following linear difference equation:

$$\begin{pmatrix} \phi_0^{n+1} \\ \phi_1^{n+1} \\ \vdots \\ \phi_{N-1}^{n+1} \\ \phi_N^{n+1} \end{pmatrix} = \begin{pmatrix} \sin(t_{n+1}) \\ \phi_1^n \\ \vdots \\ \phi_{N-1}^n \\ 0 \end{pmatrix} + \frac{\Delta t}{2(\Delta x)^2} \begin{pmatrix} 0 & & & & 0 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 0 & & & & 0 \end{pmatrix} \left[\begin{pmatrix} \phi_0^{n+1} \\ \phi_1^{n+1} \\ \vdots \\ \phi_{N-1}^{n+1} \\ \phi_N^{n+1} \end{pmatrix} + \begin{pmatrix} \phi_0^n \\ \phi_1^n \\ \vdots \\ \phi_{N-1}^n \\ \phi_N^n \end{pmatrix} \right].$$

The resulting code, given in Algorithm 11.1, is self explanatory; note that the tridiagonal matrix appearing in the implicit solves at each timestep are identical from one step to the next in the above algorithm, and thus the LU decomposition of this matrix is used at each timestep in its efficient numerical implementation.

Algorithm 11.2: Simulating Burgers' equation with CN/RKW3 in time and FD in space, together with a convenient plotting algorithm used by several of the codes appearing in §11.

View

```
function Burgers_CNRKW3_FD
% Simulate the 1D Burgers on 0<x<L with homogeneous Dirichlet BCs using CN/RKW3 in time
% (explicit on nonlinear terms, implicit on linear terms) & 2nd-order central FD in space.
%%%%%%%%%%%%%%%%%%%%%%%% Initialize the simulation paramters (user input) %%%%%%%%%%%%%%%%%%%%%%%%
L=100; Tmax=50; N=100; dt=0.5; PlotInterval=1;
dx=L/N; x=(0:N)*dx; y=-sin(pi*x/L)-sin(2*pi*x/L)+sin(6*pi*x/L); PlotXY(x,y,0,0,L,-3,3)
%%%%%%%%%%%%%% Precalculate the time-stepping coefficients used in the simulation %%%%%%%%%%%%%%
h_bar = dt*[8/15  2/15   1/3];      d=h_bar/(2*dx^2);          a= -h_bar/(2*dx^2);
beta_bar = [1     25/8   9/4];      e=beta_bar.*h_bar/(2*dx);  b=1+h_bar/dx^2;
zeta_bar = [0    -17/8  -5/4];      f=zeta_bar.*h_bar/(2*dx);  c= -h_bar/(2*dx^2);
for k=1:Tmax/dt
  for rk=1:3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ALL 3 RK SUBSTEPS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Compute nonlinear term r(y), & defer scaling until next step %  Leading-order cost:
    r=-y(2:N).*(y(3:N+1)-y(1:N-1));                                          % ~ 2N
    % Compute entire rhs, and apply all the correct scalings
    if (rk==1)
      rhs=y(2:N) +d(rk)*(y(3:N+1)-2*y(2:N)+y(1:N-1)) +e(rk)*r;               % ~ 7N
    else
      rhs=y(2:N) +d(rk)*(y(3:N+1)-2*y(2:N)+y(1:N-1)) +e(rk)*r +f(rk)*rhs;    % ~ 9N
    end
    % Solve for new y
    y(2:N)=ThomasTT(a(rk),b(rk),c(rk),rhs',N-1);                             % ~ 8N
    % Save r (in rhs) for the next timestep                                  ---------------
    if (rk<3) rhs=r; end                                                     % Total: ~ 55N
  end %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END OF RK LOOP %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  if (mod(k,PlotInterval)==0) PlotXY(x,y,k*dt,0,L,-3,3); end
end
end % function Burgers_CNRKW3_FD
```

View

```
function PlotXY(x,y,t,xmin,xmax,ymin,ymax)
% A supplemental plotting code used in several of the simulations in Chapter 11 of NR.
figure(1); clf; plot(x,y); xlabel('x'); ylabel('u');
title(sprintf('Time = %5.2f',t)); axis([xmin xmax ymin ymax]); pause(0.001);
end % function PlotXY
```

11.2.1.1 A typical tradeoff between flops and storage

We next consider the simulation of Burgers' equation with homogenous Dirichlet BCs. As before, we use the second-order central FD method to approximate the spatial derivatives. We then march the nonlinear term explicitly (with RKW3) and the linear term implicitly (with CN) using the CN/RKW3 IMEX method described in §10.5.4. This leads to a set of ODEs of the form of (10.62) with

$$[\mathbf{f}(\mathbf{y})]_i = \nu \frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2} \quad \text{and} \quad [\mathbf{g}(\mathbf{y})]_i = -y_i \frac{y_{i+1} - y_{i-1}}{2\Delta x}.$$

The resulting simulation code is implemented in Algorithm 11.2. The KS case with homogenous Dirichlet BCs may be developed in an entirely analogous fashion (see Exercise 11.8); note that the KS case requires pentadiagonal solves, as the linear operator (treated implicitly, with second-order central discretizations) has both second-derivative and fourth-derivative terms.

The leading-order cost of Algorithm 11.2 is $\sim 55N$ flops per timestep. However, this implementation actually uses *three* vectors, {`y,r,rhs`}, of length N, in addition to the vector of length N required by the `ThomasTT` algorithm[18]. The discussion in §10.4.1.3 showed that RKW3, on its own, could be completed

[18] Had the LU decomposition of A been leveraged during the three tridiagonal solves, we could reduce the flops by $9N$ per timestep,

Algorithm 11.3: Simulating Burgers' equation as in Algorithm 11.2 but with reduced storage.

View

```
function Burgers_CNRKW3_FD_RS
% (... initialization identical to that in Burgers_RKW3CN_FD ...)
for k=1:Tmax/dt
   %%%%%%%%%%%%%%%%%%%%%%%%%% FIRST RK STEP %%%%%%%%%%%%%%%%%%%%%%%%%%
   % Compute nonlinear term r(y), and defer scaling until next step %  Leading-order cost:
   z(2:N)=-y(2:N).*(y(3:N+1)-y(1:N-1));                                  % ~ 2N
   % Compute the entire RHS, and apply all the correct scalings
   y(2:N)=y(2:N) + d(1)*(y(3:N+1)-2*y(2:N)+y(1:N-1)) + e(1)*z(2:N);      % ~ 7N
   % Solve for y at end of first RK step
   y(2:N)=ThomasTT(a(1),b(1),c(1),y(2:N)',N-1);                          % ~ 8N
   %%%%%%%%%%%%%%%%%%%%%%%%%% SECOND RK STEP %%%%%%%%%%%%%%%%%%%%%%%%%%
   % Compute the entire RHS, including r(y).
   z(2:N)=y(2:N) + d(2)*(y(3:N+1)-2*y(2:N)+y(1:N-1)) - ...               % ~ 11N
          e(2)*y(2:N).*(y(3:N+1)-y(1:N-1)) + f(2)*z(2:N);
   % TRICK: now RECOMPUTE r(y) for use at the final RK step.
   y(2:N)=-y(2:N).*(y(3:N+1)-y(1:N-1));                                  % ~ 2N
   % Solve for y at end of second RK step
   z(2:N)=ThomasTT(a(2),b(2),c(2),z(2:N)',N-1);  z(N+1)=0;               % ~ 8N
   %%%%%%%%%%%%%%%%%%%%%%%%%% THIRD RK STEP %%%%%%%%%%%%%%%%%%%%%%%%%%
   % Compute entire RHS, including the computation of r(y)
   y(2:N)=z(2:N) + d(3)*(z(3:N+1)-2*z(2:N)+z(1:N-1)) - ...               % ~ 11N
          e(3)*z(2:N).*(z(3:N+1)-z(1:N-1)) + f(3)*y(2:N);
   % Solve for y at the new timestep
   y(2:N)=ThomasTT(a(3),b(3),c(3),y(2:N)',N-1);                          % ~ 8N
   if (mod(k,PlotInterval)==0) PlotXY(x,y,k*dt,0,L,-3,3); end
   %%%%%%%%%%%%%%%%%%%%%%%%%% END OF RK LOOP %%%%%%%%%%%%%%%%%%%%%%%%%%  ----------------
end                                                                      % Total: ~ 57N
end % function Burgers_CNRKW3_FD_RS
```

using only two registers of length N. A question thus arises: can we be more efficient with memory when applying the CN/RKW3 IMEX scheme to this system, requiring only *two* registers of length N?

The answer, remarkably, is yes. Note that, with care, we can **unroll** the RK loop in Algorithm 11.2 and slightly reorder the computations into the equivalent form implemented in Algorithm 11.3. The leading-order cost of Algorithm 11.3 is $\sim 57N$ flops per timestep, which is slightly more expensive than Algorithm 11.2. However, we now use only *two* registers, {`y`, `z`}, of length N, in addition to the storage required by the `ThomasTT` algorithm. Thus, we identify a tradeoff: by performing less than 4% more floating point operations per timestep, the modified algorithm reduces memory usage by 25% (from $3+1=4$ vectors to $2+1=3$ registers of length N. On a modern (cache-based) computer, in which retrieving variables from the main memory is often a significant bottleneck in the computation, this would often be a favorable tradeoff to make.

11.2.2 Time marching a spatial discretization developed via spectral methods

We now revisit the problem of Burgers' equation considered in §11.2.1, now with periodic BCs which make the system amenable to an FFT-based analysis. Due to the similarity of the Burgers' and KS problems in this setting, we develop a single code capable of treating both the Burgers' and KS problems. We use a similar time marching scheme as suggested in §11.2.1, handling linear terms implicitly (with CN) and nonlinear explicitly (with RKW3). Derivates are now handled **pseudospectrally**, which means we work in Fourier space when

but the Thomas algorithm we would have to save three additional vectors of length N in order to achieve it, which is quite likely *not* a favorable tradeoff to make on most modern (cache-based) computers for large N.

Algorithm 11.4: Simulating Burgers and KS with CN/RKW3 in time and spectral differentiation in space.

View

```
function Burgers_KS_CNRKW3_PS
% Simulate the 1D Burgers or KS equation on 0<x<L with periodic BCs using CN/RKW3 in time
% (explicit on nonlinear terms, implicit on linear terms) & pseudospectral in space.
%%%%%%%%%%%%%%%%%%%%%% Initialize the simulation paramters (user input) %%%%%%%%%%%%%%%%%%%%%%
L=200; Tmax=250; N=256; dt=0.05; PlotInt=10; alpha=1; % alpha=0 for Burgers, alpha=1 for KS
dx=L/N; x=(0:N-1)'*dx; u=0.15*randn(N,1); uhat=RFFT(u,N);
%%%%%%%%%%%%% Precalculate the time-stepping coefficients used in the simulation %%%%%%%%%%%%
h_bar=dt*[8/15 2/15 1/3]; beta_bar=[1 25/8 9/4]; zeta_bar=[0 -17/8 -5/4];
kx=(2*pi/L)*[0:N/2-1]'; if alpha==0; Aop=-kx.^2; else Aop=kx.^2-kx.^4; end;
hb2=h_bar/2; hbbb=beta_bar.*h_bar; hbzb=zeta_bar.*h_bar; Imhb2=1-h_bar/2;
for k=1:Tmax/dt
  for rk=1:3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ALL 3 RK SUBSTEPS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    uhat(fix(N/3)+1:end)=0;  % Dealias (see Section 5.7).
    r=RFFTinv(uhat,N); r=-r.*r; rhat=i*kx.*RFFT(r,N);                    % Leading-order cost:
    if (rk==1)                                                           % 2 FFTs per RK step
      uhat=(uhat+hb2(rk)*Aop.*uhat+hbbb(rk)*rhat)./(1-hb2(rk)*Aop);
    else        % Implement (10.64); note that the "solve" is now simply scalar division!
      uhat=(uhat+hb2(rk)*Aop.*uhat+hbbb(rk)*rhat+hbzb(rk)*rhat_old)./(1-hb2(rk)*Aop);
    end
    if (rk<3) rhat_old=rhat; end  % Save rhat for the next timestep
  end %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% END OF RK LOOP %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  rs(k,:)=RFFTinv(uhat,N)'; ts(k)=k*dt; % These variables are just used for plotting...
  if (mod(k,PlotInt)==0)
    pause(0.001); PlotXY(x,rs(k,:),k*dt,0,L,-1.5,1.5); title(ts(k))
  end
end
end % function Burgers_KS_CNRKW3_PS
```

calculating derivatives[19], whereas we work in physical space when calculating all nonlinear products; by so doing, the (exact) computation of derivatives costs $\sim N$ flops, and the computation of nonlinear products also costs $\sim N$ flops. When necessary, we use the FFT introduced in §5.4.1 to transform between physical space and Fourier space, which costs $\sim 5N\log_2 N$ real flops. As suggested in §5.7, we **dealias** the Fourier-space representation to eliminate the misrepresentation of high-wavenumber components (created by the nonlinear products) at lower wavenumbers. We arrange the code to compute the minimum number of FFTs possible to get the job done at each RK substep, as FFTs are the most expensive component of the code for large N. Since the linear terms are handled implicitly, and thus have a component that must be swung over to the LHS, the solve at each RK substep is thus done while the system is represented in Fourier space, for which the linear operator is diagonal and thus may be solved directly quite quickly (simply by dividing by the diagonal element). Implementation is given in Algorithm 11.4, and follows a similar structure as seen in the FD case in Algorithm 11.2; extension to 2D is relatively easy, and is considered in Exercise 11.9.

The PDE simulation codes given §11.2.1 and §11.2.2 illustrate the straightforward process of semi-discretization of the PDE using finite-differences (see §8) and spectral methods (see §5) in the spatial direction(s), followed by time marching the resulting system of ODEs (see §10). To achieve adequate accuracy with minimal computational time on a given computer, not only must care must be taken in the choice of methods used, but significant care must also be exercised in the efficient programming of these methods in order to minimize flops while streamlining storage requirements. Extension of such approaches to simulate other parabolic PDE systems in one or more dimensions follows similar reasoning; we thus focus the remainder of §11 mostly on various issues that are not simply a patching together of concepts from previous chapters.

[19] That is, multiply the Fourier coefficients of a vector by ik_{x_p} to determine the Fourier transform of its first derivative with respect to x, multiply the Fourier coefficients by $-k_{x_p}^2$ to determine the Fourier transform of its second derivative with respect to x, etc.

11.2.3 Von Neumann stability analysis

A **von Neumann stability analysis** is simply a modal analysis of a spatially- and temporally-discretized PDE system. To illustrate, consider again the 1D diffusion equation (11.3) with a second-order central FD method to approximate the spatial derivatives, and now discretize in time with the **leapfrog** scheme [introduced in (10.1c) and examined further in Exercise 10.7]; the resulting discretized system is

$$\frac{y_j^{n+1}-y_j^{n-1}}{2\Delta t} = \nu \frac{y_{j+1}^n - 2y_j^n + y_{j-1}^n}{(\Delta x)^2} \quad \Rightarrow \quad y_j^{n+1} = y_j^{n-1} + a\left(y_{j+1}^n - 2y_j^n + y_{j-1}^n\right), \tag{11.69}$$

where $a = 2\nu\Delta t/(\Delta x)^2 > 0$, the superscript denotes the timestep, and the subscript denotes the spatial grid-point. Assuming that $\mathbf{y}^n$ and $\mathbf{y}^{n-1}$ are known[20], (11.69) is trivial to march in time.

To evaluate the stability of such a strategy, consider the finite Fourier series expansion (see §5.4)

$$y_j^n = \sum_{m=-M/2}^{M/2-1} \hat{y}_m^n e^{\mathrm{i}k_m x_j} \quad \text{where} \quad k_m = 2\pi m/L_x, \quad x_j = j\Delta x, \quad \Delta x = L_x/M. \tag{11.70}$$

Inserting (11.70) into the scheme under consideration and grouping the coefficient of $e^{ik_m x_j}$ separately for each m, we can determine a propogation law for each mode of the solution, $\hat{y}_m^{n+1} = \sigma_m \hat{y}_m^n$ (that is, $\hat{y}_m^n = \sigma_m^n \hat{y}_m^0$). As with the scalar model problem considered in §10.2.1, the system is stable iff $|\sigma_m| \le 1$ for all m.

Applying such a **von Neumann stability analysis** to (11.69), we find first that

$$\sum_{m=-M/2}^{M/2-1} e^{\mathrm{i}k_m x_j} \hat{y}_m^0 \sigma_m^{n-1} \left[\sigma_m^2 = 1 + a\left(e^{\mathrm{i}k_m\Delta x} - 2 + e^{-\mathrm{i}k_m\Delta x}\right)\sigma_m\right].$$

Since this holds for any $\hat{\mathbf{y}}^0$, it follows that what is in brackets itself must be true; noting (B.41) and defining $b_m = 2a(1-\cos k_m\Delta x)$, it thus follows that

$$\sigma_m^2 + b_m\sigma_m - 1 = 0 \quad \Rightarrow \quad \sigma_m^{\pm} = \left(-b_m \pm \sqrt{b_m^2+1}\right)/2.$$

For any $M > 1$, it follows for some m that $b_m > 0$, and for other m that $b_m < 0$; for the former, $|\sigma_m^-| > 1$, and for the latter, $|\sigma_m^+| > 1$. Thus, this scheme is *unstable* at any Δt, and therefore essentially useless[21].

11.2.4 Consistency and the Dufort-Frankel scheme

Up to now, we have considered a clear two-step approach to the simulation of PDEs: first discretize the PDE in space, approximating the spatial derivatives appropriately, then march the resulting ODE in time. It is also possible to *mix* the spatial and temporal discretization methods. This is not necessarily a bad idea, as it can improve the stability properties of a PDE time marching scheme, as shown below. However, if one choses to do this, care must be taken to ensure **consistency** of the resulting method (that is, as the discretization is refined in both space and time, that the numerical solution converges to a solution of the PDE in question).

To illustrate, recall first that the leapfrog scheme applied to the 1D diffusion equation, as examined in §11.2.3, was attractively simple, but exhibited unacceptable stability properties. The **Dufort-Frankel** scheme attempts to improve the stability of this method by *mixing the spatial and temporal discretization*, replacing the $2y_j^n$ term on the RHS of the (11.69) with $(y_j^{n+1} + y_j^{n-1})$, resulting in

$$\frac{y_j^{n+1}-y_j^{n-1}}{2\Delta t} = \nu \frac{y_{j+1}^n - (y_j^{n+1}+y_j^{n-1}) + y_{j-1}^n}{(\Delta x)^2} \quad \Rightarrow \quad y_j^{n+1} = \frac{1-a}{1+a} y_j^{n-1} + \frac{a}{1+a}\left[y_{j+1}^n + y_{j-1}^n\right]. \tag{11.71}$$

[20] Recall that the PDE must be marched over the first timestep using some other method in order to initialize this multistep time marching method, as discussed further in the second paragraph of §10.4.2.

[21] For some follow-up comments on this result, see Exercise 11.7.

As in §11.2.3, we now perform a von Neumann stability analysis of (11.71), resulting in

$$\sigma_m^2 - \frac{2a\cos(k_m\Delta x)}{1+a}\sigma_m - \frac{1-a}{1+a} = 0,$$

and thus

$$\sigma_m^{\pm} = \frac{1}{2}\left[\frac{2a\cos k_m\Delta x}{1+a} \pm \sqrt{\frac{4a^2}{(1+a)^2}\cos^2(k_m\Delta x) + 4\frac{1-a^2}{(1+a)^2}}\right] = \frac{1}{1+a}\left[a\cos(k_m\Delta x) \pm \sqrt{1-a^2\sin^2(k_m\Delta x)}\right].$$

If $a^2\sin^2(k_m\Delta x) \le 1$, noting that $a > 0$, taking the absolute value of the above equation gives

$$|\sigma_m^{\pm}| \le \frac{1}{1+a}\left[\left|a\cos(k_m\Delta x)\right| + \left|\sqrt{1-a^2\sin^2(k_m\Delta x)}\right|\right] \le 1,$$

whereas if $a^2\sin^2(k_m\Delta x) > 1$, and thus $a > 1$ and $\sigma_m^{\pm} = \frac{1}{1+a}\left[a\cos(k_m\Delta x) \pm \mathrm{i}\sqrt{a^2\sin^2(k_m\Delta x) - 1}\right]$, we have

$$|\sigma_m^{\pm}| = \frac{1}{1+a}\sqrt{[a\cos(k_m\Delta x)]^2 + [a^2\sin^2(k_m\Delta x) - 1]} = \frac{\sqrt{a^2-1}}{1+a} = \sqrt{\frac{a-1}{a+1}} < 1.$$

Either way, we have $|\sigma_m^{\pm}| \le 1$ for all m and any $\Delta t > 0$, and thus this explicit scheme is *unconditionally stable*, which is quite remarkable considering it is such a simple explicit scheme.

Unfortunately, there is a high price to be paid for this remarkable stability property. To illustrate, assume now that, for some constant c, we set $\Delta t = (1/c)\Delta x \triangleq h$ in the above discretization, and consider the behavior of the discretized system as $h \to 0$. Note that the multidimensional Taylor series expansion of y_{j+k}^{n+m} near y_j^n is

$$y_{j+k}^{n+m} = y_j^n + \left(y_t\right)_j^n m\Delta t + \left(y_x\right)_j^n k\Delta x + \left(y_{tt}\right)_j^n \frac{(m\Delta t)^2}{2} + \left(y_{xx}\right)_j^n \frac{(k\Delta x)^2}{2} + \left(y_{xt}\right)_j^n m\Delta t\, k\Delta x + H.O.T.; \quad (11.72)$$

applying (11.72) to (11.71) reveals, after a minor amount of algebra, that

$$y_j^n + (y_t)_j^n\Delta t + (y_{tt})_j^n\frac{(\Delta t)^2}{2} + H.O.T. = y_j^n + \frac{1-a}{1+a}\left[-(y_t)_j^n\Delta t + (y_{tt})_j^n\frac{(\Delta t)^2}{2}\right] + \frac{2\nu\Delta t}{a+1}(y_{xx})_j^n + H.O.T.,$$

Simplifying, it follows that

$$\frac{2\Delta t}{a+1}\left[(y_t)_j^n + \frac{(\Delta t)^2}{(\Delta x)^2}(y_{tt})_j^n - \nu(y_{xx})_j^n\right] = H.O.T.$$

As $\Delta t = (1/c)\Delta x = h \to 0$, the coefficient $a \to \infty$, and thus the leading term in the above expression reveals that the PDE actually solved by this numerical method in the limit that the space/time grid is refined is

$$\frac{\partial y}{\partial t} + \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2} - \nu\frac{\partial^2 y}{\partial x^2} = 0.$$

Thus, as $h \to 0$, the numerical solution does **not** approach the solution of the 1D diffusion equation (11.3), but rather it approaches the solution of this damped 1D wave equation [see (11.39a)]. This is an example of an **inconsistent** numerical scheme; the fact that it can converge to the wrong answer as the grid is refined can be a ugly surprise unless you've done this analysis. In order to get this scheme to converge to the desired answer as you refine the space/time grid, *you must refine* Δt *faster than* Δx *in such a way that* $(\Delta t)/(\Delta x) \to 0$ *as the grid is refined*. [Alternatively, you may just set the wave speed c to be a large constant and accept the error that this method introduces.] This is the price you pay for obtaining the remarkable property of unconditional stability for such a simple explicit scheme.

11.3 Numerical simulation of hyperbolic PDEs

Parabolic PDE systems are often "forced" at the largest length scales in a problem, with energy scattering between large length scales and small length scales, and back, via nonlinear interactions, and with energy ultimately damping out at small length scales due to diffusion. Such behavior is discussed in §11.1.2.5 and illustrated in the spectrum of the prototypical KS system in Figure 11.4. In such problems, the spatial discretization of the diffusion term generally needs not be extremely accurate at the smallest length scales for the dynamics of the large and intermediate length scales to be represented adequately in a numerical simulation.

In contrast, the numerical simulation of hyperbolic PDE systems (see §11.1.3), as well as the numerical simulation of parabolic PDE systems illustrating wavelike behavior (see §11.1.5), is generally a more delicate endeavor, as the exact solution of such systems exhibits waves of a specific shape that should (if the system is nondispersive, as described in §11.1.3.2) maintain their form as they convect across the physical domain (which is, in turn, is discretized on a finite number of gridpoints). Achieving this behavior generally requires highly accurate numerical discretization in both space and time, as illustrated in the following discussion.

We introduce our study of the simulation of hyperbolic PDEs by turning our attention to the 1D wave equation

$$\frac{\partial^2 q}{\partial t^2} = c^2 \frac{\partial^2 q}{\partial x^2} \tag{11.73}$$

on $x \in [-2,2]$ for $t \geq 0$ with homogeneous Dirichlet BCs and ICs of $q(t=0) = e^{x^2/0.25}$ and $(\partial q/\partial t)_{t=0} = 0$ [that is, with equal parts right-travelling wave and left-travelling wave]. We again approach this problem by first discretizing the PDE in space, then marching the resulting ODE in time.

11.3.1 The value of high-order spatial discretization

We first investigate the use of a second-order central FD method for the spatial discretization, transforming the resulting spatially-discretized second-order ODE to a system of first-order ODEs as suggested in §10.6.1:

$$\frac{d\mathbf{x}}{dt} = \begin{bmatrix} 0 & I \\ T & 0 \end{bmatrix} \mathbf{x} \quad \text{where} \quad \mathbf{x} = \begin{bmatrix} \mathbf{q} \\ \mathbf{v} \end{bmatrix} \quad \Rightarrow \quad \begin{cases} \mathbf{q}^* = \mathbf{q}^n + h(\mathbf{v}^* + \mathbf{v}^n)/2 \\ \mathbf{v}^* = \mathbf{v}^n + hT(\mathbf{q}^* + \mathbf{q}^n)/2 \end{cases} \tag{11.74}$$

where $\mathbf{q}$ and $\mathbf{v}$ are the spatial discretizations of q and $\partial q/\partial t$, and T is a tridiagonal circulant Toeplitz matrix with $\{c^2/(\Delta x)^2, -2c^2/(\Delta x)^2, c^2/(\Delta x)^2\}$ on the extended diagonals. This system is then approximated by the simple CN form written above right[22], and iterated to convergence using the iterative CN method (10.25), after which $\mathbf{q}^{n+1} = \mathbf{q}^*$, $\mathbf{v}^{n+1} = \mathbf{v}^*$; note that an AB2 predictor is implemented in the first iteration (for $\mathbf{q}^*$) at each timestep to accelerate convergence. The resulting code, Algorithm 11.5, is self explanatory.

As seen in Figure 8.4, the second-order central FD method is not very accurate; as a result, as seen in Figure 11.8a, the accuracy of the resulting simulation suffers significantly at coarse spatial resolution (with $N = 32$). However, it is straightforward to incorporate an improved discretization of the second derivative in x, such as the fourth-order Padé expression given in (8.10). The resulting code is given in Algorithm 11.6, and its beneficial effect on the accuracy of the resulting simulation is clearly evident in Figure 11.8b.

[22] A CN method is a natural starting point for hyperbolic systems, because modes in the discretized system corresponding to wavelike behavior in the PDE have pure imaginary eigenvalues, which neither grow nor decay when marched with CN (see Figure 10.2c).

Algorithm 11.5: Simulating the 1D wave equation with iterative CN in time and second-order FD in space.

View

```
function Wave1D_ItCN_FD(L,Tmax,c,N,dt)
% This script simulates the 1D Wave equation with periodic boundary conditions.
% Iterative CN (with an AB2 predictor) is used with a second-order FD method in space.
dx=L/N; IterSteps=2; x=(-N/2:N/2-1)'*dx; t=0; q=exp(-x.^2/0.1); v=0;
PlotXY(x,q,t,-L/2,L/2,-0.2,1.2); vs=v; qs=v; a=dt*c^2/(2*dx^2); b=-dt*c^2/dx^2;
for n=1:Tmax/dt
  for m=1:IterSteps
    if m==1, qs=q+dt*(1.5*v-0.5*qs)/2; else, qs=q+dt*(vs+v)/2; end
    vs=qs+q; vs=v+a*vs([N 1:N-1],1)+b*vs([1:N],1)+a*vs([2:N 1],1);
  end
  t=t+dt; q=qs; qs=v; v=vs; PlotXY(x,q,t,-L/2,L/2,-0.2,1.2);
end
end % function Wave1D_ItCN_FD
```

Algorithm 11.6: Simulating the 1D wave equation with iterative CN in time and fourth-order Padé in space.

View

```
function Wave1D_ItCN_Pade(L,Tmax,c,N,dt)
% This script simulates the 1D Wave equation with periodic boundary conditions.
% Iterative CN (with an AB2 predictor) is used with a fourth-order Pade method in space.
dx=L/N; IterSteps=2; t=0; x=(-N/2:N/2-1)'*dx; q=exp(-x.^2/0.1); v=0;
PlotXY(x,q,t,-L/2,L/2,-0.2,1.2); vs=v; qs=v; a=0.6*dt*c^2/dx^2; b=-1.2*dt*c^2/dx^2;
for n=1:Tmax/dt
  for m=1:IterSteps
    if m==1, qs=q+dt*(1.5*v-0.5*qs)/2; else, qs=q+dt*(vs+v)/2; end
    vs=qs+q; vs=v+ThomasTT(0.1,1,0.1,a*vs([N 1:N-1],1)+b*vs([1:N],1)+a*vs([2:N 1],1),N);
  end
  t=t+dt; q=qs; v=vs; PlotXY(x,q,t,-L/2,L/2,-0.2,1.2);
end
end % function Wave1D_ItCN_Pade
```

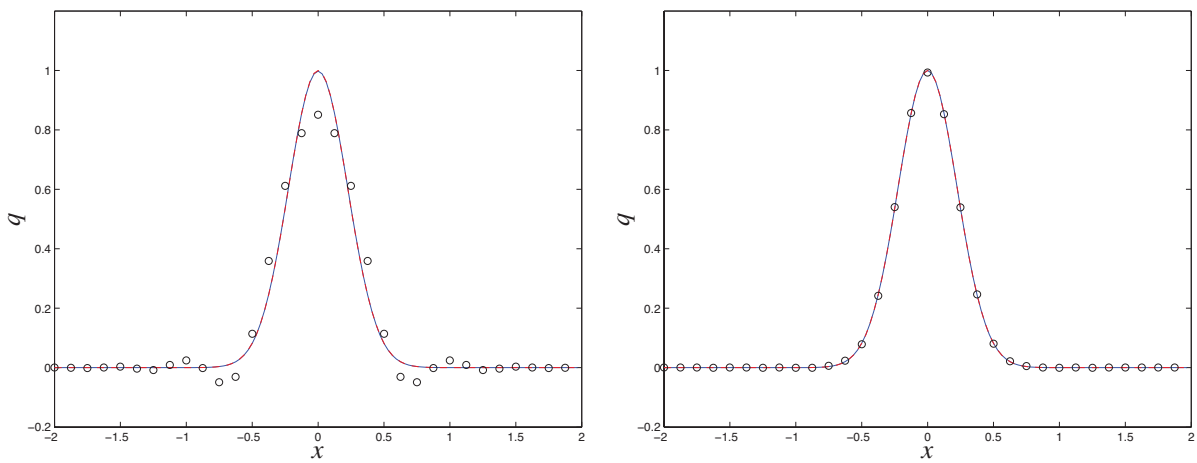


Figure 11.8: Output from (left) Algorithm 11.5 [using a second-order finite difference discretization in space] and (right) Algorithm 11.6 [using a fourth-order Padé discretization in space] with dt=0.01, illustrating (solid) the initial condition, which coincides with the exact solution at $T = 4$; (dashed) state at $T = 4$ using $N = 128$; (circles) state at $T = 4$ using $N = 32$. In both plots, the numerical solutions with $N = 128$ essentially coincide with the exact solution. Note the significantly improved accuracy at coarse spatial resolution (with $N = 32$) when using the Padé expression for the spatial derivative.

Algorithm 11.7: Simulating the 1D wave equation with Newmark in time and fourth-order Padé in space.

View

```
function [q,x]=Wave1D_Newmark_Pade(L,Tmax,c,N,dt)
% This script simulates the 1D Wave equation with periodic boundary conditions.
% Newmark's method is used in time with a fourth-order Pade method in space.
dx=L/N; t=0; x=(-N/2:N/2-1)'*dx; q=exp(-x.^2/0.1); v=0; PlotXY(x,q,t,-L/2,L/2,-0.2,1.2);
dt2=dt^2/2; beta=1/4; gamma=1/2; b1=1-2*beta; b2=2*beta; g1=1-gamma;
dd=-1.2*c^2/dx^2; ee=2.4*c^2/dx^2; aa=.1+beta*dt^2*dd; bb=1+beta*dt^2*ee;
a=ThomasTT(0.1,1,.1,dd*q([N 1:N-1],1)+ee*q([1:N],1)+dd*q([2:N 1],1),N);
for n=1:Tmax/dt
  as=q+dt*v+dt2*b1*a;
  as=ThomasTT(aa,bb,aa,-dd*as([N 1:N-1],1)-ee*as([1:N],1)-dd*as([2:N 1],1),N);
  q=q+dt*v+dt2*(b1*a+b2*as); v=v+dt*(g1*a+gamma*as); a=as;
  t=t+dt; PlotXY(x,q,t,-L/2,L/2,-0.2,1.2);
end
end % function Wave1D_Newmark_Pade
```

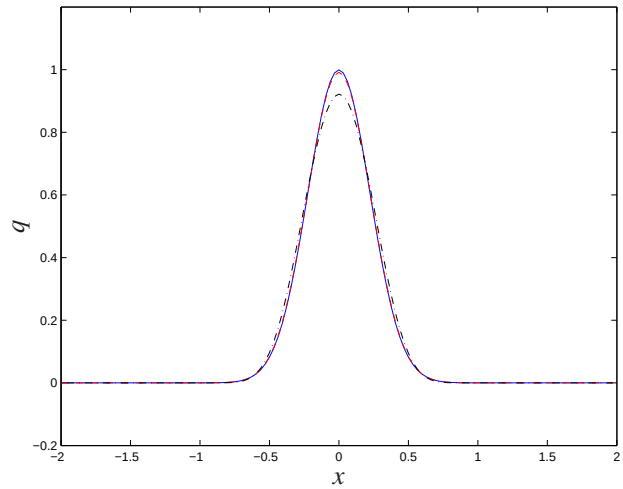


Figure 11.9: Output from Algorithm 11.7, using a Newmark discretization in time, with $N = 128$, illustrating (solid) initial condition; (dashed) state at $T = 4$ using $h = 0.02$; (dot-dashed) state at $T = 4$ using $h = 0.04$. The simulation with $h = 0.02$ exhibits very slight damping, missing the peak of the exact solution at $T = 4$ by 1%, whereas the simulation with $h = 0.04$ exhibits more significant damping, missing the peak by almost 8% [in contrast, Algorithm 11.6 exhibits a high-wavenumber instability (and is thus unusable) at $h = .04$].

11.3.2 The value of a temporal discretization designed for second-order systems

Note that (11.74) is linear and sparse, and thus the iterative approach to applying the CN method to it is perhaps unnecessary. Indeed, it is simply the *ordering* of (11.74) which lacks an easily exploitable sparsity pattern for direct application of the CN method. This issue can be alleviated by **interleaving** the **q** and **v** variables in the **x** vector, leading to a pentadiagonal circulant solve at each timestep if the CN method is applied directly. Though we usually prefer direct time-marching schemes over iterative time-marching schemes when they are affordable, as direct schemes eliminate the somewhat sticky question of how many iterations should be taken at each timestep, the iterative CN method implemented in Algorithm 11.5 is seen to perform well in this case with only a couple of iterations per timestep, and is competitive with the corresponding direct CN method on this problem in terms of efficiency, as pentadiagonal circulant systems are expensive to solve.

The iterative CN method implemented in Algorithms 11.5 and 11.6 exhibits a high-wavenumber instability when a coarse temporal resolution is used (that is, for $h \gtrsim .04$), as the discretization of the RHS in both codes (second-order FD in the former, fourth-order Padé in the latter) is only approximate, and the CN formula itself is solved only approximately at each timestep in the iterative CN approach. To eliminate this high-wavenumber instability while retaining second-order accuracy, albeit at the cost of increasing both the storage required and the number of flops per timestep, an iterative form of the CN[ϕ] method [see (10.46)] could be employed, selecting, say, $\phi = 1/8$ or $1/16$ in order to damp slightly the highest wavenumbers.

An alternative temporal discretization, which has the advantage of being direct, is the Newmark method described in §10.6.2. To implement this method, we first write the spatially-discretized form of (11.73) in the form (10.66b) as follows:

$$\frac{d^2\mathbf{q}}{dt^2} + M^{-1}K\mathbf{q} = 0 \qquad \text{where} \qquad M^{-1}K\mathbf{q} = -c^2\frac{\delta^2\mathbf{q}}{\delta x^2}.$$

As motivated by §11.3.1, we would still like to use the fourth-order Padé expression given in (8.10) for the numerical approximation of the second derivative, which may be achieved by solving $A\,\delta^2\mathbf{q}/\delta x^2 = B\,\mathbf{q}$ where A is a tridiagonal circulant Toeplitz matrix with $\{1, 0.1, 1\}$ on the extended diagonals, and B is a tridiagonal circulant Toeplitz matrix with $\{1.2/(\Delta x)^2, -2.4/(\Delta x)^2, 1.2/(\Delta x)^2\}$ on the extended diagonals. To pose in the form required for implementation of the Newmark method, we thus select $M = A$ and $K = -c^2B$, noting that the initial values of the vectors $\mathbf{q}$ and $\mathbf{v}$ are given by the initial conditions on q and $\partial q/\partial t$ in the original PDE system (11.73), and the initial value of $\mathbf{a}$ is given by solving $A\,\mathbf{a}_0 = B\mathbf{q}_0$.

Following this approach, implementation is again straightforward, as seen in Algorithm 11.7. In essence, the tridiagonal solve due to the Padé spatial discretization [see above] and the linear solve inherent to the Newmark temporal discretization [see (10.69c)] coincide; that is, *there is only one tridiagonal solve at each timestep*. The result is a time marching algorithm which is both *less expensive per timestep*, as it does not require iteration at each timestep as required by the iterative CN approach (nor a pentadiagonal circulant solve as required by the direct CN approach when $\mathbf{q}$ and $\mathbf{v}$ are interleaved in $\mathbf{x}$), and which is also *more stable than iterative CN*, as illustrated in Figure 11.9. Akin to the iterative CN[ϕ] method, if a high wavenumber instability is detected with the Newmark method, then the integration parameter γ may be adjusted to be slightly larger than the nominal values of $\gamma = 1/2$ in order to suppress the instability (see §10.6.2).

11.3.3 The value of pseudospectral methods

We have already shown (in §11.2.2) how pseudospectral methods may be used on (1D) parabolic PDEs with periodic boundary conditions. Recalling the comments at the beginning of §11.3, regarding how highly accurate discretizations are especially desirable when simulating hyperbolic systems, we now illustrate how pseudospectral simulation techniques may be partially extended to a 2D hyperbolic PDE with a peculiar mix of Dirichlet and Neumann BCs on its various components.

As in §11.2.2, we first approximate the spatial derivatives of the PDE (11.41) with BCs (11.49) as a large set of $3M$ ODEs (???), then march the resulting ODEs using the iterative CN.

The spatial domain may be discretized with a square 2D grid with $x_{ij} = i\Delta x$ and $y_{ij} = j\Delta y$ for $i = 0, 1, \ldots, N_x$ and $j = 0, 1, \ldots, N_y$ [resolutions of $N_x = 128$ and $N_y = 256$ are an appropriate starting point]. That is, there are a total of $M = (N_x + 1) \times (N_y + 1)$ gridpoints, with three variables, $\{u, v, h\}$, discretized at each gridpoint.

$b = \tanh(x/5 - 5)$

Assembling these $3M$ variables in the spatially discretized system as the vector $\mathbf{q}$, we first express the spatial discretization of (11.41) on the interior gridpoints, together with the BCs (11.49), as $3M$ coupled ODEs in the generic form $\partial\mathbf{q}/\partial t = \mathbf{f}(\mathbf{q})$.

Recalling the boundary conditions (11.49), there is a natural opportunity to apply 1D and 2D transform techniques (specifically, sine and cosine transforms) to maximize the accuracy of the simulation of this system. Recall from §5.11 the fast techniques, based on the FFT, to perform the forward and inverse sine transform

$$u_j = \sum_{n=1}^{N-1} \hat{u}_n^s \sin(k_n x_j), \qquad \hat{u}_m^s = \frac{2}{N} \sum_{j=1}^{N-1} u_j \sin(k_m x_j),$$

and the forward and inverse cosine transform

$$u_j = \sum_{n=0}^{N} \hat{u}_n^c \cos(k_n x_j), \qquad \hat{u}_m^c = \frac{2}{c_m N} \sum_{j=0}^{N} \frac{u_j}{c_j} \cos(k_m x_j),$$

where

$$k_n = \pi n / L \quad \text{and} \quad c_j \triangleq \begin{cases} 2 & \text{if } j = 0 \text{ or } j = N \\ 1 & \text{otherwise.} \end{cases}$$

Based on the boundary conditions (11.49), which direction(s) would you apply such transform techniques to for u, v, and h, and which transform would you use in each case? What is the advantage of using a transform technique like this? Describe carefully how you would calculate the first two terms on the RHS of (11.41a) following this approach.

ICs that are representative of a rock landing in the pool at $x = 10$, $y = 2$ are given by.

Using a sufficiently small timestep Δt, a stable simulation can be obtained following the approach described in questions (a) and (b), but the result is grossly inaccurate with the resolution used. Thus, refine the grid by a factor of 10 in each spatial direction and repeat the simulation. Determine (again, by trial and error) how much do you have to adjust the timestep such that the simulation is again stable, and describe why (in light of Figure **??** the timestep needs to be changed.

11.3.4 Mixing finite difference and pseudospectral methods

11.3.5 Godunov methods

Consider first the prototypical two-dimensional, linear, hyperbolic PDE in conservation form

$$\frac{\partial p(x,y,t)}{\partial t} = -\frac{\partial u(x,y)\, p(x,y,t)}{\partial x} - \frac{\partial v(x,y)\, p(x,y,t)}{\partial y}. \tag{11.75}$$

Defining the convenient C-type **augmented assignment operators** $a \mathrel{+}= b$ and $c \mathrel{-}= d$ such that $a \leftarrow a + b$ and $c \leftarrow c - d$ respectively, a **Godunov method** applied (11.75) on a uniform Cartesian 2D mesh (with constant Δx and Δy) may be written in the form

$$\frac{p_{ij}^{n+1} - p_{ij}^n}{\Delta t} = -\frac{F_{i+1/2,j}^n - F_{i-1/2,j}^n}{\Delta x} - \frac{G_{i,j+1/2}^n - G_{i,j-1/2}^n}{\Delta y}, \tag{11.76}$$

where the **fluxes** $F_{i-1/2,j}^n$ and $G_{i,j-1/2}^n$ are determined, for all i and j, by first initializing

$$F_{i-1/2,j}^n = u_{i-1/2,j}^+ p_{i-1,j}^n + u_{i-1/2,j}^- p_{i,j}^n, \qquad G_{i,j-1/2}^n = v_{i,j-1/2}^+ p_{i,j-1}^n + v_{i,j-1/2}^- p_{i,j}^n,$$

where $u^+ = \max(u,0)$, $u^- = \min(u,0)$, etc, then applying the **corner transport upwind** (CTU) terms by updating, for all i and j,

$$F^n_{i-1/2,j-1} \mathrel{-}= \Delta t \frac{u^-_{i-1/2,j-1} v^-_{i,j-1/2}}{2} \frac{\Delta p^n_{i,j-1/2}}{\Delta y}, \qquad G^n_{i-1,j-1/2} \mathrel{-}= \Delta t \frac{v^-_{i-1,j-1/2} u^-_{i-1/2,j}}{2} \frac{\Delta p^n_{i-1/2,j}}{\Delta x},$$

$$F^n_{i+1/2,j-1} \mathrel{-}= \Delta t \frac{u^+_{i+1/2,j-1} v^-_{i,j-1/2}}{2} \frac{\Delta p^n_{i,j-1/2}}{\Delta y}, \qquad G^n_{i-1,j+1/2} \mathrel{-}= \Delta t \frac{v^+_{i-1,j+1/2} u^-_{i-1/2,j}}{2} \frac{\Delta p^n_{i-1/2,j}}{\Delta x},$$

$$F^n_{i-1/2,j} \mathrel{-}= \Delta t \frac{u^-_{i-1/2,j} v^+_{i,j-1/2}}{2} \frac{\Delta p^n_{i,j-1/2}}{\Delta y}, \qquad G^n_{i,j-1/2} \mathrel{-}= \Delta t \frac{v^-_{i,j-1/2} u^+_{i-1/2,j}}{2} \frac{\Delta p^n_{i-1/2,j}}{\Delta x},$$

$$F^n_{i+1/2,j} \mathrel{-}= \Delta t \frac{u^+_{i+1/2,j} v^+_{i,j-1/2}}{2} \frac{\Delta p^n_{i,j-1/2}}{\Delta y}, \qquad G^n_{i,j+1/2} \mathrel{-}= \Delta t \frac{v^+_{i,j+1/2} u^+_{i-1/2,j}}{2} \frac{\Delta p^n_{i-1/2,j}}{\Delta x},$$

where $\Delta p^n_{i-1/2,j} = p^n_{ij} - p^n_{i-1,j}$, $\Delta p^n_{i,j-1/2} = p^n_{ij} - p^n_{i,j-1}$, and finally applying the **high-resolution correction terms** by updating, for all i and j,

$$F^n_{i-1/2,j} \mathrel{+}= \Delta t \frac{|u_{i-1/2,j}|}{2} \left(\frac{\Delta x}{\Delta t} - |u_{i-1/2,j}| \right) \frac{\Delta p^n_{i-1/2,j}}{\Delta x} \phi(\theta^n_{i-1/2,j}),$$

$$G^n_{i,j-1/2} \mathrel{+}= \Delta t \frac{|v_{i,j-1/2}|}{2} \left(\frac{\Delta y}{\Delta t} - |v_{i,j-1/2}| \right) \frac{\Delta p^n_{i,j-1/2}}{\Delta y} \phi(\theta^n_{i,j-1/2}),$$

where

$$\theta^n_{i-1/2,j} = \begin{cases} \Delta p^n_{i-3/2,j} / \Delta p^n_{i-1/2,j} & \text{if } u_{i-1/2,j} \geq 0, \\ \Delta p^n_{i+1/2,j} / \Delta p^n_{i-1/2,j} & \text{if } u_{i-1/2,j} < 0, \end{cases} \qquad \theta^n_{i,j-1/2} = \begin{cases} \Delta p^n_{i,j-3/2} / \Delta p^n_{i,j-1/2} & \text{if } v_{i,j-1/2} \geq 0, \\ \Delta p^n_{i,j+1/2} / \Delta p^n_{i,j-1/2} & \text{if } v_{i,j-1/2} < 0, \end{cases}$$

and the **flux limiter** function $\phi(\theta) \in [0,2]$ is selected as one of several possible choices, including the **monotonized central-difference** (*MC*) limiter and the **van Leer** limiter:

$$\begin{aligned} &\textbf{MC}: && \phi(\theta) = \max\{0, \min[(1+\theta)/2, 2, 2\theta]\}, \\ &\textbf{van Leer}: && \phi(\theta) = (\theta + |\theta|)/(1+|\theta|). \end{aligned}$$

Note that exact conservation of the discrete approximation of the integral of p over phase space, as implied by the continuous formulation in (11.75), follows immediately from (11.76).

Flux limiter functions, such as the ones described above, are designed to give the scheme a first-order spatial behavior with an appropriate amount of numerical dissipation in the region of large gradients of P, thereby providing a total variation diminishing (TVD) solution (that is, preventing spurious oscillations with new local minima and maxima). A simple numerical test of the algorithm described above which illustrates this TVD property, with $u(x,y)$ and $v(x,y)$ defined to give simple solid body rotation about the origin, is given in Figure 11.10 and discussed further in §**??** and §**??**.

In regions characterized by smooth variation of p, $\theta \approx 1$ and $\phi(\theta) \approx 1$, and the algorithm described above is amenable to straightforward numerical analysis. For simplicity, consider the 1D case with u constant:

$$\frac{\partial p}{\partial t} = -u \frac{\partial p}{\partial x}; \tag{11.77}$$

in this case, the Godunov method described above reduces to

$$\frac{p^{n+1}_i - p^n_i}{\Delta t} = -\frac{F^n_{i+1/2} - F^n_{i-1/2}}{\Delta x} \quad \text{with} \quad F^n_{i-1/2} = \frac{u}{2}(p^n_i + p^n_{i-1}) - \frac{u^2}{2}\frac{\Delta t}{\Delta x}(p^n_i - p^n_{i-1}),$$

and thus

$$\frac{p_i^{n+1}-p_i^n}{\Delta t}=-u\frac{(p_{i+1}^n-p_{i-1}^n)}{2\,\Delta x}+\frac{u^2\,\Delta t}{2}\frac{(p_{i+1}^n-2p_i^n+p_{i-1}^n)}{(\Delta x)^2}.$$

Now applying to this equation the multidimensional Taylor series expansion,

$$y_{i+k}^{n+m}=y_i^n+m\Delta t\left(\frac{\partial y}{\partial t}\right)_i^n+k\Delta x\left(\frac{\partial y}{\partial x}\right)_i^n+\frac{(m\Delta t)^2}{2}\left(\frac{\partial^2 y}{\partial t^2}\right)_i^n+\frac{(k\Delta x)^2}{2}\left(\frac{\partial^2 y}{\partial x^2}\right)_i^n+m\Delta t\,k\Delta x\left(\frac{\partial^2 y}{\partial x\partial t}\right)_i^n+\ldots,$$

and rearranging appropriately, gives

$$(p_t)_i^n=-u(p_x)_i^n-\frac{\Delta t}{2}(p_{tt})_i^n+\frac{u^2\Delta t}{2}(p_{xx})_i^n+\mathscr{O}((\Delta t)^2,(\Delta x)^2,\Delta x\Delta t).$$

Differentiating (11.77) with respect to t and inserting (11.77) into the RHS of the result, it is seen that the second and third terms on the RHS of the above expression cancel. Thus, in regions of smooth variation of p, the proposed scheme is *second-order accurate in both space and time*.[23] A similar analysis follows for problems in higher dimensions.

[23] Meaning that the error is bounded by a term proportional to $(\Delta x)^2$ in space and $(\Delta t)^2$ in time, giving convergence of $\mathscr{O}(1/N^2)$.

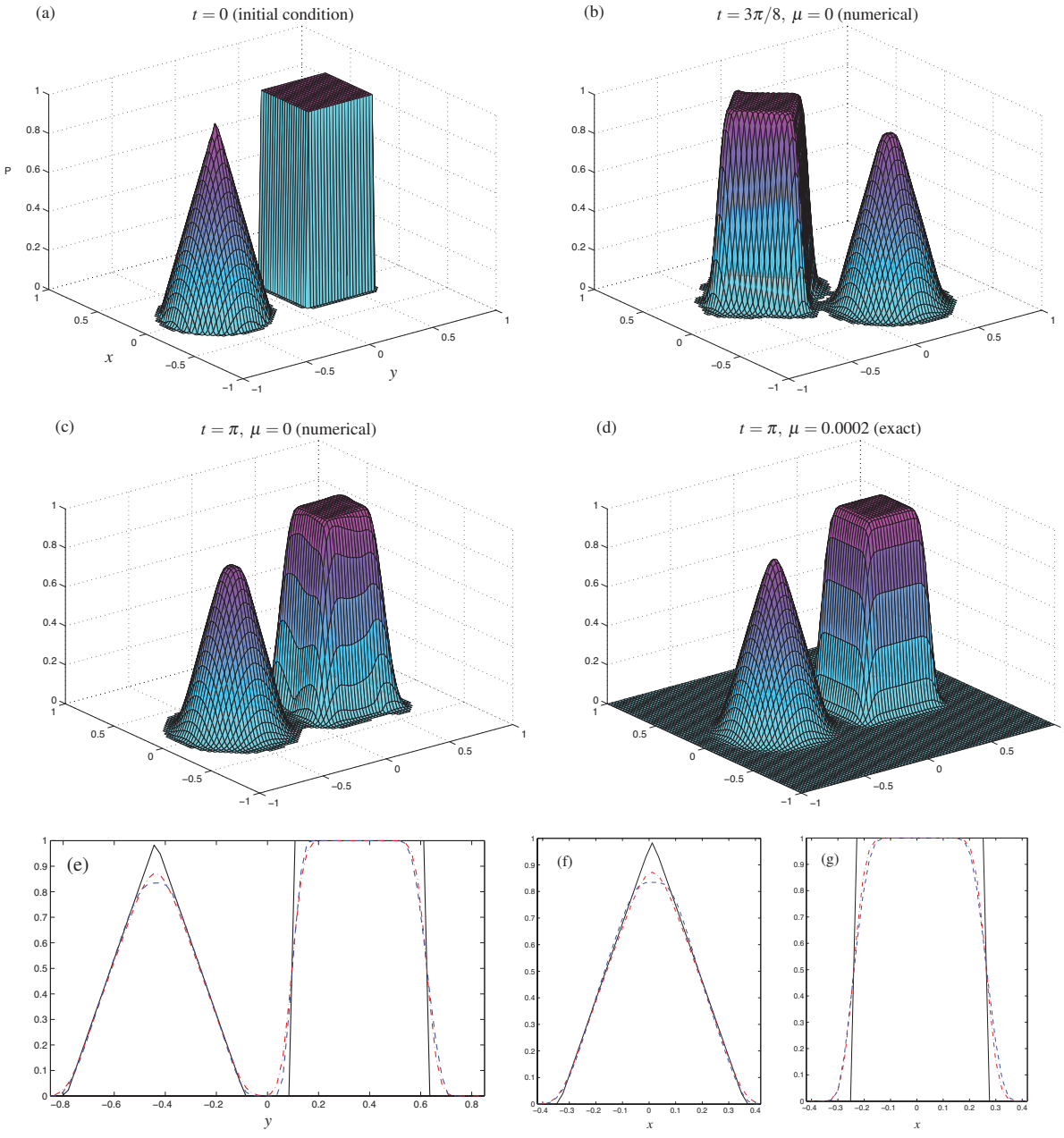


Figure 11.10: Computation of (11.75) with $u(x,y) = 2y$ and $v(x,y) = -2x$ at (a) $t = 0$, (b) $t = 3\pi/8$, and (c) $t = \pi$, using the Godunov method of §11.3.5 with an MC flux limiter, $\Delta x = 0.024$, $\Delta t = 0.012$, and tracking numerically only those cells with $p > 0.001$ and their immediate neighbor cells, as indicated by the location of the colored contours. The exact solution at $t = \pi$ for $\mu = 0$ is precisely the initial condition at $t = 0$. Also shown (d) is the exact solution at $t = \pi$ with a diffusion term $\mu \Delta p$ added to the RHS of (11.75), taking $\mu = 0.0002$; the leading-order error in c is thus seen to be a bit of targeted diffusion of p. The surface plots in (a), (c), and (d) are compared in cross section in: (e) the plane $x = 0$, (f) the plane $y = -0.45$, and (g) the plane $y = 0.35$, with solid denoting the initial condition, dashed denoting the numerical solution at $t = \pi$ using $\mu = 0$ and $\Delta x = 0.024$, and dot-dashed denoting the exact solution at $t = \pi$ for $\mu = 0.0002$.

11.4 Numerical solution of elliptic PDEs

11.4.1 Multigrid revisited: a Rosetta stone in Matlab, Fortran, and C

None of the simple splitting techniques presented in §3.2.1 are particularly efficient at solving large systems $A\mathbf{x} = \mathbf{b}$ derived from the numerical discretization of elliptic PDEs. As seen in Figure 3.3, the red/black Gauss-Seidel method is best thought of as an unbiased and numerically efficient *smoother* of the defect $\mathbf{d}$ at the highest spatial frequencies represented on the grid. The **multigrid** (a.k.a. **geometric multigrid**) method presented in this section, which is designed specifically for application to discretizations of elliptic PDEs, leverages this smoothing behavior in a clever way. To be clear, we will illustate the multigrid method here as it applies to the particular linear problem described in (3.11) with $n_x = n_y \triangleq n = 2^p$; once the basic method is understood, a multitude of generalizations are possible (see Trottenberg, Oosterlee, & Schüller 2001).

To begin, one or two iterations of the red/black Gauss-Seidel smoother are first applied to reduce the high-frequency components of the defect on a fine-grid discretization of (3.11b), with the result written here (for brevity) as $A_n \mathbf{x}_n^{(k)} \approx \mathbf{b}_n$; we denote the defect on this fine grid after these initial smoothing steps by $\mathbf{d}_n^{(k)} = A\mathbf{x}_n^{(k)} - \mathbf{b}_n$. Our goal now is to find efficiently an even better approximation $\mathbf{x}_n^{(k+1)}$ such that

$$A_n \mathbf{x}_n^{(k+1)} - \mathbf{b}_n \approx 0 \quad \Rightarrow \quad A_n(\mathbf{x}_n^{(k)} + \mathbf{v}_n^{(k)}) - \mathbf{b}_n \approx 0 \quad \Rightarrow \quad A_n \mathbf{v}_n^{(k)} \approx -\mathbf{d}_n^{(k)}; \tag{11.78}$$

however, due to the high dimension of $\mathbf{x}_n$ [with $\sim n^2$ unknowns] and the lack of an exploitable sparsity structure in A_n, the latter equation is too difficult to solve directly. The standard splitting approach to this problem, as shown in (3.8b), is thus, instead of solving (11.78), to solve the equation $M_n \mathbf{v}_n^{(k)} = -\mathbf{d}_n^{(k)}$ (where M_n contains some but not all of the information in A_n), and then to iterate in k until convergence; for example, as described previously, the Gauss-Seidel approach takes $M_n = L_n + D_n$, where $A_n = L_n + D_n + U_n$.

The multigrid approach is more clever. Following this approach, the desired relation $A_n \mathbf{v}_n^{(k)} = -\mathbf{d}_n^{(k)}$ is **restricted** (that is, approximated) on a grid which has been *coarsened* in all coördinate directions by a factor of two; dropping the k superscripts for notational clarity, we denote the restriction of the fine-grid defect onto the coarse grid as $\mathbf{d}_{n/2}$, and the matrix in the coarse-grid discretization of the governing equation (3.11) as $A_{n/2} = L_{n/2} + D_{n/2} + U_{n/2}$. Note in particular that we do *not* need to restrict $\mathbf{x}_n$ onto the coarse grid.

We can now follow one of two approaches. If $n/2$ is sufficiently small that $A_{n/2}\mathbf{v}_{n/2} = -\mathbf{d}_{n/2}$ may be solved directly, we go ahead and solve (exactly) for this correction $\mathbf{v}_{n/2}$ on the coarse grid, then **prolongate** (that is, interpolate) this correction back to the fine grid, and apply it to update $\mathbf{x}_n$ on the fine grid via (3.8c). This restriction/prolongation process introduces high-frequency errors on the fine grid; we thus apply one or two additional iterations of the red/black Gauss-Seidel smoother to $\mathbf{x}_n$, and repeat the entire process.

If, however, $n/2$ is still so large that the coarse-grid equation

$$A_{n/2}\mathbf{v}_{n/2} = -\mathbf{d}_{n/2} \tag{11.79}$$

can not easily be solved directly, we may instead apply one or two iterations of the red/black Gauss-Seidel smoother to $\mathbf{v}_{n/2}$ (using $M_{n/2} = L_{n/2} + D_{n/2}$), compute the defect of (11.79) after these smoothing steps, and then restrict this new defect to an even *coarser* grid with $n/4$ gridpoints in each direction, in effect forming an even coarser approximation of the original problem. This process of restrict/smooth/restrict/smooth is repeated until the $A\mathbf{v} = -\mathbf{d}$ problem is reduced to a small enough size that the correction $\mathbf{v}$ may easily be solved directly. The resulting correction is then prolongated to the previous level, applied to update the previous estimate of $\mathbf{v}$ at this level, and the result smoothed with one or two further iterations of the red/black Gauss-Seidel smoother to diminish the high-frequency errors at this level which might have been introduced by the restriction/prolongation process. The process of prolongate/smooth/prolongate/smooth is repeated all the way back out to the original level, and the entire cycle repeated until convergence. Note that straightforward variations on the basic **V-cycle** described here are common in the literature, including the **W-cycle** and the

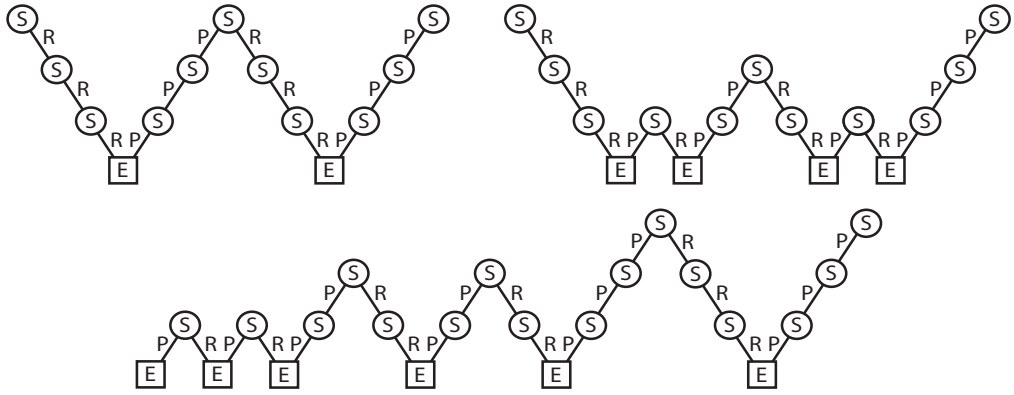


Figure 11.11: Typical restriction (R) / prolongation (P) schedules of the multigrid algorithm; note that S denotes smoothing (via one or two iterations of red/black Gauss-Seidel) and E denotes exact solution. (top-left) **V-cycles**, (top-right) a **W-cycle**, and (bottom) a **full multigrid cycle**. Note that the V and W cycles begin on the finest grid, whereas the full multigrid cycle begins on the coarsest grid.

full multigrid cycle illustrated in Figure 11.11; we note here that the simple V-cycle, as implemented in Algorithm 11.8, appears to be competitive with these more complex cycling strategies on most problems. Note also that, to accelerate convergence, the prolongation update may be applied with the SOR formula (3.8c'), for an appropriate relaxation parameter $\omega \in (0,2)$, rather than the standard formula (3.8c).

Convergence following this multigrid approach is usually quite rapid, because *the components of the defect on the fine grid which the red/black Gauss-Seidel smoother fails to diminish effectively correspond to the highest-frequency components of the defect on the coarser grids, which are effectively diminished by the red/black Gauss-Seidel smoother applied at these coarser levels*.

There is significant flexibility in the selection of the restriction (that is, fine-to-coarse) operator that is applied to the defect **d**. Denoting the vector of defect values on fine grid as **d** and the corresponding vector of restricted values on the coarse grid as $\overline{\mathbf{d}}$, the following three choices are common:

$$\begin{aligned}
&\textbf{straight injection} && \overline{d}_{i,j} = d_{2i,2j}, \\
&\textbf{half weighting} && \overline{d}_{i,j} = d_{2i,2j}/2 + (d_{2i-1,2j} + d_{2i+1,2j} + d_{2i,2j-1} + d_{2i,2j+1})/8, \\
&\textbf{full weighting} && \overline{d}_{i,j} = d_{2i,2j}/4 + (d_{2i-1,2j} + d_{2i+1,2j} + d_{2i,2j-1} + d_{2i,2j+1})/8 \\
& && \qquad + (d_{2i-1,2j-1} + d_{2i+1,2j-1} + d_{2i-1,2j+1} + d_{2i+1,2j+1})/16,
\end{aligned}$$

which may be summarized via the handy **restriction stencils**:

$$\mathcal{R}_{\text{SI}} = \begin{Bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{Bmatrix}, \quad \mathcal{R}_{\text{HW}} = \begin{Bmatrix} 0 & 1/8 & 0 \\ 1/8 & 1/2 & 1/8 \\ 0 & 1/8 & 0 \end{Bmatrix}, \quad \mathcal{R}_{\text{FW}} = \begin{Bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{Bmatrix}. \tag{11.80}$$

These restriction stencils may be thought of as simple, spatially compact, discrete approximations of a Gaussian "bump" of unit volume which may be applied to smooth **d** while restricting it onto the coarser grid. Note that the half weighting and full weighting strategies weight the red points and the black points equally, and as a result tend to work particularly well when coupled with the red/black Gauss-Seidel smoother.

The prolongation (that is, coarse-to-fine) operator that is usually applied to the correction **v** in this process is simple **bilinear interpolation** (discussion further in §7.3). Denoting the vector of values on the coarse grid

as $\overline{\mathbf{v}}$ and the corresponding vector of prolongated values on the fine grid as $\mathbf{v}$, it may be written

$$v_{i,j} = \begin{cases} \overline{v}_{i/2,j/2} & i = \text{even}, \quad j = \text{even}, \\ (\overline{v}_{i/2,(j-1)/2} + \overline{v}_{i/2,(j+1)/2})/2 & i = \text{even}, \quad j = \text{odd}, \\ (\overline{v}_{(i-1)/2,j/2} + \overline{v}_{(i+1)/2,j/2})/2 & i = \text{odd}, \quad j = \text{even}, \\ (\overline{v}_{(i-1)/2,(j-1)/2} + \overline{v}_{(i-1)/2,(j+1)/2} + \overline{v}_{(i+1)/2,(j-1)/2} + \overline{v}_{(i+1)/2,(j+1)/2})/4 & i = \text{odd}, \quad j = \text{odd}, \end{cases}$$

which may be summarized via the handy **prolongation stencil**:

$$\mathscr{P}_{\text{BI}} = \begin{Bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{Bmatrix}. \tag{11.81}$$

The original **Rosetta stone** presented a tale describing the benefits bestowed upon Egypt by the 13-year-old pharaoh Ptolemy V Epiphanes at the time of his coronation in 196 B.C. in three languages: hieroglyphic, demotic, and Greek. With such an artifact, Jean-Francois Champollion was able to decifer the hieroglyphic alphabet in 1822 A.D., an art that had been lost for nearly 2000 years. In Algorithms 11.8, 11.9, and 11.10, we present something similar. Rather than presenting a tale from ancient Egypt, we implement the multigrid algorithm described above[24,25]. Perhaps just as significant, however, is the fact that we present equivalent implementations in three languages: Matlab, Fortran, and C. A useful consequence of this to the reader is thus analogous to that of the original Rosetta stone to Jean-Francois Champollion, as it demonstrates, by example, how the following features of an advanced numerical algorithm can be implemented in a computer code in Matlab, Fortran, and C syntax:

- **parameter definitions**,
- **global variables**[26],
- **comments** (both within the code and messages displayed on the screen),
- both basic and advanced **floading-point operations**, including **mod**, **max**, and **abs**,
- **for** loops with **break** statements
- **if/then/else** statements,
- **case** statements,
- **subroutines** and **functions**, with the passing of information between them,
- **allocation**, **deallocation**, and the efficient referencing of arrays,
- **derived types** (useful for storing and passing grid parameters, etc.),
- **arrays of arrays**, a.k.a. **cell arrays** (for nonrectangular data structures)
- **random number** generation,
- **recursion**, and
- program execution **timing**.

The syntax for each of these features is slightly different from one language to the next, but the basic structure of the resulting code is essentially the same. Using this presentation as an example, it should prove straightforward to translate other codes from Matlab syntax into Fortran or C syntax. On many computers, the compiled Fortran and C executables are significantly faster than their Matlab counterpart.

There are some subtle tricks applied in the sophisticated numerical implementations that follow, the most significant of which are pointed out explicitly in the comments within the codes. These codes implement

[24] And, rather than inscribing it on a 1m by 70cm by 30cm slab of black Basalt, we provides it for you on a few humble pieces of white paper—infinitely more practical, albeit much less dramatic...

[25] Special thanks to Prof. Paolo Luchini for the original CPL code after which Algorithms 11.8, 11.9, and 11.10 were modeled (see Luchini & D'Alascio 1994), and to Anish Karandikar and Paul Belitz for assistance with the implementations in Fortran and C.

[26] Warning: the use of global variables tends to compartmentalise an algorithm less clearly, and is thus generally discouraged. However, in certain high performance computing applications such as the present, it can prevent the repeated allocation and deallocation of large matrices, thereby accelerating the code significantly. The reader is advised to use such global variables sparingly.

the multigrid algorithm described above with red/black Gauss-Seidel smoothing applied to the 2D Poisson equation (3.11a) with second-order finite differencing ($\mathscr{A}_{\text{SOFD}}$) on a uniform grid (with $n_x = 2^p$ and $n_y = 2^q$ for integers p and q with $p \geq q$) with homogeneous Dirichlet, homogeneous Neumann, or periodic boundary conditions in the x and y directions. Half weighting ($\mathscr{R}_{\text{HW}}$) is used for the restriction and bilinear interpolation ($\mathscr{P}_{\text{BI}}$) is used for the prolongation, with the (3.8c) for the prolongation update (no overrelaxation).

Algorithm 11.8: Multigrid algorithm for solving the 2D Poisson equation (Matlab version).

View

```
function PoissonMG2DTest                                % Numerical Renaissance Codebase 1.0
% A 2D Poisson solver on a uniform mesh using multigrid with red/black smoothing.  The RHS
% vector is assumed to be scaled such that the discretized Laplace operator has a 1 on the
% diagonal element.
clear; global XBC YBC N1 N2 N3 xo yo d v g verbose
% ------------------------------------------ USER INPUT ------------------------------------------
% XBC=1 for hom. Dirichlet (0:NX), =2 for periodic (-1:NX), =3 for hom. Neumann (-1:NX+1).
% Same for YBC. NX, NY must be powers of two with NX>=NY. N1,N2,N3 set how much smoothing
NX=32; NY=32; XBC=2; YBC=3; N1=2; N2=2; N3=2;                % is applied at various steps.
% ------------------------------------------ END OF USER INPUT ------------------------------------
fprintf('BCs:%2g,%2g. Smoothing:%2g,%2g,%2g.\nGrids:\n',XBC,YBC,N1,N2,N3);
for verbose=1:2
  PoissonMG2DInit(NX,NY);                                % The RHS vector b is stored in
  d{1}(3:g{1}.xm-2,3:g{1}.ym-2)=rand(g{1}.xm-4,g{1}.ym-4); % the initial value of -d [see
  i=sum(sum(d{1}(:,:)))/((g{1}.xm-4)*(g{1}.ym-4));       % (3.8a)], here taken as a zero
  d{1}(3:g{1}.xm-2,3:g{1}.ym-2)=d{1}(3:g{1}.xm-2,3:g{1}.ym-2)-i; % mean random number.
  PoissonMG2D; pause;
end
end % function PoissonMG2DTest
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function PoissonMG2DInit(NX,NY)
% This initialization routine allocates several global arrays to avoid the repeated
% memory allocation/deallocation otherwise caused by recursion.  Note that d and v are
% defined as cell arrays, which are of different size at each level l.
global XBC YBC xo yo nlev d v g
switch XBC case 1, xo=1; case 2, xo=2; case 3, xo=2; end
switch YBC case 1, yo=1; case 2, yo=2; case 3, yo=2; end
nlev=log2(NY);
for l=1:nlev
   g{l}.nx = NX/(2^(l-1)); g{l}.ny = NY/(2^(l-1)); % The g data structure contains
   g{l}.xm=g{l}.nx+XBC; g{l}.ym=g{l}.ny+YBC;       % information about the grids.
   v{l} = zeros(g{l}.xm,g{l}.ym);   d{l}=v{l};     % d=defect, v=correction.
   fprintf('%5g%5g%5g\n',l,g{l}.nx,g{l}.ny)
end
end % function PoissonMG2DInit
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function PoissonMG2D
global N1 verbose
e=MaxDefect(0); fprintf('Iter=0, max defect=%0.3e\n',e);
for i=1:N1; Smooth(1); end  % APPLY SMOOTHING STEPS BEFORE STARTING MULTIGRID (N1 times)
tic; for iter=1:10          % PERFORM UP TO 10 MULTIGRID CYCLES.
   o=e;  Multigrid(1); e = MaxDefect(iter);
   if verbose>0, fprintf('Iter=%0.6g, max defect=%0.3e, factor=%0.4f\n',iter,e,o/e); end
   if e<1E-13,   fprintf('Converged\n'), break, end
end; t=toc;
fprintf('-> Total time: %0.3g sec; Time/iteration: %0.3g sec\n',t,t/iter);
end % function PoissonMG2D
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Multigrid(l)
% The main recursive function for the multigrid algorithm. It calls the smoothing function,
% it performs the restriction and prolongation, and calls itself on the coarser grid.
global N2 N3 xo yo nlev d v g verbose
for i=1:N2; Smooth(l); end     % APPLY SMOOTHING STEPS BEFORE COARSENING (N2 times)
% Now COMPUTE THE DEFECT and RESTRICT it to the coarser grid in a single step.
% TRICK #1 we calculate the defect ONLY on the red points here, as the defect on the
% neighboring black points is zero coming out of the previous call to Smooth.
% TRICK #2: We skip a factor of /2 during the restriction here. We also skip factors of
% *4 during the smoothing and /2 during the prolongation, so the skipped factors cancel.
d{l+1}=zeros(g{l+1}.xm,g{l+1}.ym);
```

```
for ic=2:g{l+1}.xm-1; i=2*(ic-xo)+xo; for jc=2:g{l+1}.ym-1; j=2*(jc-yo)+yo;
  d{l+1}(ic,jc)=(v{l}(i+1,j)+v{l}(i-1,j)+v{l}(i,j+1)+v{l}(i,j-1))/4 -v{l}(i,j) +d{l}(i,j);
end; end;
v{l+1}=d{l+1};      % TRICK #3: this is a better initial guess for v{l+1} than v{l+1}=0.
EnforceBCs(l+1)
% Now CONTINUE DOWN TO COARSER GRID, or SOLVE THE COARSEST SYSTEM (via 20 smooth steps).
% Use same smoother on coarse grid; ie, we SKIP a *4 [i.e., *(delta x)^2] factor (TRICK 2).
if (l<nlev-1); Multigrid(l+1); else; for i=1:20; Smooth(nlev); end; end
% Now perform the PROLONGATION.  TRICK #4: Update black interior points only, as next call
% to Smooth recalculates all red points from scratch, so do not bother updating them here.
% We SKIP a /2 interpolation factor here (Trick 2).
for ic=2:g{l+1}.xm;  i=2*(ic-xo)+xo;  for jc=2:g{l+1}.ym;  j=2*(jc-yo)+yo;
    if j<g{l}.ym, v{l}(i-1,j) = v{l}(i-1,j)+(v{l+1}(ic-1,jc)+v{l+1}(ic,jc)); end
    if i<g{l}.xm, v{l}(i,j-1) = v{l}(i,j-1)+(v{l+1}(ic,jc-1)+v{l+1}(ic,jc)); end
end; end
EnforceBCs(l)
for i=1:N3; Smooth(l); end    % APPLY SMOOTHING STEPS AFTER COARSENING (N3 times)
end % function Multigrid
%%%
```

Algorithm 11.9: Multigrid algorithm for solving the 2D Poisson equation (Fortran90 version).

View

```
program PoissonMG2DTest
! A 2D Poisson solver on a uniform mesh using multigrid with red/black smoothing.  The RHS
! vector is assumed to be scaled such that the discretized Laplace operator has a 1 on the
! diagonal element.
! May be compiled on a unix machine with:  g95 PoissonMG2DTest.f90 -o PoissonMG2DTest
! ------------------------------------------ USER INPUT ------------------------------------------
! XBC=1 for hom. Dirichlet (0:NX), =2 for periodic (-1:NX), =3 for hom. Neumann (-1:NX+1).
! Same for YBC. NX, NY must be powers of two with NX>=NY. N1,N2,N3 set how much smoothing
integer :: NX=1024, NY=1024, XBC=1, YBC=3, N1=2, N2=2, N3=2 ! is applied at various steps.
! ---------------------------------------- END OF USER INPUT ----------------------------------------
integer :: xo, yo, nlev                                                    ! global variables
type :: arrays; sequence; real*8, pointer :: d(:,:); end type arrays
type grid; sequence; integer :: nx, ny, xm, ym; end type grid
type(arrays):: d(0:16), v(0:16);  type(grid) :: g(0:16)

print *, 'BCs: ',XBC,',',YBC,'. Smoothing: ',N1,',',N2,',',N3,'.  Grids:'
call PoissonMG2DInit
z=0.; do j=3,g(0)%ym-2; do i=3,g(0)%xm-2          ! The RHS vector b is stored in the
    call random_number(h); d(0)%d(i,j)=h; z=z+h   ! initial value of -d [see (3.8a)], here
end do; end do                                    ! taken as a zero-mean random number.
d(0)%d(3:g(0)%xm-2,3:g(0)%ym-2)=d(0)%d(3:g(0)%xm-2,3:g(0)%ym-2)-z/(g(0)%xm-4.)/(g(0)%ym-4.)
call PoissonMG2D
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
contains
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine PoissonMG2DInit
! This initialization routine allocates several global arrays to avoid the repeated
! memory allocation/deallocation otherwise caused by recursion.  Note that d and v are
! defined as arrays of arrays, which are of different size at each level l.
select case(XBC); case(1); xo=1; case(2); xo=2; case(3); xo=2; end select
select case(YBC); case(1); yo=1; case(2); yo=2; case(3); yo=2; end select
nlev=int(log10(real(NY))/log10(2.0))-1; call random_seed; sum = 0.0
do l=0,nlev
   g(l)%nx = NX/(2**l); g(l)%ny = NY/(2**l);   ! The g data structure contains
   g(l)%xm=g(l)%nx+XBC; g(l)%ym=g(l)%ny+YBC;   ! information about the grids.
   allocate(v(l)%d(g(l)%xm,g(l)%ym)); allocate(d(l)%d(g(l)%xm,g(l)%ym));
   v(l)%d=0.0; d(l)%d=0.0;   print *,l,g(l)%nx,g(l)%ny    ! d=defect, v=correction.
end do
end subroutine PoissonMG2DInit
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine PoissonMG2D
integer :: ta(8)
e=MaxDefect(0); print *,'Iter=0, max defect=',e
do i=1,N1; call Smooth(0); end do ! APPLY SMOOTHING BEFORE STARTING MULTIGRID (N1 times)
call date_and_time(values=ta); t = ta(5)*3600 + ta(6)*60 + ta(7) + 0.001*ta(8)
main: do iter=1,10                  ! PERFORM UP TO 10 MULTIGRID CYCLES.
   o=e; call Multigrid(0); e = MaxDefect(0);
   print *,'Iter=',iter,', max defect=',e,'factor=',o/e
   if (e<1E-13) then; print *,'Converged'; exit main; end if
end do main
call date_and_time(values=ta); t = ta(5)*3600 + ta(6)*60 + ta(7) + 0.001*ta(8) - t
print *,'-> Total time: ',t,' sec; Time/iteration: ',t/iter,' sec'
do l=0,nlev; deallocate(v(l)%d); deallocate(d(l)%d); end do
end subroutine PoissonMG2D
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
recursive subroutine Multigrid(l)
! The main recursive function of the multigrid algorithm. It calls the smoothing function,
! it performs the restriction and prolongation, and calls itself on the coarser grid.
do i=1,N2; call Smooth(l); end do    !  APPLY SMOOTHING STEPS BEFORE COARSENING (N2 times)
```

```
! Now COMPUTE THE DEFECT and RESTRICT it to the coarser grid in a single step.
! TRICK #1 we calculate the defect ONLY on the red points here, as the defect on the
! neighboring black points is zero coming out of the previous call to Smooth.
! TRICK #2: We skip a factor of /2 during the restriction here. We also skip factors of
! *4 during the smoothing and /2 during the prolongation, so the skipped factors cancel.
do jc=2,g(l+1)%ym-1; j=2*(jc-yo)+yo; do ic=2,g(l+1)%xm-1;  i=2*(ic-xo)+xo;
   d(l+1)%d(ic,jc)=d(l)%d(i,j) - v(l)%d(i,j) +                                    &
&              (v(l)%d(i+1,j)+v(l)%d(i-1,j)+v(l)%d(i,j+1)+v(l)%d(i,j-1))/4
end do; end do
v(l+1)%d=d(l+1)%d;    ! TRICK #3: this is a better initial guess for v{l+1} than v{l+1}=0.
call EnforceBCs(l+1);
! Now CONTINUE DOWN TO COARSER GRID, or SOLVE THE COARSEST SYSTEM (via 20 smooth steps).
! Use same smoother on coarse grid; ie, we SKIP a *4 [i.e., *(delta x)^2] factor (TRICK 2).
if (l<nlev-1) then; call Multigrid(l+1); else; do i=1,20; call Smooth(nlev); end do; end if
! Now perform the PROLONGATION.  TRICK #4: Update black interior points only, as next call
! to Smooth recalculates all red points from scratch, so do not bother updating them here.
! We SKIP a /2 interpolation factor here (Trick 2).
do jc=2,g(l+1)%ym; j=2*(jc-yo)+yo; do ic=2,g(l+1)%xm; i=2*(ic-xo)+xo;
   if (j<=g(l)%ym) then; v(l)%d(i-1,j)=v(l)%d(i-1,j)+(v(l+1)%d(ic-1,jc)+v(l+1)%d(ic,jc));
   end if
   if (i<=g(l)%xm) then; v(l)%d(i,j-1)=v(l)%d(i,j-1)+(v(l+1)%d(ic,jc-1)+v(l+1)%d(ic,jc));
   end if
end do; end do
call EnforceBCs(l)
do i=1,N3; call Smooth(l); end do     ! APPLY SMOOTHING STEPS AFTER COARSENING (N3 times)
end subroutine Multigrid
!!!
```

Algorithm 11.10: Multigrid algorithm for solving the 2D Poisson equation (C version).

View

```
#include <stdio.h>       /* A 2D Poisson solver on a uniform mesh using multigrid with   */
#include <stdlib.h>      /* red/black smoothing.  The RHS vector is assumed to be scaled */
#include <math.h>        /* such that the discretized Laplace operator has a 1 on the    */
#include <sys/types.h>   /* diagonal element.                                            */
#include <time.h>                /* May be compiled on a unix machine with:     */
#include <unistd.h>              /* gcc PoissonMG2DTest.c -o PoissonMG2DTest    */
#define max(A,B) ( (A) > (B) ? (A) : (B) )
void PoissonMG2DInit(); void PoissonMG2D(); void Multigrid(int l); void Smooth(int l);
void EnforceBCs(int l); double MaxDefect(int l);
/*------------------------------------------ USER INPUT ------------------------------------------*/
/* XBC=1 for hom Dirichlet (0:NX), =2 for periodic (-1:NX), =3 for hom Neumann (-1:NX+1)*/
/* Same for YBC. NX,NY must be powers of 2 with NX>=NY. N1,N2,N3 set how much smoothing */
int NX=1024, NY=1024, XBC=1, YBC=3, N1=2, N2=2, N3=2;   /* is applied at various steps. */
/*-------------------------------------- END OF USER INPUT ---------------------------------------*/
int xo, yo, nlev;                                                  /* global variables. */
typedef struct { int nx, ny, xm, ym; } grid;  grid *g;
typedef struct { double **d; } arrays;        arrays *v, *d;
/*****************************************************************************************/
int main() {
int i, j, s;  double z=0.0;
printf("BCs: %d, %d.  Smoothing: %d, %d, %d.  Grids:\n",XBC,YBC,N1,N2,N3);
PoissonMG2DInit();
for(i=3; i<g[0].xm-2; ++i) for(j=3; j<g[0].ym-2;++j) { /* The RHS vector b is stored in */
      d[0].d[i][j] = (((double) rand())/((double) RAND_MAX));  /* the initial value of  */
      z += d[0].d[i][j]; }                                     /* -d [see (3.8a)], taken here  */
for(i=3; i<g[0].xm-2; ++i) for(j=3; j<=g[0].ym-2; ++j) /* as a zero-mean random number. */
   d[0].d[i][j] = d[0].d[i][j]-z/((double)((g[0].xm-4)*(g[0].ym-4)));
PoissonMG2D();
} /*****************************************************************************************/
void PoissonMG2DInit() {
/* This initialization routine defines several global variables to avoid the repeated   */
/* memory allocation/deallocation otherwise caused by recursion.  Note that d and v are */
/* arrays of arrays which are of different size at each level l.                        */
int i,l;
switch(XBC) {case 1: xo=1; break;  case 2: xo=2; break; case 3: xo=2; break; }
switch(YBC) {case 1: yo=1; break;  case 2: yo=2; break; case 3: yo=2; break; }
nlev = (int) log2(NY)-1;
g=(grid *) calloc(nlev+1, sizeof(grid));
v=(arrays *) calloc(nlev+1, sizeof(arrays)); d=(arrays *) calloc(nlev+1, sizeof(arrays));
for (l=0; l<=nlev; ++l) {
  g[l].nx = NX/pow(2.,l); g[l].ny = NY/pow(2.,l);       /* The g data structure contains */
  printf("%d %d %d\n", l,g[l].nx,g[l].ny);              /* information about the grids.  */
  g[l].xm = g[l].nx+XBC; g[l].ym = g[l].ny+YBC;
  v[l].d = (double **) calloc( g[l].xm,  sizeof(double *) ); /* d=defect, v=correction. */
  d[l].d = (double **) calloc( g[l].xm,  sizeof(double *) );
  for(i = 0; i < g[l].xm; ++i) {
        v[l].d[i] = (double *) calloc(g[l].ym, sizeof(double));
        d[l].d[i] = (double *) calloc(g[l].ym, sizeof(double)); }}
} /*****************************************************************************************/
void PoissonMG2D() {
double e, o; int i, iter;  double t0, t1;
e=MaxDefect(0);  printf("Iter=%d, max defect=%f\n",0,e);
for (i=1; i<=N1; ++i) Smooth(0);           /* APPLY SMOOTHING BEFORE STARTING MULTIGRID */
t0 = clock(); for (iter=1; iter<=10; ++iter) {     /* PERFORM UP TO 10 MULTIGRID CYCLES. */
   o=e; Multigrid(0); e=MaxDefect(0);
   printf("Iter=%d, max defect=%f, factor=%f\n",iter,e,o/e);
   if (e<1E-13) { printf("Converged\n"); break; } } t1=clock(); t1=(t1-t0)*1.e-6;
printf("Time: %lf; Time/iteration: %f sec\n", t1, t1*max(1./iter,1./10.));
} /*****************************************************************************************/
```

```
void Multigrid(int l) {
/* The main recursive function of the multigrid algorithm. It calls the smoother,       */
/* it performs the restriction and prolongation, and calls itself on the coarser grid.  */
int i, j, ic, jc, s;
for (i=1; i<=N2; ++i) Smooth(l);                    /* APPLY SMOOTHING BEFORE COARSENING  */
/* Now COMPUTE THE DEFECT and RESTRICT it to the coarser grid in a single step.         */
/* TRICK #1 we calculate the defect ONLY on the red points here, as the defect on the   */
/* neighboring black points is zero coming out of the previous call to Smooth.          */
/* TRICK #2: We skip a factor of /2 during the restriction here. We also skip factors of*/
/* *4 during the smoothing and /2 during the prolongation; the skipped factors cancel.  */
for(i=0; i<g[l+1].xm; ++i) for(j=0;j<g[l+1].ym;++j) {v[l+1].d[i][j]=0; d[l+1].d[i][j]=0;}
for (ic=1; ic<=g[l+1].xm-2; ++ic) for (jc=1; jc<=g[l+1].ym-2; ++jc) {
   i=2*(ic-xo)+xo+1; j=2*(jc-yo)+yo+1;
   d[l+1].d[ic][jc]=d[l].d[i][j] - v[l].d[i][j] +
      (v[l].d[i][j+1]+v[l].d[i][j-1]+v[l].d[i+1][j]+v[l].d[i-1][j])*0.25;
   v[l+1].d[ic][jc]=d[l+1].d[ic][jc];  }
EnforceBCs(l+1);     /* TRICK #3: v{l+1}=d{l+1} is a better initial guess than v{l+1}=0. */
/* Now CONTINUE TO COARSER GRID, or SOLVE THE COARSEST SYSTEM (via 20 smooth steps).    */
/* Use same smoother on coarse grid, ie SKIP a *4 [i.e., *(delta x)^2] factor (TRICK 2).*/
if (l<nlev-1) Multigrid(l+1); else for (i=1;i<=20;++i) Smooth(nlev);
/* Perform the PROLONGATION.  TRICK #4: Update black interior points only, as next call */
/* to Smooth will recalculate all red points from scratch, so do not update them here.  */
/* We SKIP a /2 interpolation factor here (TRICK 2).                                    */
for (ic=1;ic<=g[l+1].xm-1;++ic) for (jc=1;jc<=g[l+1].ym-1;++jc) {
   i=2*(ic-xo)+xo+1; j=2*(jc-yo)+yo+1;
   if (j<=g[l].ym-1) v[l].d[i-1][j]=v[l].d[i-1][j]+(v[l+1].d[ic-1][jc]+v[l+1].d[ic][jc]);
   if (i<=g[l].xm-1) v[l].d[i][j-1]=v[l].d[i][j-1]+(v[l+1].d[ic][jc-1]+v[l+1].d[ic][jc]);}
EnforceBCs(l);
for (i=1;i<=N3;++i) Smooth(l);  /* APPLY SMOOTHING STEPS AFTER COARSENING (N3 times)     */
} /*************************************************************************************/
void Smooth(int l) {
/* Red/black smoothing with A from Poisson equation scaled to unit diagonal elements.   */
/* The set of points updated first, which we label as "red", includes the corners.      */
int irb, i, j, m, n;
for (irb=0; irb<=1; ++irb) { for (i=1; i<=g[l].xm-2; ++i) {
   m = 1+(1+i+irb+xo+yo) % 2; n=g[l].ym-2;  /* In C, inner loop should be on LAST index */
   for (j=m; j<=n; j+=2) v[l].d[i][j]=d[l].d[i][j] +
      (v[l].d[i][j+1]+v[l].d[i][j-1]+v[l].d[i+1][j]+v[l].d[i-1][j])*0.25;
   } EnforceBCs(l); }
} /*************************************************************************************/
void EnforceBCs(int l) {
/* Enforce the Neumann and periodic boundary conditions (nothing to do for Dirichlet)   */
int i, m, n;
switch (XBC) { case 3: m=g[l].ym-2; for (i=1;i<=m;++i)
          {v[l].d[0][i]=v[l].d[2][i]; v[l].d[g[l].xm-1][i]=v[l].d[g[l].xm-3][i];} break;
              case 2: m=g[l].ym-2; for (i=1;i<=m;++i)
          {v[l].d[0][i]=v[l].d[g[l].xm-2][i]; v[l].d[g[l].xm-1][i]=v[l].d[1][i];} break;}
switch (YBC) { case 3: n=g[l].xm-2; for (i=1;i<=n;++i)
          {v[l].d[i][0]=v[l].d[i][2]; v[l].d[i][g[l].ym-1]=v[l].d[i][g[l].ym-3];} break;
              case 2: n=g[l].xm-2; for (i=1;i<=n;++i)
          {v[l].d[i][0]=v[l].d[i][g[l].ym-2]; v[l].d[i][g[l].ym-1]=v[l].d[i][1];} break;}
} /*************************************************************************************/
double MaxDefect(int l) {
double e = 0.0; int i,j;
for (i=1; i<=g[l].nx-1; ++i) for (j=1; j<=g[l].ny-1; ++j)
   e = max(e, fabs( d[l].d[i][j] - v[l].d[i][j] +        /* Compute the maximum defect. */
      ( v[l].d[i][j+1]+v[l].d[i][j-1]+v[l].d[i+1][j]+v[l].d[i-1][j] )*0.25));
return (e);
}
```

Nonlinear multigrid

Multigrid methods may easily be generalized to solve large systems of equations derived from nonlinear PDEs such as

$$\frac{\partial^2 \lambda}{\partial x^2} + \frac{\partial^2 \lambda}{\partial y^2} + \alpha \left(\frac{\partial \lambda}{\partial x} \right)^2 + \alpha \left(\frac{\partial \lambda}{\partial y} \right)^2 + \beta \lambda^2 = c, \tag{11.82a}$$

the FD discretization of which [on $n \times n$ gridpoints, analogous to (3.11b)], we write here in the form

$$\frac{\lambda_{i+1,j} - 2\lambda_{i,j} + \lambda_{i-1,j}}{\Delta x^2} + \frac{\lambda_{i,j+1} - 2\lambda_{i,j} + \lambda_{i,j-1}}{\Delta y^2} + \alpha \left(\frac{\lambda_{i+1,j} - \lambda_{i-1,j}}{2\Delta x} \right)^2 + \alpha \left(\frac{\lambda_{i,j+1} - \lambda_{i,j-1}}{2\Delta y} \right)^2 + \beta \lambda_{i,j}^2 = c_{ij}, \tag{11.82b}$$

or in a more compact form as $\mathbf{f}_n(\mathbf{x}_n) = \mathbf{b}_n$. [Note that, if $\beta = 0$, (11.82a) may be derived from (3.11a) via the transformation $\phi = e^{\alpha\lambda}$ and $b = \alpha e^{\alpha\lambda} c$; the representation in (11.82a) is sometimes convenient for systems in which it is important to enforce $\phi > 0$, such as when ϕ represents a concentration or pressure. Note also that a sufficiently small α must be used in this formulation for (3.11b) to be diagonally dominant, or else an approach based on a splitting method will not converge.]

To accomplish this generalization, we first need a nonlinear "smoother" akin to the red/black Gauss-Seidel method in the linear setting. Noting (3.10), the red/black Gauss-Seidel method may be interpreted as looping through all the gridpoints (ordered as first the red points and then the black points in a checkerboard fashion), and at each gridpoint solving (exactly) the governing equation for the corresponding diagonal term, taking all other terms explicitly. To extend this idea to nonlinear equations with the same checkerboard dependence on the data [as in (11.82b)], we can do effectively the same thing, but instead of solving at each gridpoint for the diagonal term, when that is not possible [that is, in those discretized problems for which the unknown at the $\{i, j\}$ gridpoint appears nonlinearly, such as (11.82b) when $\beta \neq 0$] we may instead take one step of the (scalar) Newton-Raphson method (3.2), resulting in the following update formula for each gridpoint:

$$x_i \leftarrow x_i - \frac{f_i(\mathbf{x}) - b_i}{\partial f_i(\mathbf{x}) / \partial x_i}.$$

Note that repeated application of this **Newton-Raphson red/black Gauss-Seidel smoother** on its own does not converge very quickly. When the updates become small, though the Newton-Raphson component of the algorithm (applied at individual gridpoints) becomes quite precise, note that the full nonlinear system $\mathbf{f}(\mathbf{x}) = \mathbf{b}$ behaves essentially like a linear function $A\mathbf{x} = \mathbf{c}$ [for appropriately chosen A and $\mathbf{c}$], and thus convergence of the overall algorithm is approximately the same as that for the red/black Gauss-Seidel smoother in the linear setting. Thus, as in the linear case, the Newton-Raphson red/black Gauss-Seidel method may be recognized as an effective *smoother* of the defect of the nonlinear equation, though it is quite inefficient at reducing the overall *magnitude* of the defect. The **nonlinear multigrid** (a.k.a. **full approximation storage**) method described below levereges this smoothing behavior in a clever way.

To begin, one or two iterations of the Newton-Raphson red/black Gauss-Seidel smoother, as described above, are first applied to reduce the high-frequency components of the defect on a fine-grid discretization of (11.82), with the result written here (for brevity) as $\mathbf{f}_n(\mathbf{x}_n^{(k)}) \approx \mathbf{b}_n$; we denote the defect on this fine grid after these initial smoothing steps by $\mathbf{d}_n^{(k)} = \mathbf{f}_n(\mathbf{x}_n^{(k)}) - \mathbf{b}_n$. Our goal now is to find efficiently an even better approximation $\mathbf{x}_n^{(k+1)}$ such that

$$\mathbf{f}_n(\mathbf{x}_n^{(k+1)}) - \mathbf{b}_n \approx 0 \quad \Rightarrow \quad \mathbf{f}_n(\mathbf{x}_n^{(k)} + \mathbf{v}_n^{(k)}) \approx \mathbf{b}_n \quad \Rightarrow \quad \mathbf{f}_n(\mathbf{x}_n^{(k)} + \mathbf{v}_n^{(k)}) \approx \mathbf{f}_n(\mathbf{x}_n^{(k)}) - \mathbf{d}_n^{(k)}; \tag{11.83}$$

however, due to the high dimension of $\mathbf{x}_n$ [with $\sim n^2$ unknowns], the latter equation is too difficult to solve directly for the correction $\mathbf{v}_n^{(k)}$. The Newton-Raphson red/black Gauss-Seidel approach to this problem is simply to iterate on this update calculation one line at a time (ordered in a red/black fashion) and repeat until convergence; as mentioned previously, this approach is slow to converge.

Following the nonlinear multigrid approach, the desired relation $\mathbf{f}_n(\mathbf{x}_n^{(k)} + \mathbf{v}_n^{(k)}) = \mathbf{f}_n(\mathbf{x}_n^{(k)}) - \mathbf{d}_n^{(k)}$ is **restricted** on a grid which has been coarsened in all coördinate directions by a factor of two; dropping the k superscripts for notational clarity, we denote the restriction of the fine-grid defect onto the coarse grid as $\mathbf{d}_{n/2}$, the restriction of the fine-grid approximate solution onto the coarse grid as $\mathbf{x}_{n/2}$, and the coarse-grid discretization of the nonlinear operator in (11.82b) as $\mathbf{f}_{n/2}(\cdot)$. Note that we need to restrict both $\mathbf{x}_n$ and $\mathbf{d}_n$ to the coarse grid in this setting.

We can now follow one of two approaches. If $n/2$ is sufficiently small that $\mathbf{f}_{n/2}(\mathbf{x}_{n/2} + \mathbf{v}_{n/2}) = \mathbf{f}_{n/2}(\mathbf{x}_{n/2}) - \mathbf{d}_{n/2}$ may be solved essentially exactly (e.g., via the Newton-Raphson red/black Gauss-Seidel smoother applied many times), we go ahead and solve for this correction $\mathbf{v}_{n/2}$ on the coarse grid, then **prolongate** this correction back to the fine grid, and apply it to update $\mathbf{x}_n$ on the fine grid. This restriction/prolongation process introduces high-frequency errors on the fine grid; we may thus apply one or two additional iterations of the Newton-Raphson red/black Gauss-Seidel smoother to $\mathbf{x}_n$, and then repeat the entire process until convergence.

If, however, $n/2$ is still so large that the coarse-grid equation

$$\mathbf{f}_{n/2}(\mathbf{x}_{n/2} + \mathbf{v}_{n/2}) = \mathbf{f}_{n/2}(\mathbf{x}_{n/2}) - \mathbf{d}_{n/2} \tag{11.84}$$

can not easily be solved directly for $\mathbf{v}_{n/2}$, we may instead apply one or two iterations of the Newton-Raphson red/black Gauss-Seidel smoother to $\mathbf{v}_{n/2}$, compute the defect of (11.84) after these smoothing steps, and then restrict this new defect, as well as the corresponding approximate value of $\mathbf{x}_{n/2}$, to an even *coarser* grid with $n/4$ gridpoints in each direction. This process of smooth/restrict/smooth/restrict is repeated until the $\mathbf{f}(\mathbf{x} + \mathbf{v}) = \mathbf{f}(\mathbf{x}) - \mathbf{d}$ problem is reduced to a small enough size that the correction $\mathbf{v}$ may easily be solved essentially exactly. The resulting correction is then prolongated to the previous level, applied to update the previous estimate of $\mathbf{v}$ at this level, and the result smoothed with one or two further iterations of the Newton-Raphson red/black Gauss-Seidel smoother to diminish the high-frequency errors at this level which might have been introduced by the restriction/prolongation process. The process of prolongate/smooth/prolongate/smooth is repeated all the way back out to the original level, and the entire cycle repeated until convergence.

The key to the success of this method is that *the error inherent to the prolongation process is introduced to the solution only via the corrections; thus, as this correction gets small, the error introduced by this process gets proportionally small as well*. Eventually, convergence is as fast (per iteration) as for the linear multigrid strategy, noting that there is extra work involved during each iteration in the nonlinear setting (primarily due to the rextriction of $\mathbf{x}$ and the calculation of the Newton-Raphson update. Implementation of this algorithm is given in Algorithm 11.11.

Summary

An effective starting point for the simulation of PDEs is to discretize in space first, then in time. For spatial discretization, simple finite difference metheods on a structured grid are a good starting point, but spectral methods are significantly more accurate, and should be leveraged whenever the domain and boundary conditions are simple enough to make them feasible. For temporal discretization, as a good starting point, we may use the mixed RK/CN method for parabolic PDEs, and the iterative CN[27] or Newmark methods for hyperbolic PDEs. For elliptic PDEs, multigrid methods have emerged for many problems as the best approach.

The numerical solution of PDEs with complex dynamics will continue to challenge the fastest computers on the planet for the forseeable future; as the speed of these computers increases, the expectactions on the resolution of our PDE solvers inevitably increases accordingly. For this reason, special care should be used when coding a PDE solver to select the appropriate scheme, to minimize the number of flops used, and to minimize the number of full-sized storage arrays referenced. Additionally, memory references should be ordered

[27] Or, alternatively, the iterative theta method, with θ slightly larger than $1/2$, if a bit of high-frequency damping is found to be needed...

Algorithm 11.11: Multigrid algorithm for solving the 2D nonlinear elliptic PDE (11.82a).

View

```
function NonlinearMG2DTest                                    % Numerical Renaissance Codebase 1.0
% Solve (11.62a) on a uniform mesh using nonlinear multigrid with N-R r/b G-S smoothing.
% The RHS vector is assumed to be scaled such that the discretized Laplace operator has a
% 1 on the diagonal element.  This code was formed by modification of PoissonMG2DTest.m,
% to which the reader is referred for extensive comments; only the significant
% modifications for extension to the nonlinear case are marked with comments here;
% see especially those four points indicated by comment *** NOTE ***.
clear; global XBC YBC N1 N2 N3 xo yo d v x g nlev alpha betabar verbose
% ------------------------------------------ USER INPUT ------------------------------------------
NX=32; NY=32; XBC=1; YBC=1; N1=5; N2=3; N3=3; alpha=0.1, beta=0.5
% ---------------------------------------- END OF USER INPUT ----------------------------------------
fprintf('BCs:%2g,%2g. Smoothing:%2g,%2g,%2g.\nGrids:\n',XBC,YBC,N1,N2,N3);
for verbose=1:2
  NonlinearMG2DInit(NX,NY), for l=1:nlev, betabar(l)=-beta/(4*g{l}.nx^2); end
  d{1}(3:g{1}.xm-2,3:g{1}.ym-2)=rand(g{1}.xm-4,g{1}.ym-4);
  i=sum(sum(d{1}(:,:)))/((g{1}.xm-4)*(g{1}.ym-4));
  d{1}(3:g{1}.xm-2,3:g{1}.ym-2)=d{1}(3:g{1}.xm-2,3:g{1}.ym-2)-i;
  NonlinearMG2D;
end
end % function NonlinearMG2DTest
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function NonlinearMG2DInit(NX,NY)
global XBC YBC xo yo d v x g nlev
switch XBC case 1, xo=1; case 2, xo=2; case 3, xo=2; end
switch YBC case 1, yo=1; case 2, yo=2; case 3, yo=2; end
nlev=log2(NY);
for l=1:nlev
   g{l}.nx = NX/(2^(l-1)); g{l}.ny = NY/(2^(l-1));
   g{l}.xm=g{l}.nx+XBC; g{l}.ym=g{l}.ny+YBC;
   v{l}=zeros(g{l}.xm,g{l}.ym); d{l}=v{l}; x{l}=v{l};    % initialize x=solution
   fprintf('%5g%5g%5g\n',l,g{l}.nx,g{l}.ny)
end
end % function NonlinearMG2DInit
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function NonlinearMG2D
global N1 verbose
e=MaxDefect(0); fprintf('Iter=0, max defect=%0.3e\n',e);
for i=1:N1; Smooth(1); e = MaxDefect(i); end
tic; for iter=1:20   % Do up to 20 multigrid cycles (convergence slower than linear case).
   o=e;  Multigrid(1); e = MaxDefect(iter);
   if verbose>0, fprintf('Iter=%0.6g, max defect=%0.3e, factor=%0.4f\n',iter,e,e/o); end
   if e<1E-13,   fprintf('Converged\n'), break, end
end; t=toc;
fprintf('-> Total time: %0.3g sec; Time/iteration: %0.3g sec\n',t,t/iter);
end % function NonlinearMG2D
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Multigrid(l)
global N2 N3 xo yo d v x g nlev alpha betabar
for i=1:N2; Smooth(l); end
d{l+1}=zeros(g{l+1}.xm,g{l+1}.ym);  x{l+1}=d{l+1};
for ic=2:g{l+1}.xm-1; i=2*(ic-xo)+xo; for jc=2:g{l+1}.ym-1; j=2*(jc-yo)+yo;
% Compute and restrict d in a single step.                                    *** NOTE ***
d{l+1}(ic,jc)=(v{l}(i+1,j)+v{l}(i-1,j)+v{l}(i,j+1)+v{l}(i,j-1))/4-v{l}(i,j)+d{l}(i,j)  ...
  +((v{l}(i+1,j)-v{l}(i-1,j))^2+2*(x{l}(i+1,j)-x{l}(i-1,j))*(v{l}(i+1,j)-v{l}(i-1,j))  ...
   +(v{l}(i,j+1)-v{l}(i,j-1))^2+2*(x{l}(i,j+1)-x{l}(i,j-1))*(v{l}(i,j+1)-v{l}(i,j-1))  ...
   )*alpha/16 + betabar(l)*(v{l}(i,j)^2 + 2*v{l}(i,j)*x{l}(i,j));
% Restrict x as well.                                                         *** NOTE ***
x{l+1}(ic,jc)=(x{l}(i,j)+v{l}(i,j))/2+(x{l}(i+1,j)+x{l}(i-1,j)+x{l}(i,j+1)+x{l}(i,j-1) ...
                                      +v{l}(i+1,j)+v{l}(i-1,j)+v{l}(i,j+1)+v{l}(i,j-1))/8;
```

```
end; end;
v{l+1}=d{l+1};
EnforceBCs(l+1)
if (l<nlev-1); Multigrid(l+1); else; for i=1:20; Smooth(nlev); end; end
for ic=2:g{l+1}.xm; i=2*(ic-xo)+xo; for jc=2:g{l+1}.ym; j=2*(jc-yo)+yo;
    if j<g{l}.ym, v{l}(i-1,j) = v{l}(i-1,j)+(v{l+1}(ic-1,jc)+v{l+1}(ic,jc)); end
    if i<g{l}.xm, v{l}(i,j-1) = v{l}(i,j-1)+(v{l+1}(ic,jc-1)+v{l+1}(ic,jc)); end
end; end
EnforceBCs(l)
for i=1:N3; Smooth(l); end
end % function Multigrid
%%%
```

appropriately to leverage cache-based memory systems, and computations should be grouped appropriately to leverage massivly parallel computational clusters; some general high performance computing guidelines to help achieve these goals are discussed in §12. To achieve this balance, it is almost always necessary to code PDE solvers by hand, exploiting structure wherever possible. Though certain numerical algorithms (as developed in the previous chapters of this text, like the Thomas algorithm) may be used over and over as generic black-box tools, each PDE system you will encounter will generally have different structure and, along with this structure, different opportunities for maximizing the efficiency of its numerical implementation. This chapter has thus been heavy in examples, illustrating some of the typical considerations involved in the development of such PDE solvers; a more involved example is given in §13.

Exercises

Exercise 11.1 Verify (11.22b).

Exercise 11.2 It was seen in (11.30) that the convective term in the 1D Burgers', KS, and CCH equations, $-u\,\partial u/\partial x$, is energy conserving. Establish whether or not the cubic term of the 1D CCH and CH equations, $\partial^2(u^3)/\partial x^2$, and the convective term of the 3D NSE, $(\vec{u}\cdot\nabla)\vec{u}$, are energy conserving following similar derivations. Hint: in the latter case, apply the fact that the velocity field $\vec{u}$ is divergence free (i.e., $\nabla\cdot\vec{u}=0$).

Exercise 11.3 Recall that the Froude number of a 2D shallow water flow is $Fr=\sqrt{u_1^2+u_2^2}/\sqrt{gh}$, where g is the acceleration due to gravity and h is the local height of the water column. A shallow water flow is said to be subcritical if $Fr<1$, and supercritical if $Fr>1$. A stationary hydraulic jump normally takes a flow from a supercritical state to a subcritical state. Can a stationary hydraulic jump take a flow from a subcritical state to a supercritical state? Why or why not?

Exercise 11.4 Recall that the Mach number of a 3D flow governed by Euler's equation is $M=\sqrt{u_1^2+u_2^2+u_3^2}/a$, where $a=\sqrt{\gamma RT}$ is the local speed of sound. A flow is said to be subsonic if $M<1$, and supersonic if $M>1$. A stationary shock normally takes a flow from a supersonic state to a subsonic state. Can a stationary shock take a flow from a subsonic state to a supersonic state? Why or why not?

Exercise 11.5 Consider a high-velocity open channel flow bounded by a wall with a corner, as illustrated in Figure 11.12. The corner causes a **stationary oblique hydraulic jump** in the flow (see dashed line in Figure 11.12; see also photos in Figure 11.13) which is actually undular and/or turbulent, but can be approximated as a Heaviside step function in the height of the water column, h, for the purpose of analysis. Identifying the components of the velocity normal and tangential to the oblique hydraulic jump in Figure 11.12 as, respectively, $u_\perp^- = |\mathbf{u}^-|\sin\beta$ and $u_\parallel^- = |\mathbf{u}^-|\cos\beta$ upstream of the jump, and $u_\perp^+ = |\mathbf{u}^+|\sin(\beta-\theta)$ and $u_\parallel^+ = |\mathbf{u}^+|\cos(\beta-\theta)$ downstream of the jump, and defining the **normal Froude number** upstream and downstream of the hydraulic jump as $F_\perp^- = u_\perp^-/\sqrt{gh^-}$ and $F_\perp^+ = u_\perp^+/\sqrt{gh^+}$, the jump relations developed in class may be used to understand the shallow-water flow:

$$u_\parallel^- = u_\parallel^+, \qquad h^-u_\perp^- = h^+u_\perp^+, \qquad h^+ = \Big(-h^- + \sqrt{(h^-)^2+8(u_\perp^-)^2h^-/g}\Big)/2.$$

Based on these relations, develop a single transcendental equation that relates β, θ, and $F^- = |\mathbf{u}^-|/\sqrt{gh^-}$.

Exercise 11.6 Compute the dispersion relation of the piano wire model given in footnote 11 on page 347. For a string of length L, what is the frequency of oscillation of the fundamental mode ($p=1$) and the first four harmonics ($p=2$ through 5) in terms of $\{c,\kappa,b_1,b_2,L\}$? How long does it take each of these modes to

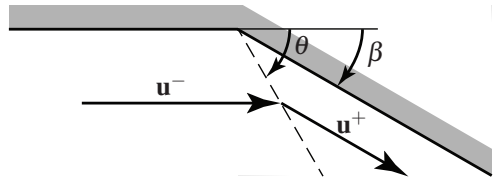


Figure 11.12: Schematic of an oblique hydraulic jump.



Figure 11.13: Photos of an oblique hydraulic jump in the lab (Bewley, Forman, Mullenax 1988).

decay to 1/2 of its initial amplitude (also in terms of $\{c, \kappa, b_1, b_2, L\}$)? Note that the problem of identifying the magnitude and phase of each mode at the initial time, depending on how and where the piano wire is struck by the hammer, was already solved in §4.10b.

Exercise 11.7 In §11.2.3, it was found that the leapfrog scheme applied to a spatial discretization of the 1D diffusion equation was unstable. Based on the result of Exercise 10.7, is this result anticipated? Discuss.

Exercise 11.8 Following Algorithm 11.2, develop a code to simulate the KS equation (11.26) with homogeneous Dirichlet BCs using second-order central FD in space CN/RKW3 in time. Then, following Algorithm 11.3, develop a code to simulate the same system using significantly reduced storage. Discuss.

Exercise 11.9 Following Algorithm 11.4 closely, develop a pseudospectral code named `CCH2D_CNRKW3_PS.m` to simulate the 2D CCH equation (11.28) on a square domain $L_x = L_y = L$. Experiment with various values of D and sufficiently large L to explore the chaotic behavior of this system, and discuss. Make sure you give the system sufficiently large initial conditions u_0 to push the unsteady dynamics out to its chaotic attractor.

Exercise 11.10 (**a**) Calculate the speeds of the Matlab, Fortran, and C implementations of the multigrid algorithm given in §11.4.1 on your computer on a 256×256 discretization. Discuss.

(**b**) Noting your results in problem (a), extend one of the three multigrid implementations provided (your choice) to solve the 3D Poisson equation on a $256 \times 256 \times 256$ grid. This generalization is easier than it might at first seem, as it mostly involves applying what is already done in the first and second dimensions to the third dimension; we will again use a second-order finite difference operator, restriction via half weighting, prolongation via bilinear interpolation, and the standard formula (3.8c) for the prolongation update, noting that the second-order finite difference stencil, the half-weighting restriction stencil, and the bilinear

interpolation prolongation stencil in the 3D case may be written

$$\mathscr{A}_{\text{SOFD}} = \frac{1}{h^2}\left\{\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & -6 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right\},$$

$$\mathscr{R}_{\text{HW}} = \left\{\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 0 & 1/12 & 0 & 0 & 0 & 0 \\ 0 & 1/12 & 0 & 1/12 & 1/2 & 1/12 & 0 & 1/12 & 0 \\ 0 & 0 & 0 & 0 & 1/12 & 0 & 0 & 0 & 0 \end{array}\right\},$$

$$\mathscr{P}_{\text{BI}} = \left\{\begin{array}{ccc|ccc|ccc} 1/8 & 1/4 & 1/8 & 1/4 & 1/2 & 1/4 & 1/8 & 1/4 & 1/8 \\ 1/4 & 1/2 & 1/4 & 1/2 & 1 & 1/2 & 1/4 & 1/2 & 1/4 \\ 1/8 & 1/4 & 1/8 & 1/4 & 1/2 & 1/4 & 1/8 & 1/4 & 1/8 \end{array}\right\}.$$

Note that special care is required specifically during the prolongation step to ensure that *all* black interior points are updated correctly (sketch the geometry to check this!).

References

Bensa, J, Bilbao, S, Kronland-Martinet, R, & Smith, J.O. III (2003) The simulation of piano string vibration: From physical models to finite difference schemes and digital waveguides, *J. Acoust. Soc. Am.* **114**, 1095-1107.

Bewley, T, Forman, S, & Mullenax, C (1988) High velocity flow in open channels, *Caltech Hy 111 Lab Report.*

Bird, R.B., Stewart, W.E. and Lightfoot, E.N. (2007) *Transport Phenomena*. Wiley.

Fahroo, F, & Ito, K (2006) Optimal Absorption Design for Damped Elastic Systems, *Proceedings of the 2006 American Control Conference*, 63-67.

Fletcher, CAJ (1991) *Computational Techniques for Fluid Dynamics, Volumes I and II*. Springer.

Garabedian, PR (1998) *Partial Differential Equations.* AMS Chelsea Publ.

Golovin, AA, & Pismen, LM (2004) Dynamic phase separation: from coarsening to turbulence via structure formation, *Chaos* **14**, 845-854.

Golovin, AA, Nepomnyashchy, AA, Davis, SH, & Zaks, MA (2001) Convective Cahn-Hilliard Models: From Coarsening to Roughening, *Phys. Rev. Lett.* **83**, 1550-1553.

Grimshaw, R, Pelinovsky, D, & Pelinovsky, E. (2010) Homogenization of the variable- speed wave equation *Wave Motion* **47**, 496-507.

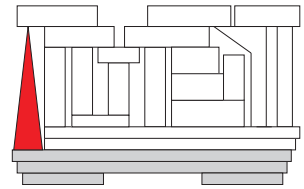
Lattes, R, & Lions, JL (1969) *The Method of Quasi-Reversibility, Applications to Partial Differential Equations*, American Elsevier.

Liepman, HW, & Roshko, A (1957) *Elements of Gasdynamics*, Dover.

Luchini, P, & D'Alascio, A (1994) Multigrid pressure correction techniques for the computation of quasi-incompressible internal flows, *International Journal for Numerical Methods in Fluids* **18**, 489-507.

Strang, G (1986) *Introduction to Applied Mathematics*. Wellesley-Cambridge Press.

Trottenberg, U, Oosterlee, CW, & Schüller, A (2001) *Multigrid*. Academic Press.



Chapter 12

High performance computing

Contents

12.1 Coding for portable speed . . . 396
12.2 Coding for cache-based memory architectures . . . 396
12.3 Coding for fine-grained parallelization at the CPU level . . . 396
12.3.1 Pipelineing and superscalar architectures . . . 397
12.3.2 Vector architectures . . . 399
12.4 Coding for coarse-grained parallelization at the system level . . . 399
12.4.1 Shared-memory parallelization . . . 400
12.4.2 Distributed-memory parallelization . . . 401
12.5 Performance tuning . . . 403
12.6 Summary . . . 403
Exercises . . . 404

The rapid growth in computer power over the last several decades has been nothing short of phenomenal, and is expected to continue for many years to come. This growth is driven largely by fierce market pressure in the highly competitive personal computer market. Due to this market pressure, technology originally developed for the world's most powerful supercomputers has rapidly been incorporated into inexpensive "commodity" hardware broadly available to everyday users. Powerful computational tools based on efficient numerical methods (such as those discussed in the present text) that leverage this advanced computational hardware have fundamentally altered the manner in which we consider a wide spectrum of problems that may now be simulated accurately in science, engineering, finance, and an increasing number of other disciplines. In order to develop efficient numerical methods that use modern computers effectively on such problems, some understanding of how these computers work is helpful.

Generally speaking, one might say that there are two characteristics of a computer that account for its power: "speed" and "complexity". **Clock speed** is the easiest measure to identify: in a digital computer, the execution of instructions on the CPU are synchronized by a clock, so, apparently, the faster the clock speed, the faster the computation. In any particular CPU/motherboard design, however, there are physical limits on the clock speed beyond which the computer will cease to function correctly. These limits are related to parasitic **resistance** (R), **capacitance** (C), and **inductance** (L) in the tiny circuits within the computer, which lead to both **characteristic time constants** (RC and L/R) in the system and, quite often, the generation of

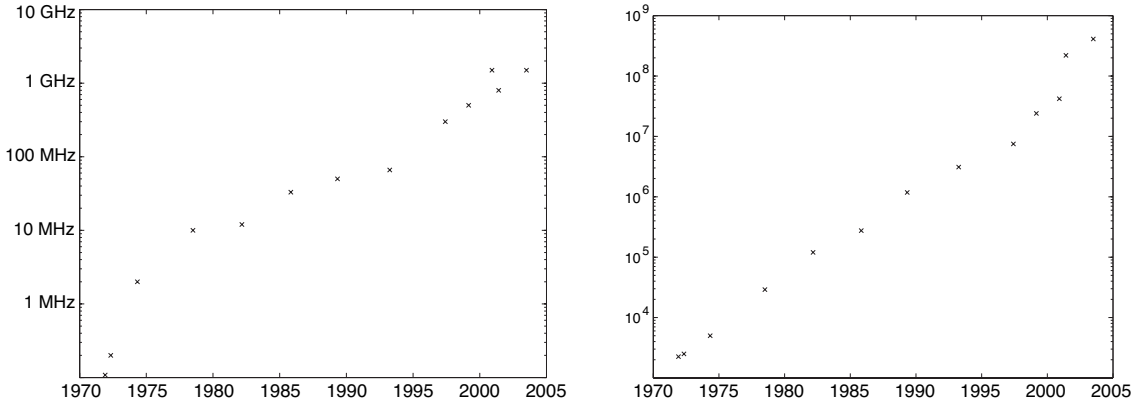


Figure 12.1: (left) Maximum rated clock speed at introduction and (right) transistor count as a function of release date for several Intel CPUs, model numbers (left to right) 4004, 8008, 8080, 8086, 286, 386, 486, Pentium, Pentium II, Pentium III, Pentium 4, Itanium, and Itanium 2. (data from: http://www.intel.com/pressroom/kits/quickreffam.htm).

a significant amount of **heat** that increases rapidly with clock speed and that must be dissipated sufficiently quickly (or else some component in the system will melt, crack, or delaminate). These physical limits are often more restrictive in the several subsystems in the computer that service the CPU than they are in the CPU itself (where the components are typically smaller), and thus these subsystems are usually synchronized at some fraction of the clock speed that synchronizes the CPU. One of the most significant of these subsystems for the purpose of numerical computations is the **system bus** that attaches the CPU to, among other things, the main (a.k.a. **core**) memory. Fortunately, the **bus speed** synchronizing this subsystem usually need not be as fast as the full clock speed in order to keep the CPU busy, but if it is too slow, the CPU will spend a substantial fraction of its available cycles idle on most problems, while it waits for the necessary data to be loaded to and from the main memory in order to prepare for the next set of calculations to begin. One often-used trick to make the most out of slower buses is to use **wide data paths** (with many wires), so that enough bits to represent multiple real numbers can be in transit to or from the main memory at any instant.

In the final design of a computer, an appropriate balance between the CPU, bus, and main memory speeds and the width of the data paths is necessary to maximize overall system performance while keeping manufacturing costs within reasonable limits. Achieving this balance is not the only thing that makes an affordable computer powerful, however. The steady growth of **chip complexity** over time is popularly characterized by **Moore's law**, which refers specifically to the trend predicted by Gordon E. Moore (cofounder of Intel) in 1965 of the approximate doubling of the number of transistors per CPU every 1 to 2 years. Remarkably, as shown in Figure 12.1 for the period 1970 to 2005, this predicted trend has held true for almost four decades; for example, the data from Intel over this period reveals a doubling of the number of transisters per CPU roughly every 22 months, combined with an order of magnitude increase in clock speed roughly every decade.

The manner in which increases in CPU complexity correlate to increases in computer power is somewhat involved. This first key point in this regard is that, more often than not, when one element of the main memory (at a particular **memory address**) is needed in a numerical calculation, nearby elements of the main memory will be needed in subsequent calculations. Thus, multiple levels of high-speed **cache memory** on and near the CPU may be used to greatly accelerate the majority of the memory references. In a cache-based memory system, the main memory is retreived and stored in the higher-speed cache memory as contiguous finite-sized vectors or **chunks**. Conversely, to make room in the cache for new data as the computation proceeds,

the main memory is updated by the results of previous computations (and, then, the corresponding sections of cache released) in a similar, chunk-wise fashion. This approach has proven to be so useful that it is usually applied at multiple levels (typically three, named **L1**, **L2**, and **L3**). The **registers**[1] referenced by the functional units performing the actual computations usually (if the cache system is doing its job well) load the specific elements of data necessary for the next calculation from the highest-speed cache (the L1 or **primary** cache, on the CPU) and then return this result to the L1 cache after the computation is completed. The L1 cache, in turn, typically loads data from & returns data to the lower-speed caches in relatively small chunks, whereas the lowest-speed cache (L3, typically on a separate chip near the CPU) loads data from & returns data to the main memory in relatively large chunks. Getting the necessary data to & from the registers in a timely fashion is in fact one of the most challenging aspects of modern CPU design; as can be seen in the representative CPU illustrated in Figure 12.2, a substantial fraction of modern CPUs is devoted to this multi-level cache-based memory system and its coördination.

The second key point by which increases in CPU complexity can result in increases in computer power is the fact that many of the computations required by numerical algorithms can almost always be performed **independently**; that is, when arranged properly, one step of a particular loop of the algorithm does not need to be completed before the next step can begin. This point can be leveraged in various different ways at both the CPU level and the system level to perform various computations in **parallel** (that is, either simultaneously or nearly simultaneously but possibly out of order). There are two classes of parallelism one can identify in a numerical algorithm: **fine-grained parallelism** and **coarse-grained parallelism**.

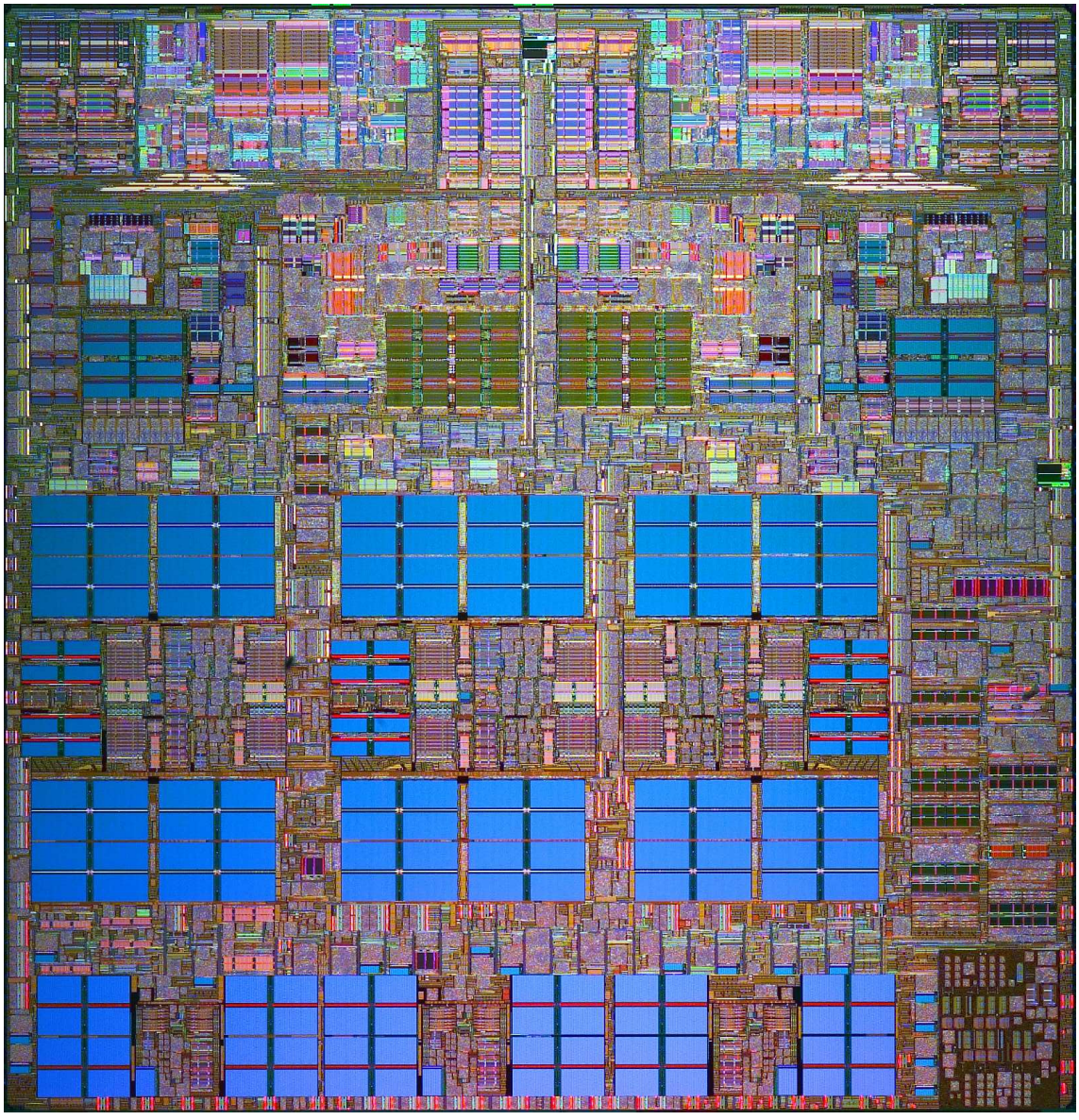
Fine-grained parallelism refers both to the independence of the individual instructions within the innermost loops of an algorithm and to the independence of one step of the innermost loop from the others (which is slightly different). Fine-grained parallelism of an algorithm may be leveraged efficiently by **pipelined**, **superscalar**, and **vector** processing elements on a single CPU, as defined and discussed further in §12.3, to perform many computations in parallel, making maximum use of the data in the L1 and L2 cache before exchanging it for other data. For example, considered again the column-wise approach to matrix/matrix multiplication introduced on page 8:

```
B=zeros(m,n)
for k=1:p
   for j=1:n
      x=X(k,j);
      for i=1:m
         B(i,j)=B(i,j)+A(i,k)*x;
      end
   end
end
```

The `i` loop in this algorithm is an example of fine-grained parallelism: note that the computer is free to work on any given step of this loop before the previous steps of the loop are finished.

In contrast, **coarse-grained parallelism** refers to the independence of any given step of one of the *outer* loops of an algorithm from the other steps of this loop. Coarse-grained parallelism of an algorithm may be leveraged by systems with several CPUs (each with their own cache) in order to divide up these outer loops into independent blocks that may be worked on separately. Coarse-grained parallelism of an algorithm may be leveraged efficiently by both **shared-memory** and **distributed-memory** computer systems, as defined and discussed further in §12.4, to perform different blocks of computations in parallel while using separate memory caches to accelerate references to different sections of the relevant matrices. For example, the `j` loop in the algorithm shown above is an example of coarse-grained parallelism: note that the compiler is

[1] Registers are the ultra high speed memory storage areas on the CPU that make the input data available to the **functional units** that perform the actual calculations and then contain the results determined by these functional units.



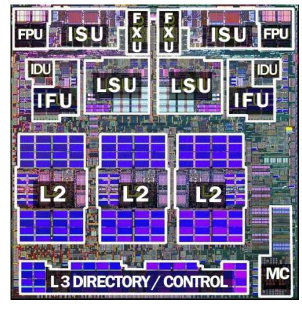


Figure 12.2: (above) Photo of the IBM Power5 chip, with (right) some of the larger functional units marked: FPU = floating-point unit, FXU = fixed-point (e.g., integer) execution unit, ISU = instruction sequencing unit, IFU = instruction fetch unit, IDU = instruction decode unit, LSU = load/store unit, and MC = memory controller. This chip has, in its lower 60%, a 1.875 MB L2 cache (split into three pieces), a controller (on the chip) for a 36 MB L3 cache (off the chip), and a controller to access the main memory. In its upper 40%, the chip actually contains two independent, superscalar CPU cores, each of which with a 64KB L1 instruction cache, a 32 KB L1 data cache, and two floating-point units. The Power5 chip, with 276 million transistors, measures almost 2cm on each side and is manufactured with a 130nm silicon-on-insulator (**SOI**) complementary metal oxide semiconductor (**CMOS**) process with copper interconnects. It was introduced in May, 2004, at a maximum clock speed of 1.9GHz, and draws nearly 200W when running at peak power. (Photo courtesey of IBM).

free to assign one block of steps of this loop to one CPU and other blocks of this loop to other CPUs for independent computation. To leverage coarse-grained parallelism to achieve the maximum effect, it should be made as coarse as possible; that is, the loop to be parallelized across separate processors should, if possible, be the outermost loop, thereby allowing the maximum amount of work to be done independently by each CPU before the CPUs need to synchronize with each other and share the results of their computations. For example, swapping the order of the `j` and `k` loops in the algorithm shown above increases the coarseness of the coarse-grained parallelism in this manner, leading to significant acceleration of the resulting code on a multiple-CPU system.

Note that, in the early days of computing (before advanced compilers were developed), programmers needed to write their numerical codes in **assembly language**[2] in order to optimize effectively both the speed and the memory usage of the resulting code. In these early days, **complex instruction set computers** (**CISC**), with relatively powerful instructions available in assembly language, were generally the most successful. With time, however, a number of factors led to an effective convergence to the dominant CPU architecture being **reduced instruction set computers** (**RISC**), in which the basic instruction set that can be executed in machine language is simplified. The factors that led to this paradigm shift included, among other things, the development of self-optimizing compilers in **high-level languages**[3] that efficiently automated a substantial fraction of the tedious effort required in assembly language programing. The simplification of the basic instruction set in RISC CPUs allows clock speeds to be substantially increased, essentially because each of the individual commands the computer may execute are restricted to be, in a way, fairly "simple". In such systems, more involved instructions are achieved by the compiler by chaining these simple commands together. Though their basic instruction sets have been somewhat streamlined, modern RISC processors, such as the IBM Power5 CPU depicted in Figure 12.2, are in fact quite complex; indeed, they are often considered as one of the more remarkable feats of modern engineering.

In this text, we would like to learn how to develop and implement numerical algorithms with near maximal efficiency on modern CPUs that account for both their **cache-based memory architecture** and their capability for **parallel computation** at both the CPU level and the system level on both computers with **multiple CPUs sharing the same main memory** as well as **interconnected networks of many independent computers**. This goal may be achieved by following a few **high performance computing guidelines** and, where necessary, introducing additional commands or directives to tell the compiler more precisely what to do in order to facilitate both parallel computations and the efficient use of memory and cache, as discussed in the remainder of this chapter. Attention to such coding issues can often play a very significant role, drastically reducing the overall execution time of a numerical code requiring a certain number of flops.

[2] **Assembly language** is a CPU-specific language that lists the commands actually executed by the CPU, with the instructions and variables listed by name. Assembly language is closely related to **machine language**, the set of instructions directly executed by the computer, with assembly language being slightly easier to read by humans, as the instructions and variables are listed by name instead of by number. Assembly language programs are translated to machine language by a program called an **assembler**.

[3] **High level languages**, such as Fortran, C, and Matlab syntax, are machine-independent languages that allow one to automate a numerical algorithm using commands that are close to that which you would write by hand, without using instructions that are dependent on a particular computer architecture. Programs written in high-level languages are translated into assembly language or machine language either all at once (such as in Fortran and C) by a program called a **compiler**, or one by one (such as in Matlab) by a program called an **interpreter**. Interpreters are convenient for experimenting with simple algorithms, but compilers are generally much more powerful for producing fast, memory-efficient machine-language code. A student studying advanced numerical methods in the 21st century should learn both Matlab syntax and either Fortran or C (or both), and be comfortable translating numerical algorithms back and forth between these languages; fortunately, such a student does *not* need to know assembly language. An advanced numerical algorithm is given in Fortran, C, and Matlab syntax in §**??** in order to facilitate the conversion between these three essential programming languages; note that this conversion is primarily just a matter of syntax.

12.1 Coding for portable speed

When going to the effort of writing a numerical code, one should plan on it ultimately being run on a variety of computer platforms. The key strategy to start with is this: *don't get in the way of the compiler optimization*; that is, keep the code simple and do not apply architecture-dependent or cache-size-dependent optimization tricks early in the code development, which might accelerate the code on your current platform but slow it down on other platforms you may want to use later. If the code is written in a clear, straightforward fashion following the various guidelines outlined in this chapter, the final compilation of the debugged code (using a good compiler with aggressive optimization enabled) should produce nearly optimal speeds without resorting to complicated tricks, which often result in substantially less straightforward code that is difficult for humans to understand, debug, and extend, and is difficult for advanced compilers to optimize further. In summary:

Guideline 12.1 *To make a code straightforward to optimize for a variety of compiler/CPU combinations, keep the initial implementation of the algorithm in numerical code straightforward. Favor the use of simple data structures where possible, such as statically-allocated arrays whose size is known at compile time.*

Note that, on unix machines, the sometimes long compile times of major codes with aggressive optimization by the compiler enabled can be substantially reduced during the iterative debugging/code optimization process using **makefiles**, which are convenient scripts which tell the unix program how to compile and link the various subprograms making up the code. Note also that, if you plan to run your code for hours, days, weeks, or even months at a time on a given machine, then there is certainly a time and place for applying architecture-dependent code optimization strategies, but it is not in the initial code development. Rather, it is in the final tweaking of an optimized, architecture-dependent version of the code, after you get a generic version of the code working and you are trying to squeeze the maximum performance possible out of a given machine (see §12.5).

Finally, be open to using both **BLAS** (introduced in §1.2.2) and **LAPACK** (a **linear algebra package** of standardized routines for many common problems in linear algebra). Versions of both of these standardized libraries of routines have, by now, been carefully hand tuned for fast execution on virtually all major CPU/operating-system combinations.

12.2 Coding for cache-based memory architectures

As discussed previously, memory is typically loaded from & returned to the main memory and loaded from & returned to the various levels of higher-speed cache memory as contiguous finite-size chunks of data, not one element at a time. The algorithms for managing such cache-based memory systems in modern RISC CPUs and multiple-CPU computer systems is in fact quite sophisticated. Fortunately, the following simple guideline is sufficient to use such cache-based architectures efficiently:

Guideline 12.2 *To make efficient use of cache, every innermost loop in a numerical algorithm should, if possible, be* **unit stride** *in all matrices that it references (that is, it should reference elements in memory sequentially, as referencing element that are far apart in memory uses cache ineffectively). Further, as many flops as possible should be included in these inner loops, in order to perform as much work as possible on the data in cache before it must be exchanged for other data in the main memory.*

12.3 Coding for fine-grained parallelization at the CPU level

As mentioned previously, many numerical algorithms may be arranged to exhibit fine-grained parallelism, that is, independence of the calculations in the innermost loops. Such algorithms are well suited for parallel

computation of many of the individual calculations in the algorithm on a single CPU. This may be accomplished in a few different ways, as discussed further in this section.

12.3.1 Pipelineing and superscalar architectures

Each command exectuted by a CPU in fact takes several clock cycles to complete. Though the number varies from CPU to CPU, it generally takes one or more clock cycles for each of the following stages:

- **intruction fetch**: load the instruction to be executed from memory,
- **instruction decode**: interpret the instruction to determine precisely what needs to be done,
- **operand fetch**: fetch the necessary data from the registers, cache, or main memory,
- **execute**: execute the instruction (e.g., add, multiply, or divide the fixed or floating-point numbers),
- **writeback**: write the result back to the registers, cache, or main memory.

The technique referred to as **pipelineing** is simply the starting of the next command before the previous command finishes. Following this approach, several independent commands may be in various stages of completion at any given instant. For example, the first in a series of independent commands can be just finishing up (performing writeback), while the second command performs an execute, the third command performs operand fetch, the fourth command performs instruction decode, and the fifth command performs instruction fetch. In this manner, the CPU can complete many more commands in a given amount of time with a given number of functional units on the chip, even though each individual command still takes the same amount of time to complete as if pipelining were not implemented. As its name implies, each stage of each command must be completed in order in a pipeline: if a certain command **stalls**[4], then later commands must wait until the command ahead completes before the flow can continue. Avoiding such stalls[5] is thus key to obtaining good performance. Note that a CPU architecture that is broken up into small enough stages that each stage is simple enough to require only a single clock cycle to execute (even when performing floating point arithmetic) is referred to as **fully pipelined**; under ideal circumstances (that is, when the L1 cache contains the necesary data), CPUs following this strategy can start a new command every single clock cycle. The process of further dividing up the more complicated stages in the pipeline into more, even simpler stages so that the clock speed can be further increased is sometimes called **superpipelining**.

Another way of implementing parallelism in a CPU is to implement multiple functional units that may operate independently, sometimes coördinated by separate pipelines. CPUs that implent this strategy are known as **superscalar** or **multiple instruction issue** processors. **Simultaneous multithreading** (**SMT**) is a recent improvement to the superscalar processing approach in which multiple "threads", or independent sets of calculatations (either from the same numerical algorithm or from completely separate jobs, potentially submitted by different users), may issue instructions at each clock cycle, per the availability of resources in the CPU, in order to make maximum use of the multiple functional units on the CPU, with the threads that are ready performing calculations during those clock cycles which would otherwise be wasted by other threads which are stalled.

The pipelining and superscaler approaches (with or without SMT) only work well if, a substantial fraction of the time, each command executed by the assembly language code is independent of the commands immediately preceding and following it. If a good compiler is used, the assembly language code quite likely contains a significant reordering of the commands in the original Fortran or C code. To some degree, the Fortran or C programmer still needs to be aware of the issues related efficient use of pipelined and superscaler architectures, however, as the complier that translates the high-level code into assembly language can more readily find the fine-grained parallelism necessary to facilitate parallel computations using both pipelining and superscaler architectures if the original code is written with either (a) independent calculations within the

[4] A command can stall, for example, if it must fetch data all the way from main memory because it suffers a **cache miss** (that is, the data it needs to execute does not reside in a high-speed cache).

[5] Stalls may be avoided by, among other things, following HPC Guideline #2 described previously so that, more often than not, the data needed for a particular command resides in cache, referred to as a **cache hit**.

inner loop or (b) inner loops for which each step of the loop is independent from the other steps of the loop. If the latter type of fine-grained parallelism is present, one strategy employed by most modern compilers is to perform an appropriate amount of **loop unrolling**, as illustrated below

Original inner loop

```
for i=1:m
   B(i,j)=B(i,j)+A(i,k)*x;
end
```

Unrolled inner loop

```
mm=mod(m,4)
for i=1:4:m-mm
   B(i  ,j)=B(i  ,j)+A(i  ,k)*x;
   B(i+1,j)=B(i+1,j)+A(i+1,k)*x;
   B(i+2,j)=B(i+2,j)+A(i+2,k)*x;
   B(i+3,j)=B(i+3,j)+A(i+3,k)*x;
end
for i=m-mm+1:m
   B(i,j)=B(i,j)+A(i,k)*x;
end
```

In the unrolled version, note that the bulk of the computations (for large m) are calculated in a loop that has many independent calculations within the loop that are independent of each other and may therefore be calculated in parallel using the pipelineing and superscalar approaches described above. If the length of the original loop (m) is not evenly divisible number of steps by which the loop is unrolled (4), then a short additional loop is necessary to pick up the $mm = 1$, 2, or 3 extra steps of the loop, as indicated above right. As emphasized by HPC Guideline #1, *do not perform loop unrolling yourself*, as the compiler can generally do it better: modern compilers know exactly how much of this unrolling to do to get the maximum benefit, which varies from machine to machine. Rather, write your code in a simple fashion with the fine-grained parallelism present in the inner loop, as illustrated above left, and let the compiler do its job restructuring the loops and reordering the calculations in the resulting assembly-language code to make maximum use of the pipelineing and superscalar architecture of your particular CPU.

All modern CPUs are both pipelined and superscalar. Though both the compiler and the **instruction sequencing unit** (**ISU**) on the CPU must often manipulate a given source code substantially to ensure that the multiple functional units are kept busy performing useful calculations following the strategies discussed above, the effective use of pipelined and superscalar architectures typically requires relatively little planning by the programmer. The following guideline is usually sufficient to use such architectures efficiently:

Guideline 12.3 *To make efficient use of pipelined and superscalar CPU architectures, the steps of as many innermost loops as possible in a numerical algorithm should be independent from the other steps of the loop. When this is not possible, then as many independent calculations as possible should be included within the innermost loops. Also, avoid calling short functions from within nested loops, as function calls usually prevent the CPU from identifying fine-grained parallelism; instead,* **inline** *such calculations, rewriting them directly within the loop in which they are performed.*

For the purpose of performing parallel computations using pipelined and superscalar architectures, **branch statements** (e.g., `IF` and `CASE` statements) can significantly slow things down. Modern CPUs perform what is known as **branch prediciton**, in which the CPU "guesses" (based on recent executions of the branch) which way the branch will go and initializes the pipeline to begin performing the subsequent computations based on this prediction while the comparison in the branch statement itself is still being evaluated. If the prediction turns out to be correct, the subsequent computations proceed without pause. However, if the prediction turns out to be wrong, the computations being performed in the pipeline must be terminated and started over from scratch following the correct direction of the branch, which significantly impedes the execution of the code. In order to avoid such incorrect branch predictions, the following guideline is recommended:

Guideline 12.4 *To avoid incorrect branch predictions, minimize the total number of branch statements executed by the numerical code by moving such statements outside of as many loops as possible.*

12.3.2 Vector architectures

Vector CPU architectures leverage fine-grained parallelism in a numerical algorithm in a very specific way, using a number of **vector registers** with enough bits to hold multiple floating-point or fixed-point numbers. In such systems, these vector registers may be loaded with vectors of data, and floating-point or fixed-point arithmetic may be performed on the entire vector, rather than on one number at a time. This type of parallelism is often referred to as **Single Instruction Multiple Data** (**SIMD**), as it is applicable only when the same sequence of instructions (in this case, in the innermost loop) is to be applied to many elements of the data vector(s) independently, which is often the case in the application of numerical methods to large-scale problems. This technique was developed in the early days of supercomputing in high-end Cray systems, in which the length of the vector registers was quite long [e.g., 64 double-precision (64-bit) floating-point numbers]. The technique has since trickled down to almost all modern CPUs, and is marketed (with slight variations) under various brand names, including MMX and SSE (Intel), 3DNow (AMD), and AltiVec/VMX (Motorola/IBM), though the vector length used in most such implementations is currently substantially shorter than the Cray systems for which the idea was originally developed [e.g., 4 single-precision (32-bit) floating-point numbers].

As opposed to the use of superscalar architectures, efficient use of vector processors often requires the use of some **compiler directives**[6,7] in the numerical code that identify at compile time which loops to **vectorize** (that is, which loops to execute as vector operations, as described above). Besides these flags, which are often necessary to notify the compiler which loops it is safe to vectorize, the main coding considerations for vector CPU architectures are quite similar to those for superscalar and pipelined architectures, as summarized here:

Guideline 12.5 *To make efficient use of vector CPU architectures, the numerical algorithm should be designed such that the steps of as many innermost loops as possible are independent from the other steps of the loop. Further, these vectorized loops should have as many steps as possible, with each step of the loop including as many operations as possible, so that the effect of vectorization of the loop is magnified. Also, the appropriate compiler directives should be included to notify the compiler which of the innermost loops can safely be vectorized.*

12.4 Coding for coarse-grained parallelization at the system level

As mentioned previously, many numerical algorithms may be arranged to exhibit coarse-grained parallelism, that is, independence of the various steps of certain outer loops. Such algorithms are well suited for parallel computation of large blocks of calculations in the algorithm on separate CPUs in a multi-CPU computer system, as discussed further in this section. In fact, in such multiple-CPU systems, there is a spectrum of possible approaches for arranging the CPUs physically and coördinating their communication with each other and with the main memory. This spectrum may be divided into two broad classes: **shared memory** and **distributed memory**.

[6] Compiler directives are simply comments in a specific format in a numerical code that have a particular interpretation in certain compilers, thereby extending the computer language (typically, C or Fortran) in a manner that is portable (that is, in a manner that will have no disruptive effect when using compilers that do not recognize such directives).

[7] Unfortunately, at the present time, the compiler directives for implementing vectorization, though similar (and fairly simple), are architecture dependent; no accepted platform-independent standard for such compiler directives has yet evolved, though the directives originally developed by Cray for this task would be a natural starting point for such a standard.

12.4.1 Shared-memory parallelization

In the **shared memory** configuration, multiple CPUs are configured to address the same main memory. To achieve this, two or more CPUs may be packaged on a single silicon chip (and share the same L2 and L3 cache, as in the Power5 processor illustrated in Figure 12.2), mounted next to each other on the same motherboard (and possibly share the same L3 cache), and/or mounted on multiple motherboards installed in separate boxes. Shared memory configurations are common at the scale of two or four CPUs per computer in PCs and workstations, and are often implemented at the scale 8, 16, or more CPUs per computer in high-end mainframes. Though very large shared-memory computers have been built, it typically gets expensive when scaling such systems up to large numbers of CPUs.

In shared-memory systems with relatively few CPUs, the CPUs may be arranged in a **symmetric multiprocessing** (**SMP**[8]) architecture in which each CPU can access directly the full memory of the computer at the same speed (that is, no CPU has preferential access to any particular segment, or **bank**, of the main memory). This setting is the easiest to program, but scales poorly as the number of CPUs increases. The SMP architecture can be implemented in a a few different ways. The simplest (appropriate for, say, 2 or 4 CPUs) is to attach all of the CPUs to a bus (via a memory controller chip often referred to as a **north bridge**), which, in turn, connects to all of the main memory. An alternative approach (appropriate for, say, 4 or 8 CPUs) is to interconnect each of the CPUs with each of several banks of memory via a so-called **crossbar**, which is a grid of wires arranged such that each CPU connects directly to each bank of memory in such a way that, when one CPU is communicating with one bank of memory, other CPUs communicating with other banks of memory do not slow it down. Such memory systems alleviate some of the bottlenecks suffered by bus configurations, but are rather complex to coördinate and expensive to manufacture for large systems with many CPUs and many banks of memory.

In shared-memory systems with a larger number of CPUs, the system is typically broken down into many **nodes**, with each node containing 1 to 8 CPUs in an SMP configuration with its own, "local" memory, with this memory shared (albeit at a reduced speed) with the other nodes via an **interconnect fabric**. Such an arrangement is referred to as a **non-uniform memory access** (**NUMA**[9]) architecture, and is a powerful alternative to SMP which scales to better to large numbers of CPUs. The NUMA approach was pioneered for large computer systems by SGI for use in their Origin and Altix shared-memory systems with up to 512 processors, and variants of this approach are now used commonly in multiprocessor machines from many vendors. Though still relatively easy to program (as each CPU can access the entire memory of the machine), one should take care when programming for this architecture to distribute the data in the memory and the computations on the CPUs in such a way as to minimize the amount of communication required over the (relatively slow) interconnect fabric, which can sometimes be an important bottleneck.

A standard set of compiler directives known as **OpenMP** has been developed that may be used to augment the Fortran and C programming languages in order to specify to the compiler how certain independent outer loops of a particular numerical algorithm may be broken up into blocks and calculated on the several CPUs of both SMP and NUMA shared-memory systems. Unfortunately, however, there is no platform-independent set of compiler directives or language extensions in the standards for C/C++ or Fortran77/90/95/2003 to lay out the data in the memory of NUMA machines in such a way as to minimize the communication required

[8] The sometimes-ambiguous abbreviation **SMP** is occasionally used to refer to **shared-memory parallelization** in general, though its preferred use is limited more strictly to refer to **symmetric multiprocessing**. The latter is a special case of the former, but the differences are important enough to the user to warrent clear distinction. Symmetric multiprocessing is therefore sometimes referred to as **true SMP** in order to distinguish it more clearly from non-uniform memory access (NUMA) architectures.

[9] As mentioned previously, all modern CPUs make extensive use of cache in order to accelerate references to the main memory. NUMA systems that use such cache-based CPUs efficiently must therefore, in addition to sharing the main memory between nodes, properly account for the fact that there may be multiple copies of a particular element of memory in the various memory caches distributed between different nodes. The classification **cache-coherent NUMA** (**ccNUMA**) is often used to indicate that this issue is taken care of in hardware by the memory controllers of the system via extra communication between nodes, which in fact is how most NUMA systems are configured today.

in the subsequent calculations over the interconnect fabric. Note that a specialized, well-designed variant of the Fortran standard, called **High Performance Fortran** (**HPF**), did appear in the early 1990s with such directives incorporated. Unfortunately, most of the useful data layout commands that were introduced into this variant of Fortran did not migrate into the Fortran95 or Fortran2003 standards. Thus, to lay out data appropriately for NUMA shared-memory architectures, at the present time one must generally resort to **vendor-specific language extensions** (such as those provided in SGI Fortran or C) or creative coding workarounds (also known as **hacks**). One such hack that can be used to ensure that the appropriate blocks of data are stored on the appropriate nodes is to allocate pointers from within a parallelized loop on the various nodes of the NUMA machine. If necessary, these pointers may later be assembled into larger data structures to make block matrices, with each block residing in the memory of the appropriate node where the computations that reference it most heavily will be performed.

Guideline 12.6 *To make efficient use of multiple-CPU computers implementing shared-memory parallelization [either in symmetric multiprocessing (SMP) or non-uniform memory access (NUMA) configurations], outer loops in the numerical algorithm should be designed/identified in which each step of the loop is independent from the other steps of the loop. Where possible, the algorithm should be reordered such that this parallelization is as coarse as possible (that is, the loops to be executed in parallel on different CPUs should, if possible, be the outermost loops). OpenMP compiler directives may be included to direct the compiler how to split up these loops in a maximally efficient fashion. Further, if designing for a NUMA architecture, lay out the data on the nodes in such a way as to minimize communication over the interconnect fabric using either HPF, vendor-specific language extensions, or coding hacks (as described above).*

12.4.2 Distributed-memory parallelization

In the **distributed memory** configuration, multiple independent computers are interconnected (for example, via ethernet) and exchange information only when explicitly instructed to do so by the numerical code. Such configurations may easily and inexpensively be scaled to systems with thousands of CPUs. Distributed memory computer systems may simply be a **network of workstations** (**NOW**), that is, a coördinated network of several independent desktop computers (a.k.a. **nodes**) communicating over ethernet and focused to work together on a useful task when such computers aren't otherwise being used by their owners. Note that a NOW is often **heterogeneous** (that is, the nodes are often dissimilar), with the number of available nodes (as well as the number of other jobs running concurrently on each node) possibly changing fairly often. A strategy must therefore be designed carefully to achieve the approporiate **load balancing** (that is, an appropriate distribution of computational tasks assigned to the various CPUs) under these dynamically-changing conditions so that all of the available nodes stay busy doing useful calculations the majority of the time without too much **communication overhead** (that is, time wasted while sending information back and forth between the nodes).

Alternatively, the network may be built as a **cluster** of dedicated, independent, stripped down (i.e., inexpensive), often identical nodes interconnected with each other using either standard gigabit ethernet or any of a number of higher-speed alternatives (e.g., InfiniBand, Myrinet, Quadrics, Dolphin, 10GigE, etc.). When a cluster is built from inexpensive commodity hardware with a dedicated interconnect network using open source software (e.g., Linux), it is commonly referred to as a **beowulf cluster**.

Following the distributed-memory approach, there is a range of possible coding strategies, including **master/slave** and **domain decomposition**:

• In the **master/slave** strategy, a **master node** runs the main numerical code, creating and dynamically updating lists of both the **worker processes** (that is substantial independent blocks of upcoming computations that need to be performed) and the **computational resources** that are available to perform these blocks of computations. The worker processes are like function calls, as they include a description of the necessary input data, the sequence of instructions to be executed, and the results to be returned. The master node distributes

the worker processes to the computational resources as they become available, combines the results they return to determine new worker processes, etc. The primary strength of the master/slave strategy is its ability to adjust dynamically to nodes going online and offline as well as the speed of the various nodes changing from time to time as other jobs on these nodes begin and end. Thus, this strategy is particularly well suited to the distribution of computationally-intensive tasks on a heterogeneous NOW or on a cluster with many other jobs coming on and off various subsets of the cluster nodes at unpredictable times and competing for the computational resources. The primary weakness of the master/slave strategy is that it suffers from a relatively large communication overhead at the beginning and end of every process.

• The **data decomposition** strategy is, in a sense, more democratic. With this strategy, the same numerical code is run on each of the nodes, with a flag on each node (indicating the node number) used to instruct that node which particular subset of the data[10] to work on. From time to time, these computations are synchronized and data is exchanged between the nodes. In some numerical algorithms, the amount of information that needs to be exchanged between nodes at such synchronization points is relatively small (for example, the information on the state of the system only within a few gridpoints of the physical boundary between neighboring subdomains); such algorithms are particularly well suited to calculation via the data decomposition method, as they pay a relatively small communication overhead. In other numerical algorithms, however, the amount of information that needs to be exchanged between nodes at such synchronization points is sometimes relatively large (for example, when, in order to perform the upcoming calculations efficiently, the entire data matrices must be redistributed across the cluster[11]); such algorithms are less well suited to calculation via the data decomposition method, as they pay a relatively large communication overhead. The primary strength of the data decomposition strategy is its ability to handle large datasets that would otherwise take a long time to transmit between nodes by reducing to a bare minimum the communication overhead (when using those particular numerical algorithms for which the data decomposition strategy is well suited). Note that the data decomposition strategy may in fact be used for problems that are so large that the entire database does not even fit into the memory of any single node. The primary weakness of the data decomposition strategy is its inability to easily reconfigure the computation to account for dynamically changing availability of the computational resources.

Following either the master/slave or domain decomposition strategy, coding for distributed-memory architectures requires significant effort and care. Nonetheless, distributed memory parallelization is the most cost-effective way to scale up computations to thousands of CPUs for computational grand-challenge problems. Two standard libraries of commands have been developed that may be used to augment the C and Fortran programming languages for distributed-memory parallelization: **PVM** and **MPI**.

• The **Parallel Virtual Machine** (**PVM**) standard was developed first (in the early 1990s) and provides particularly powerful tools for the master/slave configuration in a heterogeneous NOW. A good resource on this standard is *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing* (1994) by Geist, Beguelin, Dongarra, Jiang, Manchek, & Sunderam.

• The **Message Passing Interface** (**MPI**) was developed more recently and provides additional flexibility for the data decomposition configuration, for which it has become the dominant library of choice. A good resource on this standard is *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (1999) by Gropp, Lusk, & Skjellum.

To make efficient use of distributed memory computer systems, the parallelism of the numerical algorithm must be as coarse as possible, and the algorithm must be designed from the ground up to minimize the

[10] This subset of the data often represents a particular subset of a physical domain; the art of dividing up a physical domain into nearly uniformly sized subdomains with a minimum area between these subdomains (in order to minimize the communication overhead in a simulation following the data decomposition approach) is commonly referred to as **domain decomposition**.

[11] A typical example of this is a calculation on 3D Cartesian "*x*-*y*-*z*" grid which, for the purpose of efficient execution of the numerical algorithm, must switch from *x*-*y* planes of data stored on each node in one section of the code to *x*-*z* planes of data stored on each node in another section of the code.

communication overhead. Using the programming tools available today, the communication between nodes must be explicitly coördinated by PVM or MPI commands in the Fortran or C program, as effective tools to automate this type of parallelization are not yet available. For a more complete introduction to distributed-memory parallelization with these tools, the reader is referred to the texts cited above.

12.5 Performance tuning

Once an efficient numerical algorithm (such as those discussed in this text) is implemented in numerical code (following the several guidelines presented in this chapter), debugged, and recompiled from scratch with aggressive optimization enabled via the appropriate **compiler flags**[12], it will typically run fairly quickly without any further hand tuning required. A final step to ensure that the compiler has spotted and leveraged all of the opportunities that you can see for parallel computation in the code, and to ensure that the code isn't getting slowed down anywhere else that it perhaps shouldn't, is to **profile** the code to identify precisely where it is spending the majority of its time. The easiest way to accomplish this is to analyze the execution of the code for a few minutes with a **profiler**[13], which is a process that runs passively in the background on the computer while the code is executing and records at regular intervals (typically every 0.01 seconds) what line of code is currently being executed. In such a way, you can get a very good idea if there are any statements or loops that your code appears to be having particular challenges with, then do your best to restructure these loops or identify with compiler directives how either fine-grained or coarse-grained parallelization is to be applied to such loops. After applying such optimizations where necessary, if/when you ultimately find that your code is not hung up anywhere in particular, and/or you find that your code is spending a substantial fraction of its time executing third-party subroutines that you have called a minimum number of times and that you are confident have themselves been highly optimized by people who know your CPU architecture well (e.g., BLAS and LAPACK), then you can be confident that your job of optimizing the code is essentially finished.

12.6 Summary

Modern computers are by no means simple; in fact, they get more and more sophisticated with every generation. However, the considerations necessary to code modern computers effectively may be summarized with a small number of reasonable "high performance computing guidelines", as we have attempted to outline in this chapter, and understood in terms of a fairly rudimentary description of how such computers work.

When describing both modern computer architectures and how these architectures may be used effectively by numerical algorithms, the recurrent theme is that there are a broad spectrum of strategies available:

• In terms of the accessibility of data on the computer system, the spectrum ranges from the CPU registers, to the L1, L2, and L3 cache, to the shared main memory on an SMP node, to the shared main memory on other nodes of a NUMA architecture, to the memory on other nodes of a distributed memory network or cluster, to, ultimately, the data stored on hard disks and remote shared file systems.

• In terms of the parallelization of a numerical algorithm, the spectrum ranges from the fine-grained parallelization that may be leveraged on each individual CPU by pipelined, superscalar, and vector architectures, to the coarse-grained parallelization that may be leveraged by multiple CPU systems that share the same memory, to the still coarser-grained parallelization that may be leveraged by networks or clusters of interconnected,

[12] On a unix machine, aggressive optimization is typically enabled with the `-O3` flag at compile time, which usually enables all auto-parallelization and auto-vectorization features the compiler is capable of.

[13] On a unix machine, profiling is achieved by recompiling the code with the `-p` or `-pg` flag, running the code, then analyzing the result with the **prof** or **gprof** command. Don't forget to recompile the code *without* these flags once finished optimizing the code, as profiling slows down the exectution of the code a bit.

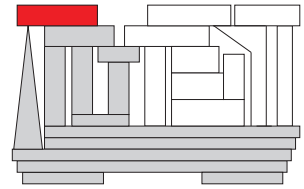
independent computers that exchange information only when explicitly instructed to do so by the numerical code. Note that all three types of parallelization may be leveraged at the same time.

Incorporating such knowledge of modern computer systems into the codes and algorithms that you develop will lead to the fastest possible execution of your numerical code, allowing you to analyze much bigger systems, or allowing you to resolve a given system with a much greater degree of fidelity, than you would otherwise be able to do with the computational hardware that you have at your disposal.

Exercises

References

Dowd, K, & Severance, C (1998) *High Performance Computing*. O'Reilly.



Chapter 13

Turbulence simulation: a case study

Contents

13.1 The incompressible Navier-Stokes equation (NSE) . . . 407
13.1.1 Notation . . . 407
13.1.2 Continuous (PDE) form of the NSE . . . 407
13.1.2.1 Triply periodic case . . . 408
13.1.2.2 Channel flow case . . . 408
13.1.2.3 Duct flow case . . . 409
13.1.2.4 Cavity flow case . . . 409
13.1.3 Conservation properties of the continuous NSE . . . 409
13.1.4 Overall strategy for numerical implementation . . . 409
13.2 Spatial discretization . . . 410
13.2.1 Stretching and staggering of the grid in the wall-bounded direction(s) . . . 410
13.2.2 Second-order finite volume formulations of the spatial derivatives of the NSE . . . 411
13.2.2.1 Channel flow case . . . 411
13.2.2.2 Duct flow case . . . 414
13.2.2.3 Cavity flow case . . . 414
13.2.3 Conservation properties of the spatially-discretized NSE . . . 415
13.2.3.1 Discrete conservation of mass . . . 415
13.2.3.2 Discrete conservation of momentum . . . 416
13.2.3.3 Discrete conservation of energy . . . 417
13.3 Temporal discretization - the fractional step algorithm . . . 420
13.3.1 All viscous terms implicit . . . 421
13.3.2 All y-derivative terms implicit . . . 421
13.3.3 All viscous terms implicit in the triply periodic case . . . 423
13.3.4 All y-derivative terms implicit in the channel-flow case . . . 424
13.3.5 Shared-memory parallelization using OpenMP . . . 426
13.3.6 Distributed-memory parallelization using MPI . . . 426
13.4 Characterizing the statistics of turbulence . . . 426
13.5 The visualization of turbulence . . . 426

13.6 Large eddy simulation . . . 426
13.7 Extensions . . . 426
13.7.1 Passive scalars . . . 426
13.7.2 Active scalars and the Boussinesq approximation of bouyancy . . . 426
13.7.3 Immersed boundary methods . . . 426
13.7.4 Coordinate transformation methods . . . 426
13.7.5 Noise generation . . . 427
13.A Diablo . . . 428
Exercises . . . 428

Chapters 1-10 of this text presented a deliberate sequence of stable, accurate, and efficient numerical methods for a variety of subproblems that must often be addressed in the numerical solution of challenging problems in science and engineering, together with just enough analysis for the computational scientist reading this text to understand the fundamentals of how these algorithms work. Chapter 11 introduced how these methods may be combined in a straightforward manner to simulate some simple PDE systems. We saw that such simulations can quickly challenge the capabilities of the hardware you have at your disposal to perform the computations required. Chapter 12 thus introduced the essential elements of high performance computing necessary to extract the maximum performance from modern computational hardware.

In the present chapter, we synthesize further the methods presented thus far to simulate what is commonly identified as a **computational grand challenge** problem in **computational fluid dynamics** (**CFD**), that is, the **direct numerical simulation** (**DNS**) and **large eddy simulation** (**LES**) of incompressible flows in simple 3D (or, as a special case, 2D) domains. The case study presented in this chapter should not be considered as an end result of your study of numerical simulation techniques, but rather as an appropriate intermediate step towards the simulation of other large-scale complex systems of interest. The present chapter is included in this text primarily to illustrate how the methods presented thus far may be used efficiently in concert. Some of the significant algorithms leveraged by the code developed in this chapter include those for

- the direct solution of banded linear systems (§2.3),
- the iterative solution of sparse (but not banded) linear systems (§3.2),
- the discretization of spatial derivatives using finite-difference (FD) methods (§8.1),
- the representation of spatial derivatives using spectral methods (§5.2.1), and
- the time marching of an ODE discretization of a PDE system with both linear terms and nonlinear terms using mixed RK/CN methods (§**??**),

all of which are implemented while paying careful attention to a variety of high-performance computing issues (§12) in order to ensure the resulting numerical code runs efficiently on modern computers. Finally, it is important to note that the code presented here has been developed with a carefully-chosen balance of efficiency, readability, and extensibility in mind, as should any any large-scale simulation code of this sort. That is, when writing a large simulation code, one should always consider both the short-term efficient simulation of the problem(s) at hand and the long-term maintainability of the code developed (i.e., continuing to be able to run it years later, on different computers with different operating systems and installed software libraries), as well as the extensibility of this code, both by the author and by others, to related problems for which it might be well suited. This means documenting the code well, as we have attempted to do both here and in the `README` text files that accompany the code. These longer-term objectives often take a substantial amount of discipline (both personal and institutional) to complete, especially if the short-term objectives have tight deadlines, which is often the case. The long-term payoff of this discipline can be quite significant.

13.1 The incompressible Navier-Stokes equation (NSE)

13.1.1 Notation

In the development of effective numerical methods for the solution of the incompressible Navier-Stokes equation, we will make use of three distinct notations. Which notation is used in any particular equation is generally self-evident.

In the initial presentation of the continuous Navier-Stokes equation (in §13.1), we use $\{u_1, u_2, u_3\}$ for the velocity components and $\{x_1, x_2, x_3\}$ for the coördinate directions. This facilitates the use of summation notation, e.g., $\partial u_j/\partial x_j \triangleq \partial u_1/\partial x_1 + \partial u_2/\partial x_2 + \partial u_3/\partial x_3$, and saves the letters v and w for different velocity vectors.

In the subsequent presentation and analysis of the spatial discretization of the NSE (in §13.2), the multiple subscripts that would arise following the above notation would get confusing. Thus, we shift in this section to the notation $\{u, v, w\}$ for the velocity individual components, $\{x, y, z\}$ for the coördinate directions, and $\{i, j, k\}$ for gridpoint enumeration.

Finally, in the presentation and discussion of the temporal discretization of the spatially-discretized NSE (in §13.3), we employ the notation $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ for the discretized velocity components. In this notation, the boldface denotes the vector formed by assembling the spatial discretization of the continuous flow variables on all of the gridpoints into a vector, which is in fact how the data is stored in the computer memory. In the computational implementation, it is most convenient to enumerate this vector with a separate index for each coördinate direction (e.g., `u1(i,j,k)`).

Finally, note that we also make use of the notation $\vec{u}$ (in the spatially-continuous case) and $\vec{\mathbf{u}}$ (in the spatially-discrete case) to denote the collection of all three velocity components; this vector notation extends naturally to $\vec{\psi}$, $\vec{\phi}$, $\vec{x}$, etc.

13.1.2 Continuous (PDE) form of the NSE

The equation governing the systems considered in this case study is the incompressible **Navier-Stokes equation** (**NSE**), given (in summation notation, for $i \in [1,2,3]$ and $j \in [1,2,3]$, normalized[1] such that $\rho = 1$) by

$$\frac{\partial u_i}{\partial t} = -\frac{\partial u_j u_i}{\partial x_j} + \mu \frac{\partial^2 u_i}{\partial x_j^2} - \frac{\partial p}{\partial x_i} + \psi_i, \tag{13.1a}$$

$$0 = \frac{\partial u_j}{\partial x_j}, \tag{13.1b}$$

in a 3D (or, as a limiting case, 2D) rectangular domain Ω defined such that $-L_x/2 \le x \le L_x/2$, $-L_y/2 \le y \le L_y/2$, $-L_z/2 \le z \le L_z/2$, with known initial conditions $\vec{u}(t=0) = \vec{u}_0$, known boundary conditions $\vec{u} = \vec{\phi}$ in 0, 1, 2, or 3 spatial directions, and periodic boundary conditions on the unknowns $\{\vec{u}, p\}$ in the remaining spatial directions. The code described in this chapter will be suitable for the following 4 cases:

- the case with 3 periodic directions, which we will refer to as the **triply periodic** case,
- the case with 2 periodic directions, which we will refer to as the **channel flow** case,
- the case with 1 periodic direction, which we will refer to as the **duct flow** case, and
- the case with 0 periodic directions, which we will refer to as the **cavity flow** case.

[1] The assumption that the density $\rho = 1$ in (13.1a) may be relaxed if we replace the pressure p with a symbol understood to denote the **density-normalized pressure**, p/ρ, and if we replace the **dynamic viscosity** μ with the **kinematic viscosity** $\nu = \mu/\rho$. We have not opted to do this in the present chapter, primarily because it is difficult to distinguish the velocity component v and the kinematic viscosity ν in print using the present font.

The first subequation of the NSE[2], (13.1a), is referred to as the **momentum equation** (with 3 components, one in each coördinate direction), whereas the second subequation of the NSE, (13.1b) is referred to as the **continuity equation**. The momentum equation is an **evolution equation** that is marched in time, whereas the continuity equation is a **constraint equation** that the velocity field must satisfy at each instant. A PDE system of this sort, comprised of both evolution equations and constraint equations, is sometimes referred to as a **differential algebraic equation** (**DAE**).

While a significant application in its own right, the Navier-Stokes equation is a valuable problem to consider as a canonical model of PDE systems in general. Though fairly simple to derive and express, Navier-Stokes systems often exhibit chaotic, multiscale dynamics (a.k.a. **turbulence**), the numerical representation of which requires considerable care and attention to a variety of subtle issues to insure the stability, accuracy, and efficiency of the numerical simulation.

13.1.2.1 Triply periodic case

The most fundamental case of interest when looking at turbulence is the decay of unforced **homogenous** (that is, statistically invariant from one *spatial point* to another) **isotropic** (that is, statistically invariant from one *direction* to another) turbulence. In the lab, such homogeneous isotropic turbulence might be approximated by passing a grid through an otherwise quiescent fluid to provide an initial quasi-random agitation, then watching the agitation of this box of turbulence decay. In the cylindrical coördinates of a coffee cup, such an initial quasi-random agitation is commonly provided with a **swizzle stick**.

In order to facilitate the study of the fundamental statistical spectrum of homogeneous isotropic turbulence computationally, one commonly provides some continuous random excitation to a triply periodic flowfield at the largest length scales in the domain (in an unsteady, approximately homogenous, isotropic fashion), then averages in time the statistics of this flow as the energy of it cascades over the higher wavenumbers.

Another case of interest in the triply periodic case is **shear-driven turbulence**. This flow may be studied by replacing $\vec{u}$ with $\vec{U}+\vec{u}$ in the governing equation (13.1), where the background flow profile $U_i(x_1,x_2,x_3) = U_1(x_2)\delta_{i1}$ is specified, then grouping all terms involving $\vec{U}$ into the forcing term $\vec{\psi}$ and simulating with a minor embellishment of the code. In the simple case that $U_1(x_2) = cx_2$, this flow is homogeneous but not isotropic.

13.1.2.2 Channel flow case

The case of channel flow is the most fundamental realization of wall-bounded turbulence. In the present study, the channel flow case is represented by taking x_2 as the wall-normal direction and by supplying a pressure gradient $\psi_i = -P_x\delta_{i1}$, where $P_x < 0$, forcing the flow in the positive x_1 direction. If the computational box is chosen to be large enough in L_x and L_z (we typically normalize $L_y = 2$ in this case), the (nonphysical) periodic boundary conditions in the x_1 and x_3 directions have minimal effect on the statistics of interest of the near-wall turbulence.

Note that one can also use no-slip BCs at the lower wall and no-shear BCs at the upper wall, together with a "sponge" region (incorporating an artificial "gentle" RHS forcing with minimal upstream influence within the so-called "sponge") in order to connect a convective outflow condition to a specified inflow profile, thus facilitating the use of a periodic channel flow code to effectively simulate a spatially-developing boundary-layer flow.

[2] As a reflection of our overall viewpoint of (13.1) defining a single system, we prefer to refer to the NSE in the singular; this viewpoint is reinforced by writing the PDE (13.1) in the form

$$E\frac{\partial \vec{q}}{\partial t} = \mathcal{N}(\vec{q}) \quad \text{where} \quad \vec{q} = \begin{pmatrix} \vec{u} \\ p \end{pmatrix} \quad \text{and} \quad E = \begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix}.$$

13.1.2.3 Duct flow case

The case of duct flow is also easily treated by the present framework. In the present code, the duct flow case is studied by taking x_2 and x_3 as the wall-bounded directions and, as in the channel flow case, supplying a pressure gradient $\psi_i = -P_x \delta_{i1}$ forcing the flow in the positive x_1 direction. Interesting corner effects may be studied in this flow that are relevant to many engineering flows of practical relevance.

13.1.2.4 Cavity flow case

Finally, the present code is also easily extended to the cavity flow case, with all directions bounded by walls and often, to make the flow interesting, nonzero boundary conditions on at least one wall. A canonical problem in this case is given by constantly translating the "lid" (that is, the surface of Ω in the positive x_2 dirction) in the x_1 direction, thereby driving a flow within the domain.

13.1.3 Conservation properties of the continuous NSE

Following closely the previous analysis of Burgers' equation [see (11.30) and the surrounding discussion], it is seen that both the nonlinear and pressure gradient terms of the NSE are energy conserving, whereas the viscous term, which dominates at large wavenumbers, constantly drains energy from the system. To see this, take the RHS forcing $\vec{\psi} = 0$ and boundary conditions $\vec{\phi} = 0$ in any of the 4 cases mentioned above, take the inner product of u_i with (13.1a), integrate over Ω, integrate by parts, and apply (13.1b), which leads to

$$\frac{\partial}{\partial t} \int_\Omega \frac{|\vec{u}|^2}{2} d\vec{x} = -\mu \int_\Omega |\vec{\nabla}\vec{u}|^2 d\vec{x} \leq 0 \quad \text{where} \quad |\vec{\nabla}\vec{u}|^2 = \sum_{i=1}^{3} \sum_{j=1}^{3} \left(\frac{\partial u_i}{\partial x_j} \right)^2 .$$

Thus, in any of the 4 cases mentioned above, in order for the flow to not decay back to $\vec{u} = 0$, some sort of driving force must be applied (that is, $\vec{\psi} \neq 0$, $\vec{\phi} \neq 0$, or both). Some of the problems of particular interest are outlined above.

Note in particular that the pressure gradient term $\partial p / \partial x_i$ in (13.1a) represents what might be called a **workless force** on the interior of the domain; over any volume, the integral of the force $\partial p / \partial x_i$ times the velocity u_i, via integration by parts and the continuity equation, only picks up a possible contribution from the boundary. To rephrase, the force $\partial p / \partial x_i$ might be said to be **orthogonal to the divergence free manifold** of u_i on the interior of the domain Ω. Yet another way of interpreting this is that the pressure gradient itself may be thought of as a **Lagrange multiplier** (see §21.1.1) which acts to enforce (13.1b) at every instant in the evolution of (13.1a), thereby keeping u_i on this divergence free manifold.

13.1.4 Overall strategy for numerical implementation

To solve the problems described above computationally, the continuous flow field must be discretized on a finite set of points in space, and the PDE governing the flow (that is, the NSE) approximated as a constrained ODE (a.k.a. a **descriptor system**) on this finite set of points. Further, the resulting ODE must be marched in time using discrete time steps. To minimize the expense of the computation, one desires to use as few spatial points as possible and as large time steps as possible while maintaining accuracy (in both space and time) and stability of the simulation. Since the flow is periodic in those directions not bounded by walls, so that there is no inflow or outflow from the domain, it is especially important when the viscosity is made small that numerical errors due to the spatial and temporal discretization of the physical problem do not accumulate in a way which causes the simulation to be unstable. Subject this restriction, a scheme with high spatial accuracy is desired. Finally, with a particular spatial discretization, it is found that certain terms of the governing equation have more restrictive time step limitations than do others in the time-marching algorithm.

The most restrictive terms in any given case should be taken implicitly to allow for stability at "large" time steps (which, however, must be kept small enough to ensure accuracy of the computation), while other less restrictive terms may be taken explicitly. These issues guide the choice of spatial and temporal discretizations of the current problem, which are discussed in detail below.

13.2 Spatial discretization

The spatial discretization used in the present code is a hybrid spectral/FD strategy. For simplicity, all spatial derivatives in all 4 cases are calculated with spectral methods in the spatially periodic direction(s) and with second-order central FD methods in the wall-bounded direction(s).

The numerical grid is chosen to be equispaced and unstaggered in the spatially periodic direction(s), allowing spectral methods (specifically, finite Fourier series expansions) to be used to accurately and efficiently compute all spatial derivatives in these direction(s) [see §5.2.1] at the corresponding gridpoints.

In the wall-bounded direction(s), on the other hand, the grid is chosen to be

- **stretched** (that is, with gridpoints clustered near the boundary of the domain in order to resolve the small-scale flow fluctuations of the flow near the walls), and
- **staggered** (that is, with the various flow variables discretized on sets of points that are offset from one another in order to tightly couple the various flowfield fluctuations governed by the discretized NSE).

Further motivation for and details of this grid stretching and staggering are described in §13.2.1.

Once the (stretched and staggered) grid is defined in the wall-bounded direction(s), a **finite volume** approach[3] is proposed to determine approximate expressions for the necessary derivatives of the flow variables in the wall-bounded direction(s), as outlined in §13.2.2.

13.2.1 Stretching and staggering of the grid in the wall-bounded direction(s)

The present code stretches the numerical grid in the wall-bounded direction(s) using a hyperbolic tangent stretching function, as illustrated in Figure 8.1. We will illustrate this stretching (as well as the subsequent staggering) by discussing stretching and staggering of the grid in the y direction, which is the wall-bounded direction in the channel flow case. Note that the duct flow case additionally stretches and staggers the grid in the z direction, and the cavity flow case additionally stretches and staggers the grid in both the x and z directions; both cases will be discussed further at the end of this section.

We initialize the stretched and staggered grid in the channel flow by first defining

$$y_j = \tanh\left(C\left(\frac{2\,(j-1)}{NY}-1\right)\right), \quad y_{j+1/2} = \frac{1}{2}\left(y_j + y_{j+1}\right). \tag{13.2a}$$

A stretching parameter of $C = 1.75$ is appropriate for many of the flows of interest. The subscript j (for the integer $j \in [0,\ldots,NY+2]$) is used to enumerate what we will call the **base grid**, whereas the subscript $j+1/2$ (for the integer $j \in [0,\ldots,NY+1]$) is used to enumerate what we will call the **fractional grid** (that is, the set of gridpoints midway between the points in the base grid). Once this grid is appropriately scaled,

- the component of velocity normal to this wall, v, will be discretized on the base grid, wheras
- the components of velocity parallel to this wall, u and w, as well as the pressure, p, will be discretized on the fractional grid.

[3] That is, a FD approach designed such that certain discrete conservation properties hold via simple telescoping arguments, as discussed further in §13.2.3.

The primary motivation for staggering the wall-normal component of velocity from the other flow variables in this way is to couple the pressure at the nodes with j even to the pressure at the nodes with j odd. This is a natural result of a staggered grid, but is not the case in non-staggered configurations[4].

Next, we scale this grid to put the desired gridpoints on the wall (at $y = \pm L_y/2$). There are essentially two choices that may be made here: scale the grid such that two planes of the base grid coincide with the wall (e.g., the planes $j = 1$ and $j = NY + 1$), or scale the grid such that two planes from the fractional grid coincide with the wall (e.g., the planes $j = 3/2$ and $j = NY + 1/2$). There are perhaps equally convincing arguments that can be made for either choice. In the present work, we make the latter of these two choices. This choice will be especially convenient when applying the multigrid strategy to the Poisson equation for the pressure update in the fractional step algorithm, to be presented later in this chapter. Selecting $C_1 = 2y_{NY+1/2}/L_y$, we rescale the grid such that

$$y_j \leftarrow y_j/C_1, \quad y_{j+1/2} \leftarrow y_{j+1/2}/C_1 \quad \forall j. \tag{13.2b}$$

Once the grid is rescaled in this fashion, $j = 3/2$ corresponds to the lower wall (at $y = -L_y/2$) and $j = NY + 1/2$ corresponds to the upper wall (at $y = L_y/2$). To simplify the subsequent wall-normal FD calculations, we also make the following definitions:

$$\Delta y_{j+1/2} = y_{j+1} - y_j, \quad \Delta y_j = y_{j+1/2} - y_{j-1/2} = \frac{1}{2}\left(\Delta y_{j+1/2} + \Delta y_{j-1/2}\right).$$

For notational convenience in the numerical implementation, which does not allow fractional indices, we also define

$$y_{f_j} = y_{j+1/2}, \quad \Delta y_{f_j} = \Delta y_{j+1/2}.$$

Also, note that numerical implementation requires standard **ASCII** (**American Standard Code for Information Interchange**) symbols (that is, non-Greek), so, e.g., Δy and Δy_f are denoted in the code as `DY` and `DYF`. The resulting stretched and staggered grid in a single wall bounded direction, y, as appropriate for the channel flow case, is illustrated in Figure X. Extension to the duct and cavity flow cases is straightforward, as illustrated in Figure Y.

13.2.2 Second-order finite volume formulations of the spatial derivatives of the NSE

13.2.2.1 Channel flow case

To interpolate the flow quantities to the adjacent gridpoints when necessary, the following interpolation formula is used for v

$$\overline{v}_{j+1/2} = \frac{1}{2}\left(v_{j+1} + v_j\right) \qquad \text{(“fully”-second order)}, \tag{13.3}$$

[4]Consider the discretization of the NSE in the channel, duct, or cavity case with a nonstaggered (and unstretched) grid in the wall-bounded direction(s). Label the gridpoints for which the sum of the indices enumerating the FD directions are even as red, and the others as black. Then the discretization of the NSE on each red gridpoint depends only on the pressure at the neighboring black gridpoints, whereas the discretization of the NSE on each black gridpoint depends only on the pressure at the neighboring red gridpoints. That is, the pressure at the red points and the black points are completely decoupled. This can ultimately lead to artificial oscillations in the pressure field in the numerical solution. Gently stretching the grid in the wall-bounded direction(s) fails to alleviate this phenomenon significantly; however, staggering the numerical grid in the FD direction(s) removes this problem completely.

and the following interpolation formulae are used for both u and w (illustrated here for u)

$$\overline{u}_j = \frac{1}{2}\left(u_{j+1/2} + u_{j-1/2}\right) \qquad \text{("quasi"-second order),} \tag{13.4}$$

$$\breve{u}_j = \frac{1}{2\Delta y_j}\left(\Delta y_{j+1/2} u_{j+1/2} + \Delta y_{j-1/2} u_{j-1/2}\right) \qquad \text{("quasi"-second order),} \tag{13.5}$$

$$\breve{\breve{u}}_j = \frac{1}{2\Delta y_j}\left(\Delta y_{j-1/2} u_{j+1/2} + \Delta y_{j+1/2} u_{j-1/2}\right) \qquad \text{("fully"-second order).} \tag{13.6}$$

Interpolation for p is not required in the fractional grid formulation. As, by definition, $y_{j+1/2}$ is midway between y_j and y_{j+1}, the interpolation formula for $\overline{v}_{j+1/2}$ is second-order accurate. As y_j is *not* midway between $y_{j+1/2}$ and $y_{j-1/2}$ due to the grid stretching, only the interpolation formula for $\breve{\breve{u}}_j$ is truly second-order accurate. The formula for $\overline{u}_j$ and $\breve{u}_j$ are only second-order accurate in the sense that, as NY is increased with the stretching function (13.2a) fixed, $\Delta y_{j+1/2}/\Delta y_{j-1/2} \to 1$, and thus both forms approach a second order form. We will make use of the $\overline{u}_j$ and $\breve{u}_j$ interpolation forms in the finite volume formulation that follows.

The motivation for using interpolation forms which are only second order accurate in the sense described above stems from conservation issues, which are described in the following section. Though the "proper" second-order interpolation formula $\breve{\breve{u}}_j$ could be used everywhere (and, in fact, would be slightly more accurate on a finite grid), the discretization error of such an interpolation formula results in spurious sources and sinks of energy on a marginally-resolved stretched grid, which can lead to numerical instabilities. Proper use of the above interpolation formulae prevents discretization errors from contributing to the total energy of the flow, even on a stretched grid. Note that a sufficiently smooth grid stretching function is used to minimize the inaccuracies caused by these interpolation formulae for reasonable values of NY.

With the spatial discretization of the flow quantities described above, the individual momentum equations are solved at the corresponding velocity points and the continuity equation is enforced on the cells surrounding at the pressure points. The spatial discretization of all derivatives required in the channel case are now made precise. The first component of the momentum equation, (13.1a), to be evaluated at $(i, j+1/2, k)$ for integer values of i, j, and k, is discretized as follows (suppressing the dependencies of all flow variables on the RHS on i and k for notational clarity)

$$\left.\frac{\partial u}{\partial t}\right|_{(i,j+1/2,k)} = -\left[\frac{\delta_s u^2_{j+1/2}}{\delta x} + \frac{(v\overline{u})_{j+1} - (v\overline{u})_j}{\Delta y_{j+1/2}} + \frac{\delta_s (wu)_{j+1/2}}{\delta z}\right] - \frac{\delta_s p_{j+1/2}}{\delta x} + \psi_1$$
$$+\mu\left[\frac{\delta_s^2 u_{j+1/2}}{\delta x^2} + \left(\frac{u_{j+3/2} - u_{j+1/2}}{\Delta y_{j+1}} - \frac{u_{j+1/2} - u_{j-1/2}}{\Delta y_j}\right)/\Delta y_{j+1/2} + \frac{\delta_s^2 u_{j+1/2}}{\delta z^2}\right].$$

Note that all derivatives in the x and z directions are computed in Fourier space according to

$$\widehat{\frac{\delta_s q}{\delta x}} = \mathrm{i}k_x \hat{q} \qquad \widehat{\frac{\delta_s q}{\delta z}} = \mathrm{i}k_z \hat{q} \qquad \widehat{\frac{\delta_s^2 q}{\delta x}} = -k_x^2 \hat{q} \qquad \widehat{\frac{\delta_s^2 q}{\delta z}} = -k_z^2 \hat{q},$$

where the hat indicates the Fourier transform in the x and z directions with corresponding wavenumbers k_x and k_z, q is an arbitrary flow quantity, and the s subscript is used to emphasize that the derivative is evaluated spectrally. Note also that the convective terms involving derivatives in the y direction are computed with "quasi"-second-order accurate FD formulae, whereas the viscous terms involving derivatives in the y direction are evaluated with "fully"-second-order accurate FD formulae. The second component of the momentum equation, evaluated at (i, j, k) for integer values of i, j, and k, is discretized as follows

$$\left.\frac{\partial v}{\partial t}\right|_{(i,j,k)} = -\left[\frac{\delta_s (\breve{u}v)_j}{\delta x} + \frac{\overline{v}^2_{j+1/2} - \overline{v}^2_{j-1/2}}{\Delta y_j} + \frac{\delta_s (\breve{w}v)_j}{\delta z}\right] - \frac{p_{j+1/2} - p_{j-1/2}}{\Delta y_j} + \psi_2$$
$$+\mu\left[\frac{\delta_s^2 v_j}{\delta x^2} + \left(\frac{v_{j+1} - v_j}{\Delta y_{j+1/2}} - \frac{v_j - v_{j-1}}{\Delta y_{j-1/2}}\right)/\Delta y_j + \frac{\delta_s^2 v_j}{\delta z^2}\right].$$

The third component of the momentum equation, evaluated at $(i, j+1/2, k)$ for integer values of i, j, and k, is discretized as follows

$$\left.\frac{\partial w}{\partial t}\right|_{(i,j+1/2,k)} = -\left[\frac{\delta_s (uw)_{j+1/2}}{\delta x} + \frac{(v\overline{w})_{j+1} - (v\overline{w})_j}{\Delta y_{j+1/2}} + \frac{\delta_s w^2_{j+1/2}}{\delta z}\right] - \frac{\delta_s p_{j+1/2}}{\delta z} + \psi_3$$
$$+\mu\left[\frac{\delta_s^2 w_{j+1/2}}{\delta x^2} + \left(\frac{w_{j+3/2} - w_{j+1/2}}{\Delta y_{j+1}} - \frac{w_{j+1/2} - w_{j-1/2}}{\Delta y_j}\right)/\Delta y_{j+1/2} + \frac{\delta_s^2 w_{j+1/2}}{\delta z^2}\right].$$

The divergence of the velocity field and the Laplacian of the pressure field, evaluated at $(i, j+1/2, k)$ for integer values of i, j, and k, is

$$\frac{\delta_s u_{j+1/2}}{\delta x} + \frac{v_{j+1} - v_j}{\Delta y_{j+1/2}} + \frac{\delta_s w_{j+1/2}}{\delta z},$$
$$\frac{\delta_s^2 p_{j+1/2}}{\delta x^2} + \left(\frac{p_{j+3/2} - p_{j+1/2}}{\Delta y_{j+1}} - \frac{p_{j+1/2} - p_{j-1/2}}{\Delta y_j}\right)/\Delta y_{j+1/2} + \frac{\delta_s^2 p_{j+1/2}}{\delta z^2}.$$

Note that the above two operators are required by the Poisson equation to update the pressure in the fractional step algorithm presented in §13.3.

13.2.2.2 Duct flow case

The duct flow case follows as a straightforward extension of the channel and cavity cases presented above and below, and thus, for brevity, is left as an exercise.

13.2.2.3 Cavity flow case

The spatial discretization of all derivatives required in the cavity case are now made precise. In the following equations, we will need to interpolate flow variables in at most one spatial direction (x, y, or z); the various interpolation formulae described in the first paragraph §13.2.2.1 are thus used again here, replacing Δy with Δx or Δz as appropriate. The first component of the momentum equation, (13.1a), to be evaluated at $\{i, j+1/2, k+1/2\}$ for integer values of i, j, and k, is discretized as follows (indicating only the dependencies of flow variables on the RHS on indices other than $\{i, j+1/2, k+1/2\}$, for notational clarity)

$$\left.\frac{\partial u}{\partial t}\right|_{(i,j+1/2,k+1/2)} = -\left[\frac{\overline{u}^2_{i+1/2}-\overline{u}^2_{i-1/2}}{\Delta x_i}+\frac{(\check{v}\overline{u})_{j+1}-(\check{v}\overline{u})_j}{\Delta y_{j+1/2}}+\frac{(\check{w}\overline{u})_{k+1}-(\check{w}\overline{u})_k}{\Delta z_{k+1/2}}\right]-\frac{p_{i+1/2}-p_{i-1/2}}{\Delta x_i}+\psi_1$$
$$+\mu\left[\frac{\frac{u_{i+1}-u_i}{\Delta x_{i+1/2}}-\frac{u_i-u_{i-1}}{\Delta x_{i-1/2}}}{\Delta x_i}+\frac{\frac{u_{j+3/2}-u_{j+1/2}}{\Delta y_{j+1}}-\frac{u_{j+1/2}-u_{j-1/2}}{\Delta y_j}}{\Delta y_{j+1/2}}+\frac{\frac{u_{k+3/2}-u_{k+1/2}}{\Delta z_{k+1}}-\frac{u_{k+1/2}-u_{k-1/2}}{\Delta z_k}}{\Delta z_{k+1/2}}\right].$$

The second component of the momentum equation, evaluated at $\{i+1/2, j, k+1/2\}$, is discretized as follows (indicating only the dependencies of flow variables on the RHS on indices other than $\{i+1/2, j, k+1/2\}$)

$$\left.\frac{\partial v}{\partial t}\right|_{(i+1/2,j,k+1/2)} = -\left[\frac{(\check{u}\overline{v})_{i+1}-(\check{u}\overline{v})_i}{\Delta x_{i+1/2}}+\frac{\overline{v}^2_{j+1/2}-\overline{v}^2_{j-1/2}}{\Delta y_j}+\frac{(\check{w}\overline{v})_{k+1}-(\check{w}\overline{v})_k}{\Delta y_{k+1/2}}\right]-\frac{p_{j+1/2}-p_{j-1/2}}{\Delta y_j}+\psi_2$$
$$+\mu\left[\frac{\frac{v_{i+3/2}-v_{i+1/2}}{\Delta x_{i+1}}-\frac{v_{i+1/2}-v_{i-1/2}}{\Delta x_i}}{\Delta x_{i+1/2}}+\frac{\frac{v_{j+1}-v_j}{\Delta y_{j+1/2}}-\frac{v_j-v_{j-1}}{\Delta x_{j-1/2}}}{\Delta x_j}+\frac{\frac{v_{k+3/2}-v_{k+1/2}}{\Delta z_{k+1}}-\frac{v_{k+1/2}-v_{k-1/2}}{\Delta z_k}}{\Delta z_{k+1/2}}\right].$$

The third component of the momentum equation, evaluated at $\{i+1/2, j+1/2, k\}$, is discretized as follows (indicating only the dependencies of flow variables on the RHS on indices other than $\{i+1/2, j+1/2, k\}$)

$$\left.\frac{\partial w}{\partial t}\right|_{(i+1/2,j+1/2,k)} = -\left[\frac{(\check{u}\overline{w})_{i+1}-(\check{u}\overline{w})_i}{\Delta x_{i+1/2}}+\frac{(\check{v}\overline{w})_{j+1}-(\check{v}\overline{w})_j}{\Delta y_{j+1/2}}+\frac{\overline{w}^2_{k+1/2}-\overline{w}^2_{k-1/2}}{\Delta x_k}\right]-\frac{p_{k+1/2}-p_{k-1/2}}{\Delta z_k}+\psi_3$$
$$+\mu\left[\frac{\frac{w_{i+3/2}-w_{i+1/2}}{\Delta x_{i+1}}-\frac{w_{i+1/2}-w_{i-1/2}}{\Delta x_i}}{\Delta x_{i+1/2}}+\frac{\frac{w_{j+3/2}-w_{j+1/2}}{\Delta y_{j+1}}-\frac{w_{j+1/2}-w_{j-1/2}}{\Delta y_j}}{\Delta y_{j+1/2}}+\frac{\frac{w_{k+1}-w_k}{\Delta z_{k+1/2}}-\frac{w_k-w_{k-1}}{\Delta z_{k-1/2}}}{\Delta z_k}\right].$$

The divergence of the velocity field and the Laplacian of the pressure field, evaluated at $\{i+1/2, j+1/2, k+1/2\}$, are discretized as follows (indicating only the dependencies of flow variables on points other than $\{i+1/2, j+1/2, k+1/2\}$)

$$\frac{u_{i+1}-u_i}{\Delta x_{i+1/2}}+\frac{v_{j+1}-v_j}{\Delta y_{j+1/2}}+\frac{w_{k+1}-w_k}{\Delta z_{k+1/2}},$$
$$\frac{\frac{p_{i+3/2}-p_{i+1/2}}{\Delta x_{i+1}}-\frac{p_{i+1/2}-p_{i-1/2}}{\Delta x_i}}{\Delta x_{i+1/2}}+\frac{\frac{p_{j+3/2}-p_{j+1/2}}{\Delta y_{j+1}}-\frac{p_{j+1/2}-p_{j-1/2}}{\Delta y_j}}{\Delta y_{j+1/2}}+\frac{\frac{p_{k+3/2}-p_{k+1/2}}{\Delta z_{k+1}}-\frac{p_{k+1/2}-p_{k-1/2}}{\Delta z_j}}{\Delta z_{k+1/2}}.$$

13.2.3 Conservation properties of the spatially-discretized NSE

An important property that can significantly improve the stability of a nonlinear simulation code is that it conserve as many global properties as possible that the original PDE conserves. We now show, in the channel flow case (even on a stretched grid), that the spatial discretization used in the present code conserves mass to within machine precision, and that errors due to the spatial discretization of the convective terms do not affect the total momentum or energy of the flow. Discrete conservation in the triply periodic, duct, and cavity cases follow as a straightforward extension, and are left as an exercise.

13.2.3.1 Discrete conservation of mass

To show that the total mass is conserved exactly by the spatial discretization in the channel-flow case, the continuity equation is integrated over the volume under consideration, with the integrals evaluated with a rectangular rule in the Fourier directions[5]

$$\begin{aligned}\int_\Omega \left(\frac{\delta u}{\delta x}+\frac{\delta v}{\delta y}+\frac{\delta w}{\delta z}\right) dV &= \int_z\int_x \sum_{j=1}^{NY} \Delta y_{j+1/2}\left(\frac{\delta_s u_{j+1/2}}{\delta x}+\frac{v_{j+1}-v_j}{\Delta y_{j+1/2}}+\frac{\delta_s w_{j+1/2}}{\delta z}\right) dx\,dz \\ &= \int_z\int_x (v_{NY+1}-v_1)\,dx\,dz \\ &= 0 \quad \Rightarrow \text{Mass is conserved.}\end{aligned}$$

Note that spectral differentiation in x corresponds to multiplying by ik_x at each wavenumber in Fourier space, whereas spectral integration in x corresponds to picking out the $k_x = 0$ wavenumber; thus, spectral integration of a spectral derivative gives exactly zero. In the wall-normal direction, note that the sum **telescopes**; that is, the positive term for one value of j in the sum exactly cancels the negative term for the next value of j in the sum, so the total sum in the y direction adds up to only a term at each boundary.

[5] For convenience in these expressions, we define v both a half a cell outside the walls and a half a cell inside the walls as the value of v to be prescribed on the boundary, where v is not defined in the staggered discretization; that is,

$$v_{NY+1} = v_{NY} = v_{NY+1/2}, \quad v_1 = v_2 = v_{3/2}.$$

13.2.3.2 Discrete conservation of momentum

To show that total momentum is conserved in each direction in the channel-flow case with no extra forcing (i.e., $\psi_i = -P_x\delta_{i1}$, $\phi = 0$), each component of the momentum equation in (13.1a) is integrated[6] over Ω:

$$\begin{aligned}
\frac{\partial}{\partial t}\int_\Omega u\,dV &= \int_\Omega\left(-\frac{\delta_s u^2}{\delta x}-\frac{\delta\, v\overline{u}}{\delta y}-\frac{\delta_s wu}{\delta z}+\mu\left(\frac{\delta_s^2 u}{\delta x^2}+\frac{\delta^2 u}{\delta y^2}+\frac{\delta_s^2 u}{\delta z^2}\right)-\frac{\delta_s p}{\delta x}-P_x\right)dV\\
&= \int_z\int_x\sum_{j=2}^{NY-1}\Delta y_{j+1/2}\left[-\frac{(v\overline{u})_{j+1}-(v\overline{u})_j}{\Delta y_{j+1/2}}+\frac{\mu}{\Delta y_{j+1/2}}\left(\frac{u_{j+3/2}-u_{j+1/2}}{\Delta y_{j+1}}-\frac{u_{j+1/2}-u_{j-1/2}}{\Delta y_j}\right)-P_x\right]dx\,dz\\
&= \int_z\int_x\left[\mu\left(\frac{u_{NY+1/2}-u_{NY-1/2}}{\Delta y_{NY}}-\frac{u_{5/2}-u_{3/2}}{\Delta y_2}\right)-L_y P_x\right]dx\,dz,\\
\frac{\partial}{\partial t}\int_\Omega v\,dV &= \int_\Omega\left(-\frac{\delta_s \breve{u}v}{\delta x}-\frac{\delta\,\overline{v}^2}{\delta y}-\frac{\delta_s \breve{w}v}{\delta z}+\mu\left(\frac{\delta_s^2 v}{\delta x^2}+\frac{\delta^2 v}{\delta y^2}+\frac{\delta_s^2 v}{\delta z^2}\right)-\frac{\delta p}{\delta y}\right)dV\\
&= \int_z\int_x\sum_{j=2}^{NY}\Delta y_j\left[-\frac{\overline{v}^2_{j+1/2}-\overline{v}^2_{j-1/2}}{\Delta y_j}+\frac{\mu}{\Delta y_j}\left(\frac{v_{j+1}-v_j}{\Delta y_{j+1/2}}-\frac{v_j-v_{j-1}}{\Delta y_{j-1/2}}\right)-\frac{p_{j+1/2}-p_{j-1/2}}{\Delta y_j}\right]dx\,dz\\
&= \int_z\int_x\left[-p_{NY+1/2}+p_{3/2}\right]dx\,dz\\
\frac{\partial}{\partial t}\int_\Omega w\,dV &= \int_\Omega\left(-\frac{\delta_s uw}{\delta x}-\frac{\delta\, v\overline{w}}{\delta y}-\frac{\delta_s w^2}{\delta z}+\mu\left(\frac{\delta_s^2 w}{\delta x^2}+\frac{\delta^2 w}{\delta y^2}+\frac{\delta_s^2 w}{\delta z^2}\right)-\frac{\delta_s p}{\delta z}\right)dV\\
&= \int_z\int_x\sum_{j=2}^{NY-1}\Delta y_{j+1/2}\left[-\frac{(v\overline{w})_{j+1}-(v\overline{w})_j}{\Delta y_{j+1/2}}+\frac{\mu}{\Delta y_{j+1/2}}\left(\frac{w_{j+3/2}-w_{j+1/2}}{\Delta y_{j+1}}-\frac{w_{j+1/2}-w_{j-1/2}}{\Delta y_j}\right)\right]dx\,dz\\
&= \int_z\int_x\left[\mu\left(\frac{w_{NY+1/2}-w_{NY-1/2}}{\Delta y_{NY}}-\frac{w_{5/2}-w_{3/2}}{\Delta y_2}\right)\right]dx\,dz
\end{aligned}$$

In the limit that $\mu \to 0$ with $P_x = 0$, momentum is conserved in the x and z directions. Numerical differencing errors on the interior also do not contribute to a loss of momentum conservation in the y direction; note that, in fact, the numerical code may be implemented in such a manner that $\int_z\int_x v\,dx\,dz = 0$ exactly for all y and t. For cases in which $\mu \neq 0$, it is seen that choosing

$$P_x = \frac{1}{V}\int_x\int_z\mu\left(\frac{u_{NY+1/2}-u_{NY-1/2}}{\Delta y_{NY}}-\frac{u_{5/2}-u_{3/2}}{\Delta y_2}\right)dx\,dz < 0, \tag{13.7}$$

where $V = L_x L_y L_z$ is the volume of the domain under consideration, maintains the x_1 component of momentum (*i.e.*, the **bulk velocity** $u_B = \frac{1}{V}\int_\Omega u\,dV > 0$) constant by balancing the skin friction integrated over the walls with the force applied by the mean pressure gradient[7].

[6] Note that $du/dt = dw/dt = 0$ on the walls, so the RHS of the u and w components of the momentum equation must be zero at the wall points; thus, these points are skipped in the corresponding sums listed here.

[7] Note that small round-off errors can slowly accumulate to drive the quantity $\frac{1}{V}\int_\Omega u\,dV$ away from the target value of u_B in a long numerical simulation. This can be corrected easily by increasing or decreasing P_x proportionally. An expression of the form

$$P_x = \frac{1}{V}\int_x\int_z\mu\left(\frac{u_{NY+1/2}-u_{NY-1/2}}{\Delta y_{NY}}-\frac{u_{5/2}-u_{3/2}}{\Delta y_2}\right)dx\,dz + k\left(\frac{1}{V}\int_\Omega u\,dV - u_{B\text{target}}\right),$$

for a small positive gain k (determined by trial and error) is thus usually employed.

13.2.3.3 Discrete conservation of energy

The viscous terms of the NSE result in energy dissipation at the small scales, which is replenished by the action of the pressure gradient P_x on the bulk flow. To show that energy is conserved in cases with forcing $\phi = \psi = 0$ and viscosity $\mu = 0$, the momentum equation in (13.1a) is multiplied by $\vec{u}$ and integrated over the volume under consideration (underbraced sums telescope and therefore cancel):

$$
\begin{aligned}
\frac{\partial}{\partial t}\int_\Omega \frac{u^2+v^2+w^2}{2}\,dV = \int_\Omega \Bigg[& u\left(-\frac{\delta_s u^2}{\delta x}-\frac{\delta\, v\overline{u}}{\delta y}-\frac{\delta_s wu}{\delta z}+\mu\left(\frac{\delta_s^2 u}{\delta x^2}+\frac{\delta^2 u}{\delta y^2}+\frac{\delta_s^2 u}{\delta z^2}\right)-\frac{\delta_s p}{\delta x}-P_x\right) \\
& +v\left(-\frac{\delta_s \breve{u}v}{\delta x}-\frac{\delta\, \overline{v}^2}{\delta y}-\frac{\delta_s \breve{w}v}{\delta z}+\mu\left(\frac{\delta_s^2 v}{\delta x^2}+\frac{\delta^2 v}{\delta y^2}+\frac{\delta_s^2 v}{\delta z^2}\right)-\frac{\delta p}{\delta y}\right) \\
& +w\left(-\frac{\delta_s uw}{\delta x}-\frac{\delta\, v\overline{w}}{\delta y}-\frac{\delta_s w^2}{\delta z}+\mu\left(\frac{\delta_s^2 w}{\delta x^2}+\frac{\delta^2 w}{\delta y^2}+\frac{\delta_s^2 w}{\delta z^2}\right)-\frac{\delta_s p}{\delta z}\right)\Bigg]\,dV
\end{aligned}
$$

$$
\begin{aligned}
= \int_z\int_x \Bigg[& \sum_{j=2}^{NY-1} \Delta y_{j+1/2}\, u_{j+1/2}\left(-\frac{\delta_s u^2_{j+1/2}}{\delta x}-\frac{(v\overline{u})_{j+1}-(v\overline{u})_j}{\Delta y_{j+1/2}}-\frac{\delta_s (wu)_{j+1/2}}{\delta z}-\frac{\delta_s p_{j+1/2}}{\delta x}\right) \\
& +\sum_{j=2}^{NY} \Delta y_j\, v_j\left(-\frac{\delta_s (\breve{u}v)_j}{\delta x}-\frac{\overline{v}^2_{j+1/2}-\overline{v}^2_{j-1/2}}{\Delta y_j}-\frac{\delta_s (\breve{w}v)_j}{\delta z}-\frac{p_{j+1/2}-p_{j-1/2}}{\Delta y_j}\right) \\
& +\sum_{j=2}^{NY-1} \Delta y_{j+1/2}\, w_{j+1/2}\left(-\frac{\delta_s (uw)_{j+1/2}}{\delta x}-\frac{(v\overline{w})_{j+1}-(v\overline{w})_j}{\Delta y_{j+1/2}}-\frac{\delta_s w^2_{j+1/2}}{\delta z}-\frac{\delta_s p_{j+1/2}}{\delta z}\right)\Bigg]\,dx\,dz
\end{aligned}
$$

$$
\begin{aligned}
= \int_z\int_x \Bigg[& \sum_{j=2}^{NY} \Delta y_{j+1/2}\left(p_{j+1/2}-\frac{u^2_{j+1/2}+\left(v^2_{j+1}+v^2_j\right)/2+w^2_{j+1/2}}{2}\right)\left(\frac{\delta_s u_{j+1/2}}{\delta x}+\frac{\delta_s w_{j+1/2}}{\delta z}\right)dx\,dz \\
& -\sum_{j=2}^{NY}\frac{1}{2}\left(\underbrace{u_{j+1/2}u_{j+3/2}v_{j+1}-u_{j-1/2}u_{j+1/2}v_j}+u_{j+1/2}u_{j+1/2}v_{j+1}-u_{j+1/2}u_{j+1/2}v_j\right) \\
& +\sum_{j=2}^{NY} \Delta y_{j+1/2}\, p_{j+1/2}\left(\frac{v_{j+1}-v_j}{\Delta y_{j+1/2}}\right)-\sum_{j=2}^{NY-1}\left(\frac{v^2_{j+1}+v^2_j}{4}(v_{j+1}-v_j)\right) \\
& -\sum_{j=2}^{NY}\frac{1}{2}\left(\underbrace{w_{j+1/2}w_{j+3/2}v_{j+1}-w_{j-1/2}w_{j+1/2}v_j}+w_{j+1/2}w_{j+1/2}v_{j+1}-w_{j+1/2}w_{j+1/2}v_j\right)\Bigg]\,dx\,dz
\end{aligned}
$$

$$
= \int_z\int_x \sum_{j=2}^{NY} \Delta y_{j+1/2}\left(p_{j+1/2}-\frac{u^2_{j+1/2}+\frac{v^2_{j+1}+v^2_j}{2}+w^2_{j+1/2}}{2}\right)\left(\frac{\delta_s u_{j+1/2}}{\delta x}+\frac{v_{j+1}-v_j}{\Delta y_{j+1/2}}+\frac{\delta_s w_{j+1/2}}{\delta z}\right)dx\,dz
$$

$= 0 \quad \Rightarrow$ energy is conserved.

Some additional algebra used in the derivation outlined above now follows. Note first that, in the spectral directions, we may apply what we will call **spectral integration by parts** in our analysis. This property follows from the fact that spectral integration involves simply scaling the $k_x = 0$ mode of the integrand and

noting from (5.36) that

$$q = \frac{\delta_s p}{\delta x} u \quad \Rightarrow \quad \hat{q}_0 = \sum_j (\mathrm{i} k_{x_j} \hat{p}_j) \hat{u}_{-j}, \qquad \text{and}$$

$$r = \frac{\delta_s u}{\delta x} p \quad \Rightarrow \quad \hat{r}_0 = \sum_i (\mathrm{i} k_{x_i} \hat{u}_i) \hat{p}_{-i} = \sum_j \hat{u}_{-j} (-\mathrm{i} k_{x_j} \hat{p}_j) = -\hat{q}_0.$$

Thus, in the spectral directions x and z, it follows that

$$\int_x u \frac{\delta_s p}{\delta x} dx = -\int_x \frac{\delta_s u}{\delta x} p \, dx \quad \text{and} \quad \int_z w \frac{\delta_s p}{\delta z} dz = -\int_w \frac{\delta_s w}{\delta z} p \, dz.$$

Now consider the spectral derivative of a product, such as $\delta_s uv/\delta x$. Recall that nonlinear products in a pseudospectral code are computed by transforming to physical space, performing the product, then transforming back to Fourier space, whereas spectral differentiation is performed by simply multiplying by $\mathrm{i}k_x$. Recall also that nonlinear products scatter energy to higher wavenumbers. *If* the PDE system under consideration is "fully resolved" in the spectral directions, both products and derivatives would be computed exactly, and we could apply what we will call the **spectral chain rule for differentiation**

$$\frac{\delta_s \breve{u} v}{\delta x} = \frac{\delta_s \breve{u}}{\delta x} v + \breve{u} \frac{\delta_s v}{\delta x}.$$

However, to make them affordable, turbulence simulations are inevitably conducted with "marginal resolution", using as few Fourier modes as possible in each direction while still achieving the desired accuracy on the quantities of interest in the simulation (typically, some of the time-averaged statistics). Thus, we expect significant energy to cascade to wavenumbers outside the range of wavenumbers represented in the numerical simulation. As discussed in §5.7, there are two primary ways of handling the necessary truncation of the Fourier series representation of the flow field under consideration:

A) allow the cascade of energy to higher wavenumbers (due to the nonlinear products) to alias back to lower wavenumbers, hoping that the spurious effects of this aliasing will be minimal, or
B) zero out all higher-order variations resulting from nonlinear products, using the 2/3 dealiasing rule.

Method A creates spurious energy sources, as the spectral chain rule for differentiation shown above does *not* hold when the infinite Fourier series are truncated. Method B constantly drains off the energy of all unresolved modes after each nonlinear product, and thus energy is not conserved in this case either. However, the spectral chain rule for differentiation shown above *does* apply when 2/3 dealiasing is applied to all nonlinear products. Together with the rest of the energy conservation proof provided above, this guarantees that no spurious numerical energy *sources* ever appear in the flow due to the discretization and Fourier series truncation. Thus, in order to insure stability of the computation, we use 2/3 dealiasing in the spectral directions in the present numerical code, acknowledging the extra dissipation that this method applies at the unresolved scales.

The spectral chain rule for differentiation and spectral integration by parts and may thus be applied to the integral of $v\delta_s(uv)/\delta x$ in the spectral direction x, resulting in

$$\int_x v \frac{\delta_s uv}{\delta x} dx = \int_x \left(v^2 \frac{\delta_s u}{\delta x} + uv \frac{\delta_s v}{\delta x} \right) dx = \int_x \left(v^2 \frac{\delta_s u}{\delta x} - v \frac{\delta_s uv}{\delta x} \right) dx \quad \Rightarrow \quad \int_x v \frac{\delta_s uv}{\delta x} dx = \frac{1}{2} \int_x v^2 \frac{\delta_s u}{\delta x} dx.$$

Note that, since the spectral integral of a spectral derivative is zero, it follows from the spectral chain rule for differentiation and the above identity that

$$\int_x \frac{\delta_s u^3}{\delta x} dx = \int_x u \frac{\delta_s u^2}{\delta x} dx = \int_x u^2 \frac{\delta_s u}{\delta x} dx = 0.$$

Note also that, applying the rectangular rule approximation of integration in y and the above identity, we may write

$$\begin{aligned}\int_x \sum_{j=2}^{NY} \Delta y_j\, v_j \frac{\delta_s\,(\breve{u}v)_j}{\delta x}\,dx &= \int_x \sum_{j=2}^{NY} \frac{1}{2}\left(\Delta y_{j+1/2}\, v_j \frac{\delta_s u_{j+1/2} v_j}{\delta x} + \Delta y_{j-1/2}\, v_j \frac{\delta_s u_{j-1/2} v_j}{\delta x}\right) dx \\ &= \int_x \sum_{j=2}^{NY} \frac{1}{4}\left(\Delta y_{j+1/2}\, v_j^2 \frac{\delta_s u_{j+1/2}}{\delta x} + \Delta y_{j-1/2}\, v_j^2 \frac{\delta_s u_{j-1/2}}{\delta x}\right) dx \\ &= \int_x \sum_{j=2}^{NY-1} \Delta y_{j+1/2} \frac{v_j^2 + v_{j+1}^2}{4} \frac{\delta_s u_{j+1/2}}{\delta x}\,dx.\end{aligned}$$

The other convective terms in the spectral directions x and z are handled similarly. Finally, the step involving the rectangular rule approximation of integration of the term $v\delta\overline{v}^2/\delta y$ may be written

$$\begin{aligned}\sum_{j=2}^{NY} v_j(\overline{v}^2_{j+1/2} - \overline{v}^2_{j-1/2}) &= \sum_{j=2}^{NY} \frac{v_j}{4}(\underbrace{v_{j+1}v_j - v_{j-1}^2}+\underbrace{v_{j+1}^2 - v_j v_{j-1}}+v_{j+1}v_j - v_j v_{j-1}) = \sum_{j=2}^{NY} \frac{1}{4}(v_j^2 v_{j+1} - v_j^2 v_{j-1}) \\ &= \sum_{j=2}^{NY} \frac{v_j^2}{4}\left[\left(v_{j+1} - v_j\right) + \left(v_j - v_{j-1}\right)\right] \qquad = \sum_{j=2}^{NY} \frac{v_j^2 + v_{j+1}^2}{4}\left(v_{j+1} - v_j\right).\end{aligned}$$

13.3 Temporal discretization - the fractional step algorithm

The temporal discretization used in the present work, referred to as a **fractional step** algorithm, is an extension of the mixed RKW3/CN method developed in §10.4.2 applied to the spatial discretization described above of the constrained system (13.1). We first write the time discretization of the spatially-discretized momentum equation (13.1a) as

$$\frac{\mathbf{u}_i^{rk}-\mathbf{u}_i^{rk-1}}{\overline{h}^{rk}} = \overline{\beta}^{rk}\,\mathbf{r}_i(\vec{\mathbf{u}}^{rk-1}) + \overline{\zeta}^{rk}\,\mathbf{r}_i(\vec{\mathbf{u}}^{rk-2}) + \frac{1}{2}\left(A_i(\vec{\mathbf{u}}^{rk}) + A_i(\vec{\mathbf{u}}^{rk-1})\right) - \frac{\delta \mathbf{p}^{rk-1}}{\delta x_i} - \frac{\delta \mathbf{q}}{\delta x_i} + \boldsymbol{\psi}_i^{rk-1}, \tag{13.8}$$

where $\mathbf{q} \triangleq \mathbf{p}^{rk} - \mathbf{p}^{rk-1}$, and thus pressure is, in effect, treated with IE over each RK substep. In this discretization, $\mathbf{r}_i(\vec{\mathbf{u}})$ represents those (possibly nonlinear) RHS terms of the spatial discretization of (13.1) to be treated explicitly using RKW3 and $A_i(\vec{\mathbf{u}})$ represent those (linear) RHS terms to be treated implicitly using CN over each RK substep; note that there are a couple of possible choices we can make in this regard, as detailed in the following two subsections. The forcing term $\boldsymbol{\psi}$ is handled with simple EE over each RK substep. The constants $\overline{h}^{rk}$, $\overline{\beta}^{rk}$, and $\overline{\zeta}^{rk}$ are all defined as in 10.63.

The fractional step algorithm approximates the computation of (13.8) by breaking it into two steps. The first step calculates an intermediate update to the components of the flow velocity, $\mathbf{v}_i$, neglecting the influence of the pressure update term $-\delta\mathbf{q}/\delta x_i$ on the RHS:

$$\frac{\mathbf{v}_i-\mathbf{u}_i^{rk-1}}{\overline{h}^{rk}} = \overline{\beta}^{rk}\,\mathbf{r}_i(\vec{\mathbf{u}}^{rk-1}) + \overline{\zeta}^{rk}\,\mathbf{r}_i(\vec{\mathbf{u}}^{rk-2}) + \frac{1}{2}\left(A_i(\vec{\mathbf{v}}) + A_i(\vec{\mathbf{u}}^{rk-1})\right) - \frac{\delta \mathbf{p}^{rk-1}}{\delta x_i} + \boldsymbol{\psi}_i^{rk-1}. \tag{13.9a}$$

The second step then adds the influence of the formerly-neglected pressure update term to the components intermediate velocity field $\mathbf{v}_i$, and also updates $\mathbf{p}$ itself:

$$\frac{\mathbf{u}_i^{rk}-\mathbf{v}_i}{\overline{h}^{rk}} = -\frac{\delta \mathbf{q}}{\delta x_i}, \qquad \mathbf{p}^{rk} = \mathbf{p}^{rk-1} + \mathbf{q}. \tag{13.9b}$$

The pressure update $\mathbf{q}$ is calculated in such a way as to insure that the spatial discretization of the velocity field at the new RK substep, $\vec{\mathbf{u}}^{rk}$, is exactly divergence free, thereby enforcing the spatial discretization of the continuity equation (13.1b). Noting the derivation[8] in §5.1, this is done simply by defining $\mathbf{q}$ as the solution to the equation

$$\frac{\delta^2 \mathbf{q}}{\delta x_1^2} + \frac{\delta^2 \mathbf{q}}{\delta x_2^2} + \frac{\delta^2 \mathbf{q}}{\delta x_3^2} = \frac{1}{\overline{h}^{rk}}\left(\frac{\delta \mathbf{v}_1}{\delta x_1} + \frac{\delta \mathbf{v}_2}{\delta x_2} + \frac{\delta \mathbf{v}_3}{\delta x_3}\right). \tag{13.10}$$

For convenience, we will apply the same boundary conditions on the intermediate field $\vec{\mathbf{v}}$ and the final field $\vec{\mathbf{u}}^{rk}$; thus, it follows from the normal component of (13.9b) evaluated at the wall(s) that the appropriate boundary conditions on $\mathbf{q}$ on the walls are homogeneous Neumann, i.e., $(\delta\mathbf{q}/\delta n)_{\text{wall}} = 0$. In the directions in which the flow velocity $\vec{\mathbf{u}}$ is periodic, the variables $\mathbf{p}$ and $\mathbf{q}$ are also periodic.

Note that the representation of (13.8) as the two-step process (13.9a)-(13.9b) is only approximate, as we have replaced $A_i(\vec{\mathbf{u}}^{rk})$ on the RHS of (13.8) with $A_i(\vec{\mathbf{v}})$ on the RHS of (13.9a) in order to convert the problem (13.8) to a set of two equations that may be solved one step at a time. As we have been careful to include the explicit term $-\delta\mathbf{p}^{rk-1}/\delta x_i$ in (13.9a), the effect of $\mathbf{q}$ only represents a small "pressure update" over the RK substep, and thus this approximation is, in fact, quite acceptable.

[8] Note that we have selected the constant in this projection as $c = \overline{h}^{rk}$, and thus the velocity update formula given in (5.45c) takes the form shown in (13.9b).

To rearrange and summarize, the equations to be solved by the fractional step algorithm are as follows. First, compute the explicit right-hand sides

$$\mathbf{R}_i = \mathbf{u}_i^{rk-1} + \overline{h}^{rk}\left[\overline{\beta}^{rk}\,\mathbf{r}_i(\vec{\mathbf{u}}^{rk-1}) + \overline{\zeta}^{rk}\,\mathbf{r}_i(\vec{\mathbf{u}}^{rk-2}) + \frac{1}{2}A_i(\vec{\mathbf{u}}^{rk-1}) - \frac{\delta \mathbf{p}^{rk-1}}{\delta x_i} + \boldsymbol{\psi}_i^{rk-1}\right]. \tag{13.11a}$$

Then, solve the implicit systems for the intermediate velocity components $\mathbf{v}_i$,

$$\left(I - \frac{\overline{h}^{rk}}{2}A_i\right)\mathbf{v}_i = \mathbf{R}_i, \tag{13.11b}$$

while enforcing the desired boundary conditions for $\vec{\mathbf{u}}^{rk}$ on the intermediate velocity components $\mathbf{v}_i$. Then solve the Poisson equation for the pressure update $\mathbf{q}$,

$$\frac{\delta^2 \mathbf{q}}{\delta x_j^2} = \frac{1}{\overline{h}^{rk}}\frac{\delta \mathbf{v}_j}{\delta x_j}, \tag{13.11c}$$

while enforcing homogeneous Neumann boundary conditions on $\mathbf{q}$. Finally, update the velocity and pressure accordingly,

$$\mathbf{u}_i^{rk} = \mathbf{v}_i - \overline{h}^{rk}\frac{\delta \mathbf{q}}{\delta x_i}, \qquad \mathbf{p}^{rk} = \mathbf{p}^{rk-1} + \mathbf{q}. \tag{13.11d}$$

13.3.1 All viscous terms implicit

As mentioned above, there are a couple of possible choices for which terms to take with RKW3 and which terms to take with CN over each RK substep in the temporal discretization described above. The simplest is to take all (nonlinear) convective terms with RKW3 and all (linear) viscous terms with CN over each RK substep, that is[9],

$$\mathbf{r}_i(\vec{\mathbf{u}}) = -\frac{\delta \mathbf{u}_1 * \mathbf{u}_i}{\delta x_1} - \frac{\delta \mathbf{u}_2 * \mathbf{u}_i}{\delta x_2} - \frac{\delta \mathbf{u}_3 * \mathbf{u}_i}{\delta x_3}, \tag{13.12a}$$

$$A_i(\vec{\mathbf{u}}) = \mu\left(\frac{\delta^2}{\delta x_1^2} + \frac{\delta^2}{\delta x_2^2} + \frac{\delta^2}{\delta x_3^2}\right)\mathbf{u}_i. \tag{13.12b}$$

This strategy is appropriate when the grid is essentially equally clustered in all three directions.

13.3.2 All y-derivative terms implicit

For systems in which the grid is clustered very tightly in one direction only, such as near the wall in the channel flow case, the wall-normal convective term, if treated explicitly, can result in a significant constraint on the time step for numerical stability. For such systems, it is advantageous to take the wall-normal viscous and wall-normal convective term with CN and all other terms with RKW3, that is[9],

$$\mathbf{r}_i(\vec{\mathbf{u}}) = \mu\left(\frac{\delta^2 \mathbf{u}_i}{\delta x_1^2} + \frac{\delta^2 \mathbf{u}_i}{\delta x_3^2}\right) - \frac{\delta \mathbf{u}_1 * \mathbf{u}_i}{\delta x_1} - \frac{\delta \mathbf{u}_3 * \mathbf{u}_i}{\delta x_3}, \tag{13.13a}$$

$$A_i(\vec{\mathbf{u}}) = \mu\frac{\delta^2 \mathbf{u}_i}{\delta x_2^2} - \frac{\delta \mathbf{u}_2 * \mathbf{u}_i}{\delta x_2}. \tag{13.13b}$$

[9]The notation $\mathbf{u}_1 * \mathbf{u}_2$ is used to denote the pointwise product of the vector u_1 with the vector u_2 at each gridpoint in physical space.

As this definition of $A_i(\vec{\mathbf{u}})$ is not linear, we can not apply CN to this term directly. However, it is significant to note that, when rearranged properly, this formulation may indeed be approximated by a formulation which is linear in the implicit variables without loss of overall accuracy of the method. To see how this may be accomplished, noting that $(\mathbf{v}_2 - \mathbf{u}_2^{rk-1}) \sim O(h)$ and therefore $(\mathbf{v}_2 - \mathbf{u}_2^{rk-1}) * (\mathbf{v}_2 - \mathbf{u}_2^{rk-1}) \sim O(h^2)$, we note the following identity:

$$0 \approx (\mathbf{v}_2 - \mathbf{u}_2^{rk-1}) * (\mathbf{v}_2 - \mathbf{u}_2^{rk-1}) = \mathbf{v}_2 * \mathbf{v}_2 - 2\mathbf{v}_2 * \mathbf{u}_2^{rk-1} + \mathbf{u}_2^{rk-1} * \mathbf{u}_2^{rk-1} \quad \Rightarrow \quad \mathbf{v}_2 * \mathbf{v}_2 \approx 2\mathbf{v}_2 * \mathbf{u}_2^{rk-1} - \mathbf{u}_2^{rk-1} * \mathbf{u}_2^{rk-1}.$$

Thus, applying this approximation, we may rewrite our temporal discretization of our system, reëxpressing $A_2(\vec{\mathbf{v}})$ as implied by the above identity and moving the extra explicit term so generated onto the RHS, such that[10]

$$A_2(\vec{\mathbf{v}}) = \mu \frac{\delta^2 \mathbf{v}_2}{\delta x_2^2} - 2\frac{\delta \mathbf{u}_2^{rk-1} * \mathbf{v}_2}{\delta x_2}, \quad \mathbf{R}_2 \leftarrow \mathbf{R}_2 + \frac{\overline{h}^{rk}}{2} \frac{\delta \mathbf{u}_2^{rk-1} * \mathbf{u}_2^{rk-1}}{\delta x_2}.$$

Notice that this modified form of $A_2(\vec{\mathbf{v}})$ is indeed linear in the unknown $\mathbf{v}_2$, though now the operator A_2 is itself a function of $\mathbf{u}_2^{rk-1}$. After the $\mathbf{v}_2$ equation is solved[11], we may use $\mathbf{v}_2$ to approximate $\mathbf{u}_2^{rk}$ in the expressions for $A_1(\vec{\mathbf{u}}^{rk})$ and $A_3(\vec{\mathbf{u}}^{rk})$.

$$A_1(\vec{\mathbf{u}}^{rk}) = \mu \frac{\delta^2 \mathbf{v}_1}{\delta x_2^2} - \frac{\delta \mathbf{v}_2 * \mathbf{v}_1}{\delta x_2}.$$
$$A_3(\vec{\mathbf{u}}^{rk}) = \mu \frac{\delta^2 \mathbf{v}_3}{\delta x_2^2} - \frac{\delta \mathbf{v}_2 * \mathbf{v}_3}{\delta x_2}.$$

Notice that these modified operations are also linear in the unknowns $\mathbf{v}_1$ and $\mathbf{v}_3$ respectively, though the linear operators A_1 and A_3 are themselves a function of $\mathbf{v}_2$.

[10] Fortunately, the new term added to R_2 via this manipulation exactly cancels one of the existing terms of $A_2(\vec{\mathbf{u}}^{rk-1})$ in R_2 [see (13.11a) and (13.12a)], thereby simplifying the computation of R_2.

[11] Note that, in this approach, the $\mathbf{v}_2$ momentum equation must be solved before the $\mathbf{v}_1$ and $\mathbf{v}_3$ momentum equations.

13.3.3 All viscous terms implicit in the triply periodic case

In the channel-flow case, all spatial derivatives are calculated spectrally. As introduced in §13.3.1, time stepping in this case may be accomplished with a mixed strategy with all viscous terms treated with CN over each RK substep, and all convective terms treated with RKW3. Each RK substep in this case thus proceeds as follows:

1. Initialize $\widehat{\mathbf{R}}_i$ with $\widehat{\mathbf{u}}^{rk}$ and the explicit part of the CN term
   $\widehat{\mathbf{R}}_i = \left\{1 - \frac{\nu}{2}\overline{h}^{rk}(\mathbf{k}_x^2 + \mathbf{k}_y^2 + \mathbf{k}_z^2)\right\} * \widehat{\mathbf{u}}_i$

2. Account for the pressure gradient term using EE
   $\widehat{\mathbf{R}}_1 \leftarrow \widehat{\mathbf{R}}_1 - \overline{h}^{rk}\mathrm{i}\mathbf{k}_x * \hat{\mathbf{p}}, \quad \widehat{\mathbf{R}}_2 \leftarrow \widehat{\mathbf{R}}_2 - \overline{h}^{rk}\mathrm{i}\mathbf{k}_y * \hat{\mathbf{p}}, \quad \widehat{\mathbf{R}}_3 \leftarrow \widehat{\mathbf{R}}_3 - \overline{h}^{rk}\mathrm{i}\mathbf{k}_z * \hat{\mathbf{p}}.$

3. Add the RK terms from the previous timestep, stored in $\widehat{\mathbf{F}}_i$, to the RHS
   if $(rk > 1)$ then $\widehat{\mathbf{R}}_i \leftarrow \widehat{\mathbf{R}}_i + \overline{\zeta}^{rk}\overline{h}^{rk}\widehat{\mathbf{F}}_i$

4. Convert the velocity to physical space
   $\widehat{\mathbf{u}}_i \rightarrow \mathbf{u}_i$

5. Calculate the nonlinear terms and store in $\widehat{\mathbf{F}}_i$
   Note that while there are 9 nonlinear terms, the ordering used here requires only 6 FFT calls.
   (a) $\mathbf{F}_1 = \mathbf{u}_1 * \mathbf{u}_1$
   (b) $\mathbf{F}_2 = \mathbf{u}_1 * \mathbf{u}_2$
   (c) $\mathbf{F}_3 = \mathbf{u}_1 * \mathbf{u}_3$
   (d) $\mathbf{S} = \mathbf{u}_2 * \mathbf{u}_2$
   (e) $\mathbf{F}_1 \rightarrow \widehat{\mathbf{F}}_1$
   (f) $\mathbf{F}_2 \rightarrow \widehat{\mathbf{F}}_2$
   (g) $\mathbf{F}_3 \rightarrow \widehat{\mathbf{F}}_3$
   (h) $\mathbf{S} \rightarrow \widehat{\mathbf{S}}$
   (i) $\widehat{\mathbf{F}}_1 \leftarrow -\mathrm{i}\mathbf{k}_x * \widehat{\mathbf{F}}_1 - \mathrm{i}\mathbf{k}_y * \widehat{\mathbf{F}}_2 - \mathrm{i}\mathbf{k}_z * \widehat{\mathbf{F}}_3$
   (j) $\widehat{\mathbf{F}}_2 \leftarrow -\mathrm{i}\mathbf{k}_x * \widehat{\mathbf{F}}_2 - \mathrm{i}\mathbf{k}_y * \widehat{\mathbf{S}}$
   (k) $\widehat{\mathbf{F}}_3 \leftarrow -\mathrm{i}\mathbf{k}_x * \widehat{\mathbf{F}}_3$
   (l) $\mathbf{S} = \mathbf{u}_2 * \mathbf{u}_3$
   (m) $\mathbf{S} \rightarrow \widehat{\mathbf{S}}$
   (o) $\widehat{\mathbf{F}}_2 \leftarrow \widehat{\mathbf{F}}_2 - \mathrm{i}\mathbf{k}_z * \widehat{\mathbf{S}}$
   (p) $\widehat{\mathbf{F}}_3 \leftarrow \widehat{\mathbf{F}}_3 - \mathrm{i}\mathbf{k}_y * \widehat{\mathbf{S}}$
   (q) $\mathbf{S} = \mathbf{u}_3 * \mathbf{u}_3$
   (r) $\mathbf{S} \rightarrow \widehat{\mathbf{S}}$
   (s) $\widehat{\mathbf{F}}_3 \leftarrow \widehat{\mathbf{F}}_3 - \mathrm{i}\mathbf{k}_z * \widehat{\mathbf{S}}$

6. Add the RK terms from the present timestep to the RHS
   $\widehat{\mathbf{R}}_i \leftarrow \widehat{\mathbf{R}}_i + \overline{\beta}^{rk}\overline{h}^{rk}\widehat{\mathbf{F}}_i$

7. Solve for the intermediate velocity. Since the system is diagonal, this is easy.
   $\widehat{\mathbf{u}}_i = \widehat{\mathbf{R}}_i / [1 + \frac{\nu}{2}\overline{h}^{rk}(\mathbf{k}_x^2 + \mathbf{k}_y^2 + \mathbf{k}_z^2)]$

8. Calculate the pressure update $\mathbf{q}$ that will make the velocity divergence free
   $\widehat{\mathbf{q}} = -(\mathrm{i}\mathbf{k}_x\widehat{\mathbf{u}}_1 + \mathrm{i}\mathbf{k}_y\widehat{\mathbf{u}}_2 + \mathrm{i}\mathbf{k}_z\widehat{\mathbf{u}}_3)/(\mathbf{k}_x^2 + \mathbf{k}_y^2 + \mathbf{k}_z^2)$

9. Project the velocity to get a divergence free field
   $\widehat{\mathbf{u}}_1 \leftarrow \widehat{\mathbf{u}}_1 - \mathrm{i}\mathbf{k}_x * \widehat{\mathbf{q}}, \quad \widehat{\mathbf{u}}_2 \leftarrow \widehat{\mathbf{u}}_2 - \mathrm{i}\mathbf{k}_y * \widehat{\mathbf{q}}, \quad \widehat{\mathbf{u}}_3 \leftarrow \widehat{\mathbf{u}}_3 - \mathrm{i}\mathbf{k}_z * \widehat{\mathbf{q}}.$

10. Finally, update the pressure field using $\mathbf{q}$
    $\widehat{\mathbf{p}} \leftarrow \widehat{\mathbf{p}} + \widehat{\mathbf{q}}/\overline{h}^{rk}$

In all, we have 9 FFT calls per RK substep.

13.3.4 All y-derivative terms implicit in the channel-flow case

In the channel-flow case, the wall-normal (y) derivatives are approximated with FD methods, while the wall-parallel (x and z) derivatives are calculated spectrally. Time-stepping in this case may be accomplished with either a slight modification of the all-viscous-terms-implicit approach described above, or, as introduced in §13.3.2, an alternative mixed strategy with the viscous and convective terms involving wall-normal derivatives treated with CN over each RK substep, and the remaining viscous and convective terms treated with RKW3. Following the spatial discretization used in §13.2.2.1, each RK substep in this case proceeds as follows:

1. Initialize $\widehat{\mathbf{R}}_i$ with $\widehat{\mathbf{u}}^{rk}$
   $\widehat{\mathbf{R}}_i = \widehat{\mathbf{u}}_i$
2. Account for the pressure gradient term using EE
   $\widehat{\mathbf{R}}_1 \leftarrow \widehat{\mathbf{R}}_1 - \overline{h}^{rk} \mathrm{i}\mathbf{k}_x * \hat{\mathbf{p}}, \quad \widehat{\mathbf{R}}_2 \leftarrow \widehat{\mathbf{R}}_2 - \overline{h}^{rk} (\hat{\mathbf{p}}_x, \quad \widehat{\mathbf{R}}_3 \leftarrow \widehat{\mathbf{R}}_3 - \overline{h}^{rk} \mathrm{i}\mathbf{k}_z * \hat{\mathbf{p}}.$
3. If $(RK > 1)$ then add the term from the previous RK step
   $\widehat{R}_i = \widehat{R}_i + \overline{\zeta}_{RK} \overline{\beta}_{RK} \widehat{F}_i$
4. Add the pressure gradient to the RHS
   $\widehat{R}_1 = \widehat{R}_1 - \overline{h}_{RK} \hat{\imath} k_x \widehat{P}$
   $\widehat{R}_2(k_x, k_z, j) = \widehat{R}_2(k_x, k_z, j) - \overline{h}_{RK} \frac{\widehat{P}(k_x, k_z, j) - \widehat{P}(k_x, k_z, j-1)}{\Delta Y(j)}$
   $\widehat{R}_3 = \widehat{R}_3 - \overline{h}_{RK} \hat{\imath} k_z \widehat{P}$
5. Add P_x, the background pressure gradient that drives the flow
   $\widehat{R}_1(k_x = 0, k_z = 0, j) = \widehat{R}_1(k_x = 0, k_z = 0, j) - \overline{h}_{RK} P_x$
6. Create a storage variable F that will contain all RK terms and start with the viscous terms involving horizontal derivatives.
   $\widehat{F}_i = -\nu (k_x^2 + k_z^2) \widehat{u}_i,$
7. Convert the velocity to physical space
   $\widehat{u}_i \rightarrow u_i$
8. Add the nonlinear terms involving horizontal derivatives to $\widehat{F}$
   $\widehat{F}_1 = \widehat{F}_1 - \hat{\imath} k_x \widehat{u_1 u_3} - \hat{\imath} k_x \widehat{u_1 u_1}$
   $\widehat{F}_2 = \widehat{F}_2 - \hat{\imath} k_x \widehat{u_1} u_2 - \hat{\imath} k_z \widehat{u_3} u_2$
   $\widehat{F}_3 = \widehat{F}_3 - \hat{\imath} k_z \widehat{u_1 u_3} - \hat{\imath} k_z \widehat{u_3 u_3}$
   (Note that we need 5 independent FFTs here)
9. Now, we are done building the Runge-Kutta terms, add to the right hand side. We will need to keep $\widehat{F}_i$ for the next RK step, so it should not be overwritten below this point.
   $\widehat{R}_i = \widehat{R}_i + \overline{\beta}_{RK} \overline{h}_{RK} \widehat{F}_i$
10. Convert the right hand side arrays to physical space
    $\widehat{R}_i \rightarrow R_i,$
11. Compute the vertical viscous terms and add to the RHS as the explicit part of Crank-Nicolson.

$$R_1(i,j,k) = R_1(i,j,k) + \frac{\nu \overline{h}_{RK}}{2} \left(\frac{u_1(i,j+1,k) - u_1(i,j,k)}{\Delta Y(j+1)} - \frac{u_1(i,j,k) - u_1(i,j-1,k)}{\Delta Y(j)} \right) / \Delta Y_F(j)$$

$$R_2(i,j,k) = R_2(i,j,k) + \frac{\nu \overline{h}_{RK}}{2} \left(\frac{u_2(i,j+1,k) - u_2(i,j,k)}{\Delta Y_F(j)} - \frac{u_2(i,j,k) - u_2(i,j-1,k)}{\Delta Y_F(j-1)} \right) / \Delta Y(j)$$

$$R_3(i,j,k) = R_3(i,j,k) + \frac{\nu\overline{h}_{RK}}{2}\left(\frac{u_3(i,j+1,k)-u_3(i,j,k)}{\Delta Y(j+1)} - \frac{u_3(i,j,k)-u_3(i,j-1,k)}{\Delta Y(j)}\right)/\Delta Y_F(j)$$

12. Compute the nonlinear terms involving vertical derivatives and add to the RHS as the explicit part of Crank-Nicolson.
$S_1 = \overline{u}_1 * u_2$
$R_1(i,j,k) = R_1(i,j,k) - \frac{\overline{h}_{RK}}{2}(S_1(i,j+1,k) - S_1(i,j,k))/\Delta Y_F(j)$
$S_1 = \overline{u}_3 * u_2$
$R_3(i,j,k) = R_3(i,j,k) - \frac{\overline{h}_{RK}}{2}(S_1(i,j+1,k) - S_1(i,j,k))/\Delta Y_F(j)$

13. Solve the tridiagonal system for the intermediate wall-normal velocity:

$$\begin{aligned} v_2(i,j,k) - \frac{\nu\overline{h}_{RK}}{2}&\left(\frac{v_2(i,j+1,k)-v_2(i,j,k)}{\Delta Y_F(j)} - \frac{vv_2(i,j,k)-v_2(i,j-1,k)}{\Delta Y_F(j-1)}\right)/\Delta Y(j) \\ &+\overline{h}_{RK}\left(\overline{v}_2(i,j,k)\overline{u}_2(i,j,k) - \overline{v}_2(i,j-1,k)\overline{u}_2(i,j-1,k)\right)/\Delta Y(j) \quad = R_2(i,j,k) \end{aligned}$$

14. Now that we have the new intermediate wall-normal velocity, v_2, solve for the intermediate v_1 and v_3 using this new velocity.

$$\begin{aligned} v_1(i,j,k) - \frac{\nu\overline{h}_{RK}}{2}&\left(\frac{v_1(i,j+1,k)-v_1(i,j,k)}{\Delta Y(j+1)} - \frac{v_1(i,j,k)-v_1(i,j-1,k)}{\Delta Y(j)}\right)/\Delta Y_F(j) \\ &+\overline{h}_{RK}\left(\overline{v}_1(i,j+1,k)v_2(i,j+1,k) - \overline{v}_1(i,j,k)v_2(i,j,k)\right)/\Delta Y_F(j) \quad = R_1(i,j,k) \end{aligned}$$

$$\begin{aligned} v_3(i,j,k) - \frac{\nu\overline{h}_{RK}}{2}&\left(\frac{v_3(i,j+1,k)-v_3(i,j,k)}{\Delta Y(j+1)} - \frac{v_3(i,j,k)-v_3(i,j-1,k)}{\Delta Y(j)}\right)/\Delta Y_F(j) \\ &+\overline{h}_{RK}\left(\overline{v}_3(i,j+1,k)v_2(i,j+1,k) - \overline{v}_3(i,j,k)v_2(i,j,k)\right)/\Delta Y_F(j) \quad = R_3(i,j,k) \end{aligned}$$

15. Convert the intermediate velocity to Fourier space
$v_i \rightarrow \widehat{v}_i$

16. Solve the tridiagonal system for the pressure correction:

$$\begin{aligned} &-(k_x^2+k_z^2)\widehat{q}(k_x,k_z,j) + \left(\frac{\widehat{q}(k_x,k_z,j+1)-\widehat{q}(k_x,k_z,j)}{\Delta Y(j+1)} - \frac{\widehat{q}(k_x,k_z,j)-\widehat{q}(k_x,k_z,j-1)}{\Delta Y(j)}\right)/\Delta Y_F(j) \\ &= \hat{i}k_x\widehat{v}_1(k_x,k_z,j) + \hat{i}k_z\widehat{v}_3(k_x,k_z,j) + (\widehat{v}_2(k_x,k_z,j+1) - \widehat{v}_2(k_x,k_z,j))/\Delta Y_F(j) \end{aligned}$$

(Note that in order to avoid an extra storage array, we can store q in R_1 which is no longer needed for this RK step. Also notice that a factor of $\overline{h}_{RK}$ has been absorbed into q)

17. Now, use the pressure update to obtain a divergence-free velocity field.
$\widehat{u}_1^{RK+1} = \widehat{v}_1 - \hat{i}k_x\widehat{q}$
$\widehat{u}_2^{RK+1}(k_x,k_z,j) = \widehat{v}_2(k_x,k_z,j) - (\widehat{q}(k_x,k_z,j) - \widehat{q}(k_x,k_z,j-1))/\Delta Y(j)$
$\widehat{u}_3^{RK+1}\widehat{v}_3 - \hat{i}k_z\widehat{q}$
(In order to avoid an extra storage array, only one set of velocity arrays are defined, and this update is done in place.)

18. Finally, update the pressure field using q
    $\widehat{P} = \widehat{P} + \widehat{q}/\overline{h}_{RK}$
    (We need to divide by $\overline{h}_{RK}$ since this constant has been absorbed into $\overline{q}$ in the steps above.)

In all, we have 14 FFT calls per Runge-Kutta substep, and 11 full-sized storage arrays.

13.3.5 Shared-memory parallelization using OpenMP

13.3.6 Distributed-memory parallelization using MPI

13.4 Characterizing the statistics of turbulence

13.5 The visualization of turbulence

The discriminant.

Data Explorer

13.6 Large eddy simulation

13.7 Extensions

13.7.1 Passive scalars

13.7.2 Active scalars and the Boussinesq approximation of bouyancy

Introduce the compressible Navier-Stokes equation. Discuss computational issues. Define Mach number. Introduce Boussinesq.

13.7.3 Immersed boundary methods

13.7.4 Coordinate transformation methods

13.7.5 Noise generation

Neglecting viscosity, the equation governing compressible flow (known as the **inviscid Euler equation**), in conservation form, is

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0, \tag{13.14a}$$

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_j u_i}{\partial x_j} + \frac{\partial p}{\partial x_i} = 0. \tag{13.14b}$$

Subtracting $\partial^2\rho/\partial t^2$ from $\partial^2 p/\partial x_i^2$ and applying $\partial/\partial t$ of (13.14a) then (13.14b), we obtain

$$\frac{\partial^2 p}{\partial x_i^2} - \frac{\partial^2 \rho}{\partial t^2} = \frac{\partial}{\partial x_i}\Big(\frac{\partial p}{\partial x_i} + \frac{\partial \rho u_i}{\partial t}\Big) = \frac{\partial}{\partial x_i}\Big(-\frac{\partial \rho u_j u_i}{\partial x_j}\Big).$$

Now applying the ideal gas law $p = \rho RT$ where we approximate the temperature as nearly constant, $T \approx T_0$, and thus the speed of sound may also be approximated as constant, $c = \sqrt{RT} \approx \sqrt{RT_0} \triangleq c_0$, we obtain the inviscid approximation of **Lighthill's equation**,

$$\Big(\frac{\partial^2}{\partial x_i^2} - \frac{1}{c_0^2}\frac{\partial^2}{\partial t^2}\Big)p = -\frac{\partial^2 T_{ij}}{\partial x_i \partial x_j} \quad \text{where} \quad T_{ij} = \rho u_i u_j. \tag{13.15a}$$

It is convenient that, for low Mach number flows, incompressible simulations may be used to approximate the noise generated by the flow. This may be accomplished by calculating the **accoustic pressure field** radiated according to the wave equation given by (13.15a), with the **Lighthill stress tensor** T_{ij} on the RHS taken as a source term using the **hydrodynamic velocity field** well approximated by the incompressible flow simulation. The boundary conditions at solid surfaces on this system for the accoustic pressure field are derived in a similar manner, with

$$\Big(\frac{\partial^2}{\partial x_i^2} - \frac{1}{c_0^2}\frac{\partial^2}{\partial t^2}\Big)p = b \quad \text{where} \quad b = \frac{\partial^2 p}{\partial n^2}, ?? \tag{13.15b}$$

where, again, the RHS term b is taken as a source term using the hydrodynamic pressure field well approximated by the incompressible flow simulation.

Note that, by writing the PDE (13.15a) as a first order system

$$\mathbf{q}' = \begin{pmatrix} 0 & 1 \\ c_0^2\dfrac{\partial^2}{\partial x_i^2} & 0 \end{pmatrix}\mathbf{q} + \begin{pmatrix} 0 \\ c_0^2\dfrac{\partial^2 T_{ij}}{\partial x_i \partial x_j} \end{pmatrix} \quad \text{where} \quad \mathbf{q} = \begin{pmatrix} p \\ p' \end{pmatrix},$$

discretizing in space, and enforcing the boundary conditions (13.15b), ths system may easily be solved using the techniques already presented (e.g., Crank-Nicholson in time and Fourier/Padé in space). Approaches based on **Green's functions** may also be employed. ?

13.A Diablo

The numerical algorithm described in the first half of this chapter is implemented in the open-source code **diablo**, available at http://numerical-renaissance.com/diablo. Note that the diablo code is distributed under the **GNU General Public License**:

> Diablo is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License (distributed with diablo) for more details.

There are several README files that accompany the code to help get the reader started running it. For clarity, some of the details of the numerical implementation of the spatial and temporal discretization schemes described above are now described further, in order of appearance in the code. Consistent with the notation used above, the RHS of the momentum equations are accumulated in the arrays $\mathbf{R}_i$. The Runge-Kutta terms will be stored in $\mathbf{F}_i$ (since they are relatively expensive to calculate) and saved for use at the next RK substep. An extra storage array called $\mathbf{S}$ is also defined.

Care has been taken in the numerical implementations in order to minimize both the number of FFTs per timestep and the number of full-sized storage arrays. The algorithms detailed in the following two subsections use 11 full-sized storage arrays. In these presentations, $\widehat{\mathbf{u}}_i$, $\widehat{\mathbf{p}}$, $\widehat{\mathbf{R}}_i$, $\widehat{\mathbf{F}}_i$, and $\widehat{\mathbf{S}}_i$ denote the Fourier-space representations of $\mathbf{u}_i$, $\mathbf{p}$, $\mathbf{R}_i$, $\mathbf{F}_i$, and $\mathbf{S}$. Note that the physical- and Fourier-space representations of any given array occupy the same location in memory, with the FFT transforming from one representation to the other performed in place in the computer memory.

As in the previous section, in physical space, the notation $\mathbf{u}_1 * \mathbf{u}_2$ denotes the pointwise product of the vector $\mathbf{u}_1$ with the vector $\mathbf{u}_2$ at each gridpoint. In Fourier space, $\mathbf{k}_x * \hat{\mathbf{u}}_1$ denotes the multiplication of each Fourier coefficient of $\mathbf{u}_1$ by the corresponding streamwise wavenumber[12], $\hat{\mathbf{R}}_i / \mathbf{k}_x^2$ denotes the division of each Fourier coefficient of $\mathbf{u}_1$ by the square of the corresponding streamwise wavenumber, etc.

Exercises

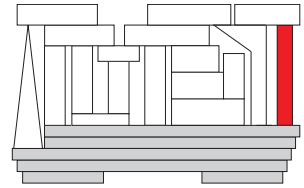
References

Pozrikidis, C (1997) *Introduction to Theoretical and Computational Fluid Dynamics*. Oxford.

[12] Note that this is done without ambiguity even though, $\mathbf{k}_x$ is a one-dimensional array of wavenumbers, whereas $\mathbf{u}_1$ is a three-dimensional array of Fourier coefficients.

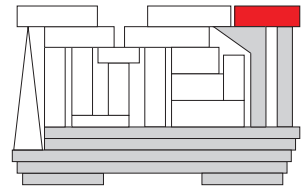
Part III
Optimization

14 Linear programming 431
15 Derivative-free minimization 433
16 Derivative-based minimization 453



Chapter 14

Linear programming



Chapter 15

Derivative-free minimization

Contents

15.1 Bracketing approaches for scalar minimization . . . 435
15.1.1 Golden section search . . . 436
15.1.2 Inverse quadratic interpolation . . . 438
15.1.3 Brent's method . . . 438
15.2 Lattice based derivative-free optimization via global surrogates . . . 440
15.2.1 Introduction . . . 440
15.2.2 Applying lattice theory to the coördination of derivative-free optimization . . . 442
15.2.3 Global optimization strategies leveraging kriging-based interpolation . . . 448
15.A LABDOGS . . . 450
15.B αDOGS . . . 451
Exercises . . . 451

It is often the case that one needs to minimize[1] a scalar function with respect to the scalar or vector $\mathbf{x}$; we thus investigate this problem in some detail in both this chapter and the next. Two cases of particular interest have already been mentioned: one arises in the iterative solution of nonlinear equations (see §3.1.2), and the other in the iterative solution of high-dimensional linear equations (see §3.2.3). Optimization is also a problem of interest in its own right in order to tune certain parameters affecting a system to achieve a desired objective (stated as the minimization of a cost function), as discussed further §20.

Minimization algorithms fall into two broad catagories: derivative-free methods, as discussed in §15, and derivative-based methods, to be discussed in §16. Derivative-free methods are appropriate for cases in which the function is not smooth at small scales in the optimization space $\mathbf{x}$, and thus the local gradient and Hessian information is not useful in determining the minimum of $J(\mathbf{x})$. Derivative-based methods, on the other hand, are well suited for smooth functions, for which gradient and Hessian information may be leveraged to greatly accelerate convergence to the minimum point.

The fundamental ideas underlying most effective derivative-free methods are twofold:

- *keep function evaluations far apart until convergence is approached*, and

[1] Maximization of $J(\mathbf{x})$ is equivalent to minimization of $-J(\mathbf{x})$; we thus focus on minimization, without loss of generality.

- *interpolate based on existing function evaluations* to identify promising new regions of the optimization space to explore with additional function evaluations.

We will encounter these themes repeatedly in both §15.1 and §15.2.

View

Algorithm 15.1: A simple bracketing algorithm.

```
function [AA,AB,AC,JA,JB,JC]=Bracket(AA,AB,JA,X,P,V) % Numerical Renaissance Codebase 1.0
JB=ComputeJ(X+AB*P,V);  if JB>JA; [AA,AB]=Swap(AA,AB); [JA,JB]=Swap(JA,JB); end
AC=AB+2*(AB-AA); JC=ComputeJ(X+AC*P,V);
end
end % function Bracket
```

Algorithm 15.2: An improved bracketing algorithm.

View

```
function [AA,AB,AC,JA,JB,JC]=BracketPress(AA,AB,JA,X,P,V)
% Numerical Renaissance Codebase 1.0
% INPUT: {AA,AB} are guesses of A near a minimum of J(A)=ComputeJ(X+A*P), with JA=J(AA).
% OUTPUT: {AA,AB,AC} bracket the minimum of J(A), with values {JA,JB,JC}.
JB=ComputeJ(X+AB*P,V);  if JB>JA; [AA,AB]=Swap(AA,AB); [JA,JB]=Swap(JA,JB); end
AC=AB+2*(AB-AA); JC=ComputeJ(X+AC*P,V);
while JB>=JC                          % At this point, {AA,AB,AC} has JA>=JB>=JC.
  flag=0; AL=AB+100*(AC-AB);          % Will allow exploration out to AL during this iteration.
  T=(AB-AA)*(JB-JC); D=(AB-AC)*(JB-JA); N=(AB-AC)*D-(AB-AA)*T; D=2.*(T-D);
  if (D==0), AN=AL; else; AN=AB+N/D; end                % Do a parabolic fit [see (15.2)]
  if (AB-AN)*(AN-AC) > 0.                               % Fitted point AN between AB and AC.
    JN=ComputeJ(X+AN*P,V);                              % Evaluate fitted point AN.
    if      JN<JC, AA=AB; AB=AN; JA=JB; JB=JN; return; % {AB,AN,AC} is a bracketing triplet!
    elseif JN>JB, AC=AN;         JC=JN;        return; % {AA,AB,AN} is a bracketing triplet!
    else    AN=AC+4*(AC-AB); JN=ComputeJ(X+AN*P,V);     % Fit not useful. Compute new AN.
    end
  elseif (AN-AC)*(AL-AN) > 0.                           % Fitted point AN between AC and AL.
    JN=ComputeJ(X+AN*P,V);                              % Evaluate fitted point AN.
    if JN<JC                                            % Function still not increasing.
      AB=AC; AC=AN; AN=AC+4*(AC-AB); JB=JC; JC=JN; JN=ComputeJ(X+AN*P,V); % Compute new AN.
    end
  elseif (AL-AN)*(AC-AL)>=0.                            % Fitted point at or beyond AL limit.
    AN=AL; JN=ComputeJ(X+AN*P,V);                       % Evaluate limit point AL.
  else
    AN=AC+4*(AC-AB); JN=ComputeJ(X+AN*P,V);             % All other cases: compute new AN.
  end
  AA=AB; AB=AC; AC=AN; JA=JB; JB=JC; JC=JN;             % {AB,AC,AN} -> {AA,AB,AC}
end
end % function BracketPress
```

15.1 Bracketing approaches for scalar minimization

We first seek a reliable approach to minimize a scalar function of a scalar argument, $J(a)$, when a good initial guess for the minimum is not necessarily available. To do this, we begin with a "bracketing" approach analogous to that which was used for finding the root of a nonlinear scalar equation (see §**??**). Recall that **bracketing a root** means finding a *pair* $\{x_a, x_b\}$ for which $f(x_a)$ and $f(x_b)$ have opposite signs, so that a root must exist between x_a and x_b if the function $f(x)$ is continuous and bounded.

Analogously, **bracketing a minimum** means finding a *triplet* $\{a_a, a_b, a_c\}$ for which a_b is between a_a and a_c and for which $J(a_b) < J(a_a)$ and $J(a_b) < J(a_c)$, so that a minimum must exist between a_a and a_c if the function $J(a)$ is continuous and bounded. Such an initial bracketing triplet may often be found by trial and error. At times, it is convenient to have an automatic procedure to find such a bracketing triplet. For functions which are large and positive for sufficiently large $|a|$, a very simple approach is to start with an initial guess for the bracket and then geometrically scale out the downhill end until a bracketing triplet is found, as implemented in Algorithm 15.1. An accelerated approach, suggested by Press *et al.* (1986), is based on inverse quadratic interpolation (see §15.1.2), and is implemented in Algorithm 15.2.

15.1.1 Golden section search

Once the minimum of the non-quadratic function is bracketed, all that remains to be done is to refine these brackets. A simple algorithm to accomplish this, analogous to the bisection technique developed for scalar root finding, is called the golden section search. As illustrated in Figure 15.1a, let W^k be defined as the ratio of the smaller interval to the width of the bracketing triplet $\{a_a^k, a_b^k, a_c^k\}$ such that

$$W^k = \frac{a_b^k - a_a^k}{a_c^k - a_a^k} \qquad \Rightarrow \qquad 1 - W^k = \frac{a_c^k - a_b^k}{a_c^k - a_a^k}.$$

We now pick a new trial point a_n^k and define $Z^k = \dfrac{a_n^k - a_b^k}{a_c^k - a_a^k} \quad \Rightarrow \quad a_n^k = a_b^k + Z^k(a_c^k - a_a^k)$.

There are two possibilities:

a) if $J(a_n^k) > J(a_b^k)$, then $\{a_n^k, a_b^k, a_a^k\}$ becomes the new bracketing triplet (as in Figure 15.1b), and

b) if $J(a_n^k) < J(a_b^k)$, then $\{a_b^k, a_n^k, a_c^k\}$ becomes the new bracketing triplet.

Minimizing the width of this new (refined) bracketing triplet in the worst case, we take the width of both of these triplets as identical:

$$W^k + Z^k = 1 - W^k \qquad \Rightarrow \qquad Z^k = 1 - 2W^k. \tag{15.1}$$

Using the same algorithm for the refinement at each iteration k, we would like to ensure that a self-similar situation, of sorts, develops in which the ratio W^k is constant from one iteration to the next, *i.e.*, $W^k = W^{k+1}$. Thus, dropping the superscripts on W and Z, which we assume approach constants once the iterations become self-similar, and describing the width of the new bracket in terms of both the variables at iteration k and the variables at iteration $k+1$ (see Figure 15.1), we may write

a) $Z^k d^k = W^{k+1} d^{k+1} \quad \Rightarrow \quad W/Z = d^k/d^{k+1} = 1/(W+Z)$ if $\{a_n^k, a_b^k, a_a^k\}$ is the new bracketing triplet, or

b) $Z^k d^k = W^{k+1} d^{k+1} \quad \Rightarrow \quad W/Z = d^k/d^{k+1} = 1/(1-W)$ if $\{a_b^k, a_n^k, a_c^k\}$ is the new bracketing triplet.

Inserting (15.1), both of these conditions reduce to the relation

$$W^2 - 3W + 1 = 0,$$

which (because $0 < W < 1$) implies that

$$W = \frac{3-\sqrt{5}}{2} \approx 0.381966 \qquad \Rightarrow \qquad 1 - W = \frac{\sqrt{5}-1}{2} \approx 0.618034 \qquad \text{and} \qquad Z = \sqrt{5} - 2 \approx 0.236068.$$

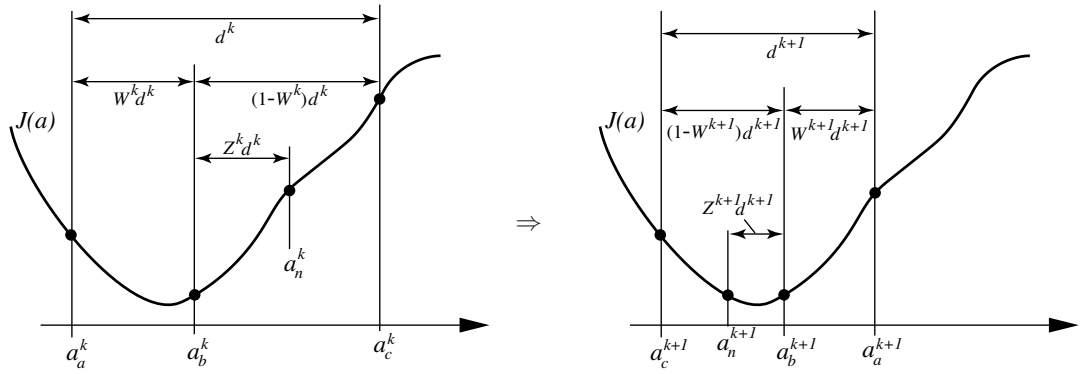
These proportions are referred to as the golden section, and are prevalent in Renaissance art and architecture.

To summarize, the golden section algorithm takes an initial bracketing triplet $\{a_a^0, a_b^0, a_c^0\}$, computes a new data point at $a_n^0 = a_b^0 + Z(a_c^0 - a_a^0)$ where $Z = 0.236068$, and then:

a) if $J(a_n^0) > J(a_b^0)$, the new triplet is $\{a_a^1, a_b^1, a_c^1\} = \{a_n^0, a_b^0, a_a^0\}$, or

b) if $J(a_n^0) < J(a_b^0)$, the new triplet is $\{a_a^1, a_b^1, a_c^1\} = \{a_b^0, a_n^0, a_c^0\}$.

The process continues on the new (refined) bracketing triplet in an iterative fashion until the desired tolerance is reached. Even if the initial bracketing triplet is not in the ratio of the golden section, repeated application of this algorithm quickly brings the triplet into this ratio as it is refined. Note that convergence is attained linearly: each bracket of the minimum is 0.618034 times the width of the previous bracket. This is slightly slower than the convergence of the bisection algorithm for nonlinear root-finding, in which each bracket of the root was 0.5 times the width of the previous bracket.



k'th iteration $(k+1)$'th iteration

Figure 15.1: Two steps of a golden section search. Note that $\{a_a^k, a_b^k, a_c^k\}$ is referred to as the bracketing triplet at iteration k, where a_b^k is between a_a^k and a_c^k and a_b^k is assumed to be closer to a_a^k than it is to a_c^k. A new guess is made at point a_n^k and the bracket is refined by retaining those three of the four points which maintain the tightest bracket. The reduction of the interval continues at the following iterations in a self-similar fashion.

View

Algorithm 15.3: The golden section search for 1D minimization.

```
function [AB,JB]=Golden(AA,AB,AC,JA,JB,JC,T,X,P,V)    % Numerical Renaissance Codebase 1.0
% INPUT: {AA,AB,AC} bracket a minimum of J(A)=ComputeJ(X+A*P), with values {JA,JB,JC}.
% OUTPUT: AB locally minimizes J(A), with accuracy T*abs(AB) and value JB.
% WARNING: this routine is slow.  Use Brent instead.
if abs(AB-AA) > abs(AC-AB); [AA,AC]=Swap(AA,AC); [JA,JC]=Swap(JA,JC); end  % Reorder data
for ITER=1:50
  if abs(AC-AB) < T*abs(AB)+1e-25, ITER, return, end
  AN = AB + 0.236068*(AC-AB);   JN = ComputeJ(X+AN*P,V);
  if (JN > JB)
    AC=AA; JC=JA; AA=AN; JA=JN;  % Center new triplet on AB (AB already in position)
  else
    AA=AB; JA=JB; AB=AN; JB=JN;  % Center new triplet on AN (AC already in position)
  end
  if V, disp(sprintf('%9.5f %9.5f %9.5f %9.5f %9.5f %9.5f',AA,AB,AC,JA,JB,JC)); end
end
end % function Golden
```

View

```
function [J]=ComputeJ(X,V)                              % Numerical Renaissance Codebase 1.0
J=X+X.^6+1*sin(15*X);
if V;
  switch V; case 1, s='kx'; case 2, s='rx'; case 3, s='bx'; case 4, s='gx'; end;
  plot(X,J,s);
end
end % function ComputeJ
```

View

```
% script BracketGoldenBrentTest                        % Numerical Renaissance Codebase 1.0
clear; figure(1); clf; hold on;
AA=0; AB=0.1; JA=ComputeJ(AA,1);
disp('Bracket...'); [AA,AB,AC,JA,JB,JC] = Bracket(AA,AB,JA,0,1,1), pause;
disp('Golden...  '); [A,J] = Golden(AA,AB,AC,JA,JB,JC,0.0001,0,1,2), pause;
disp('InvQuad...'); [A,J] = InvQuad(AA,AB,AC,JA,JB,JC,0.0001,0,1,3), pause;
disp('Brent...   '); [A,J] = Brent(AA,AB,AC,JA,JB,JC,0.0001,0,1,4)
```

View

15.1.2 Inverse quadratic interpolation

Recall from §**??** that, when a function $f(x)$ is **locally linear** (meaning that, over the local interval of interest, its shape is well-approximated by a linear function), the false position method is an efficient technique to find the root of the function based on function evaluations alone. The false position method is based on the construction of successive linear interpolations of recent function evaluations, taking each new estimate of the root of $f(x)$ as that value of x for which the value of the linear interpolant is zero.

Analogously, when a function $J(a)$ is **locally quadratic** (meaning that, over the local interval of interest, its shape is well-approximated by a quadratic function), the minimum point of the function may be found via an efficient technique based on function evaluations alone. At the heart of this technique is the construction of successive quadratic interpolations based on recent function evaluations, taking each new estimate of the minimum of $J(a)$ as that value of a for which the value of the quadratic interpolant is minimum. For example, given data points $\{a_a, J_a\}$, $\{a_b, J_b\}$, and $\{a_c, J_c\}$, the quadratic interpolant is given by the Lagrange interpolant:

$$Q(a) = J_a \frac{(a-a_b)(a-a_c)}{(a_a-a_b)(a_a-a_c)} + J_b \frac{(a-a_a)(a-a_c)}{(a_b-a_a)(a_b-a_c)} + J_c \frac{(a-a_a)(a-a_b)}{(a_c-a_a)(a_c-a_b)},$$

as described in §7.3.2. Setting $dQ(a)/da = 0$ to find the critical point of this quadratic yields

$$0 = J_a \frac{2a-a_b-a_c}{(a_a-a_b)(a_a-a_c)} + J_b \frac{2a-a_a-a_c}{(a_b-a_a)(a_b-a_c)} + J_c \frac{2a-a_a-a_b}{(a_c-a_a)(a_c-a_b)}.$$

Multiplying by $(a_a-a_b)(a_b-a_c)(a_c-a_a)$ and then solving for a gives the desired value of a which is a critical point of the interpolating quadratic:

$$\begin{aligned} a &= \frac{1}{2} \frac{J_a(a_b+a_c)(a_b-a_c) + J_b(a_a+a_c)(a_c-a_a) + J_c(a_a+a_b)(a_a-a_b)}{J_a(a_b-a_c) + J_b(a_c-a_a) + J_c(a_a-a_b)} \\ &= \ldots = a_b + \frac{1}{2} \frac{(a_b-a_c)^2(J_b-J_a) - (a_b-a_a)^2(J_b-J_c)}{(a_b-a_a)(J_b-J_c) - (a_b-a_c)(J_b-J_a)} \end{aligned} \tag{15.2}$$

Since the points $\{a_a, a_b, a_c\}$ bracketed a minimum, not a maximum, the critical point found by the above formula is a minimum point of the interpolating quadratic, and will lie somewhere between a_a and a_c. Together with a few ad hoc checks to prevent the minimization algorithm from stalling, (15.2) is used at the heart of Algorithm 15.4.

15.1.3 Brent's method

As with the false position technique for accelerated bracket refinement for the problem of scalar root finding, the inverse quadratic technique can also stall for a variety of scalar functions $J(a)$ one might attempt to minimize, and a careful implementation should fall back on a simpler, more pedestrian appraoch when such stalling is detected.

A hybrid technique, referred to as Brent's method (Algorithm 15.5), has thus been developed which combines the reliable convergence benefits of the golden section search with the ability of the inverse quadratic interpolation technique to home in rapidly on the solution when the minimum point is approached. Switching in a reliable fashion from one technique to the other without stalling requires a few careful ad hoc checks, and in fact slightly slows down the convergence of the pure inverse quadratic code for some well behaved functions. However, the remarkably robust convergence properties of Brent's method generally render it overall the most suitable choice for 1D minimization.

Algorithm 15.4: Inverse quadratic interpolation for 1D minimization.

View

```
function [AB,JB]=InvQuad(AA,AB,AC,JA,JB,JC,T,X,P,V) % Numerical Renaissance Codebase 1.0
% INPUT: {AA,AB,AC} bracket a minimum of J(A)=ComputeJ(X+A*P), with values {JA,JB,JC}.
% OUTPUT: AB locally minimizes J(A), with accuarcy T*abs(AB) and value JB.
% WARNING: this routine stalls on functions like J(A)=A^6+A.  Use Brent instead.
if AA>AC; [AA,AC]=Swap(AA,AC); [JA,JC]=Swap(JA,JC); end
for ITER=1:500;                                          % {AA,AB,AC} is starting triplet
  T1=T*abs(AB)+1.E-25;  T3=T1*0.99;                      % Initialize
  AM=0.5*(AA+AC); if abs(AC-AA)<4.*T1; ITER, return; end          % Check convergence
  T=(AB-AA)*(JB-JC); D=(AB-AC)*(JB-JA); N=(AB-AC)*D-(AB-AA)*T; D=2*(T-D);  % Parabolic fit
  AINC=N/D;                                                       % [see (15.2)]
  if (AB-AA<T1 & AINC<=0); AINC=T3; elseif (AC-AB<T1 & AINC>=0); AINC=-T3; end
  if abs(AINC)<T1; AN=AB+T3*sign(AINC); else; AN=AB+AINC; end; JN=ComputeJ(X+AN*P,V);
  if (AB-AA)*(AN-AB)>0;                                  % N is between B and C
    if (JN > JB) AC=AN; JC=JN;                           % {AA,AB,AN} is new triplet
    else         AA=AB; JA=JB; AB=AN; JB=JN; end;        % {AB,AN,AC} is new triplet
  else                                                   % N is between A and B
    if (JN > JB) AA=AN; JA=JN;                           % {AN,AB,AC} is new triplet
    else         AC=AB; JC=JB; AB=AN; JB=JN; end;        % {AA,AN,AB} is new triplet
  end
  if V, disp(sprintf('%19.15f %19.15f %19.15f',AA,AB,AC)); end
end
end % function InvQuad
```

Algorithm 15.5: Brent's algorithm for 1D minimization.

View

```
function [AB,JB]=Brent(AA,AB,AC,JA,JB,JC,TOL,X,P,V)  % Numerical Renaissance Codebase 1.0
% INPUT: {AA,AB,AC} bracket a minimum of J(A)=ComputeJ(X+A*P), with values {JA,JB,JC}.
% OUTPUT: AB locally minimizes J(A), with accuarcy TOL*abs(AB) and value JB.
AINC=0; AL=min(AC,AA); AR=max(AC,AA);
if (abs(AB-AA) > abs(AC-AB)); [AA,AC]=Swap(AA,AC); [JA,JC]=Swap(JA,JC); end;
for ITER=1:50;
  if ITER<3; AINT=2*(AR-AL); end; TOL1=TOL*abs(AB)+1E-25; TOL2=2*TOL1; FLAG=0; % Initialize
  AM=(AL+AR)/2; if (AR-AL)/2+abs(AB-AM)<TOL2; ITER, return; end        % Check convergence
  if (abs(AINT)>TOL1 | ITER<3)
    % Perform a parabolic fit based on points {AA,AB,AC} [see (15.2)]
    T=(AB-AA)*(JB-JC); D=(AB-AC)*(JB-JA); N=(AB-AC)*D-(AB-AA)*T; D=2*(T-D);
    if D<0.; N=-N; D=-D; end; T=AINT; AINT=AINC;
    if (abs(N)<abs(D*T/2) & N>D*(AL-AB) & N<D*(AR-AB)) % AINC=N/D within reasonable range?
      AINC=N/D; AN=AB+AINC; FLAG=1;                    % Success! AINC is new increment.
      if (AN-AL<TOL2 | AR-AN<TOL2); AINC=abs(TOL1)*sign(AM-AB); end  % Fix if AN near ends
    end
  end
  % If parabolic fit unsuccessful, do golden section step based on bracket {AL,AB,AR}
  if FLAG==0; if AB>AM; AINT=AL-AB; else; AINT=AR-AB; end; AINC=0.381966*AINT; end
  if abs(AINC)>TOL1; AN=AB+AINC; else; AN=AB+abs(TOL1)*sign(AINC); end
  JN=ComputeJ(X+AN*P,V);
  if JN<=JB                                  % Keep 6 (not necessarily distinct) points
    if AN>AB; AL=AB; else; AR=AB; end        % defining the interval from one iteration
    AC=AA; JC=JA; AA=AB; JA=JB; AB=AN; JB=JN; % to the next:
  else                                       % {AL,AB,AR} bracket the minimum
    if AN<AB; AL=AN; else; AR=AN; end        % AB=Lowest point, most recent if tied w/ AA
    if (JN<=JA | AA==AB)                     % AA=Second-to-lowest point.
      AC=AA; JC=JA; AA=AN; JA=JN;            % AC=Third-to-lowest point
    elseif (JN<=JC | AC==AB | AC==AA)        % AN=Newest point
      AC=AN; JC=JN;                          % Parabolic fit based on {AA,AB,AC}
    end                                      % Golden section search based on {AL,AB,AR}
  end
  if V, disp(sprintf('%d %9.5f %9.5f %9.5f %9.5f %9.5f',FLAG,AA,AB,AC,AL,AR)); end
end
end % function Brent
```

15.2 Lattice based derivative-free optimization via global surrogates

As mentioned previously, the core idea of all efficient search algorithms[2] for derivative-free optimization is to *keep function evaluations far apart until convergence is approached*. **Generalized Pattern Search** (**GPS**) algorithms accomplish this by coördinating the search with an underlying grid which is refined, and coarsened, as appropriate. Rather than using the cubic grid (the typical choice), we recommend for this purpose the use of lattices derived from n-dimensional sphere packings. Such lattices are significantly more uniform and have much higher kissing numbers (that is, they have many more nearest neighbors) than their cubic grid counterparts; both of these facts make them much better suited for coördinating GPS algorithms. One of the most efficient subclasses of GPS algorithms, known as the **Surrogate Management Framework** (**SMF**), alternates between an exploratory **search** over an interpolating **surrogate** function which summarizes the trends exhibited by the existing function evaluations, and an exhaustive **poll** which checks the function on neighboring points to confirm or confute the local optimality of any given **candidate minimum point** (**CMP**) on the underlying grid.

The algorithm presented here uses efficient lattices based on n-dimensional sphere packings (see §**??**) to coördinate such surrogate-based optimizations while incorporating an efficient global search strategy based on both the predictor and the uncertainty of a kriging model (see §7.5.3) of the function, thereby developing an extremely efficient algorirthm for Lattice Based Derivative-free Optimization via Global Surrogates, dubbed LABDOGS. A review of §**??** and 7.5.3 is thus highly recommended before proceeding.

15.2.1 Introduction

The minimization of computationally expensive, high-dimensional functions is often most efficiently performed via gradient-based optimization algorithms such as steepest descent, conjugate gradient, and L-BFGS, as discussed further in §16. In complex dynamic systems for which an accurate computer model is available, the gradient required by such algorithms may often be found by adjoint analysis (§20). However, when the function in question is not sufficiently smooth to leverage gradient information effectively during its optimization (see, e.g., Figure 15.2), a derivative-free approach is necessary. Such a scenario is evident, for example, when optimizing a finite-time-average approximation of an infinite-time-average statistic of a chaotic system such as a turbulent flow. Such an approximation may be determined via simulation or experiment. The truncation of the averaging window used to determine this approximation renders derivative-based optimization strategies ill suited, as the truncation error, though small, is effectively decorrelated from one flow simulation/experiment to the next. This effective decorrelation of the truncation error is reflected, for example, by the exponential growth, over the entire finite time horizon considered, of the adjoint field related to the optimization problem of interest in the simulation-based setting.

Due to the sometimes significant expense associated with performing repeated function evaluations (in the above example, turbulent flow simulations or experiments), a derivative-free optimization algorithm which, with reasonable confidence, converges to within an accurate tolerance of the global minimum of a nonconvex function of interest with a minimum number of function evaluations is desired. It is noted that, in the general case, proof of convergence of an optimization algorithm to a global minimum is possible only when, in the limit that the total number of function evaluations, N, approaches infinity, the function evaluations become dense in the feasible region of parameter space (Torn & Zilinskas, 1987). Though the algorithm developed presented here, when implemented properly, satisfies this condition, so do far inferior approaches, such as the

[2] The problem of optimization is, in general, **NP-hard** (that is, the only available rigorous solutions to problems of this class incorporate exhaustive searches). Thus, if searching over an infinite number of possibilities of the optimization variables, such problems would require infinite time to solve completely. There have been a variety of methods posed over the years to explore for solutions of derivative-free problems which do not have built-in safeguards to keep function evaluations far apart until convergence is approached, including the so-called **genetic algorithms** (**GA**) and **simulated annealing** (**SA**) algorithms. With the appropriate heuristics, as discussed in this chapter, much more efficient algorithms are possible; we thus do not discuss the GA and SA approaches further here.

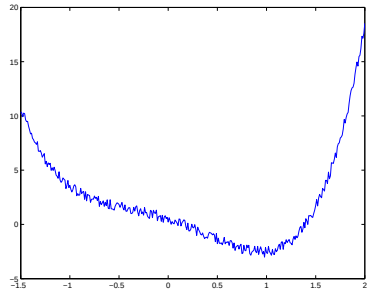


Figure 15.2: Prototypical nonsmooth optimization problem for which local gradient information is ill-suited to accelerate the optimization algorithm.

rather unintelligent algorithm which we call **Exhaustive Sampling** (**ES**), which simply covers the feasible parameter space with a grid, evaluates the function at *every* gridpoint, refines the grid by a factor of two, and repeats until terminated. Thus, a guarantee of global convergence is *not* sufficient to establish the efficiency of an optimization algorithm. If function evaluations are relatively expensive, and thus only a relatively small number of function evaluations can be afforded, effective heuristics for rapid convergence are certainly just as important, if not significantly more important, than rigorous proofs of the behavior of the optimization algorithm in the limit that $N \to \infty$, a limit that might be argued to be of limited relevance when function evaluations are expensive. Careful attention to such heuristics thus forms an important foundation for the present study.

If, for the moment, we give up on the goal of global convergence, perhaps the simplest grid-based derivative-free optimization algorithm, which we identify with the name **Successive Polling** (**SP**), proceeds as follows. Start with a coarse grid and evaluate the function at some starting point on this grid, identified as the first **candidate minimum point** (**CMP**). Then, poll (that is, evaluate) the function values on gridpoints which neighbor the CMP in parameter space, at a sufficient number of gridpoints to *positively span*[3] the feasible neighborhood of the CMP [this step ensures convergence, as discussed further in Torczon 1997 and Coope & Price 2001]. When polling:

(a) If any poll point is found to have a function value lower than that of the CMP, immediately consider this new point the new CMP and terminate the present poll step.
(b) If all poll points are found to have function values higher than that of the CMP, refine the grid by a factor of two.

A new poll step is then initiated, either around the new CMP or on the refined grid, and the process repeated until terminated. Though the basic SP algorithm described above, on its own, is not very efficient, there are a variety of effective techniques for accelerating it. All grid-based schemes which effectively build on the basic SP idea described above are classified as **Generalized Pattern Search** (**GPS**) algorithms.

The most efficient subclass of GPS algorithms, known as the **Surrogate Management Framework** (**SMF**; see Booker *et al.*, 1999), leverages inexpensive interpolating "surrogate" functions (often, kriging interpolations are used) to interpolate the available function evaluations and provide suggested regions of parameter space in which to perform new function evaluations between each poll step. SMF algorithms thus alternate beween two steps:

(i) **Search** over the inexpensive interpolating function to identify, based on the existing function evaluations, the most promising gridpoint at which to perform a new function evaluation. Perform a function evaluation

[3] That is, such that any feasible point in the neighborhood of the CMP can be reached via a *linear combination with non-negative coefficients* of the vectors from the CMP to the poll points. For further discussion, see §1.3.

at this point, update the interpolating function, and repeat. The search step is terminated when this search algorithm returns a gridpoint at which either the function has already been evaluated or the function, once evaluated, has a value greater than that of the CMP.
(ii) **Poll** the neighborhood of the new CMP identified by the search algorithm, following rules (a) and (b) above.

Note that there is substantial flexibility during the search step described above. An effective search strategy is essential for an efficient SMF algorithm. In the case that the search behaves poorly and fails to return improved function values, which often happens when the function of interest is very flat (such as near the minimum of the Rosenbrock test function), the SMF algorithm essentially reduces to the SP algorithm. If, however, the surrogate-based search is effective, the SMF algorithm will converge to a minimum far faster than the simple SP search. As the search and poll steps are essentially independent of one another, we will discuss them each in turn in the sections that follow, then present how we have combined them in our highly efficient new optimization algorithm and its realization in the open-source software package dubbed *Checkers*.

Note also that the interpolating surrogate function of the SMF may be used to order the function evaluations at the poll points such that those poll points which are most likely to have a function value lower than that of the CMP are evaluated first. By so doing, the poll steps will, on average, terminate much sooner, and the computational cost of the overall algorithm may be substantially reduced.

15.2.2 Applying lattice theory to the coördination of derivative-free optimization

At the heart of the SMF algorithm lies the discretizing grid or "lattice" to which all function evaluations are restricted, and which defines the set of points from which the poll set is selected at each poll step. Like in the game of Checkers (contrast American Checkers with Chinese Checkers), cubic grids are not the only possibility for discretizing parameter space in such an application. As the underlying lattice is the foundation for any GPS algorithm, we first define and compare the characteristics of various lattice alternatives to cubic grids.

As discussed in §**??**, there are two key drawbacks with cubic approaches to the coördination of derivative-free optimization algorithms. First, the *discretization of the optimization space is less uniform* when using the cubic grid as opposed to the available alternatives, as measured by the packing density Δ, the covering thickness Θ, and the normalized mean-squared quantization error G, as summarized in Table **??**. Second, the *configuration of nearest-neighbor gridpoints is poor* when using the cubic grid, as measured by the kissing number τ, which is an indicator of the degree of flexibility available when selecting a positive basis from nearest neighbors on the lattice. As seen by comparing the $n = 2$, $n = 8$, and $n = 24$ cases in Table **??**, these drawbacks become increasingly substantial as the dimension n is increased.

Recall in particular that the poll points described above must be selected to form a positive basis around the CMP (that is, a set of vectors such that any point in the feasible parameter space neighboring the CMP can be reached by a linear combination of these vectors with non-negative coefficients). Assuming computationally expensive function evaluations, minimizing the number of poll points while maintaining a positive basis around the CMP is of key importance in maximizing the efficiency of the SP algorithm. This highlights an obvious shortcoming of defining the poll points based on a cubic grid, where a complete poll step performed on a positive basis based on the nearest neighbors of a CMP requires $2n$ function evaluations; in more well-behaved lattices such as A_n, the positive basis requires only $n+1$ function evaluations (see Figure 15.3). In fact, most alternative lattices developed as n-dimensional sphere packings also require only $n+1$ nearest-neighbor points to form a positive basis. Thus, independent of the benefits in increased uniformity and decreased mean-square quantization error provided by the alternative lattices considered here, a factor of nearly 2 increase in SP efficiency is realized immediately as a direct consequence of the more convenient configuration of the nearest-neighbor points on these alternative lattices.

Note that it is possible to construct a positive basis with only $n+1$ points, referred to as a *minimal*

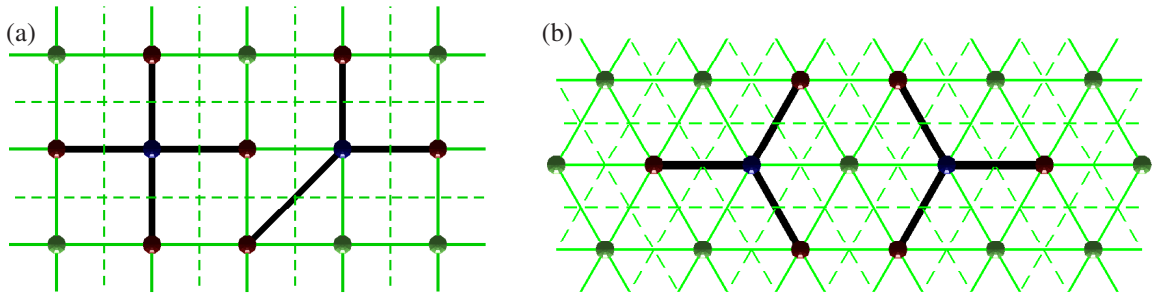


Figure 15.3: Configuration of the nearest-neighbor gridpoints in the 2D (a) square and (b) hexagonal lattices.

positive basis, in the n-dimensional cubic case if one poll point is used which is not a nearest-neighbor point, as indicated in $n = 2$ dimensions in Figure 15.3a. However, the vector to this oddball point is a factor of $\sqrt{n}$ longer than the remaining n vectors. Additionally, this oddball vector is at a much larger angle to all of the other vectors in the positive basis than these vectors are to themselves. As a consequence, the region to which the optimal point is effectively localized via polling a set of points distributed in such a fashion is increased greatly from that tight region resulting from a poll on a well-distributed positive basis on nearest-neighbor points, as possible when using the A_2 configuration as depicted in Figure 15.3b. By providing a poorly distributed poll set, the efficiency of the SP algorithm is significantly compromised following the oddball vector approach depicted in Figure 15.3a[4].

Note that it is not yet clear which of the four standard metrics introduced above (that is, Δ, Θ, G, τ) is/are in fact most relevant when selecting a lattice for coördinating a derivative-free optimization algorithm, and some experimentation will thus be required to make this selection optimally for each n. The one thing that is clear, however, is that the cubic grid is inferior (by orders of magnitude for even modest values of n) to the available alternative lattices by all four of these metrics, as discussed further and tabulated comprehensively in §**??**.

To extend lattice theory, as summarized in §**??**, to coördinate a derivative-free optimization algorithm, a few additional steps are needed, as described below.

Enumerating nearest-neighbor lattice points

All lattice points $\tilde{\mathbf{x}}^i \in \mathbb{R}^n$ which are nearest neighbors of the origin $\tilde{\mathbf{x}} = 0$ in any real lattice defined by a basis matrix B may be enumerated via the following algorithm.

0. Initialize $m = 1$.

1. Define a distribution of points $\tilde{\mathbf{z}}^i$ such that each element of each of these vectors is selected from the set of integers $\{-m,\ldots,0,\ldots,m\}$, and that *all possible vectors* that can be created in such a fashion, except the origin, are present (without duplication) in this distribution.

2. Compute the distance of each transformed point $\tilde{\mathbf{y}}^i = B\tilde{\mathbf{z}}^i$ in this distribution from the origin, and eliminate those points in the distribution that are farther from the origin than the minimum distance computed in the set.

[4]Taking this idea one step further, a relatively new class of methods, referred to as Mesh Adaptive Direct Search (MADS), polls based on a cubic grid but using *several* non-nearest-neighbor gridpoints. Though this approach has received much attention in recent years (see, e.g., Abramson, Audet, & Dennis 2005) and shows some promise, a poll of this sort has the unfortunate consequence of effectively localizing the minimum point to a much larger region of parameter space than does a poll based on nearest-neighbor points on a grid of the same density. We believe that a MADS-type approach is rendered unnecessary when a lattice with a significantly higher kissing number than that of the cubic grid is used.

3. Count the number of points remaining in the distribution. If this number equals the (known) kissing number of the lattice under consideration, as listed in Table 2 or 3, then determine an orthogonal $\hat{B}$ from B via Gram Schmidt orthogonalization, set $\tilde{\mathbf{x}}^i = \hat{B}^T \tilde{\mathbf{y}}^i$ for all i, and exit; otherwise, increment m and repeat from step 1.

Though this simple algorithm is not particularly efficient, it need not be, as the nearest neighbor distribution is identical around every lattice point, and thus this algorithm need only be run once during the initialization of the optimization code.

Testing for a positive basis

Given a subset of the nearest-neighbor lattice points, we will at times need an efficient test to determine whether or not the vectors to these points from the CMP form a positive basis of the feasible domain around the CMP. Without loss of generality, we will shift this problem so that the CMP corresponds to the origin in the discussion that follows.

A set of vectors $\{\tilde{\mathbf{x}}^1, \ldots, \tilde{\mathbf{x}}^k\}$ for $k \geq n+1$ is said to positively span $\mathbb{R}^n$ if any point in $\mathbb{R}^n$ may be reached via a linear combination of these vectors with non-negative coefficients. Since the $2n$ basis vectors $\{\mathbf{e}^1, \ldots, \mathbf{e}^n, -\mathbf{e}^1, \ldots, -\mathbf{e}^n\}$ positively span $\mathbb{R}^n$, a convenient test for whether or not the vectors $\{\tilde{\mathbf{x}}^1, \ldots, \tilde{\mathbf{x}}^k\}$ positively span $\mathbb{R}^n$ is to determine whether or not each vector in the set $\{\mathbf{e}^1, \ldots, \mathbf{e}^n, -\mathbf{e}^1, \ldots, -\mathbf{e}^n\}$ can be reached by a positive linear combination of the vectors $\{\tilde{\mathbf{x}}^1, \ldots, \tilde{\mathbf{x}}^k\}$. That is, for each vector $\mathbf{e}$ in the set $\{\mathbf{e}^1, \ldots, \mathbf{e}^n, -\mathbf{e}^1, \ldots, -\mathbf{e}^n\}$, a solution $\mathbf{z}$, with $z_i \geq 0$ for $i = 1, \ldots, n$, to the equation $\tilde{X}\mathbf{z} = \mathbf{e}$ is desired, where $\tilde{X} = \begin{pmatrix} \tilde{\mathbf{x}}^1 & \ldots & \tilde{\mathbf{x}}^k \end{pmatrix}$. If such a $\mathbf{z}$ exists for each vector $\mathbf{e}$, then the vectors $\{\tilde{\mathbf{x}}^1, \ldots, \tilde{\mathbf{x}}^k\}$ positiviely span $\mathbb{R}^n$; if such a $\mathbf{z}$ does not exist, then the vectors $\{\tilde{\mathbf{x}}^1, \ldots, \tilde{\mathbf{x}}^k\}$ do not positiviely span $\mathbb{R}^n$.

Thus, testing a set of vectors to determine whether or not it positively spans $\mathbb{R}^n$ may be reduced simply to testing for the existence of a solution to $2n$ well-defined linear programs in standard form. Techniques to perform such tests are well developed and readily available (see §14). Further, if a set of vectors positiviely spans $\mathbb{R}^n$, it is a simple matter to check whether or not this set of vectors is also a positive basis of $\mathbb{R}^n$, if such a check is necessary, simply by checking whether or not any subset of $k-1$ vectors chosen from this set also positively span $\mathbb{R}^n$. Note that a positive basis with k vectors will necessarily have k in the range $n+1 \leq k \leq 2n$.

Selecting a positive basis from the nearest-neighbor lattice points

Section 15.2.2 described how to enumerate all points which are nearest neighbors of the origin of a lattice (and thus, with the appropriate shift, all points which are nearest neighbors of any CMP on a lattice), and Section 15.2.2 described how to test a subset of such points to see if the vectors to these points form a positive basis around the CMP. We now present a general algorithm to solve the problem of selecting a positive basis from the nearest-neighbor points using the minimum number of new poll points possible, while creating the maximum achievable angular uniformity between the vectors from the CMP to each of these points. This problem has an interesting connection to Tammes' problem (Tammes 1930), which may be summarized as the question "Where should k repulsive individuals settle on a planet in order to be as far away from each other as possible?". In the present incarnation of this problem, we have an n-dimensional planet, and we need to distribute $k = n+m$ such repulsive individuals on a well-selected subset of a discrete set of locations (that is, the nearest-neighbor lattice points) at which the individuals are allowed to settle. Ideally[5], for $m = 1$, the solution to this discrete Tammes' problem will produce a positive basis with good angular uniformity; if it does not, we may successively increment m by one and try again until we succeed in producing a positive basis.

[5] That is, for a "good" lattice, such as that depicted in Figure 15.4.

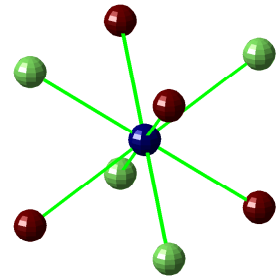


Figure 15.4: Two different positive bases on the bcc lattice D_3^*, shown in green and red around the blue CMP. Note the complete radial and angular uniformity as well as the flexibility in the orientation of the basis.

We have studied three algorithms for solving the problem of finding a positive basis while leveraging this discrete Tammes formulation:

Algorithm A. If the kissing number τ of the lattice under consideration is relatively large (that is, if $\tau \gg n$; for example, for the Leech lattice Λ_{24}), then a straightforward algorithm can first be used to solve Tammes' problem on a continuous sphere in n dimensions. This can be done simply and quickly by modeling the $n+m$ repulsive individuals with identical negatively charged particles, initializing the location of each such particle on the sphere randomly, and then, at each iteration, using a straightforward force-based algorithm[6] to move each particle along the surface of the sphere a small amount in the direction that the other particles are tending to push it, and iterating until the set of particles approaches an equilibrium. Then, each equilibrium point so determined may be quantized to the closest nearest-neighbor lattice point, as enumerated in §15.2.2.

Algorithm B. If the kissing number τ of the lattice under consideration is relatively small (that is, if τ is *not* well over an order of magnitude larger than n), then it turns out to be more expedient to solve the discrete Tammes' problem directly. To accomplish this, we distribute the $n+m$ negatively charged particles randomly on $n+m$ nearest-neighbor points, and then, at each iteration, move a few (two or three[7]) of these particles that are furtherest from equilibrium in the force-based model described above (that is, those particles which have the highest force component projected onto the surface of the sphere) into new positions selected from the available locations (enumerated in §15.2.2) which minimize the maximum force (projected onto the sphere) over the entire set of particles. Though each iteration of this algorithm involves an exhaustive search for placing the two or three particles in question, it converges quickly when τ is $O(100)$ or less.

Algorithm C. For intermediate kissing numbers τ, a hybrid approach may be used: a "good" initial distribution may be found using Algorithm *A*, then this distribution may be refined using Algorithm *B*.

In each of these algorithms, to minimize the number of new function evaluations required at each poll step, a check is first made to determine whether any previous function evaluations have already been performed on the set of nearest-neighbor lattice points. If so, then negatively-charged particles are fixed at these locations, while the remaining negatively-charged particles are adjusted via one of the three algorithms described above. By so doing, previously-calculated function values may be used with maximum effectiveness during the polling procedure. When performing the poll step of a surrogate-based search, in order to orient the new poll set favorably, a negatively-charged particle is also fixed at the nearest neighbor point with the lowest value of the surrogate function; when polling, this poll point is evaluated first.

[6] In this model, the repulsive force exerted by any two particles on each other is proporational to the inverse square of the distance between the particles.

[7] Moving more than two or three particles at a time in this algorithm makes each iteration computationally intensive, and has little impact on overall convergence of the algorithm.

The iterative algorithms described above, though in practice quite effective, are not guaranteed to converge from arbitrary initial conditions to a positive basis for a given value of m, even if such a positive basis exists. To address this issue, if either algorithm fails to produce a positive basis, the algorithm may be repeated using a new random starting distribution. Our numerical tests have indicated that this repeated random initialization scheme usually generates a positive basis within a few initializations when such a positive basis indeed exists.

Since at times there exists no minimal positive basis on the nearest-neighbor lattice points, particularly when the previous function evaluations being leveraged are poorly configured, the number of new random initializations is limited to a prespecified value. Once this value is reached, m is increased by one and the process repeated. As the cost of each function evaluation increases, the user can increase the number of random initializations attempted using one of the above algorithms for each value of m in order to avoid the computation of extraneous poll points that might in fact be unnecessary if sufficient exploration by the discrete Tammes algorithm is performed.

Numerical tests have demonstrated the efficacy of this rather simple basis-finding strategy, which reliably generates a positive basis, even when leveraging a relatively poor configuration of previous function evaluations, while keeping computational costs to a minimum. Additionally, the strategy itself lacks any explicit dependence on the lattice being used; the only inputs to it are the dimension of the problem, the locations of the nearest-neighbor lattice points, and the identification of those nearest-neighbor lattice points for which previous function evaluations are available.

Implementation of feasible domain boundaries

When implementing a global search in n dimensions, or even when implementing a local search on a function which is ill-defined for certain nonphysical values of the parameters (such as negative concentrations of chemicals), it is important to restrict the optimization algorithm to look only over a prespecified bounded "feasible" region of parameter space. For simplicity, the present work assumes rectangular constraints on this feasible domain (that is, simple upper and lower bounds on each parameter value). An efficient n-dimensional lattice with packing radius ρ_n (see §**??**) is used to quantize the interior of the feasible domain, efficient $(n-1)$-dimensional lattices with packing radius $\rho_{n-1} = \rho_n/2$ are used to quantize the portions of the boundary of the feasible domain with one active constraint (that is, the "faces"), efficient $(n-2)$-dimensional lattices with packing radius $\rho_{n-2} = \rho_n/4$ are used to quantize the portions of the boundary of the feasible domain with two active constraints (that is, the "edges"), etc. The present section describes how to keep from violating the boundaries of the feasible domain, and how to move on and off of these boundaries as appropriate, while carefully restricting all function evaluations to the interior and boundary lattices in order to coördinate an efficient search.

We distinguish between two scenarios in which the polling algorithm as described thus far must be adjusted to avoid violating the $(n-1)$-dimensional boundaries[8] of the feasible domain. In the first scenario, the CMP is relatively far (that is, greater than ρ_n but less than $2\rho_n$) from the boundary of the feasible domain, and thus one or more of the poll points as determined by one of the algorithms proposed in §15.2.2 might land slightly outside this boundary. In this scenario, an effective remedy is simply to *eliminate* all lattice points which land outside of the feasible domain from the list of potential poll points (see §15.2.2), and then to *augment* to this restricted list of potential poll points all those lattice points on the nearby $(n-1)$-dimensional constraint surface which are less than $2\rho_n$ from the CMP. From this modified list of potential poll points, the poll set may be selected in the usual fashion using one of the algorithms described in §15.2.2.

In the second scenario, the CMP is relatively close (that is, less than ρ_n) to the boundary of the feasible domain. In this scenario, it is most effective simply to shift the CMP onto the nearest lattice point on the $(n-1)$-dimensional constraint surface. With the CMP on the feasible domain boundary, each poll step explores a minimum positive basis selected on the lattice quantizing the $(n-1)$-dimensional boundary and, in addition,

[8] That is, the portions of the boundary with a single active constraint.

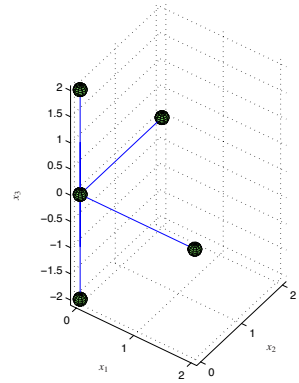


Figure 15.5: A scenario in which a CMP at $\mathbf{x} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$ sits on an $(n-2) = 1$-dimensional edge of an $n = 3$-dimensional feasible region with bounds $x_1 \geq 0$ and $x_2 \geq 0$. Note that the feasible neighborhood of this edge is positively spanned by the nearest neighbors on the integer lattice, and that two additional vectors are added to the poll set to facilitate moving off of each of these active constraint boundaries.

polls an additional lattice point on the interior of the feasible domain to allow the algorithm to move back off this constraint boundary. Ideally, this additional point would be located on a inward-facing vector normal to the $(n-1)$-dimensional feasible domain boundary a distance ρ_n from the CMP; we thus choose the interior lattice point closest to this location.

Multiple active constraints are handled in an analogous manner (see Figure 15.5). In an n-dimensional optimization problem with $p \geq 2$ active constraints, the CMP is located on an active constraint "surface" of dimension $n-p$. An efficient $(n-p)$-dimensional lattice with packing radius $\rho_{n-p} = \rho_n/2^p$ is used to quantize this active constraint surface, and a poll set is constructed by creating a positive basis selected from the points neighboring the CMP within the $(n-p)$-dimensional active constraint surface, together with p additional points located on the $(n-p+1)$-dimensional constraint surfaces neighboring the CMP. Ideally, these p additional points would be located on vectors normal to the $(n-p)$-dimensional active constraint surface a distance $\rho_{n-p+1} = \rho_n/2^{p-1}$ from the CMP; we thus choose the lattice points on the $(n-p+1)$-dimensional feasible domain boundaries closest to these locations.

In practice, it is found that, once an optimization routine moves onto $p \geq 1$ feasible domain boundaries, it only somewhat infrequently moves back off. To account for this, the p additional poll points mentioned in the previous paragraph are polled *after* the other poll points forming the positive basis within the $(n-p)$-dimensional active constraint surface.

We now have all of the ingredients necessary to coördinate an SP algorithm, as laid out in §15.2.1, with any of the lattices listed in Table 2, while both reusing previous function evaluations and respecting sharp bounds on the feasible region of parameter space. Numerical testing of such an algorithm is reported in §VII-A.

The purpose of the search step of an SMF algorithm is to interpolate, and extrapolate, the trends exhibited by the existing function evaluations in order to suggest new regions of parameter space, perhaps far from the CMP, where the function value is anticipated, with some reasonable degree of probability, to be lower than that of the CMP. There are a variety of possibile ways of accomplishing this; we leverage here the kriging interpolation strategy (Krige 1951; Matheron 1963; Jones 2001; Rasmussen & Williams 2006).

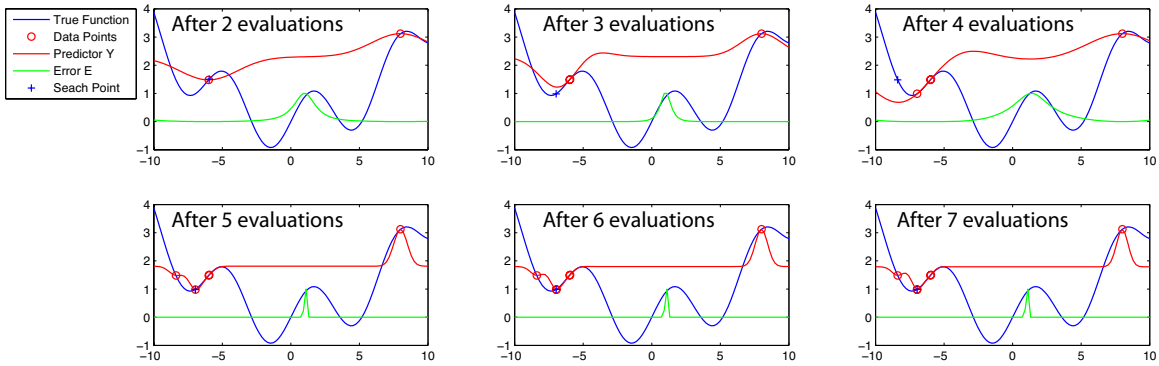


Figure 15.6: Convergence of a search algorithm based on minimizing the kriging predictor, $J(\mathbf{x}) = \hat{f}(\mathbf{x})$, at each iteration. This algorithm does not necessarily converge even to a local minimum, and after six steps the search has effectively stalled.

15.2.3 Global optimization strategies leveraging kriging-based interpolation

The previous section reviewed the kriging interpolation strategy which, based on a sparse set of observed function values $f^o(\mathbf{x}^i)$ for $i = 1,\ldots,N$, develops a function predictor $\hat{f}(\mathbf{x})$ and a model of the uncertainty $s^2(\mathbf{x})$ associated with this prediction for any given set of parameter values $\mathbf{x}$. Leveraging this kriging model, a sophisticated search algorithm can now be developed for the derivative-free optimization algorithm summarized in §15.2.1.

The effectiveness of the various kriging-based search strategies which one might propose may be tested by applying them repeatedly to simple test problems via the following procedure:

- a search function $J(\mathbf{x})$ is first developed based on a kriging model fit to the existing function evaluations,
- a gradient-based search is used to minimize this (computationally inexpensive, smoothly-varying) search function,
- the function $f(\mathbf{x})$ is sampled at the point $\tilde{\mathbf{x}}$ which minimizes the search function[9],
- the kriging model is updated, and the search is repeated.

In the present work, we consider a scalar test problem with multiple minima, $f(x) = \sin(x) + x^2$, on the interval $[-5,5]$, and use two starting points to initialize the search. Ineffective search strategies will not converge to the global minimum of $f(x)$ in this test, and may not converge even to a local minimum. More effective search strategies converge to the global minimum following this approach, and the number of function evaluations required for convergence indicates of the effectiveness of the search strategy used.

Perhaps the most obvious strategy commonly used in such scenarios consists of fitting a kriging model to the known data, then searching the kriging predictor itself, $J(\mathbf{x}) = \hat{f}(\mathbf{x})$, for its minimum value. This simple approach has been implemented in a variety of examples with reasonably good results (see Booker *et al*, 1999). However, as shown clearly in Figure 15.6, this approach can easily break down. The kriging predictor does not necessarily model the function accurately, and its minimization fails to guarantee convergence to even a local minimum of the function $f(\mathbf{x})$. This observed fact can be motivated informally by identifying the kriging predictor as an *interpolating* function which only under extraördinary conditions predicts a function value significantly lower than all of the previously-computed function values; under ordinary conditions, a strategy of minimizing the predictor will thus often stall in the vicinity of the previously-evaluated points.

To avoid the shortcomings of a search defined solely by the minimization of the predictor, another strategy explored by Booker *et al* (1999) is to evaluate the function at *two* points in parameter space during the search:

[9] For the moment, to focus our attention on the behavior of the search algorithm itself, no underlying grid is used to coördinate the search in order to keep function evaluations far apart.

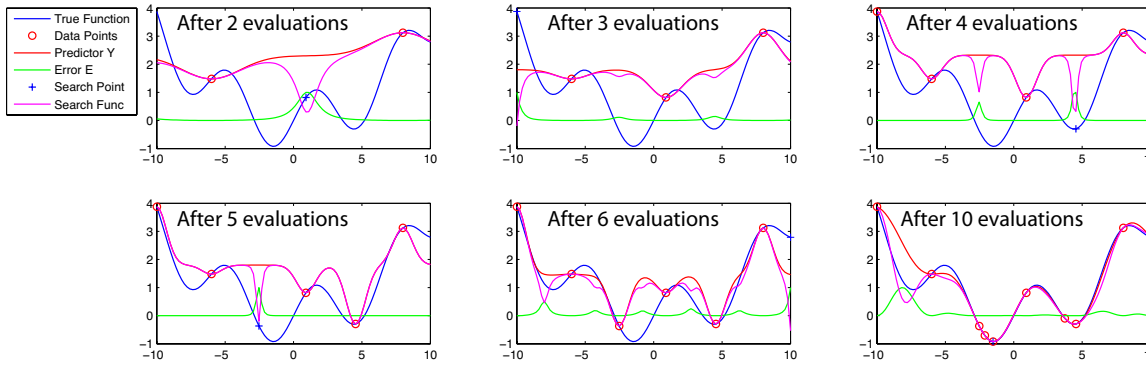


Figure 15.7: Convergence of a search algorithm based on minimizing the search function $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ at each iteration, taking $c = ???$. Note that the global minimum found after just a few iterations.

one point chosen to minimize the predictor, and the other point chosen to maximize the predictor uncertainty. Such a heuristic provides a guarantee of global convergence, as the seach becomes dense in the parameter space as the total number of function evaluations, N, approaches infinity (see §15.2.1). However, this approach generally does not converge quickly as compared with the improved methods described below, as the extra search point has no component associated with the predictor, and is thus often evaluated in relatively "poor" regions of parameter space.

We are thus motivated to develop a flexible strategy to explore in the vicinity of the minima of the predictor. To achive this, consider the minimization of $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$, where c is some constant (Cox & John, 1997; Jones 2001). A search coördinated by this function will tend to explore regions of parameter space where both the predictor of the function value is relatively low *and* the uncertainty of this prediction in the kriging model is relatively high. With this strategy, the search is driven to regions of higher uncertainty, with the $-c \cdot s^2(\mathbf{x})$ term in $J(\mathbf{x})$ tending to cause the algorithm to explore *away* from previously evaluated points. Additionally, minimizing $\hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ allows the algorithm to explore the vicinity of *multiple* local minima in successive iterations in order to determine, with an increasing degree of certainty, which local "bowl" in fact has the deepest minimum. The parameter c provides a natural means to tune the degree to which the search is driven to regions of higher uncertainty, with smaller values of c focusing the search more on refining the vicinity of the lowest function value(s) already found, and larger values of c focusing the search more on exploring regions of parameter space which are still relatively poorly sampled. This parameter may tuned based on knowledge of the function being minimized: if the function is suspected of having multiple minima, c can be made relatively large to ensure a more exploratory search, whereas if the function is suspected of having a single minimum, c can be made relatively small to ensure a more focused search in the vicinity of the CMP. For an appropriate intermediate value of c, the resulting algorithm is often quite effective at both global exploration and local refinement of the minimum, as illustrated in Figure 15.7. The strategy of searching $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ also extends naturally to multiple dimensions, as illustrated for a two-dimensional problem in Figure 15.8. Note also that, in the spirit of Booker *et al* (1999) [who effectively suggested, in the present notation, exploring based on both $c = 0$ and $c \to \infty$ at each search step], one can, for pathological functions, perform a search using multiple but finite values of c at each search step, returning a set of points designed to focus, to varying degrees, on the competing objectives of global exploration and local refinement.

Minimizing $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ is not the only strategy to take advantage of the estimate of the uncertainty of the predictor provided by the kriging model. Another effective search strategy involves maximizing the probability of achieving a target level of improvement on the current CMP [Kushner 1964, Stuckman 1988, Perttunen 1991, Elder 1992, Mockus 1994]. If the current CMP has a function value $f_{\min}$, then this

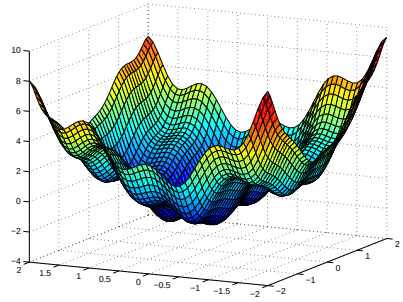


Figure 15.8: The $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ search function for a search in two dimensions.

search strategy seeks that $\mathbf{x}$ for which the probability of finding a function value $f(\mathbf{x})$ less than some prespecified target value f_{target} [that is, for which $f(\mathbf{x}) \leq f_{\text{target}} < f_{\text{min}}$] is maximized in the kriging model. If $f(\mathbf{x})$ is known to be a positive function, a typical target value in this approach is $f_{\text{target}} = (1-\delta) f_{\text{min}}$, where δ might be selected somewhere in the range of 0.01 to 0.2. As for the parameter c discussed in the previous paragraph, the parameter δ in this strategy tunes the degree to which the search is driven to regions of higher uncertainty, with smaller values of δ focusing the search more on refining the vicinity of the lowest function value(s) already found, and larger values of δ focusing the search more on exploring regions of parameter space which are still relatively poorly sampled. For commensurate values of c and δ, the strategies of minimizing $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ and maximizing the probability of achieving a target level of improvement are in fact equivalent (Jones 2001); for simplicity, we thus focus on the former strategy in the present work[10].

Since the search function $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ is inexpensive to compute, continuous, and smooth, but in general has multiple minima, an efficient gradient-based search, initialized from several well-selected points in parameter space, may be used to to minimize it. As the uncertainty $s^2(\mathbf{x})$ goes to zero at each sample point, $J(\mathbf{x})$ will tend to dip between each sample point. Thus, a search is initialized on $2n \cdot N$ total points forming a positive basis near (say, a distance of $\rho_n/2$) to each of the N sample points, and each of these starting points is marched to a local minimum of the function $J(\mathbf{x})$ using an efficient gradient-based search (which is constrained to remain within the feasible domain of $\mathbf{x}$). The lowest point of the trajectories so generated will very likely be the global minimum of $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$. For simplicity, the necessary gradients for this search may be computed via a simple second-order central finite difference scheme applied to the kriging model.

15.A LABDOGS

To recap: the LABDOGS (Lattice Based Derivative-free Optimization via Global Surrogates) algorithm consists of an SMF-based optimization (see §15.2.1) coördinated by efficient n-dimensional lattices (see §**??** and§15.2.2) while leveraging a kriging interpolant (see §7.5.3) to perform a highly efficient global search based on the search function $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ (see §15.2.3). The full algorithm has been implemented in an efficient numerical code, is available for free download at the website for this text.

[10] Even more sophisticated search strategies can be proposed, as elegantly reviewed by Jones 2001. However, the simplicity, flexibility, and performance given by the strategy of minimizing $J(\mathbf{x}) = \hat{f}(\mathbf{x}) - c \cdot s^2(\mathbf{x})$ make this approach appear to be quite adequate for our present purposes.

15.B αDOGS

Exercises

References

Conway, JH, & Sloane, NJA (1998) *Sphere Packings, Lattices, and Groups*, Springer.

Fletcher, R (1987) *Practical Methods of Optimization*. Wiley.

Jones, DR (2001) A Taxonomy of Global Optimization Methods Based on Response Surfaces. *Journal of Global Optimization* **21**, 345-383.

Luenberger, DG (1984) *Linear and Nonlinear Programming*. Addison-Wesley.

Nocedal, J, & Wright, SJ (2006) *Numerical Optimization*. Springer.

Press, WH, Flannery, BP, Teukolsky, SA, & Vetterlig, WT (1986-2007) *Numerical Recipes, The Art of Scientific Computing*. Cambridge.

Related references

Audet, C, & Dennis, Jr, JE (2003) Analysis of generalized pattern searches. *SIAM Journal on Optimization* **13**, 889903.

Audet, C, Dennis, Jr, JE, & Moore, DW (2000) A surrogate-model-based method for constrained optimization. *AIAA Paper 00-4891*.

Booker, A, Dennis, JR, Frank, P, Serafini, D, Torczon, V, & Trosset, M (1999) A rigorous framework for optimization of expensive functions by surrogates. *Structural and Multidisciplinary Optimization* **17**, 113.

Conway, JH, & Sloane, NJA (1984) On the Voronoi regions of certain lattices. *SIAM J. Alg. Disc. Meth.,* **5**, 294-302.

Coope, ID, & Price, CJ (2001) On the convergence of grid-based methods for unconstrained optimization. *SIAM J. Optim.,* **11**, 859869.

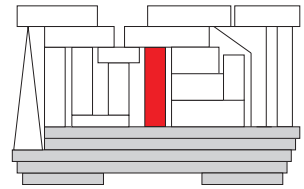
Cox, DD & John, S (1997) SDO: A statistical method for global optimization. In *Multidisciplinary Design Optimization: State of the Art* (edited by Alexandrov, N, & Hussaini, MY), 315329. SIAM.

Curtis, RT (1976) A new combinatorial approach to M24, *Math. Proc. Camb. Phil. Soc.*, **79**, 25-42.

Tammes, PML (1930) On the origin of number and arrangement of the places of exit on the surface of pollen-grains. *Recueil des travaux botaniques néerlandais*, **27**, 1-84.

Torczon, V (1997) On the convergence of pattern search algorithms. *SIAM J. Optim.*, **7**, 1-25.

Torn, A, & Zilinskas, A (1987) *Global Optimization*, Springer.



Chapter 16

Derivative-based minimization

Contents

16.1 The Newton-Raphson method revisited . . . 454
16.2 Gradient-based approaches . . . 454
16.2.1 Steepest descent for quadratic functions . . . 455
16.2.2 Conjugate gradient for quadratic functions . . . 456
16.2.3 Verification of the conjugate gradient algorithm† . . . 460
16.2.4 Preconditioned conjugate gradient . . . 462
16.2.5 Extension to non-quadratic functions . . . 465
16.2.6 Extension to non-smooth problems: bundle methods . . . 466
16.3 Quasi-Newton methods . . . 467
16.3.1 The BFGS method . . . 467
16.3.2 The limited-memory BFGS method . . . 467
Exercises . . . 467

As introduced in §15, the problem of minimization has a broad range of applications.

In this chapter, we explore three classes of methods for solving the minimization problem which build on local computation of the derivatives of the scalar function $J(\mathbf{x})$, where $\mathbf{x} = \mathbf{x}_{n\times 1}$. The Newton-Raphson method discussed in §16.1 builds on calculations of both the **graident vector** $\mathbf{g}(x)$ (of first derivatives of J with respect to the components of $\mathbf{x}$) as well as the **Hessian matrix** $H(x)$ (of second derivatives of J with respect to the components of $\mathbf{x}$). In contrast, the gradient-based methods for multivariable minimization discussed in §16.2 build upon calculations of just the graident vector $\mathbf{g}(x)$ and the function $J(\mathbf{x})$ itself. The quasi-Newton methods for multivariable minimization developed in §16.3 also build on calculations of just the graident vector $\mathbf{g}(x)$ and the function $J(\mathbf{x})$, but uses these calculations to approximate the Hessian matrix as the iterations proceed. Note that $J(\mathbf{x})$ is a scalar, $\mathbf{g}(\mathbf{x})$ is a vector of order n, and $H(x)$ is a matrix of order $n \times n$. For large n, the latter is prohibitively expensive to compute and store, and thus gradient-based methods and quasi-Newton methods are preferred. Thus, §16.3 concludes with a reduced storage variant of the quasi-Newton method introduced earlier in the section which only requires the storage of a pre-determined number of vectors of order n, rather than an entire Hessian matrix of order $n \times n$.

16.1 The Newton-Raphson method revisited

If the function $J(\mathbf{x})$ is locally quadratic and a good initial guess of the desired minimum point is available, then minimization of $J(\mathbf{x})$, for both scalar and vector $\mathbf{x}$, can be accomplished quite effectively simply by applying the iterative Newton-Raphson method developed in §3.1.1 to the gradient of $J(\mathbf{x})$. That is, given the gradient $\mathbf{g}(\mathbf{x}) = \nabla J(\mathbf{x})$, the Newton-Raphson technique developed previously may be applied directly to find a solution of $\mathbf{g}(\mathbf{x}) = 0$. Note that this approach works just as well for scalar or vector $\mathbf{x}$, but just as readily converges to a maximum point (or to a saddle point) as it does to a minimum point. As in the root finding case discussed in §3.1.1, convergence to the desired minimum point is obtained quadratically, provided a sufficiently accurate initial guess of this minimum point is available. Recall from §3.1.1 that the Newton-Raphson method requires both the evaluation of $\mathbf{g}(\mathbf{x})$, in this case taken to be the gradient of the cost $J(\mathbf{x})$, and the evaluation of the Jacobian of $\mathbf{g}(\mathbf{x})$, given in this case by

$$h_{ij}^k = \frac{\partial g_i}{\partial x_j}\Big|_{\mathbf{x}=\mathbf{x}^k} = \frac{\partial^2 J}{\partial x_i \partial x_j}\Big|_{\mathbf{x}=\mathbf{x}^k},$$

and referred to as the **Hessian** $H(\mathbf{x})$ of the function $J(\mathbf{x})$. Unfortunately, computation and storage of the Hessian matrix, which has n^2 elements, is prohibitively expensive for large n.

16.2 Gradient-based approaches

We now develop a reliable technique to minimize a multivariable convex quadratic function $J(\mathbf{x})$ with resepect to $\mathbf{x} = \mathbf{x}_{n\times 1}$. The method we will develop

- does not require a particularly good initial guess of the minimum,
- is efficient for $n \gg 1$, as it does not require computation and storage of the Hessian matrix, and
- is easily generalized to $J(\mathbf{x})$ which are nonquadratic and even nonconvex.

Perhaps the most straightforward strategy to minimize $J(\mathbf{x})$ is to update the vector $\mathbf{x}$ iteratively, proceeding at each step in a downhill direction $\mathbf{p}$ a distance which minimizes $J(\mathbf{x})$ in this direction. In the simplest such algorithm, referred to as the **steepest descent** algorithm, the direction $\mathbf{p}$ is taken as exactly opposite the grdient direction $\mathbf{g} = \nabla J(\mathbf{x})$, which is the direction of maximum *increase* of the function $J(\mathbf{x})$. As the iteration $k \to \infty$, this approach usually converges to one of the minima of $J(\mathbf{x})$, provided that $J(\mathbf{x})$ is at least locally convex and is sufficiently smooth. Note that, if $J(\mathbf{x})$ has multiple minima, this technique will only find one of the minimum points, and the one it finds (a **local minimum**) might not necessarily be the one with the smallest value of $J(\mathbf{x})$ (the **global minimum**).

Though the above approach is simple, it is usually quite slow. As we will show, it is not always the best idea to proceed in the direction of steepest descent of the cost function. A descent direction $\mathbf{p}^k$ chosen to be a linear combination of the direction of steepest descent $\mathbf{r}^k$ and the step taken at the previous iteration $\mathbf{p}^{k-1}$ is often much more effective. The "momentum" carried by such an approach allows the iteration to turn more directly down narrow valleys without oscillating between one descent direction and another, a phenomenon often encountered when momentum is lacking. A particular choice of the momentum term results in a remarkable orthogonality property amongst the set of various descent directions (namely, ${\mathbf{p}^k}^T A\,\mathbf{p}^j = 0$ for $j \neq k$) and the set of descent directions are referred to as a conjugate set. Searching in a series of mutually conjugate directions leads to *exact* convergence of the iterative algorithm in n iterations, assuming a quadratic cost function and no numerical round-off errors[1].

In the following, we will discuss the steepest descent (§5.3.1) and conjugate gradient (§5.3.2, 5.3.3) approaches for quadratic functions first, then discuss their extension to nonquadratic functions (§5.3.4). Though

[1] Note that, for large n, the accumulating round-off error due to the finite-precision arithmetic of the calculations is significant, so exact convergence in n iterations usually can not be obtained.

the quadratic and nonquadratic cases are handled essentially identically in most regards, the line minimizations required by the algorithms may be done directly for quadratic functions, but should be accomplished by a more reliable bracketing procedure (e.g., Brent's method) for nonquadratic functions. Exact convergence in n iterations (again, neglecting numerical round-off errors) is possible only in the quadratic case, though nonquadratic functions may also be minimized quite effectively with the conjugate gradient algorithm when the appropriate modifications are made. A complete proof of the convergence of the conjugate gradient method for quadratic functions, which is somewhat involved, is given in §5.3.5.

16.2.1 Steepest descent for quadratic functions

Consider a quadratic function $J(\mathbf{x})$ of the form

$$J(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x}$$

where A is positive definite. The geometry of this problem is illustrated in Figure 16.1.

We will begin at some initial guess $\mathbf{x}^0$ and move at each step of the iteration k in a direction downhill $\mathbf{r}^k$ such that

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_{k+1}\mathbf{r}^k,$$

where α_{k+1} is a parameter for the descent which will be determined. In this manner, we proceed iteratively towards the minimum of $J(\mathbf{x})$. Noting the derivation of ∇J in §5.0.1, define $\mathbf{r}^k$ as the direction of steepest descent such that

$$\mathbf{r}^k = -\nabla J(\mathbf{x}^k) = \mathbf{b} - A\mathbf{x}^k.$$

Now that we have figured out what *direction* we will update $\mathbf{x}^k$, we need to figure out the parameter of descent α_{k+1}, which governs the *distance* we will update $\mathbf{x}^k$ in this direction. This may be done by selecting the scalar α which minimizes $J(\mathbf{x}^k + \alpha\,\mathbf{r}^k)$. Dropping the superscripts $()^k$ for the time being for notational clarity, note first that

$$J(\mathbf{x} + \alpha\,\mathbf{r}) = \frac{1}{2}(\mathbf{x} + \alpha\,\mathbf{r})^T A(\mathbf{x} + \alpha\,\mathbf{r}) - \mathbf{b}^T(\mathbf{x} + \alpha\,\mathbf{r})$$

and thus

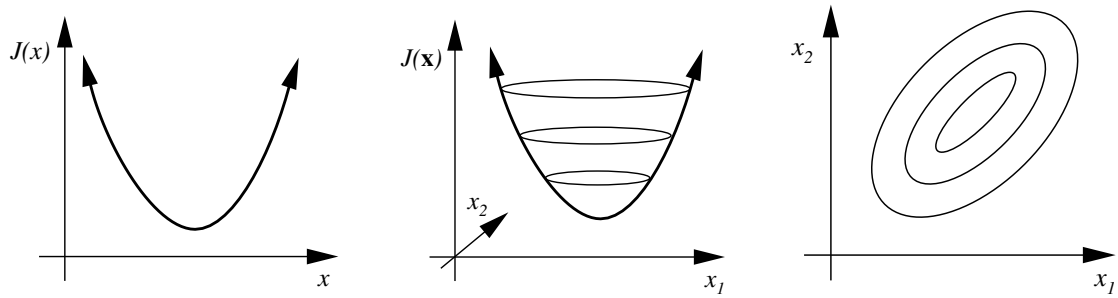
$$\begin{aligned}\frac{\partial J(\mathbf{x} + \alpha\,\mathbf{r})}{\partial \alpha} &= \frac{1}{2}\mathbf{r}^T A(\mathbf{x} + \alpha\,\mathbf{r}) + \frac{1}{2}(\mathbf{x} + \alpha\,\mathbf{r})^T A\mathbf{r} - \mathbf{b}^T\mathbf{r} \\ &= \alpha\,\mathbf{r}^T A\mathbf{r} + \mathbf{r}^T A\mathbf{x} - \mathbf{r}^T\mathbf{b} \\ &= \alpha\,\mathbf{r}^T A\mathbf{r} + \mathbf{r}^T(A\mathbf{x} - \mathbf{b}) \\ &= \alpha\,\mathbf{r}^T A\mathbf{r} - \mathbf{r}^T\mathbf{r}.\end{aligned}$$

Setting $\partial J(\mathbf{x} + \alpha\mathbf{r})/\partial\alpha = 0$ yields

$$\alpha = \frac{\mathbf{r}^T\mathbf{r}}{\mathbf{r}^T A\mathbf{r}}.$$

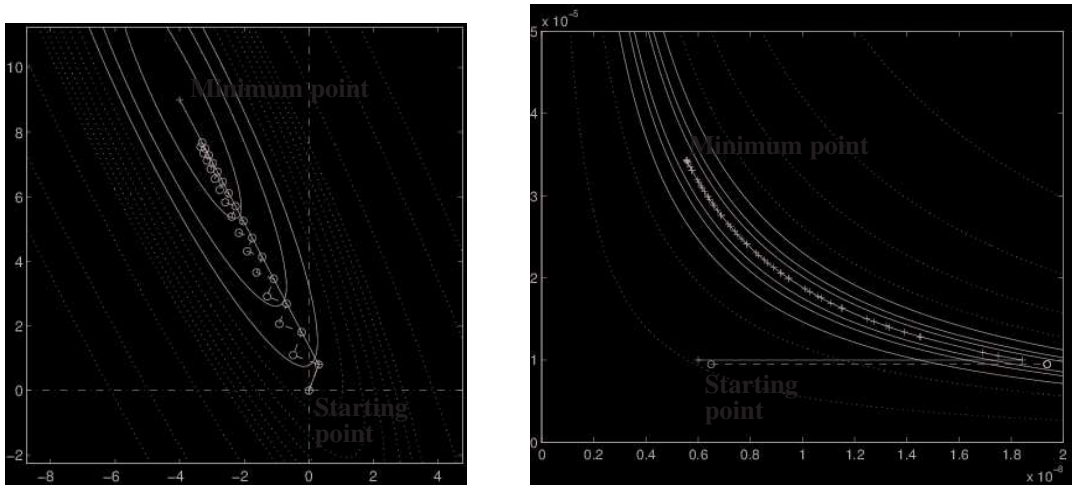
Thus, from the value of $\mathbf{x}$ at each iteration k, we can determine explicitly both the direction of steepest descent $\mathbf{r}$ and the parameter α which minimizes J when $\mathbf{x}$ is updated in the direction $\mathbf{r}$.

The operation count of the steepest descent algorithm described above is dominated by the two matrix vector products, $A\mathbf{x}$ and $A\mathbf{g}$; the leading-order computational cost of the entire algorithm is thus $\sim (4n^2)$ flops. A faster technique for computing such an algorithm is discussed at the end of the next section.



One variable | Two variables (oblique view) | Two variables (top view)

Figure 16.1: Geometry of the minimization problem for quadratic functions. The ellipses in the two figures on the right indicate isosurfaces of constant J.



a) Quadratic $J_3(\phi_1, \phi_2)$. b) Non-quadratic $J_4(\phi_1, \phi_2)$.

Figure 16.2: Convergence of: (○) simple gradient, and (+) the conjugate gradient algorithms when applied to find minima of two test functions of two scalar control variables x_1 and x_2 (horizontal and vertical axes). Contours illustrate the level surfaces of the test functions; contours corresponding to the smallest isovalues are solid, those corresponding to higher isovalues are dotted.

16.2.2 Conjugate gradient for quadratic functions

As discussed earlier, and shown in Figure 16.2, proceeding in the direction of steepest descent at each iteration is not necessarily the most efficient strategy. By so doing, the path of the algorithm can be very jagged. Due to the successive line minimizations and the lack of momentum from one iteration to the next, the steepest descent algorithm must tack back and forth 90° at each turn. We now show that, by slight modification of the steepest descent algorithm, we arrive at the vastly improved conjugate gradient algorithm. This improved algorithm retains the correct amount of momentum from one iteration to the next to successfully negotiate functions $J(\mathbf{x})$ with narrow valleys.

Note that in "easy" cases for which the condition number is approximately unity, the level surfaces of J are approximately circular, and convergence with either the steepest descent or the conjugate gradient algorithm will be quite rapid. In poorly conditioned problems, the level surfaces become highly elongated ellipses, and

Algorithm 16.1: Steepest descent for quadratic functions.

```
function [x,res_save,x_save]=SDquad(A,b)          % Numerical Renaissance Codebase 1.0
% Minimize J=(1/2) x T A x - b T x using the steepest descent method
maxiter=20; minres=1e-10; x=0*b;
for iter=1:maxiter
  g=A*x-b;                                        % determine gradient
  res=g'*g; res_save(iter)=res; x_save(:,iter)=x; % compute residual
  if (res < minres), break; end                   % test for convergence
  alpha=res/(g'*A*g);                             % compute alpha
  x=x-alpha*g;                                    % update x
end
end % function SDquad
```

View

the zig-zag behavior is amplified.

Instead of minimizing in a single search direction at each iteration, as we did for the method of steepest descent, now consider searching *simultaneously* in m directions, which we will denote $\mathbf{p}^0, \mathbf{p}^1, \ldots \mathbf{p}^{m-1)}$. Take

$$\mathbf{x}^m = \mathbf{x}^0 + \sum_{j=1}^{m} \alpha_j \mathbf{p}^{j-1},$$

and note that

$$J(\mathbf{x}^m) = \frac{1}{2}\left(\mathbf{x}^0 + \sum_{j=1}^{m} \alpha_j \mathbf{p}^{j-1}\right)^T A \left(\mathbf{x}^0 + \sum_{j=1}^{m} \alpha_j \mathbf{p}^{j-1}\right) - \mathbf{b}^T \left(\mathbf{x}^0 + \sum_{j=1}^{m} \alpha_j \mathbf{p}^{j-1}\right).$$

Taking the derivative of this expression with respect to α_k,

$$\begin{aligned}\frac{\partial J(\mathbf{x}^m)}{\partial \alpha_k} &= \frac{1}{2}\left(\mathbf{p}^{k-1}\right)^T A \left(\mathbf{x}^0 + \sum_{j=1}^{m} \alpha_j \mathbf{p}^{j-1}\right) + \frac{1}{2}\left(\mathbf{x}^0 + \sum_{j=1}^{m} \alpha_j \mathbf{p}^{j-1}\right)^T A \left(\mathbf{p}^{k-1}\right) - \mathbf{b}^T \mathbf{p}^{k-1} \\ &= \alpha_k (\mathbf{p}^{k-1})^T A \mathbf{p}^{k-1} + (\mathbf{p}^{k-1})^T A \mathbf{x}^0 - (\mathbf{p}^{k-1})^T \mathbf{b} + \sum_{\substack{j=1 \\ j\neq k}}^{m} \alpha_j (\mathbf{p}^{k-1})^T A \mathbf{p}^{j-1}.\end{aligned}$$

We seek a technique to select all the $\mathbf{p}^j$ in such a way that they are orthogonal through A, or conjugate, such that

$$(\mathbf{p}^k)^T A \mathbf{p}^j = 0 \qquad \text{for} \quad j \neq k.$$

IF we can find such a sequence of $\mathbf{p}^j$, then we obtain

$$\begin{aligned}\frac{\partial J(\mathbf{x}^m)}{\partial \alpha_k} &= \alpha_k (\mathbf{p}^{k-1})^T A \mathbf{p}^{k-1} + (\mathbf{p}^{k-1})^T \left(A\mathbf{x}^0 - \mathbf{b}\right) \\ &= \alpha_k (\mathbf{p}^{k-1})^T A \mathbf{p}^{k-1} - (\mathbf{p}^{k-1})^T \mathbf{r}^0,\end{aligned}$$

and thus setting $\partial J/\partial \alpha_k = 0$ results in

$$\alpha_k = \frac{(\mathbf{p}^{k-1})^T \mathbf{r}^0}{(\mathbf{p}^{k-1})^T A \mathbf{p}^{k-1}}.$$

The remarkable thing about this result is that it is independent of $\mathbf{p}^j$ for $j \neq k$! Thus, so long as we can find a way to construct a sequence of $\mathbf{p}^k$ which are all conjugate, then each of these minimizations may effectively be done independently.

The conjugate gradient technique is simply an efficient technique to construct a sequence of $\mathbf{p}^k$ which are conjugate. It entails just redefining the descent direction $\mathbf{p}^k$ at each iteration after the first to be a linear combination of the direction of steepest descent, $\mathbf{r}^k$, and the descent direction at the previous iteration, $\mathbf{p}^{k-1}$, such that

$$\mathbf{p}^k = \mathbf{r}^k + \beta_k \mathbf{p}^{k-1} \qquad \text{and} \qquad \mathbf{x}^k = \mathbf{x}^{k-1} + \alpha_k \mathbf{p}^{k-1},$$

where β_k and α_k are given by

$$\beta_k = \frac{(\mathbf{r}^k)^T \mathbf{r}^k}{(\mathbf{r}^{k-1})^T \mathbf{r}^{k-1}}, \qquad \alpha_k = \frac{(\mathbf{r}^k)^T \mathbf{r}^k}{(\mathbf{p}^{k-1})^T A \mathbf{p}^{k-1}}.$$

Verification that this choice of the β_k results in conjugate directions and that this choice of the α_k is equivalent to that mentioned previously (minimizing J in the direction $\mathbf{p}^{k-1}$ from the point $\mathbf{x}^{k-1}$) involves a straightforward proof by induction, which is deferred to §5.3.5.

As seen by comparison of Algorithms 16.1 and **??**, implementation of the conjugate gradient algorithm involves only a slight modification of the steepest descent algorithm, though, as seen in Figure 16.2, its convergence results are vastly superior. To leading order, the operation count is the same (`CGquad.m` requires only $2n$ more flops per iteration), and the storage requirements are only slightly increased (`CGquad.m` defines an extra n'th-order vector $\mathbf{p}$).

View

Algorithm 16.2: Conjugate gradient for quadratic functions.

```
function [x,res_save,x_save]=CGquad(A,b)                % Numerical Renaissance Codebase 1.0
% Minimize J=(1/2) x T A x - b T x using the conjugate gradient method
maxiter=20; minres=1e-10; x=0*b;
for iter=1:maxiter
  g=A*x-b;                                              % determine gradient
  res=g'*g; res_save(iter)=res; x_save(:,iter)=x;       % compute residual
  if (res < minres), break; end                         % test for convergence
  if (iter==1); p=-g;                                   % Set up a steepest descent step
  else;         p=-g+(res/res_old)*p;                   % Set up a conjugate gradient step
  end
  alpha=res/(p'*A*p); x=x+alpha*p; res_old=res;         % compute alpha and update x
end
end % function CGquad
```

View

```
% script CGquadTest                                     % Numerical Renaissance Codebase 1.0
n=2; A=randn(n); A=A'*A; b=randn(n,1);
[x,SDres,SDxs]=SDquad(A,b); SDerr=norm(A*x-b)
[x,CGres,CGxs]=CGquad(A,b); CGerr=norm(A*x-b)
figure(1); semilogy(SDres,'b-'); hold on; semilogy(CGres,'r-'); hold off;
if n==2
  figure(2); plot(SDxs(1,:),SDxs(2,:),'b-x'); hold on; plot(CGxs(1,:),CGxs(2,:),'r-x');
  z=A\b; plot(z(1),z(2),'bo'); axis equal;
  a=axis; ah=(a(2)-a(1))/10; av=(a(4)-a(3))/10; a=[a(1)-ah a(2)+ah a(3)-av a(4)+av];
  axis(a); [X,Y] = meshgrid(a(1):(a(2)-a(1))/100:a(2), a(3):(a(4)-a(3))/100:a(4));
  J=(1/2)*(X.^2*A(1,1)+X.*Y*(A(1,2)+A(2,1))+Y.^2*A(2,2))-(b(1)*X+b(2)*Y);
  contour(X,Y,J,30); hold off;
end
```

View

Algorithm 16.3: A more efficient implementation of the conjugate gradient algorithm for quadratic functions.

```
function [x,res_save]=CGquadFast(A,b)                   % Numerical Renaissance Codebase 1.0
% Minimize J=(1/2) x T A x - b T x using the standard cg method (fast implementation)
minres=1e-20; x=0*b; alpha=0; maxiter=500;
for iter=1:maxiter
  if (iter==1); g=A*x-b; else; g=g+alpha*d; end         % determine gradient
  res=g'*g;  res_save(iter)=res;                        % compute residual
  if (res < minres), break; end                         % test for convergence
  if (iter==1);          p=-g;                          % Set up a steepest descent step
  else;                  p=-g+(res/res_old)*p;          % Set up a conjugate gradient step
  end
  d=A*p;                % <-- perform the (expensive) matrix/vector product
  alpha=res/(p'*d); x=x+alpha*p; res_old=res;           % compute alpha and update x
end; iter
end % function CGquadFast
```

A faster implementation of the conjugate gradient algorithm for quadratic functions may be derived by leveraging the following two equations:

$$\mathbf{x}^k = \mathbf{x}^{k-1} + \alpha_k \mathbf{p}^{k-1}, \quad \mathbf{r}^k = \mathbf{b} - A\mathbf{x}^k \qquad \Rightarrow \qquad \mathbf{r}^k = \mathbf{b} - A(\mathbf{x}^{k-1} + \alpha_k \mathbf{p}^{k-1}) = \mathbf{r}^{k-1} - \alpha_k A\mathbf{p}^{k-1}.$$

The convenient aspect of this update formula for $\mathbf{r}$ is that it depends on the matrix/vector product $A\mathbf{p}^{k-1}$, which needs to be computed anyway during the computation of α. Thus, for quadratic functions, an implementation which costs only $\sim (2n^2)$ flops per iteration is possible, as implemented in Algorithm **??**.

16.2.3 Verification of the conjugate gradient algorithm†

The conjugate gradient algorithm motivated and discussed in §16.2.2 is now verified for the quadratic function

$$J(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} \tag{16.1}$$

for symmetric positive definite A. The proof given in this section, which is self contained, is adapted from that in Luenberger (1984).

We begin with a few of definitions:

(a) The Euclidean space of all n'th-order vectors is denoted E^n.

(b) A **subspace** M of E^n is a subset such that, if $\mathbf{a}$ and $\mathbf{b}$ are vectors in M, then $\lambda\mathbf{a}+\mu\mathbf{b}$ is also in M for all pairs of real numbers λ and μ. Examples in E^3 include lines or planes that contain the origin.

(c) A **convex set** C in E^n is a subset such that, if $\mathbf{a}$ and $\mathbf{b}$ are vectors in C, then $\lambda\mathbf{a}+(1-\lambda)\mathbf{b}$ is also in C for $0<\lambda<1$. A convex set thus contains the line segment between any two points in the set.

(d) A **linear variety** V in E^n is a subset such that, if $\mathbf{a}$ and $\mathbf{b}$ are vectors in V, then $\lambda\mathbf{a}+(1-\lambda)\mathbf{b}$ is also in V for all real numbers λ. A linear variety thus contains the entire line passing through any two points in the set, rather than just the line segment between them. A linear variety that contains the origin is a subspace.

(e) A **hyperplane** in an n-dimensional linear vector space is an $(n-1)$-dimensional linear variety.

We now prove an important intermediate result.

Expanding Subspace Theorem. *Let $\{\mathbf{p}^0,\mathbf{p}^1,\ldots,\mathbf{p}^{n-1}\}$ be a sequence of nonzero vectors in E^n that are orthogonal through A (that is, $(\mathbf{p}^k)^T A\,\mathbf{p}^j = 0$ for $j\neq k$). The subspace of E^n that is spanned by the first k of these vectors is denoted $\mathscr{B}_k$. Then for any $\mathbf{x}^0\in E^n$, with $\mathbf{r}^0=\mathbf{b}-A\mathbf{x}^0$, the sequence of $\mathbf{x}^k$ generated according to*

$$\alpha_k = \frac{(\mathbf{p}^{k-1})^T\mathbf{r}^{k-1}}{(\mathbf{p}^{k-1})^T A\mathbf{p}^{k-1}}, \tag{16.2a}$$

$$\mathbf{x}^k = \mathbf{x}^{k-1}+\alpha_k\mathbf{p}^{k-1}, \tag{16.2b}$$

$$\mathbf{r}^k = -\nabla J(\mathbf{x}^k) = \mathbf{b}-A\mathbf{x}^k, \tag{16.2c}$$

for $k=1,2,\ldots,n-1$ has the property that $\mathbf{x}^k$ minimizes $J(\mathbf{x})$ in (16.1) on the line $\mathbf{x}=\mathbf{x}^{k-1}+\alpha\mathbf{p}^{k-1}$ for all real α, as well as on the linear variety $\mathbf{x}^0+\mathscr{B}_k$.

Proof. It need only be shown that $\mathbf{x}^k$ minimizes $J(\mathbf{x})$ on the linear variety $\mathbf{x}^0+\mathscr{B}_k$, which contains the line $\mathbf{x}=\mathbf{x}^{k-1}+\alpha\mathbf{p}^{k-1}$ for all real α. Since $J(\mathbf{x})$ is a strictly convex function, the conclusion will hold if it can be shown that $\mathbf{r}^k$ is orthogonal to $\mathscr{B}_k$ (that is, the gradient of $J(\mathbf{x})$ evaluated at $\mathbf{x}=\mathbf{x}^k$ is orthogonal to the subspace $\mathscr{B}_k$).

We prove that $\mathbf{r}^k\perp\mathscr{B}_k$ by induction. Since $\mathscr{B}_0$ is empty, the hypothesis is true for $k=0$. Assuming that it is true for $k-1$, that is, assuming $\mathbf{r}^{k-1}\perp\mathscr{B}_{k-1}$, we now show that $\mathbf{r}^k\perp\mathscr{B}_k$. By premultiplying (16.2b) by $(-A)$ and applying (16.2c), it follows that

$$\mathbf{r}^k = \mathbf{r}^{k-1}-\alpha_k A\mathbf{p}^{k-1}. \tag{16.3}$$

† The complete proof given in this subsection is fairly involved, and may be skipped upon first read, in favor of the heuristic motivation for the conjugate gradient algorithm given in §16.2.2. However, it is useful to see the complete derivation of this algorithm in order to remove the "mystery" which might otherwise be associated with some of the motivating comments and formulae given in §16.2.2.

Premultiplying (16.3) by $(\mathbf{p}^{k-1})^T$ gives

$$(\mathbf{p}^{k-1})^T\mathbf{r}^k = (\mathbf{p}^{k-1})^T\mathbf{r}^{k-1} - \alpha_k(\mathbf{p}^{k-1})^T A\mathbf{p}^{k-1} = 0$$

by the definition of α_k in (16.2a). Similarly, premultiplying (16.3) by $(\mathbf{p}^i)^T$, for $i < k-1$, gives

$$(\mathbf{p}^i)^T\mathbf{r}^k = (\mathbf{p}^i)^T\mathbf{r}^{k-1} - \alpha_k(\mathbf{p}^i)^T A\mathbf{p}^{k-1} = 0,$$

where the first term on the rhs vanishes by the induction hypothesis while the second term vanishes by the assumption that the vectors $\mathbf{p}^i$ are orthogonal through A. Thus, $\mathbf{r}^k$ is orthogonal to the vectors $\{\mathbf{p}^0,\mathbf{p}^1,\ldots,\mathbf{p}^{k-1}\}$, and therefore $\mathbf{r}^k \perp \mathscr{B}_k$. □

A form of the *Conjugate Gradient Algorithm* is now proposed: starting at any point $\mathbf{x}^0 \in E^n$, initialize $\mathbf{p}^0 = \mathbf{r}^0 = \mathbf{b} - A\mathbf{x}^0$ and proceed as follows, for $k = 1,2,\ldots,n-1$ or until $||\mathbf{r}^k||$ vanishes:

$$\alpha_k = \frac{(\mathbf{p}^{k-1})^T\mathbf{r}^{k-1}}{(\mathbf{p}^{k-1})^T A\mathbf{p}^{k-1}}, \tag{16.4a}$$

$$\mathbf{x}^k = \mathbf{x}^{k-1} + \alpha_k\mathbf{p}^{k-1}, \tag{16.4b}$$

$$\mathbf{r}^k = \mathbf{b} - A\mathbf{x}^k, \tag{16.4c}$$

$$\beta_k = -\frac{(\mathbf{r}^k)^T A\mathbf{p}^{k-1}}{(\mathbf{p}^{k-1})^T A\mathbf{p}^{k-1}}, \tag{16.4d}$$

$$\mathbf{p}^k = \mathbf{r}^k + \beta_k\mathbf{p}^{k-1}. \tag{16.4e}$$

Denoting, e.g., $[\mathbf{p}^0,\mathbf{p}^1,\ldots,\mathbf{p}^k]$ as the subspace spanned by the vectors in the set $\{\mathbf{p}^0,\mathbf{p}^1,\ldots,\mathbf{p}^k\}$, we now prove the main result:

Conjugate Gradient Theorem. *The conjugate gradient algoritghm* (16.4a)-(16.4e)*, if not terminating at or before* $\mathbf{x}^{k-1}$*, obeys the following:*

$$[\mathbf{r}^0,\mathbf{r}^1,\ldots,\mathbf{r}^k] = [\mathbf{r}^0,A\mathbf{r}^0,\ldots,A^k\mathbf{r}^0], \tag{16.5a}$$

$$[\mathbf{p}^0,\mathbf{p}^1,\ldots,\mathbf{p}^k] = [\mathbf{r}^0,A\mathbf{r}^0,\ldots,A^k\mathbf{r}^0], \tag{16.5b}$$

$$(\mathbf{p}^k)^T A\,\mathbf{p}^j = 0 \quad \text{for} \quad j < k, \tag{16.5c}$$

$$\alpha_k = \frac{(\mathbf{r}^{k-1})^T\mathbf{r}^{k-1}}{(\mathbf{p}^{k-1})^T A\mathbf{p}^{k-1}}, \tag{16.5d}$$

$$\beta_k = \frac{(\mathbf{r}^k)^T\mathbf{r}^k}{(\mathbf{r}^{k-1})^T\mathbf{r}^{k-1}}. \tag{16.5e}$$

Proof. We first prove (16.5a), (16.5b), and (16.5c) simultaneously by induction. Clearly, the hypotheses are true for $k = 0$. Assuming that they are true up to $k-1$, we now show that they are true for k. By premultiplying (16.4b) by $(-A)$ and applying (16.4c), it follows that

$$\mathbf{r}^k = \mathbf{r}^{k-1} - \alpha_k A\mathbf{p}^{k-1}. \tag{16.6}$$

By the induction hypothesis both $\mathbf{r}^{k-1}$ and $A\mathbf{p}^{k-1}$ belong to $[\mathbf{r}^0,A\mathbf{r}^0,\ldots,A^k\mathbf{r}^0]$, the first by (16.5a) and the second by A times (16.5b). Thus $\mathbf{r}^k \in [\mathbf{r}^0,A\mathbf{r}^0,\ldots,A^k\mathbf{r}^0]$. Furthermore $\mathbf{r}^k$, if it is nonzero, is not an element of $[\mathbf{r}^0,A\mathbf{r}^0,\ldots,A^{k-1}\mathbf{r}^0]$ because, by the induction hypothesis on (16.5c) and the Expanding Subspace Theorem, $\mathbf{r}^k \perp \mathscr{B}_k$. Thus, we conclude that

$$[\mathbf{r}^0,\mathbf{r}^1,\ldots,\mathbf{r}^k] = [\mathbf{r}^0,A\mathbf{r}^0,\ldots,A^k\mathbf{r}^0],$$

which proves (16.5a).

Relation (16.5b) follows directly from the induction hypothesis on (16.5b), (16.4e), and (16.5a).

To prove (16.5c), note that postmultiplying the transpose of (16.4e) times $A\mathbf{p}^i$ gives

$$(\mathbf{p}^k)^T A\mathbf{p}^i = (\mathbf{r}^k)^T A\mathbf{p}^i + \beta_k (\mathbf{p}^{k-1})^T A\mathbf{p}^i.$$

For $i = k-1$ the right side is zero by the definition of β_k in (16.4d). For $i < k-1$ both terms vanish. The first term vanishes by the induction hypothesis on (16.5b) which implies $A\mathbf{p}^i \in [\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^{i+1}]$, the induction hypothesis on (16.5c) which guarantees that the directions $\{\mathbf{p}^0, \mathbf{p}^1, \ldots \mathbf{p}^{k-1}\}$ are mutually orthogonal through A, and the Expanding Subspace Theorem which guarantees that $\mathbf{r}^k$ is orthogonal to $[\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^{k-1}]$. The second term vanishes by the induction hypothesis on (16.5c). This proves (16.5c).

To prove (16.5d), note (16.4a) and that postmultiplying the transpose of (16.4e) by $\mathbf{r}^k$ gives

$$(\mathbf{p}^k)^T \mathbf{r}^k = (\mathbf{r}^k)^T \mathbf{r}^k + \beta_k (\mathbf{p}^{k-1})^T \mathbf{r}^k,$$

where the second term on the RHS is zero by the Expanding Subspace Theorem.

To prove (16.5e), note that $(\mathbf{r}^k)^T \mathbf{r}^{k-1} = 0$ because $\mathbf{r}^{k-1} \in [\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^{k-1}]$ by (16.5a) and (16.5b) and $\mathbf{r}^k$ is orthogonal to $[\mathbf{p}^0, \mathbf{p}^1, \ldots, \mathbf{p}^{k-1}]$ by the Expanding Subspace Theorem. By (16.6) we have

$$-A\mathbf{p}^{k-1} = \frac{1}{\alpha_k}(\mathbf{r}^k - \mathbf{r}^{k-1});$$

premultiplication by $(\mathbf{r}^k)^T$ gives

$$-(\mathbf{r}^k)^T A\mathbf{p}^{k-1} = \frac{1}{\alpha_k}(\mathbf{r}^k)^T \mathbf{r}^k. \tag{16.7}$$

Substituting (16.7) into the numerator of (16.4d) and substituting (16.5d) into the denominator of (16.4d) completes the proof. □

Equation (16.5c) of this theorem establishes that the set directions $\mathbf{p}^k$ constructed iteratively by the Conjugate Gradient Algorithm are in fact mutually orthogonal through A, which allows the Expanding Subspace Theorem to be applied. Equations (16.5d) and (16.5e) are identities yielding convenient formulae for the computation of α_k and β_k.

Note by (16.4e) and the Expanding Subspace Theorem that $||\mathbf{p}^k|| \neq 0$ as long as $||\mathbf{r}^k|| \neq 0$. Thus, since A is assumed to be positive definite, it is readily confirmed from (16.4a) and (16.4d) that the values of α_k and β_k are finite until the algorithm terminates.

16.2.4 Preconditioned conjugate gradient

Assuming exact arithmetic, the conjugate gradient algorithm converges in exactly n iterations for an n'th-order quadratic minimization problem. For large n, however, we often can not afford to perform n iterations. We often seek to perform approximate minimization of an n-diminsional problem with a total number of iterations $m \ll n$. Unfortunately, convergence of the conjugate gradient algorithm to the minimum of J, though monotonic, is often irregular, with large reductions in J not occuring until iterations well after the iteration m at which we would like to truncate the iteration sequence.

The uniformity of the convergence is governed by the condition number c of the matrix A, which (for symmetric positive-definite A) is just equal to the ratio of its maximum and minimum eigenvalues, $\lambda_{\max}/\lambda_{\min}$. For small c, convergence of the conjugate gradient algorithm is quite rapid even for high-order problems with $n \gg 1$.

We therefore seek to solve a better conditioned but equivalent problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ which, once solved, will allow us to easily extract the solution of the original problem $A\mathbf{x} = \mathbf{b}$ for a poorly-conditioned symmetric

positive definite A. To accomplish this, premultiply $A\mathbf{x} = \mathbf{b}$ by P^{-1} for some symmetric preconditioning matrix P:

$$P^{-1}A\mathbf{x} = P^{-1}\mathbf{b} \qquad \Rightarrow \qquad \underbrace{(P^{-1}AP^{-1})}_{\tilde{A}}\underbrace{P\mathbf{x}}_{\tilde{\mathbf{x}}} = \underbrace{P^{-1}\mathbf{b}}_{\tilde{\mathbf{b}}}$$

Note that the matrix $P^{-1}AP^{-1}$ is symmetric positive definite. We will defer discussion of the construction of an appropriate P to the end of the section; suffice it to say for the moment that, if P^2 is somehow "close" to A, then the problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ is a well conditioned problem (because $\tilde{A} = P^{-1}AP^{-1} \approx I$) and can be solved rapidly (with a small number of iterations) using the conjugate gradient approach.

The computation of $\tilde{A}$ might be prohibitively expensive and destroy any sparsity structure of A. We now show that it is not actually necessary to compute $\tilde{A}$ and $\tilde{\mathbf{b}}$ in order to solve the original problem $A\mathbf{x} = \mathbf{b}$ in a well conditioned manner. To begin, we write the conjugate gradient algorithm for the well conditioned problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$. For simplicity, we use a short-hand ("pseudo-code") notation:

for $i = 1 : m$

$$\tilde{\mathbf{r}} \leftarrow \begin{cases} \tilde{\mathbf{b}} - \tilde{A}\tilde{\mathbf{x}}, & i = 1 \\ \tilde{\mathbf{r}} - \alpha\tilde{\mathbf{d}}, & i > 1 \end{cases}$$

$$res_{\text{old}} = res, \qquad res = \tilde{\mathbf{r}}^T\tilde{\mathbf{r}}$$

$$\tilde{\mathbf{p}} \leftarrow \begin{cases} \tilde{\mathbf{r}}, & i = 1 \\ \tilde{\mathbf{r}} + \beta\tilde{\mathbf{p}} \quad \text{where} \quad \beta = res/res_{\text{old}}, & i > 1 \end{cases}$$

$$\alpha = res/(\tilde{\mathbf{p}}^T\tilde{\mathbf{d}}) \quad \text{where} \quad \tilde{\mathbf{d}} = \tilde{A}\tilde{\mathbf{p}}$$

$$\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}} + \alpha\tilde{\mathbf{p}}$$

end

For clarity of notation, we have introduced a tilde over each vector and matrix involved in this optimization. Note that, in converting the poorly-conditioned problem $A\mathbf{x} = \mathbf{b}$ to the well-conditioned problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$, we made the following definitions: $\tilde{A} = P^{-1}AP^{-1}$, $\tilde{\mathbf{x}} = P\mathbf{x}$, and $\tilde{\mathbf{b}} = P^{-1}\mathbf{b}$. Define now some new intermediate variables $\mathbf{r} = P\tilde{\mathbf{r}}$, $\mathbf{p} = P^{-1}\tilde{\mathbf{p}}$, and $\mathbf{d} = P\tilde{\mathbf{d}}$. With these definitions, we now rewrite *exactly* the above algorithm for solving the well-conditioned problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$, but substitute in the non-tilde variables:

for $i = 1 : m$

$$P^{-1}\mathbf{r} \leftarrow \begin{cases} P^{-1}\mathbf{b} - (P^{-1}AP^{-1})P\mathbf{x}, & i = 1 \\ P^{-1}\mathbf{r} - \alpha P^{-1}\mathbf{d}, & i > 1 \end{cases}$$

$$res_{\text{old}} = res, \qquad res = (P^{-1}\mathbf{r})^T(P^{-1}\mathbf{r})$$

$$P\mathbf{p} \leftarrow \begin{cases} P^{-1}\mathbf{r}, & i = 1 \\ P^{-1}\mathbf{r} + \beta P\mathbf{p} \quad \text{where} \quad \beta = res/res_{\text{old}}, & i > 1 \end{cases}$$

$$\alpha = res/[(P\mathbf{p})^T(P^{-1}\mathbf{d})] \quad \text{where} \quad P^{-1}\mathbf{d} = (P^{-1}AP^{-1})P\mathbf{p}$$

$$P\mathbf{x} \leftarrow P\mathbf{x} + \alpha P\mathbf{p}$$

end

Now define $M = P^2$ and simplify:

$$
\begin{array}{l}
\text{for } i = 1 : m \\
\quad \mathbf{r} \leftarrow \begin{cases} \mathbf{b} - A\mathbf{x}, & i = 1 \\ \mathbf{r} - \alpha \mathbf{d}, & i > 1 \end{cases} \\
\quad res_{\text{old}} = res, \qquad res = \mathbf{r}^T \mathbf{s} \quad \text{where} \quad \mathbf{s} = M^{-1}\mathbf{r} \\
\quad \mathbf{p} \leftarrow \begin{cases} \mathbf{s}, & i = 1 \\ \mathbf{s} + \beta \mathbf{p} \quad \text{where} \quad \beta = res/res_{\text{old}}, & i > 1 \end{cases} \\
\quad \alpha = res/[\mathbf{p}^T \mathbf{d}] \quad \text{where} \quad \mathbf{d} = A\mathbf{p} \\
\quad \mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{p} \\
\text{end}
\end{array}
$$

This is practically identical to the original conjugate gradient algorithm for solving the problem $A\mathbf{x} = \mathbf{b}$. The new variable $\mathbf{s} = M^{-1}\mathbf{r}$ may be found by solution of the system $M\mathbf{s} = \mathbf{r}$ for the unknown vector $\mathbf{s}$. Thus, when implementing this method, we seek an M for which we can solve this system quickly (e.g., an M which is the product of sparse triangular matrices). Recall that if $M = P^2$ is somehow "close" to A, the problem here (which is actually the solution of $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$ via standard conjugate gradient) is well conditioned and converges in a small number of iterations. There are a variety of heuristic techniques to construct an appropriate M. One of the most popular is **incomplete Cholesky factorization**, which constructs a triangular H with $HH^T = M \approx A$ with the following strategy:

$$
\begin{array}{l}
H = A \\
\text{for } k = 1 : n \\
\quad H(k,k) = \sqrt{H(k,k)} \\
\quad \text{for } i = k+1 : n \\
\quad\quad \text{if } H(i,k) \neq 0 \text{ then } H(i,k) = H(i,k)/H(k,k) \\
\quad \text{end} \\
\quad \text{for } j = k+1 : n \\
\quad\quad \text{for } i = j : n \\
\quad\quad\quad \text{if } H(i,j) \neq 0 \text{ then } H(i,j) = H(i,j) - H(i,k)H(j,k) \\
\quad\quad \text{end} \\
\quad \text{end} \\
\text{end}
\end{array}
$$

Once H is obtained with this appraoch such that $HH^T = M$, solving the system $M\mathbf{s} = \mathbf{r}$ for $\mathbf{s}$ is similar to solving Gaussian elimination by leveraging an LU decomposition: one first solves the triangular system $H\mathbf{f} = \mathbf{r}$ for the intermediate variable $\mathbf{f}$, then solves the triangular system $H^T\mathbf{s} = \mathbf{f}$ for the desired quantity $\mathbf{s}$.

Note that the triangular factors H and H^T are zero everywhere A is zero. Thus, if A is sparse, the above algorithm can be rewritten in a manner that leverages the sparsity structure of H (akin to the backsubstitution in the Thomas algorithm). Though it is sometimes takes a bit of effort to write an algorithm that efficiently leverages such sparsity structure, as it usually must be done on a case-by-case basis, the benefits of preconditioning are often quite significant and well worth the coding effort which it necessitates.

Algorithm 16.4: The preconditioned conjugate gradient algorithm for quadratic functions.

```
function [x,res_save]=CGquadPrecon(A,b)                    % Numerical Renaissance Codebase 1.0   View
% Minimize J=(1/2) x T A x - b T x using the cg method with preconditioning
minres=1e-20; x=0*b; alpha=0; maxiter=500;  N=size(b,1);
for i=1:N; for j=1:N; if (abs(A(i,j))>.1 | i==j); M(i,j)=A(i,j); else; M(i,j)=0; end; end; end;
nm=norm(A-M), P=sqrtm(M); conA=cond(A), conPAP=cond(inv(P)*A*inv(P)),
for iter=1:maxiter
  if (iter==1); g=A*x-b; else; g=g+alpha*d; end  % determine gradient
  s=M\g;
  res=g'*s;                                      % compute residual
  if (mod(iter,100)==1); p=-s;                   % Set up a steepest descent step
  else;                  p=-s+(res/reso)*p;      % Set up a conjugate gradient step
  end
  d=A*p;                 % <---- perform the (expensive) matrix/vector product
  alpha=res/(p'*d);                              % compute alpha
  x=x+alpha*p;                                   % update x
  if (res < minres), break; end                  % test for convergence
  reso=res;
end
end % function CGquadPrecon
```

16.2.5 Extension to non-quadratic functions

At each iteration of the conjugate gradient method, there are five things to be done:

1. Determine the (negative of) the gradient direction, $\mathbf{r} = -\nabla J$,
2. compute the residual $\mathbf{r}^T\mathbf{r}$,
3. determine the necessary momentum β and the corresponding update direction $\mathbf{p} \leftarrow \mathbf{r} + \beta\mathbf{p}$,
4. determine the (scalar) parameter of descent α which minimizes $J(\mathbf{x} + \alpha\mathbf{p})$, and
5. update $\mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{p}$.

We now must extend the codes developed above for quadratic problems to nonquadratic problems, creating the two new routines `CGnonquad.m`. Essentially, the algorithm is the same, but $J(\mathbf{x})$ now lacks the special quadratic structure we assumed in the previous sections. Generalizing the results of the previous sections to nonquadratic problems entails only a few modifications:

1'. Replace the line which determines the gradient direction with a call to a function, which we will call `compute_grad.m`, which calculates the gradient ∇J of the nonquadratic function J.

3'. As the function is not quadratic, but the momentum term in the conjugate gradient algorithm is computed using a local quadratic approximation of J, the momentum sometimes builds up in the wrong direction. Thus, the momentum should be reset to zero (*i.e.*, take $\beta = 0$) every R iterations in `cg_nq.m` ($R = 20$ is often a good choice).

4'. Replace the direct computation of α with a call to an (appropriately modified) version of Brent's method to determine α based on a series of function evaluations.

Finally, when working with nonquadratic functions, it is often adventageous to compute the momentum term β according to the formula

$$\beta = \frac{(\mathbf{r}^k - \mathbf{r}^{k-1})^T \mathbf{r}^k}{(\mathbf{r}^{k-1})^T \mathbf{r}^{k-1}}$$

Algorithm 16.5: The preconditioned conjugate gradient algorithm for quadratic functions.

View

```
function [x,res_save]=CGquadPrecon(A,b)                    % Numerical Renaissance Codebase 1.0
% Minimize J=(1/2) x T A x - b T x using the cg method with preconditioning
minres=1e-20; x=0*b; alpha=0; maxiter=500;  N=size(b,1);
for i=1:N; for j=1:N; if (abs(A(i,j))>.1 | i==j); M(i,j)=A(i,j); else; M(i,j)=0; end; end; end;
nm=norm(A-M), P=sqrtm(M); conA=cond(A), conPAP=cond(inv(P)*A*inv(P)),
for iter=1:maxiter
  if (iter==1); g=A*x-b; else; g=g+alpha*d; end  % determine gradient
  s=M\g;
  res=g'*s;                                      % compute residual
  if (mod(iter,100)==1); p=-s;                   % Set up a steepest descent step
  else;                  p=-s+(res/reso)*p;      % Set up a conjugate gradient step
  end
  d=A*p;                 % <---- perform the (expensive) matrix/vector product
  alpha=res/(p'*d);                              % compute alpha
  x=x+alpha*p;                                   % update x
  if (res < minres), break; end                  % test for convergence
  reso=res;
end
end % function CGquadPrecon
```

The "correction" term $(\mathbf{r}^{k-1})^T\mathbf{r}^k$ is zero using the conjugate gradient approach when the function J is quadratic. When the function J is not quadratic, this additional term often serves to nudge the descent direction towards that of a steepest descent step in regions of the function which are locally not well approximated by a quadratic. This approach is referred to as the **Polak-Ribiere** variant of the conjugate gradient method for nonquadratic functions.

16.2.6 Extension to non-smooth problems: bundle methods

This section still under construction.

16.3 Quasi-Newton methods

16.3.1 The BFGS method

Consider the problem of the minimization of a scalar real function J of a vector (possibly complex) argument $\mathbf{x}$. As introduced earlier in this chapter, minimization of $J(\mathbf{x})$ corresponds to finding a minimum of $\mathbf{g}(\mathbf{x}) = \nabla J(x)$ with the Hessian matrix $H(\mathbf{x}) = \nabla^2 J(x) \geq 0$. To proceed, we will consider a locally quadratic approximation of $J(\mathbf{x})$ near some point $\mathbf{x}_k$, and thus a locally linear approximation of $\mathbf{g}(\mathbf{x})$ near $\mathbf{x}_k$ which may be written

$$\mathbf{f}(\mathbf{x}) = \nabla J(\mathbf{x}_k) + \nabla^2 J(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k) = \mathbf{g}_k + H_k(\mathbf{x} - \mathbf{x}_k) \approx \nabla J(\mathbf{x}) = \mathbf{g}(\mathbf{x}).$$

In this section, we will construct a matrix B_k iteratively, starting from $B_0 = I$, such that $B_k \underset{k\to\infty}{\longrightarrow} H_k$. To accomplish this, at each itertion we will adjust B_k to ensure that it is a consistent approximation of the Hessian for the gradient vector just calculated; that is, if $\mathbf{x}_{k+1}$ and $\mathbf{x}_k$ are related via an update formula

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k = \mathbf{x}_k + \mathbf{s}_k$$

for some α_k (ideally, as in the previous section, a value which minimizes J in the direction $\mathbf{p}_k$ from the point $\mathbf{x}_k$), then we would like that the gradient at the new point, $\mathbf{g}(x_{k+1})$, be zero in

the Hessian H_k with some $B_k > 0$ which will will build up iteratively from the identity matrix via successive combinations of gradient vectors

16.3.2 The limited-memory BFGS method

This section still under construction.

Exercises

References

Part IV
Control

17 Signals & systems: transform-based methods **471**

18 Classical control design **527**

19 Linear Circuits **587**

20 Linear systems: state-space methods **615**

21 State-space & model-predictive control design **669**

22 State estimation & adaptive observation **695**

Our treatment of the related subjects of **linear systems** and **linear control & estimation** is spread over §§17-22. In §17 we consider **transform-based representations** of linear systems in **transfer-function form**; §18 then follows up immediately with a treatment of the feedback control problem for systems represented in such transfer-function forms[2]. Transform-based approaches are well suited to the special case of **single-input single-output** (**SISO**) systems, or at least systems in which the inputs and outputs may be considered one at a time for the purpose of analysis. The tools for characterization and control of transfer-function forms builds, to a degree, on spectral representations and the Fourier transform, as laid out in §5; however, a thorough understanding of that material is not essential to digest most of the concepts in §§17-18. In §19, we illustrate how many of the filters considered in §§17-18 may be implemented in analog electronics.

Our discussion of linear systems, begun in §17, is continued in §20 with a focus on system representations in **state-space form**; §21 then follows up immediately with a treatment of the feedback control problem for systems represented in such state-space forms[3]. State-space approaches are well suited to the more general case of **multiple-input multiple-output** (**MIMO**) systems[4]. The tools for characterization and control of state-space forms builds heavily on a deep understanding of advanced linear algebra, as laid out in §4.

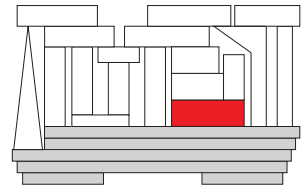
A thorough understanding transform-based approaches and transfer-function forms is helpful to build one's intuition regarding the characterization and control of linear systems. Thus, even if one's ultimate goal is the characterization and feedback control of MIMO systems in state-space form, transform-based representations and transfer-function forms are important foundational concepts that should not be skipped. In other words, §20 is separated from §17, and §21 is separated from §18, for pedagogical reasons only, as the subjects of linear systems and linear control are both somewhat abstract upon first read, and are perhaps easier to understand if broken into smaller chunks. We thus encourage the study of §§17-21 in order.

Finally, §22 is dedicated to a close look at the difficult problem of estimation in both linear and non-linear systems. This chapter brings together many of the interdisciplinary concepts presented elsewhere in *Numerical Renaissance*, and provides an appropriate capstone for the entire text.

[2] For historical reasons, continuous-time control strategies based on the Laplace-transform (§17.3), and discrete-time control strategies based on the Z transform (§17.4), are often called **classical** control techniques.

[3] For historical reasons, control strategies based on state-space representations are sometimes called **modern** control techniques.

[4] Note that, when building from the SISO case to the MIMO case, we make occasional reference to the **single-input multiple-output** (**SIMO**) and **multiple-input single-output** (**MISO**) cases.



Chapter 17

Signals & systems: transform-based methods

Contents

17.1 Introduction to transforms: Fourier, Laplace, and Z . . . 472
17.1.1 The relation of the Laplace and Z transforms to the Fourier transform . . . 473
17.1.2 The remarkable utility of such transforms . . . 473
17.2 A survey of the dynamics of some physical systems . . . 474
17.3 Laplace transform methods . . . 491
17.3.1 The Laplace Transform of derivatives and integrals of functions . . . 493
17.3.2 Using the Laplace Transform to solve unforced linear differential equations . . . 494
17.3.3 Continuous-time (CT) transfer functions . . . 494
17.3.3.1 Proper, strictly proper, and improper CT systems . . . 496
17.4 Z transform methods . . . 499
17.4.1 The Z Transform of translated sequences . . . 499
17.4.2 Using the Z Transform to solve unforced linear difference equations . . . 500
17.4.3 Discrete-time (DT) transfer functions . . . 501
17.4.3.1 The transfer function of a DAC – $G(s)$ – ADC cascade . . . 502
17.4.3.2 Causal, strictly causal, and noncausal DT systems . . . 503
17.4.4 Reconciling the Laplace and Z transforms . . . 504
17.4.4.1 Tustin's approximation . . . 506
17.4.4.2 Tustin's approximation with prewarping . . . 507
17.5 Frequency-domain analyses and filters . . . 508
17.5.1 The Bode plot . . . 508
17.5.1.1 Sketching Bode plots of real systems by hand . . . 510
17.5.2 Low-pass, high-pass, band-pass, and band-stop filters . . . 513
17.5.2.1 Maximal flatness filters: Butterworth and Bessel . . . 513
17.5.2.2 Equiripple filters: Chebyshev, inverse Chebyshev, and elliptic† . . . 516
Exercises . . . 521

By way of introduction, we begin this chapter, in §17.1, by presenting briefly the three essential classes of transforms at the heart of signal analysis: Fourier, Laplace, and Z. The **Fourier transform**, which comes in four forms appropriate for either **continuous signals** {defined throughout the domain of interest} or **discrete signals** {defined only at regularly-spaced intervals over the domain of interest}, and for signals defined on either **infinite domains** {that is[1], $t \in (-\infty,\infty)$} or **bounded domains** {that is, $t \in [0,T)$}, is built on *sinusoidal* basis functions, $e^{i\omega t} = \cos(\omega t) + i\sin(\omega t)$, as studied in depth in §5.

Before discussing in depth the Laplace and Z transforms and their extensive utility in control theory, we endeavor to make this discussion a bit more concrete by diverging for a bit and presenting next, in §17.2, some ODE models for a handful of simple physical systems (a.k.a. "plants"); we return to these motivating examples at several points later in the presentation in both this chapter and the next. Note also that the realization of various ODEs of interest (particularly as "controllers") as *electric circuits* is considered in §19.

The **Laplace transform**, which is appropriate for the analysis of **continuous signals** on **semi-infinite domains** $t \in [0,\infty)$, as well as for the analysis of the **differential systems** that govern their evolution from given initial conditions at $t = 0$, is built on *exponential* basis functions, e^{st}, and is developed in §17.3.

The Z **transform**, which is appropriate for the analysis of **discrete signals** on **semi-infinite domains** $t_k \in \{0,h,2h,3h,\dots\}$, as well as for the analysis of the **difference systems** that govern their evolution from given initial conditions around $t_0 = 0$, is built on *polynomial* basis functions, z^{k-1}, and is developed in §17.4.

The use of Fourier transforms in signal analysis, introduced in §5, is then extended significantly in §17.5.

17.1 Introduction to transforms: Fourier, Laplace, and Z

Recall first the **tetralogy** of Fourier transforms developed in §5:
The forward and inverse **infinite Fourier series transform** [see (5.8)], defined for *continuous signals* $u(t)$ on *bounded domains* $t \in [0,T)$ with $\omega_n = 2\pi n/T$ for $n \in \{\dots,-2,-1,0,1,2,\dots\}$, are defined by

$$\hat{u}_n = \frac{1}{T}\int_0^T u(t)e^{-i\omega_n t}\,dt \quad \Leftrightarrow \quad u(t) = \sum_{n=-\infty}^{\infty} \hat{u}_n e^{i\omega_n t}, \tag{17.1a}$$

the forward and inverse **infinite Fourier integral transform** [see (5.15)], defined for *continuous signals* $u(t)$ on *infinite domains* $t \in (-\infty,\infty)$ with $\omega \in (-\infty,\infty)$, are defined[2] by

$$\hat{u}(\omega) = \frac{1}{2\pi}\int_{-\infty}^{\infty} u(t)e^{-i\omega t}\,dt \quad \Leftrightarrow \quad u(t) = \int_{-\infty}^{\infty} \hat{u}(\omega)e^{i\omega t}\,d\omega, \tag{17.1b}$$

the forward and inverse **finite Fourier series transform** [see (5.23)], defined for *discrete signals* $u_k = u(t_k)$ on *bounded domains* $t_k = kh$ for $k = \{0,\dots,N-1\}$ and $h = T/N$ with[3] $\omega_n = 2\pi n/T$ for $n \in \{-N/2,\dots,N/2\}$, are defined by

$$\hat{u}_n = \frac{1}{N}\sum_{k=0}^{N-1} u_k e^{-i\omega_n t_k} \quad \Leftrightarrow \quad u_k = \sum_{n=-N/2}^{N/2} \hat{u}_n e^{i\omega_n t_k}, \tag{17.1c}$$

and the forward and inverse **finite Fourier integral transform** [see (5.70)], defined for *discrete signals* $u_k = u(t_k)$ on *infinite domains* $t_k = kh$ for $k = \{\dots,-2,-1,0,1,2,\dots\}$ with $\omega \in (-\pi/h,\pi/h)$, are defined by

$$\hat{u}(\omega) = \frac{h}{2\pi}\sum_{k=-\infty}^{\infty} u_k e^{-i\omega t_k} \quad \Leftrightarrow \quad u_j = \int_{-\pi/h}^{\pi/h} \hat{u}(\omega)e^{i\omega t_j}\,d\omega. \tag{17.1d}$$

[1] As in §5, the physical coördinate over which such transforms may be applied may be interpreted as time or space, and is denoted without loss of generality in the present chapter as t; see also footnote 19 on page 158.

[2] Recall from Footnote 10 on page 151 that there are alternative definitions of the Fourier integral, so different authors will place the factor of 2π in these formulae in different ways.

[3] Note in particular the discussion in §5.5 of the peculiar component of the discrete signal at the **Nyquist frequency** $\omega_{N/2} = \pi N/T$.

Similarly, the forward and inverse **Laplace transform** [developed in §17.3] are defined by

$$U(s) = \int_0^\infty u(t) e^{-st} dt \quad \Leftrightarrow \quad u(t) = \frac{1}{2\pi \mathrm{i}} \int_{\gamma - \mathrm{i}\infty}^{\gamma + \mathrm{i}\infty} U(s) e^{st} ds, \tag{17.2}$$

and the forward and inverse Z **transform** [developed in §17.4] are defined by

$$U(z) = \sum_{k=0}^{\infty} u_k z^{-k} \quad \Leftrightarrow \quad u_k = \frac{1}{2\pi \mathrm{i}} \oint_\Gamma U(z) z^{k-1} dz. \tag{17.3}$$

17.1.1 The relation of the Laplace and Z transforms to the Fourier transform

At the outset, note that the Laplace transform at right in (17.2) is simply a representation, or "expansion", of a continuous function $u(t)$ on $t \in [0, \infty)$ as a linear combination of a set of exponential basis functions e^{st} with the coefficient function $U(s)$ as weights. Similarly, the Z transform at right in (17.3) is simply a representation of a discrete function u_k on $k = 0, 1, 2, \ldots$ as a linear combination of a set of polynomial basis functions z^{k-1} with the coefficient function $U(z)$ as weights. The Laplace and Z transforms are thus remarkably similar to the corresponding Fourier transforms (17.1b) and (17.1d), respectively, which similarly represent continuous and discrete functions on infinite domains as a linear combination of a set of complex exponential basis functions with the Fourier coefficients as weights. Indeed, noting the definition of the Laplace transform in (17.2) and the infinite Fourier integral expansion in (17.1b), it follows that

$$\left.\begin{aligned} U(s) &= \int_0^\infty u(t) e^{-st} dt \\ \hat{u}(\omega) &= \frac{1}{2\pi} \int_{-\infty}^\infty u(t) e^{-\mathrm{i}\omega t} dt \end{aligned}\right\} \quad \Rightarrow \quad \hat{u}(\omega) = \frac{1}{2\pi} U(\mathrm{i}\omega) \quad \text{if} \quad u(t) = 0 \ \text{ for } \ t < 0. \tag{17.4}$$

Similarly, noting the definition of the Z transform in (17.3) and the finite Fourier integral expansion in (17.1d), it follows that

$$\left.\begin{aligned} U(z) &= \sum_{k=0}^{\infty} u_k z^{-k} \\ \hat{u}(\omega) &= \frac{h}{2\pi} \sum_{k=-\infty}^{\infty} u_k e^{-\mathrm{i}\omega t_k} \end{aligned}\right\} \quad \Rightarrow \quad \hat{u}(\omega) = \frac{h}{2\pi} U(e^{\mathrm{i}\omega h}) \quad \text{if} \quad u_k = 0 \ \text{ for } \ k < 0. \tag{17.5}$$

17.1.2 The remarkable utility of such transforms

The utility of the Fourier transform in the identification and analysis of the various sinusoidal components of a signal at different spatial frequencies or temporal wavenumbers has already been encountered in §5. (Indeed, any aspiring audiophile is already well familiar with the need to route the "low-frequency sinusoidal components" of an audio signal to a woofer, to route the "high-frequency sinusoidal components" of an audio signal to a tweeter, and to dampen the "highest-frequency sinusoidal components" of an audio signal associated with noise, which can come from a variety of sources; the Fourier transform simply makes this decomposition of a signal into sinusoidal components at different frequencies mathematically precise.)

The Laplace and Z transforms are similarly natural for the analysis of the *evolution* of **continuous-time** (**CT**) systems and **discrete-time** (**DT**) systems from initial conditions, governed by differential equations and difference equations respectively. As such transform methods are centrally based on an *abstraction* (the temporal frequency ω or spatial wavenumber k in the case of the Fourier transforms, the exponential scaling s in the case of the Laplace transform, and the base of the polynomial expansion, z, in the case of the Z

transform), they require a bit of analysis before their utility is fully apparent; §17.3 and §17.4 are intended to make this utility evident.

It should be noted at the outset, however, that all of these transforms are **linear**: that is, if X and Y are the (Fourier, Laplace, or Z) transforms of x and y, then $\alpha X + \beta Y$ is the corresponding transform of $\alpha x + \beta y$. Further, all of these transforms are **invertible**: that is, knowledge of the untransformed variable x over the appropriate region of the physical domain is sufficient to reconstruct the transformed variable X over the abstracted domain, and knowledge of the transformed variable X over the appropriate region of the abstracted domain is sufficient to reconstruct the untransformed variable x over the physical domain. These two points are essential to the practical utility of analysis, filtering, and control techniques based on such transforms.

17.2 A survey of the dynamics of some physical systems

We now describe a few simple physical systems that are considered further, in the analysis and control settings, in the remainder of this work. For the sake of brevity, rather than building up the subject of 3D dynamics and fluid mechanics from first principles, we instead simply write the low-dimensional ODEs governing the dynamics of a few representative mechanical, fluid, chemical, automotive, structural, and aerospace systems.

Example 17.1 A (linear) mass/spring/damper system

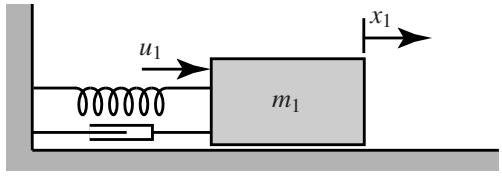


Figure 17.1: The single mass/spring/damper system set up in Example 17.1.

Recalling **Newton's second law**, $f = ma$ where m is the mass of the body, f is the force applied to the body, and a is the resulting linear acceleration of the body, the motion of the simple **mass/spring/damper**[4] system illustrated in Figure 17.1 is governed by

$$m_1 \frac{d^2 x_1}{dt^2} = u_1 - k x_1 - c \frac{dx_1}{dt}, \tag{17.6}$$

where x_1 is the deflection of the mass from its rest position, u_1 is the applied force, $\{m_1, k, c\}$ are constants, and the spring and damper have been modeled as linear in the deflection and velocity, respectively.

Identifying a SISO model by taking the **output** of the system as $y = x_1$, the **input** to the system as $u = u_1$, and defining $a_1 = c/m_1$, $a_0 = k/m_1$, and $b_0 = 1/m_1$, we may rewrite (17.6) in a standard **input/output ODE form** as

$$\frac{d^2 y}{dt^2} + a_1 \frac{dy}{dt} + a_0 y = b_0 u. \tag{17.7}$$

If the spring and damper are removed ($k = c = 0$), the system reduces to the **double integrator** $d^2y/dt^2 = b_0 u$.

Recalling **Newton's second law of rotation**, $\tau = I\alpha$ where I is the moment of inertia of the about the axis of rotation, τ is the applied torque, and α is the resulting angular acceleration, analogous rotational systems are easily identified that are governed by the same ODEs as those identified above. Hard disk read/write head/arm assemblies are an important engineering example system that fit such a model.

[4] A.k.a. **dashpot** or **shock absorber**. Note that shock absorbers often exhibit significant nonlinear characteristics for large or fast motions, in which case the linear model used here should be considered as only approximate.

Example 17.2 A (linear) cascade mass/spring system with viscous friction

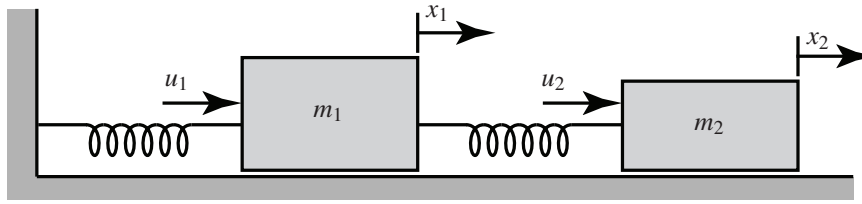


Figure 17.2: The cascade mass/spring system with viscous friction set up in Example 17.2.

The equations governing the motion of each mass in the cascade system illustrated in Figure 17.2 also follow immediately from Newton's second law:

$$m_1 \frac{d^2x_1}{dt^2} = u_1 - k_1 x_1 + k_2(x_2 - x_1) - \mu_1 m_1 g \frac{dx_1}{dt}, \tag{17.8a}$$

$$m_2 \frac{d^2x_2}{dt^2} = u_2 \qquad - k_2(x_2 - x_1) - \mu_2 m_2 g \frac{dx_2}{dt}, \tag{17.8b}$$

where x_1 and u_1 are the deflection of and applied force on the first mass, x_2 and u_2 are the deflection of and applied force on the second mass, $\{m_1, m_2, k_1, k_2, \mu_1, \mu_2\}$ are constants, and $g = 9.8$ m/sec^2. Note that the linear damping in this case is modeled as arising from the **viscous friction** between the blocks and the horizontal surface, assuming this interface is lubricated; in this case, the friction force is appropriately (and conveniently) modeled as proportional to both the weight of the respective block[5] and the velocity of the respective motion at the interface, and is of a sign that opposes this motion.

To manipulate a set of ODEs like (17.8) algebraically, it is convenient to first express it in operator form:

$$\left[m_1 \frac{d^2}{dt^2} + \mu_1 m_1 g \frac{d}{dt} + k_1 + k_2\right] x_1 + \left[-k_2\right] x_2 = u_1, \qquad \Rightarrow \qquad \mathscr{L}_1 x_1 + \mathscr{L}_2 x_2 = u_1, \tag{17.9a}$$

$$\left[-k_2\right] x_1 + \left[m_2 \frac{d^2}{dt^2} + \mu_2 m_2 g \frac{d}{dt} + k_2\right] x_2 \qquad = u_2, \qquad \Rightarrow \qquad \mathscr{L}_3 x_1 + \mathscr{L}_4 x_2 = u_2. \tag{17.9b}$$

Identifying a SISO model by taking, for example, the **output** of the system as $y = x_2$ and the **input** to the system as $u = u_1$ (and taking $u_2 = 0$), we may thus rewrite (17.8) by subtracting $\mathscr{L}_3$ times (17.9a) from $\mathscr{L}_1$ times (17.9b), noting that, e.g., $\mathscr{L}_1 \mathscr{L}_3 x_1 = \mathscr{L}_3 \mathscr{L}_1 x_1$, thus leading to a more standard ODE form:

$$(\mathscr{L}_1 \mathscr{L}_4 - \mathscr{L}_3 \mathscr{L}_2) x_2 = -\mathscr{L}_3 u_1 \qquad \Rightarrow \qquad \frac{d^4 y}{dt^4} + a_3 \frac{d^3 y}{dt^3} + a_2 \frac{d^2 y}{dt^2} + a_1 \frac{dy}{dt} + a_0 y = b_0 u \tag{17.10}$$

where $a_3 = (\mu_1 + \mu_2) g$, $a_2 = k_2/m_2 + (k_1 + k_2)/m_1 + \mu_1 \mu_2 g^2$, $a_1 = \mu_1 g k_2/m_2 + \mu_2 g (k_1 + k_2)/m_1$, $a_0 = k_1 k_2/(m_1 m_2)$, and $b_0 = k_2/(m_1 m_2)$.

Finally, as in (17.7), note that there are more derivatives on the output y than there are on the input u in the SISO ODE model given in (17.10); this property is essentially ubiquitous in mechanical systems with inertia, and is discussed further in §17.3.3.1.

[5] Or, the component of this weight normal to the interface if the interface is at an incline.

Example 17.3 A (nonlinear) mass/elastic-conveyer-belt system with dry friction

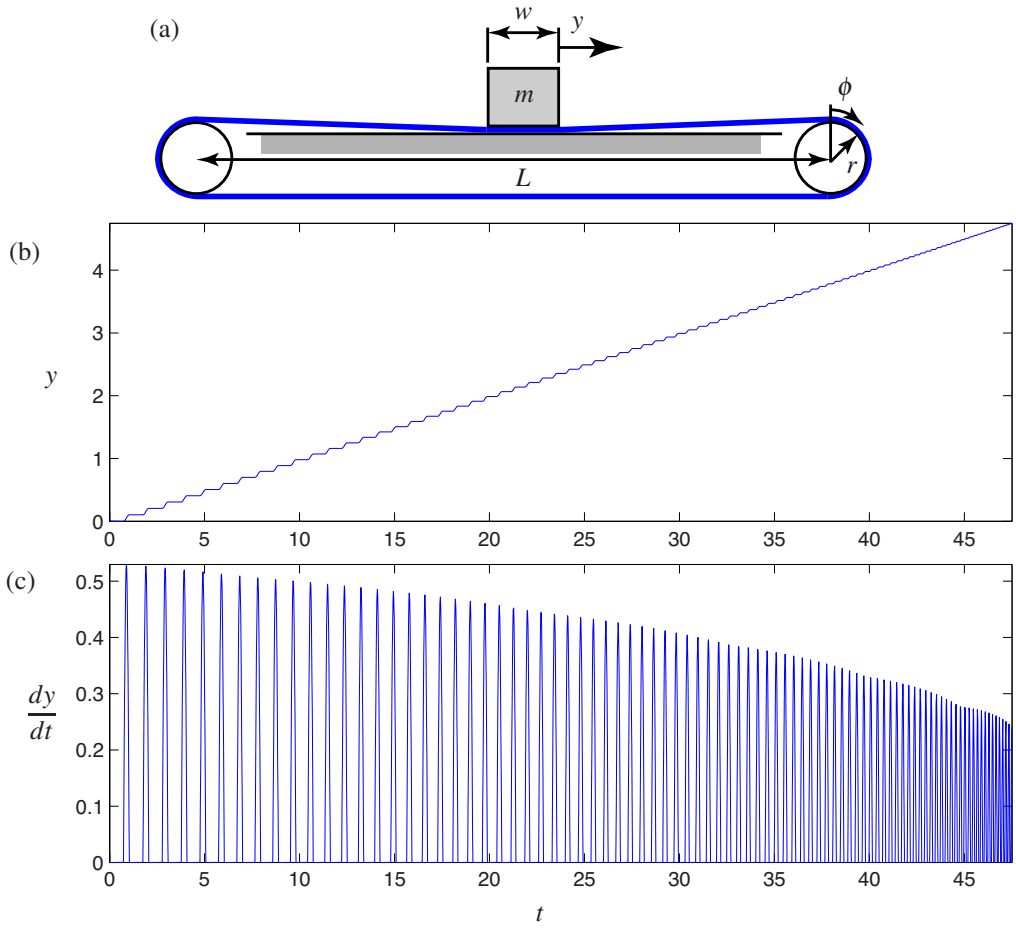


Figure 17.3: (a) The elastic conveyer belt system with dry friction set up in Example 17.3. To simplify the analysis (slightly), we assume that $w \ll L$ and $r \ll L$. The (b) position and (c) velocity components of the step response of this system as a function of time are also shown; note the **stick/slip** behavior that results from the nonlinear friction model. Note also that the frequency of the resulting **jerks** increases as the mass approaches the end of the belt.

We now consider the more problematical elastic conveyer belt system illustrated in Figure 17.3a.

In this system, the sections of the pretensioned elastic belt (that is, the "springs") acting to pull the mass to the left and right are effectively changing in "length" as the driven pulley, on the right end of the system, drags the mass across the table to the right. The modeling of the force applied by the belt thus requires some care. We first assume that the belt does not slip on the driven pulley and that the mass doesn't slip on the belt, though the idler pulley on the left end of the system is free to rotate and the belt, though it makes contact with the table under the mass, slides (with friction) across the table. We will refer to the "length" of the portion of the belt tending to pull the mass to the right as the distance from the mass directly to the driven pulley, $\ell_1 = L/2 - y$, and the "length" of the portion of the belt tending to pull the mass to the left as the distance from the mass to the driven pulley around the idler, $\ell_2 = 3L/2 + y$. For any given amount of rotation of the driven pulley $\phi(t)$, measured in radians, there is a corresponding nominal position of the mass $\overline{y}(t)$ at which

the force applied by the pretensioned belt to the left and right sides of the mass is equal. The actual position of the mass, $y(t)$, may thus be written

$$y = \overline{y} + y' \quad \text{where} \quad \overline{y} = r\phi, \tag{17.11a}$$

where $y'(t)$ denotes the (small) perturbation of the mass from the nominal position $\overline{y}(t)$. The total force applied by the belt to the mass, which opposes the perturbation y', is then given by

$$f_{\text{belt}} = -y'(k_0/\ell_1 + k_0/\ell_2) \tag{17.11b}$$

where k_0, the spring constant per unit length of the belt, is a constant.

The friction force f_{friction} caused by the dry contact of the portion of the belt under the mass with the table is accurately modeled in two parts. If you have ever tried to push a heavy object without wheels across a level surface[6], you probably recall that it takes more force to get the object moving than it takes to keep it moving, and that once the object is moving, the force required to keep it moving is approximately independent of the speed at which it is moving (this latter property is know as **Coulomb's law**). That is,

- if the velocity of the mass is zero (i.e., the system is **stuck**), the magnitude of the friction force f_{friction} precisely matches the force applied to the mass by the belt, with a sign that opposes the force applied by the belt, up to a maximum absolute value of $\mu_s mg$, where mg is the weight of the mass[7,8], whereas
- if the velocity of the mass is nonzero (i.e., the system is **unstuck**), the magnitude of the friction force f_{friction} is $\mu_k mg$, with a sign that opposes the motion of the belt (and the mass that sits thereon),

where μ_s is the **coefficient of static friction** and μ_k is the **coefficient of kinetic friction**; thus,

$$f_{\text{friction}} = \begin{cases} -\min(|f_{\text{belt}}|, \mu_s mg)\,\text{sgn}\,(f_{\text{belt}}) & \text{if } dy/dt = 0, \\ -\mu_k mg\,\text{sgn}\,(dy/dt) & \text{if } dy/dt \neq 0. \end{cases} \tag{17.11c}$$

Typically, $\mu_s > \mu_k$; representative values[9] of these two coefficients for a rubber belt and a metal surface are $\mu_s \approx 1.0$ and $\mu_k \approx 0.5$.

The motion of the mass is thus governed by

$$m\frac{d^2y}{dt^2} = f_{\text{belt}} + f_{\text{friction}}, \tag{17.11d}$$

where f_{belt} and f_{friction} are given above, with $\overline{y} = y' = \phi = 0$ corresponding to the mass at the center of the conveyer belt with no net force applied by the belt to the mass.

The motion that the above system exhibits is illustrated in Figure 17.3b-c; for the purpose of this numerical simulation, we take $m = 1$ kg, $r = 0.1$ m, $k_0 = 500$ N, and $L = 10$ m. This system may be simulated accurately using, e.g., the standard RK4 technique (see §10.4.1.1); however, care must be taken in order to switch accurately between the "stuck" and "unstuck" conditions. The code used to perform the simulation illustrated in Figure 17.3b-c is available as `Example_17_3.m` in the *NRC*.

The **stick/slip** behavior illustrated in Figure 17.3b-c is a nonlinear phenomenon that defies any reasonably accurate linear approximation. Some physical systems are like this, with systems exhibiting dry friction being particularly "sticky" to deal with. Fortunately, many[10] "highly nonlinear"[11] systems are *not* like this, and can be treated adequately via **linearization**, as illustrated in the several examples presented next.

[6] Most students attempt this at least once when moving into or out of college and/or graduate school...

[7] This type of frictional force is often referred to as **stiction**.

[8] If the belt is at an angle θ from horizontal, the normal force $mg\cos(\theta)$ across the interface should be used instead.

[9] Tables of such coefficients, for different materials in contact, are broadly available on the web.

[10] Indeed, it is our experience that *most* control problems encountered in practice may be treated effectively with linear methods.

[11] The phrase "highly nonlinear", like "mostly dead" and "very unique", should be avoided in scientific writing.

Example 17.4 A simple rolling cart system, and its linearization

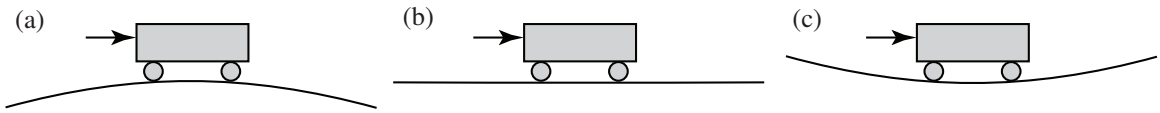


Figure 17.4: A second-order system governed by (17.12) in: (a) an unstable configuration with $\alpha < 0$, (b) a neutrally-stable configuration with $\alpha = 0$, and (c) an oscillatory configuration with $\alpha > 0$.

We now consider now the dynamics of a simple rolling cart (see Figure 17.4) governed by

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + mg\sin\left(\frac{dy}{dx}\right) = u, \qquad \text{where} \qquad y = \alpha x^2. \tag{17.12}$$

Combining these two equations, applying the identity (B.72) [i.e., $\sin(\varepsilon) = \varepsilon - \varepsilon^3/3! + \ldots$], and **linearizing** (i.e., performing the necessary Taylor series expansions and, assuming x and u are small, neglecting all terms that are quadratic or higher in x and/or u) leads to

$$\frac{d^2x}{dt^2} + a_1\frac{dx}{dt} + a_0 x = b_0 u \quad \text{where} \quad a_1 = \frac{c}{m}, \;\; a_0 = 2\alpha g, \;\; b_0 = \frac{1}{m}. \tag{17.13}$$

Example 17.5 Inverted and hanging pendulum/cart systems, and their linearization

It is straightforward to derive the full nonlinear equations of motion of the inverted and hanging pendulum/cart systems illustrated in Figure 17.5. Define P_x and P_y as the forces the pendulum exerts on the cart in the $\mathbf{e}^1$ and $\mathbf{e}^2$ directions (and, thus, the cart exerts the opposite forces on the pendulum), $x(t)$ as the horizontal position of the cart, $\theta(t)$ as the angle of the pendulum (measured counterclockwise from upright), and $\mathbf{r}(t)$ as a vector from a (stationary) coordinate system origin to the center of mass of the pendulum. Writing $\mathbf{r}(t)$ as a function of $x(t)$ and $\theta(t)$ (known as a **kinematic** relationship), differentiating twice, and reärranging gives

$$\mathbf{r} = [x - \ell\sin\theta\,]\,\mathbf{e}^1 + [\,\ell\cos\theta\,]\,\mathbf{e}^2, \tag{17.14a}$$

$$\frac{d^2\mathbf{r}}{dt^2} = \left[\frac{d^2x}{dt^2} - \ell\cos\theta\,\frac{d^2\theta}{dt^2} + \ell\sin\theta\left(\frac{d\theta}{dt}\right)^2\right]\mathbf{e}^1 - \left[\ell\sin\theta\,\frac{d^2\theta}{dt^2} + \ell\cos\theta\left(\frac{d\theta}{dt}\right)^2\right]\mathbf{e}^2 \tag{17.14b}$$

$$= \left[\cos\theta\,\frac{d^2x}{dt^2} - \ell\frac{d^2\theta}{dt^2}\right]\mathbf{e}^{\perp} - \left[\sin\theta\,\frac{d^2x}{dt^2} + \ell\left(\frac{d\theta}{dt}\right)^2\right]\mathbf{e}^{\parallel}, \tag{17.14c}$$

where $\mathbf{e}^{\perp} = \mathbf{e}^1\cos\theta + \mathbf{e}^2\sin\theta$ is the direction perpendicular to the pendulum, and $\mathbf{e}^{\parallel} = \mathbf{e}^2\cos\theta - \mathbf{e}^1\sin\theta$ is the direction parallel to the pendulum (see Figure 17.5a). We then write Newton's second law for the acceleration in the $\mathbf{e}^1$ direction of the cart and pendulum, and Newton's second law of rotation for the pendulum:

$$m_c\frac{d^2x}{dt^2} = P_x + u, \tag{17.15a}$$

$$m_p\left[\frac{d^2\mathbf{r}}{dt^2}\cdot\mathbf{e}^1\right] = m_p\left[\frac{d^2x}{dt^2} - \ell\cos\theta\,\frac{d^2\theta}{dt^2} + \ell\sin\theta\left(\frac{d\theta}{dt}\right)^2\right] = -P_x \tag{17.15b}$$

$$I_p\frac{d^2\theta}{dt^2} = -P_y\ell\sin\theta - P_x\ell\cos\theta; \tag{17.15c}$$

we are also interested in Newton's second law for the acceleration of the pendulum in the $\mathbf{e}^{\perp}$ direction:

$$m_p\left[\frac{d^2\mathbf{r}}{dt^2}\cdot\mathbf{e}^{\perp}\right] = m_p\left[\cos\theta\,\frac{d^2x}{dt^2} - \ell\frac{d^2\theta}{dt^2}\right] = -m_p g\sin\theta - P_y\sin\theta - P_x\cos\theta. \tag{17.15d}$$

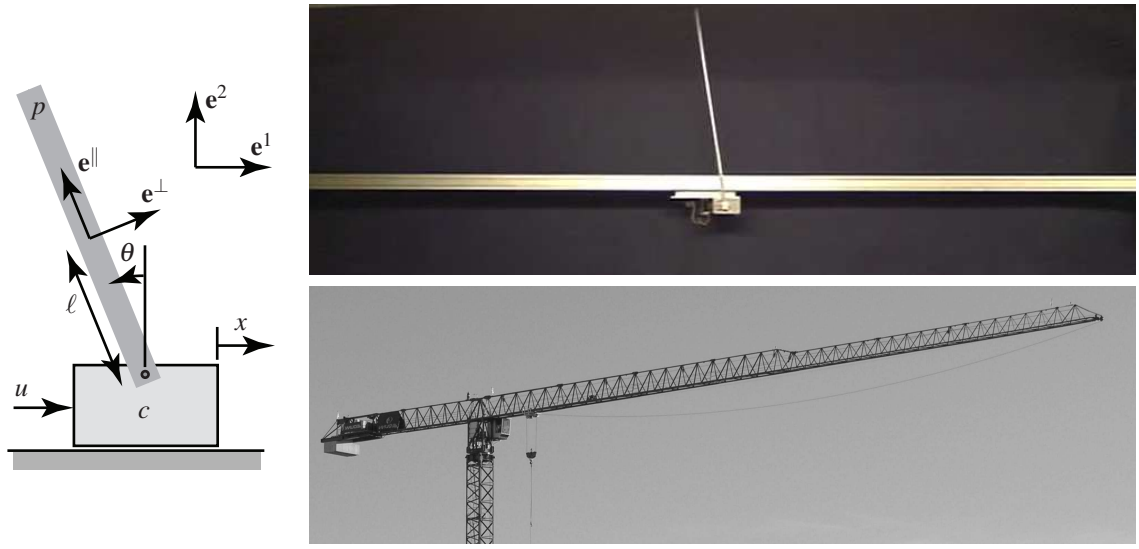


Figure 17.5: Pendulum/cart systems: (a) schematic, (b) lab realization in the **inverted** configuration $\theta(t) \approx 0$ [see (17.17)], and (c) a large-scale realization in the **hanging** configuration $\theta(t) \approx \pi$ [see (17.18)].

Note that $\{m_p, m_c\}$ are the masses of the pendulum and cart, I_p is the moment of inertia of the pendulum about its center of mass, ℓ is the distance from the center of mass of the pendulum to the point where it is pivotally attached to the cart, and g is the acceleration due to gravity; all of these parameters are positive.

First combining (17.15a) and (17.15b), then combining (17.15c) and (17.15d), leads to the two nonlinear equations of motion:

$$(m_c + m_p)\frac{d^2x}{dt^2} \quad - m_p \ell \cos\theta \frac{d^2\theta}{dt^2} + m_p \ell \sin\theta \left(\frac{d\theta}{dt}\right)^2 = u, \tag{17.16a}$$

$$-m_p \ell \cos\theta \frac{d^2x}{dt^2} + (I_p + m_p \ell^2)\frac{d^2\theta}{dt^2} - m_p g \ell \sin\theta \qquad = 0. \tag{17.16b}$$

Linearization of this system is performed by taking $x = \overline{x} + x'$, $\theta = \overline{\theta} + \theta'$, and $u = \overline{u} + u'$ in (17.16), expanding with Taylor series, multiplying out, applying the fact that the **nominal** condition $\{\overline{x}, \overline{\theta}, \overline{u}\}$ is itself also a solution of (17.16), and keeping only those terms which are linear in the perturbation (primed) quantities, as terms that are quadratic or higher in the perturbations are negligible if the perturbations are sufficiently small. Often, a nonlinear system is linearized about a *stationary* (a.k.a. **equilibrium**) nominal condition; such an equilibrium condition might be **stable**, such as the **hanging pendulum** configuration with $\{\overline{x} = 0, \overline{\theta} = \pi, \overline{u} = 0\}$, or **unstable**, such as the **inverted pendulum** configuration with $\{\overline{x} = 0, \overline{\theta} = 0, \overline{u} = 0\}$. More generally, the nominal condition about which small perturbations of a nonlinear system are modeled in a linearization may also be an *unsteady* trajectory of the system considered, which we denote $\{\overline{x}(t), \overline{\theta}(t), \overline{u}(t)\}$ for the problem considered in (17.16); this is called a **tangent linear** approximation of the equations of motion governing perturbations $\{x'(t), \theta'(t), u'(t)\}$ of the system from the "target" nominal trajectory $\{\overline{x}(t), \overline{\theta}(t), \overline{u}(t)\}$.

Taking $\{\overline{x} = 0, \overline{\theta} = 0, \overline{u} = 0\}$, the linearized equations of motion of the **inverted pendulum** are

$$(m_c + m_p)\frac{d^2x'}{dt^2} \qquad - m_p \ell \frac{d^2\theta'}{dt^2} \qquad = u', \tag{17.17a}$$

$$-m_p \ell \frac{d^2x'}{dt^2} + (I_p + m_p \ell^2)\frac{d^2\theta'}{dt^2} - m_p g \ell \theta' = 0. \tag{17.17b}$$

Taking $\{\overline{x}=0,\overline{\theta}=\pi,\overline{u}=0\}$, the linearized equations of motion of the **hanging pendulum** are

$$(m_c+m_p)\frac{d^2x'}{dt^2} \qquad +m_p\ell\frac{d^2\theta'}{dt^2} \qquad =u', \tag{17.18a}$$

$$m_p\ell\frac{d^2x'}{dt^2}+(I_p+m_p\ell^2)\frac{d^2\theta'}{dt^2}+m_pg\ell\theta'=0. \tag{17.18b}$$

Finally, considering an unsteady nominal trajectory $\{\overline{x}(t),\overline{\theta}(t),\overline{u}(t)\}$ gives

$$\begin{aligned}(m_c+m_p)\frac{d^2(\overline{x}+x')}{dt^2}-m_p\ell\cos(\overline{\theta}+\theta')\frac{d^2(\overline{\theta}+\theta')}{dt^2}+m_p\ell\sin(\overline{\theta}+\theta')\Big(\frac{d(\overline{\theta}+\theta')}{dt}\Big)^2 &= \overline{u}+u',\\ -m_p\ell\cos(\overline{\theta}+\theta')\frac{d^2(\overline{x}+x')}{dt^2} \qquad +(I_p+m_p\ell^2)\frac{d^2(\overline{\theta}+\theta')}{dt^2}-m_pg\ell\sin(\overline{\theta}+\theta') &= 0;\end{aligned}$$

leveraging (B.46) and (B.47), multiplying out, applying the condition that $\{\overline{x}(t),\overline{\theta}(t),\overline{u}(t)\}$ itself satisfies (17.16), and keeping only those terms linear in the perturbation (primed) quantities then gives the **tangent linear** approximation of the equations of motion governing perturbations of the pendulum system, $\{x'(t),\theta'(t),u'(t)\}$, in the vicinity of any "target" nominal trajectory $\{\overline{x}(t),\overline{\theta}(t),\overline{u}(t)\}$:

$$\begin{aligned}(m_c+m_p)\frac{d^2x'}{dt^2}-m_p\ell\cos(\overline{\theta})\frac{d^2\theta'}{dt^2}+m_p\ell\Big[\frac{d^2\overline{\theta}}{dt^2}\sin(\overline{\theta})\,\theta'+\Big(\frac{d\overline{\theta}}{dt}\Big)^2(\cos\overline{\theta})\theta'+2\frac{d\overline{\theta}}{dt}\sin(\overline{\theta})\frac{d\theta'}{dt}\Big] &= u'.\\ -m_p\ell\cos(\overline{\theta})\frac{d^2x'}{dt^2}+(I_p+m_p\ell^2)\frac{d^2\theta'}{dt^2}-m_p\ell\Big[g(\cos\overline{\theta})\,\theta'-\frac{d^2\overline{x}}{dt^2}\sin(\overline{\theta})\,\theta'\Big] &= 0,\end{aligned}$$

Example 17.6 The Mobile Inverted Pendulum problem, and its linearization

The derivation of the equations of motion of the Mobile Inverted Pendulum (MIP; see Figure 17.6) is related to that of the classical inverted pendulum (Example 17.5). Define P_x and P_y as the forces that the MIP body exerts on the wheels in the $\mathbf{e}^1$ and $\mathbf{e}^2$ directions, $x(t)$ as the horizontal position of the center of the wheels, $\phi(t)$ as the angle of rotation of the wheels as they roll (measured counterclockwise from a reference position), $\theta(t)$ as the angle of the MIP body (measured counterclockwise from upright, with $-\pi/2<\theta(t)<\pi/2$), and $\mathbf{r}(t)$ as a vector from a stationary coordinate system origin to the center of mass of the MIP body. Writing $\mathbf{r}(t)$ as a function of $x(t)$ and $\theta(t)$, differentiating twice, and reärranging again gives (17.14).

A motor is attached which applies an input torque τ that tends to rotate the body in one direction and the wheels in the other. We assume for the moment that the two wheels of the vehicle move together (that is, the vehicle isn't turning), and that a stiction force between the wheels and the ground is generated such that the wheels do not slip, and thus

$$r\phi=x. \tag{17.19}$$

We then write Newton's second law for the acceleration in the $\mathbf{e}^1$ direction of the wheel centers and the center-of-mass of the MIP body, and Newton's second law of rotation for the MIP body and the wheels:

$$m_w\frac{d^2x}{dt^2}=P_x-f, \tag{17.20a}$$

$$m_b\Big[\frac{d^2\mathbf{r}}{dt^2}\cdot\mathbf{e}^1\Big]=m_b\Big[\frac{d^2x}{dt^2}-\ell\cos\theta\frac{d^2\theta}{dt^2}+\ell\sin\theta\Big(\frac{d\theta}{dt}\Big)^2\Big]=-P_x, \tag{17.20b}$$

$$I_b\frac{d^2\theta}{dt^2}=-\tau-P_y\ell\sin\theta-P_x\ell\cos\theta, \tag{17.20c}$$

$$I_w\frac{d^2\phi}{dt^2}=\tau-rf; \tag{17.20d}$$

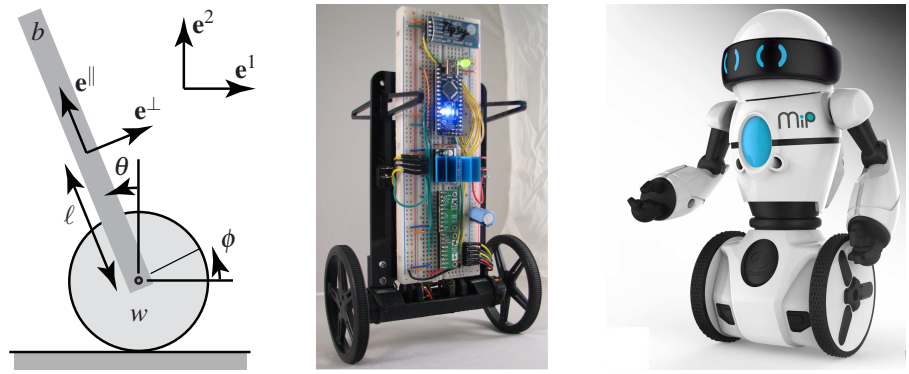


Figure 17.6: MIP, a Mobile Inverted Pendulum; (left) schematic, (center) prototype, (right) toy product commercialized by WowWee Robotics and the UCSD Coordinated Robotics Lab.

we are also interested in Newton's second law for the acceleration of the MIP body in the $\mathbf{e}^\perp$ direction:

$$m_b\left[\frac{d^2\mathbf{r}}{dt^2}\cdot\mathbf{e}^\perp\right] = m_b\left[\cos\theta\,\frac{d^2x}{dt^2} - \ell\,\frac{d^2\theta}{dt^2}\right] = -m_b\,g\,\sin\theta - P_y\sin\theta - P_x\cos\theta. \tag{17.20e}$$

Note that $\{m_b, m_w, I_b, I_w\}$ are the masses and moments of inertia (about their respective centers of mass) of the body and the sum of both wheels moving together, r is the radius of the wheels, ℓ is the distance from the center of mass of the MIP body to the axis of rotation of the wheels, g is the acceleration due to gravity; all of these parameters are positive. First combining (17.19), (17.20a), (17.20b) and (17.20d), then combining (17.20c) and (17.20e), leads to the nonlinear equations of motion of the MIP:

$$[I_w + (m_w+m_b)r^2]\,\frac{d^2\phi}{dt^2} + m_b\,r\,\ell\cos\theta\,\frac{d^2\theta}{dt^2} - m_b\,r\,\ell\sin\theta\left(\frac{d\theta}{dt}\right)^2 = \tau, \tag{17.21a}$$

$$m_b\,r\,\ell\cos\theta\,\frac{d^2\phi}{dt^2} + (I_b + m_b\,\ell^2)\,\frac{d^2\theta}{dt^2} - m_b\,g\,\ell\sin\theta = -\tau. \tag{17.21b}$$

Linearization of this system is performed by taking $\phi = \overline{\phi} + \phi'$, $\theta = \overline{\theta} + \theta'$, and $u = \overline{u} + u'$ in (17.21), applying the fact that the **nominal** condition $\{\overline{\theta}, \overline{\phi}, \overline{u}\}$ is itself also a solution of (17.21), and keeping only those terms which are linear in the perturbation (primed) quantities.

Taking $\{\overline{\phi} = 0, \overline{\theta} = 0, \overline{u} = 0\}$, the linearized equations of motion of the MIP about its upright state are

$$[I_w + (m_w+m_b)r^2]\,\frac{d^2\phi'}{dt^2} + m_b\,r\,\ell\,\frac{d^2\theta'}{dt^2} = \tau', \tag{17.22a}$$

$$m_b\,r\,\ell\,\frac{d^2\phi'}{dt^2} + (I_b + m_b\,\ell^2)\,\frac{d^2\theta'}{dt^2} - m_b\,g\,\ell\,\theta' = -\tau'. \tag{17.22b}$$

Considering an unsteady nominal trajectory $\{\overline{\phi}(t), \overline{\theta}(t), \overline{u}(t)\}$ and applying the same manipulations as before gives the **tangent linear** approximation of the equations governing the perturbations $\{\phi'(t), \theta'(t), u'(t)\}$ of the MIP in the vicinity of the nominal trajectory $\{\overline{\phi}(t), \overline{\theta}(t), \overline{u}(t)\}$:

$$[I_w + (m_w+m_b)r^2]\,\frac{d^2\phi'}{dt^2} + m_b\,r\,\ell\cos\overline{\theta}\,\frac{d^2\theta'}{dt^2} + m_b\,r\,\ell\left[2\sin\overline{\theta}\,\frac{d\overline{\theta}}{dt}\,\frac{d\theta'}{dt} - \sin\overline{\theta}\,\frac{d^2\overline{\theta}}{dt^2}\,\theta' + \cos\overline{\theta}\left(\frac{d\overline{\theta}}{dt}\right)^2\theta'\right] = \tau',$$

$$m_b\,r\,\ell\cos\overline{\theta}\,\frac{d^2\phi'}{dt^2} + (I_b + m_b\,\ell^2)\,\frac{d^2\theta'}{dt^2} - m_b\,g\,\ell\,(\cos\overline{\theta})\,\theta' + m_b\,r\,\ell\sin\overline{\theta}\,\frac{d^2\overline{\phi}}{dt^2}\,\theta' = -\tau'. \tag{17.23}$$

Example 17.7 A (linear) temperature bath system with a transport delay

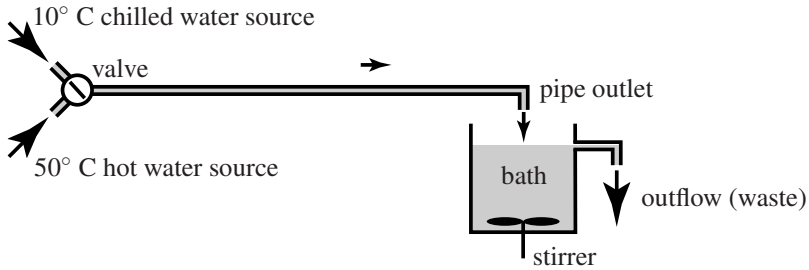


Figure 17.7: The temperature bath system set up in Example 17.7. The valve allows an adjustment of the flow temperature between $T_{\text{valve, min}} = 10^\circ$ C and $T_{\text{valve, max}} = 50^\circ$ C, and maintains a constant flow rate of $dV/dt = 6$ liters per minute in the inflow pipe; the wastewater flows out from the bath at precisely the same rate. The volume of fluid in the inflow pipe and the bath at any given time are $V_{\text{pipe}} = 1.2$ liters and $V_{\text{bath}} = 20$ liters. We assume further that (i) the inflow pipe is perfectly insulated, (ii) its walls have negligible thermal capacity, (iii) there is negligible heat diffusion within the fluid as it flows through the inflow pipe, and (iv) the bath is stirred quickly enough that it is maintained at essentially uniform temperature.

Performing a **control volume** analysis of the temperature bath system illustrated in Figure 17.7 to compute the **thermal energy** of the bath at time $t + \Delta t$ (for small Δt), at which time the bath has lost ΔV of the liquid it had at time t and gained ΔV of the new liquid from the pipe outlet, it follows that

$$V_{\text{bath}}\, T_{\text{bath}}(t+\Delta t) = (V_{\text{bath}} - \Delta V) T_{\text{bath}}(t) + \Delta V\, T_{\text{outlet}}(t) = (V_{\text{bath}} - \Delta V) T_{\text{bath}}(t) + \Delta V\, T_{\text{valve}}(t-d),$$

where $d = V_{\text{pipe}} / dV/dt = 12$ s represents the **convective transport delay** (that is, the time it takes the fluid to flow from the valve to the pipe outlet), and thus

$$T_{\text{bath}}(t) + \frac{dT_{\text{bath}}(t)}{dt}\Delta t + \ldots = T_{\text{bath}}(t) + \frac{\Delta V}{V_{\text{bath}}}[T_{\text{valve}}(t-d) - T_{\text{bath}}(t)];$$

taking $y(t) = T_{\text{bath}}(t)$, $u(t) = T_{\text{valve}}(t)$, and $a_0 = b_0 = dV/dt\,/\,V_{\text{bath}} = 0.005\ \text{s}^{-1}$, in the limit of small Δt we have

$$\frac{dT_{\text{bath}}(t)}{dt} = \frac{dV/dt}{V_{\text{bath}}}\Big[T_{\text{valve}}(t-d) - T_{\text{bath}}(t)\Big] \qquad \Rightarrow \qquad \frac{dy(t)}{dt} + a_0\, y(t) = b_0\, u(t-d). \tag{17.24}$$

Example 17.8 A Corvette ZR1 with a throttle delay, and its linearization

The equations of motion for an automobile at cruise (see Figure 17.8) may be written

$$\begin{aligned} m\frac{dv(t)}{dt} &= u(t-d) - f_d(t) - [f_r(t) + f_f(t)] \\ &= u(t-d) - C_d A \cdot 0.5\,\rho\, v(t)^2 - mg[C_0 + C_1 v(t)^{2.5}], \end{aligned} \tag{17.25}$$

where

- $u(t)$ denotes the "control" force applied to accelerate the vehicle by the engine,
- $f_d(t) = C_d A \cdot 0.5 \rho v(t)^2$ models the aerodynamic drag,
- $f_r(t) + f_f(t) = mg[C_0 + C_1 v(t)^{2.5}]$ models the rolling drag (see p. 117 of Gillespie 1992), and
- $w(t)$ denotes the "disturbances" (caused by headwind/tailwind, road inclination, modeling errors, etc.).

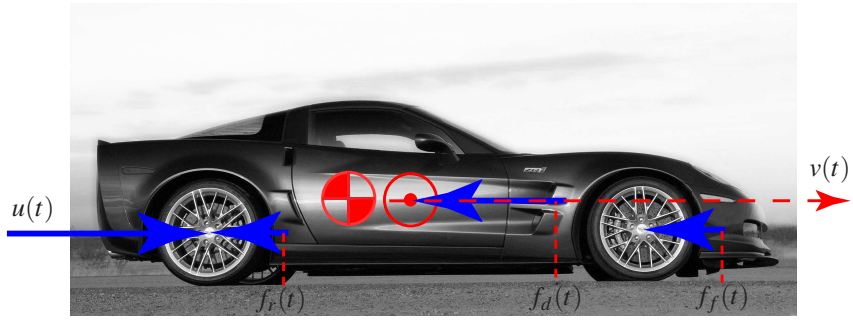


Figure 17.8: Coördinate system for the analysis of the **Corvette ZR1** at cruise, as set up in Example 17.8 (to clarify the drawing, the vertical and lateral forces are not marked).

Note that this model accounts for a slight delay d between the actuation of the throttle and its effect on the force applied to accelerate the vehicle. In our model of the vehicle depicted in Figure 17.8, we will take $C_d = 0.36, A = 2.07, \rho = 1.2, m = 1520, g = 9.8, C_0 = .01, C_1 = 1.2 \cdot 10^{-6}$, and $d = 0.04$, where all variables are in **SI units** (length in meters, mass in kilograms, time in seconds, force in Newtons, etc.)

At an equilibrium target car velocity, $v(t) = \overline{v}$, the corresponding throttle position $u(t) = \overline{u}$ is given by

$$d\overline{v}/dt = d\overline{u}/dt = 0, \quad \overline{u} = C_d A \cdot 0.5\,\rho\,\overline{v}^2 + mg[C_0 + C_1 \overline{v}^{2.5}]. \tag{17.26}$$

We now show how to linearize the dynamics of this system, taking $v(t) = \overline{v} + v'(t)$ and $u(t) = \overline{u} + u'(t)$ where $v'(t)$ and $u'(t)$ denote perturbations to the equilibrium car velocity and throttle position respectively.

Recall that any smooth function $f(x)$ may be expanded about $x = \overline{x}$ via a Taylor Series as follows:

$$f(\overline{x} + x') = f(x)\Big|_{x=\overline{x}} + \frac{df(x)}{dx}\Big|_{x=\overline{x}}(x') + \frac{d^2 f(x)}{dx^2}\Big|_{x=\overline{x}}\frac{(x')^2}{2!} + \frac{d^3 f(x)}{dx^3}\Big|_{x=\overline{x}}\frac{(x')^3}{3!} + \ldots;$$

thus, the function $f(v) = v^{2.5}$ may be expanded about $v = \overline{v}$ as follows:

$$[\overline{v} + v']^{2.5} = \overline{v}^{2.5} + 2.5\overline{v}^{1.5}(v') + O[(v')^2]. \tag{17.27}$$

Considering small perturbations (17.25) about the equilibrium condition $\{\overline{v}, \overline{u}\}$, by substituting $v(t) = \overline{v} + v'(t)$ and $u(t) = \overline{u} + u'(t)$ into (17.25), applying (17.27), then applying (17.26), then finally eliminating all terms which are quadratic or higher in the perturbation (primed) quantities, leads to a linear equation as follows:

$$\begin{aligned}
m\frac{d\{\overline{v} + v'(t)\}}{dt} &= \{\overline{u} + u'(t-d)\} - C_d A \cdot 0.5\,\rho\,\{\overline{v} + v'(t)\}^2 - mg[C_0 + C_1\{\overline{v} + v'(t)\}^{2.5}], \\
m\frac{d\{\overline{v} + v'(t)\}}{dt} &= \overline{u} + u'(t-d) - C_d A \cdot 0.5\,\rho\,\{\overline{v}^2 + 2\overline{v}[v'(t)] + [v'(t)]^2\} \\
&\qquad - mg[C_0 + C_1\{\overline{v}^{2.5} + 2.5\overline{v}^{1.5}v'(t) + O[(v'(t))^2]\}], \\
\Rightarrow \quad m\frac{d\{v'(t)\}}{dt} &= u'(t-d) - C_d A \cdot 0.5\,\rho\,\{2\overline{v}[v'(t)] + [v'(t)]^2\} - mg[C_1\{2.5\overline{v}^{1.5}v'(t) + O[(v'(t))^2]\}], \\
m\frac{d\{v'(t)\}}{dt} &= u'(t-d) - C_d A\,\rho\,\overline{v}[v'(t)] - mgC_1 2.5\overline{v}^{1.5}[v'(t)],
\end{aligned}$$

thus resulting in the linear ODE

$$\left(\frac{d}{dt} + a_0\right) v'(t) = b_0 u'(t-d) \qquad \text{where} \qquad a_0 = C_d A\,\rho\,\overline{v}/m + 2.5 C_1\, g\overline{v}^{1.5} > 0. \tag{17.28}$$

Example 17.9 A three-story building during an earthquake, and its linearization

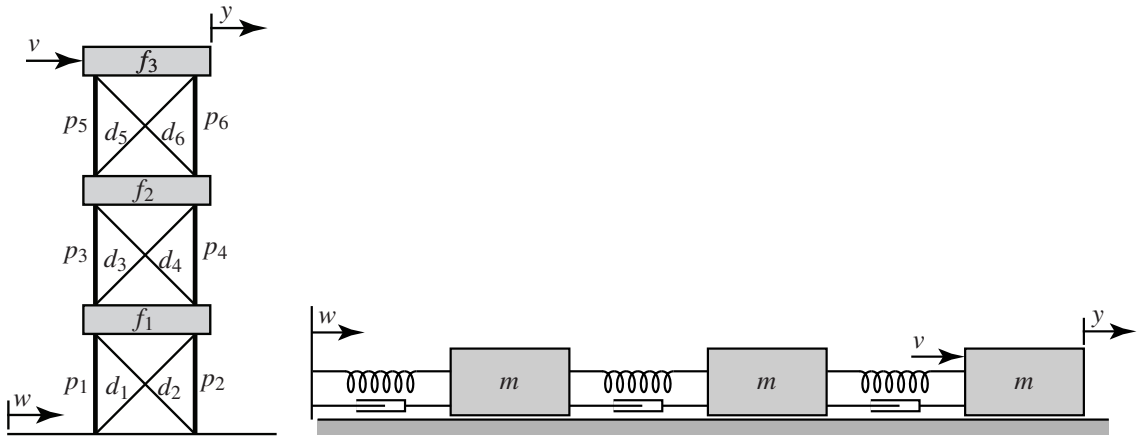


Figure 17.9: (a) The three-story building considered in Example 17.9, and (b) a cascade spring/mass/damper system which provides an equivalent model for the linearized horizontal dynamics of this structure.

We now analyze the dynamics of the three-story building illustrated in Figure 17.9 during an earthquake; note that we already analyzed the statics of this building in Example 2.1.

Each of the three floors is of mass $m = 1000$ kg, and the diagonals are nominally at 45° angles. The lengths of the pillars and floors are nominally $\ell_p = \ell_f \triangleq \ell = 5$ m; by the Pythagorean theorem, the lengths of the diagonals are nominally $\ell_d = \ell\sqrt{2}$. All joints are assumed to be **pinned**, so no members bear bending loads. The vertical pillars are under compression. The diagonal stabilizers are under tension, and each has a spring constant k and damping coefficient c; note that the structure is **pretensioned** (see Example 2.1), so the diagonal members remain under tension even as the building is deformed. An earthquake is modeled as horizontal motion of the ground, $w(t)$. We are primarily interested in the horizontal motion of the top floor, $y(t)$, which, we will see, can deflect a lot even for relatively small ground motion $w(t)$ if the building is forced at a critical **resonant frequency** of the structure by the earthquake.

We now model the horizontal motion of the ground floor, $x_1(t)$, the second floor, $x_2(t)$, and the top floor $x_3(t) = y(t)$, as a function of both the horizontal motion of the ground, $w(t)$, and the force applied to the top floor, $v(t)$. It will be seen that we can neglect the vertical motions of the floors, which (for small deflections) are small as compared with the horizontal motions.

Assume first that the horizontal position of the third floor is perturbed a small amount to the right of the horizontal position of the second floor; that is, $0 < (x_3 - x_2)/\ell \ll 1$. Denote by $\theta_6 = \sin^{-1}[(x_3 - x_2)/\ell] \approx (x_3 - x_2)/\ell$ the (clockwise) angle that the sixth pillar is deflected from its nominally vertical orientation. Note that, since $\cos\theta_6 = 1 + O([(x_3 - x_2)/\ell]^2)$, the perturbations in the vertical forces and vertical deflections of the floor are *quadratic* in the horizontal perturbation quantities; that is, they are negligible as compared with the horizontal forces and deflections if these quantities are small. The (clockwise) angles that the other pillars are deflected from their nominally vertical orientations may be defined and computed similarly,

$$\theta_5 \approx \theta_6 \approx (x_3 - x_2)/\ell, \quad \theta_3 \approx \theta_4 \approx (x_2 - x_1)/\ell, \quad \theta_1 \approx \theta_2 \approx (x_1 - w)/\ell, \tag{17.29}$$

and also result in negligible perturbations in the vertical forces and vertical positions of the floors; we thus focus on the horizontal dynamics in the remainder of this example.

Denote by δ_5 the changes in length of the fifth diagonal member from its nominal (pretentioned) length ℓ_d. Noting that $|x_3 - x_2|/\ell \ll 1$ [and, therefore, $|\delta_5|/\ell \ll 1$], we again appeal to the Pythagorean theorem:

$$\ell^2 + [\ell + (x_3 - x_2)]^2 = [\ell_d + \delta_5]^2 \;\Rightarrow\; 2\ell(x_3 - x_2) + (x_3 - x_2)^2 = 2\sqrt{2}\ell\delta_5 + \delta_5^2 \;\Rightarrow\; \delta_5 \approx (x_3 - x_2)/\sqrt{2}.$$

Again, when performing linearizations of this sort, terms which are quadratic in the perturbation quantities are negligible as compared with terms which are linear in the perturbation quantities, which are assumed to be small. The changes in lengths of the other diagonal members may be computed similarly:

$$\begin{aligned} \delta_5 &\approx \quad (x_3-x_2)/\sqrt{2}, \quad \delta_3 \approx \quad (x_2-x_1)/\sqrt{2}, \quad \delta_1 \approx \quad (x_1-w)/\sqrt{2}, \\ \delta_6 &\approx -(x_3-x_2)/\sqrt{2}, \quad \delta_4 \approx -(x_2-x_1)/\sqrt{2}, \quad \delta_2 \approx -(x_1-w)/\sqrt{2}. \end{aligned} \tag{17.30}$$

Finally, denote by $\phi_5 = \tan^{-1}[(\ell+x_3-x_2)/\ell] - \pi/4$ the angle that the fifth diagonal member is rotated from its nominal $\pi/4$ radian orientation (again, measured clockwise from vertical). Noting the identities (B.49), (B.77), and (B.74),

$$\tan(x+y) = \frac{\tan x + \tan y}{1-\tan x \tan y}, \quad \frac{1}{1-\varepsilon} = 1+\varepsilon+\varepsilon^2+\ldots, \quad \tan(\varepsilon) = \varepsilon + \frac{\varepsilon^3}{3} + \ldots,$$

when $\phi_5 \ll 1$ it follows that

$$\tan(\pi/4+\phi_5) = \frac{1+\tan\phi_5}{1-\tan\phi_5} = [1+\phi_5+O(\phi_5^3)][1+\phi_5+O(\phi_5^2)] = 1+2\phi_5+O(\phi_5^2);$$

thus, $\phi_5 \approx (x_3-x_2)/(2\ell)$. The (clockwise) angles that the other diagonal members are rotated from their nominal orientations ($\pm\pi/4$ radians from vertical) may be computed similarly:

$$\phi_5 \approx \phi_6 \approx (x_3-x_2)/(2\ell), \quad \phi_3 \approx \phi_4 \approx (x_2-x_1)/(2\ell), \quad \phi_1 \approx \phi_2 \approx (x_1-w)/(2\ell). \tag{17.31}$$

We are now in a position to add up all of the horizontal forces on the floors when the structure undergoes small horizontal movements. The horizontal acceleration of the top floor is acted upon by the external force v, the horizontal component of the force from the two rotated pillars [see (17.29)], and the horizontal component of the force from the two extended [see (17.30)] and rotated [see (17.31)] diagonal members; noting the nominal loading computed in (2.7) and the identity (B.46) [$\sin(x+y) = \sin x \cos y + \cos x \sin y$], neglecting terms that are quadratic or higher in the perturbations, this may be summed up as follows,

$$\begin{aligned} m\frac{d^2x_3}{dt^2} &= v + p_5 \sin\theta_5 + p_6 \sin\theta_6 - \left(d_5 + k\,\delta_5 + c\frac{d\delta_5}{dt}\right)\sin\left(\frac{\pi}{4}+\theta_5\right) - \left(d_6 + k\,\delta_6 + c\frac{d\delta_5}{dt}\right)\sin\left(-\frac{\pi}{4}+\theta_6\right) \\ &\approx v + 2p_5\frac{x_3-x_2}{\ell} - \left[d_5 + \left(k + c\frac{d}{dt}\right)\frac{x_3-x_2}{\sqrt{2}}\right]\frac{1+(x_3-x_2)/\ell}{\sqrt{2}} \\ &\qquad - \left[d_6 - \left(k + c\frac{d}{dt}\right)\frac{x_3-x_2}{\sqrt{2}}\right]\frac{-1+(x_3-x_2)/\ell}{\sqrt{2}} \\ &\approx v + 2p_5\frac{x_3-x_2}{\ell} - 2d_5\frac{x_3-x_2}{\sqrt{2}\ell} - 2\left(k + c\frac{d}{dt}\right)\frac{x_3-x_2}{2} \\ &\approx v - \overline{k}_3(x_3-x_2) - c\left(\frac{dx_3}{dt} - \frac{dx_2}{dt}\right); \end{aligned} \tag{17.32a}$$

note that the horizontal forces of the other two floors may be summed up in a similar fashion,

$$m\frac{d^2x_2}{dt^2} = -\overline{k}_2(x_2-x_1) + \overline{k}_3(x_3-x_2) - c\left(\frac{dx_2}{dt} - \frac{dx_1}{dt}\right) + c\left(\frac{dx_3}{dt} - \frac{dx_2}{dt}\right), \tag{17.32b}$$

$$m\frac{d^2x_1}{dt^2} = -\overline{k}_1(x_1-w) + \overline{k}_2(x_2-x_1) - c\left(\frac{dx_1}{dt} - \frac{dw}{dt}\right) + c\left(\frac{dx_2}{dt} - \frac{dx_1}{dt}\right), \tag{17.32c}$$

where, noting the solution of the statics of this building derived in Example 2.1,

$$\overline{k}_1 = k - 2p_1/\ell + \sqrt{2}d_1/\ell = k - 5880\,\text{kg/sec}^2, \tag{17.33a}$$

$$\overline{k}_2 = k - 2p_3/\ell + \sqrt{2}d_3/\ell = k - 3919\,\text{kg/sec}^2, \tag{17.33b}$$

$$\overline{k}_3 = k - 2p_5/\ell + \sqrt{2}d_5/\ell = k - 1960\,\text{kg/sec}^2. \tag{17.33c}$$

For the purpose of clear visualization of the essence of this problem, we may thus ignore the vertical *separation* of the floors, and model small horizontal motions of this building linearly as a cascade spring/mass/damper system, as illustrated in Figure 17.9b, with spring constants $\overline{k}_1$, $\overline{k}_2$, and $\overline{k}_3$ and damping $\overline{c}_1 = \overline{c}_2 = \overline{c}_3 = c$.

The three second-order equations governing x_1, x_2, and x_3 may thus be rewritten as

$$\left(m\frac{d^2}{dt^2} + 2c\frac{d}{dt} + \overline{k}_1 + \overline{k}_2\right)x_1 = \left(c\frac{d}{dt} + \overline{k}_1\right)w + \left(c\frac{d}{dt} + \overline{k}_2\right)x_2 \quad \Rightarrow \quad \mathscr{L}_1 x_1 = \mathscr{L}_2 w + \mathscr{L}_3 x_2, \tag{17.34a}$$

$$\left(m\frac{d^2}{dt^2} + 2c\frac{d}{dt} + \overline{k}_2 + \overline{k}_3\right)x_2 = \left(c\frac{d}{dt} + \overline{k}_2\right)x_1 + \left(c\frac{d}{dt} + \overline{k}_3\right)x_3 \quad \Rightarrow \quad \mathscr{L}_4 x_2 = \mathscr{L}_5 x_1 + \mathscr{L}_6 x_3, \tag{17.34b}$$

$$\left(m\frac{d^2}{dt^2} + c\frac{d}{dt} + \overline{k}_3\right)x_3 = \left(c\frac{d}{dt} + \overline{k}_3\right)x_2 + v \quad \Rightarrow \quad \mathscr{L}_7 x_3 = \mathscr{L}_8 x_2 + v. \tag{17.34c}$$

The task of eliminating x_1 and x_2 from these three second-order ODEs, thereby determining a single sixth-order ODE relating $y = x_3$ to v and w, is algebraically involved; it is thus helpful (as in Example 17.2) to use the streamlined notation introduced above right for the scalar linear differential operators $\mathscr{L}_i$. Premultiplying (17.34a) by $\mathscr{L}_5$ and (17.34b) by $\mathscr{L}_1$ and combining to eliminate x_1 (noting, e.g., that $\mathscr{L}_1\mathscr{L}_5 = \mathscr{L}_5\mathscr{L}_1$) leads to

$$\mathscr{L}_1\mathscr{L}_4 x_2 = (\mathscr{L}_5\mathscr{L}_2 w + \mathscr{L}_5\mathscr{L}_3 x_2) + \mathscr{L}_1\mathscr{L}_6 x_3 \quad \Rightarrow \quad (\mathscr{L}_1\mathscr{L}_4 - \mathscr{L}_5\mathscr{L}_3)x_2 = \mathscr{L}_5\mathscr{L}_2 w + \mathscr{L}_1\mathscr{L}_6 x_3; \tag{17.35}$$

premultiplying (17.35) by $\mathscr{L}_8$ and (17.34c) by $(\mathscr{L}_1\mathscr{L}_4 - \mathscr{L}_5\mathscr{L}_3)$ and combining to eliminate x_2 then leads to

$$(\mathscr{L}_1\mathscr{L}_4\mathscr{L}_7 - \mathscr{L}_5\mathscr{L}_3\mathscr{L}_7 - \mathscr{L}_8\mathscr{L}_1\mathscr{L}_6)x_3 = (\mathscr{L}_1\mathscr{L}_4 - \mathscr{L}_5\mathscr{L}_3)v + (\mathscr{L}_8\mathscr{L}_5\mathscr{L}_2)w,$$

which, denoting $y = x_3$, may be rewritten as

$$\begin{aligned}\left(\frac{d^6}{dt^6} + a_5\frac{d^5}{dt^5} + a_4\frac{d^4}{dt^4} + a_3\frac{d^3}{dt^3} + a_2\frac{d^2}{dt^2} + a_1\frac{d}{dt} + a_0\right)y = \\ \left(b_4\frac{d^4}{dt^4} + b_3\frac{d^3}{dt^3} + b_2\frac{d^2}{dt^2} + b_1\frac{d}{dt} + b_0\right)v + \left(\overline{b}_3\frac{d^3}{dt^3} + \overline{b}_2\frac{d^2}{dt^2} + \overline{b}_1\frac{d}{dt} + \overline{b}_0\right)w.\end{aligned} \tag{17.36}$$

Symbolic manipulation tools may now be used to do the necessary (but tedious) algebraic simplifications (for Matlab implementation, see `Example_17_8.m` in the *NRC*) in order to determine the coefficients. As seen by running this code, for $k = 10000$ and $c = 10$, the coefficients work out to be:

$$a_5 = .05,\ \ a_4 = 32.361,\ \ a_3 = .76881,\ \ a_2 = 237.95,\ \ a_1 = 1.0706,\ \ a_0 = 201.40, \tag{17.37a}$$

$$b_4 = .001,\ \ b_3 = .00004,\ \ b_2 = .024320,\ \ b_1 = .00036480,\ \ b_0 = .10706, \tag{17.37b}$$

$$\overline{b}_3 = .000001,\ \ \overline{b}_2 = .0018240,\ \ \overline{b}_1 = 1.0706,\ \ \overline{b}_0 = 201.40; \tag{17.37c}$$

the values of the coefficients for other values of k and c may be determined similarly. The Bode plot of this system is considered in Exercise 17.6, the design of a passive vibration damper for this system is considered in Exercise 18.2, and this system is considered in state-space form in Exercise 20.1.

Example 17.10 The launch of a Saturn V rocket, and its linearization

The dynamics of a Saturn V rocket during liftoff (see Figure 17.10) may be considered in the $x-z$ plane and the $y-z$ plane separately, as the rocket is not spinning. Considering the dynamics in one of these planes, there are three equations governing the motion of the vehicle, two of the form $d^2x/dt^2 = f/m$ and one of the form $d^2\theta/dt^2 = \tau/J$:

$$m\frac{d^2z(t)}{dt^2} = f_t\cos[\theta(t)-u(t)] - f_d(t)\cos[\alpha(t)+\theta(t)] + w(t)\sin[\alpha(t)+\theta(t)] - f_g, \tag{17.38a}$$

$$m\frac{d^2x(t)}{dt^2} = f_t\sin[\theta(t)-u(t)] - f_d(t)\sin[\alpha(t)+\theta(t)] - w(t)\cos[\alpha(t)+\theta(t)], \tag{17.38b}$$

$$J\frac{d^2\theta(t)}{dt^2} = f_t D\sin[u(t)] \quad - f_d(t)L\sin[\alpha(t)] \quad - w(t)L\cos[\alpha(t)], \tag{17.38c}$$

which may be taken together with the **kinematic** condition

$$\frac{v_x(t)}{|\mathbf{v}(t)|} = \sin[\alpha(t)+\theta(t)]. \tag{17.39}$$

Note that the three second-order ODES in (17.38) may easily be rewritten as six first-order equations, and describe the evolution in time of the six state variables listed in Figure 17.10.

Taking the disturbance force $w(t)$, the horizontal velocity $v_x(t)$, and the angles $\{\theta(t),\alpha(t),u(t)\}$ to be small, the equation for the vertical acceleration, (17.38a), reduces upon linearization to

$$m\frac{d^2z(t)}{dt^2} = f_t - 10\left|\frac{dz(t)}{dt}\right|^2 - f_g, \tag{17.40a}$$

whereas the equations for the horizontal and angular acceleration, (17.38b)-(17.38c), reduce to

$$m\frac{d^2x(t)}{dt^2} = f_t\,[\theta(t)-u(t)] - f_d(t)\,[\alpha(t)+\theta(t)] - w(t). \tag{17.40b}$$

$$J\frac{d^2\theta(t)}{dt^2} = f_t D u(t) \quad - f_d(t)L\alpha(t) \quad - Lw(t). \tag{17.40c}$$

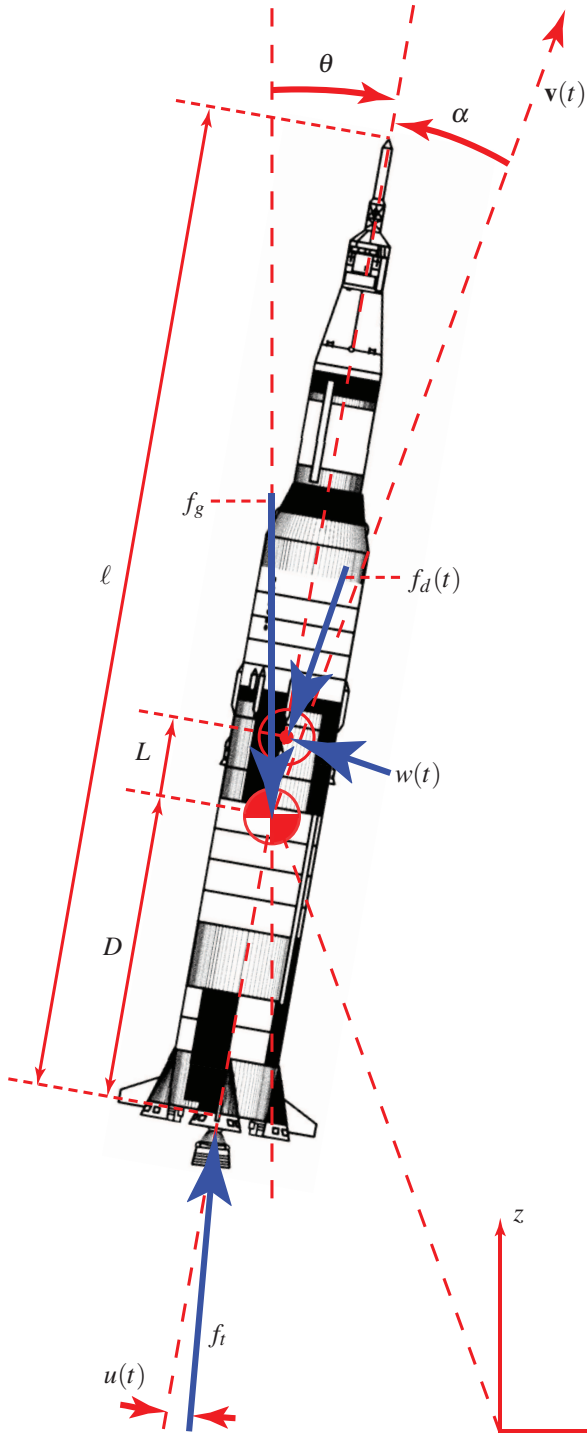
Note that (17.40a) can be marched in order to compute $dz(t)/dt = v_z(t) = |\mathbf{v}(t)|$ at any instant t_1. Given this value of $\overline{v} = |\mathbf{v}(t_1)|$, which varies only slowly in time due to the large mass of the rocket (and the fact that its thrust only slightly exceeds its weight), the linearized form of the auxiliary equation (17.39) may be written

$$\frac{dx(t)}{dt} = \overline{v}\,[\alpha(t)+\theta(t)]. \tag{17.41}$$

Considering $\overline{v}$ as essentially constant, defining $\overline{f}_d = 10\overline{v}^2$, and combining (17.40b)-(17.40c) and (17.41) to eliminate α leads to

$$m\frac{d^2x(t)}{dt^2} + \frac{\overline{f}_d}{\overline{v}}\frac{dx(t)}{dt} - f_t\,\theta(t) = -f_t\,u(t) - w(t), \tag{17.42a}$$

$$\frac{\overline{f}_d L}{\overline{v}}\frac{dx(t)}{dt} + J\frac{d^2\theta(t)}{dt^2} - \overline{f}_d L\,\theta(t) = f_t D\,u(t) - Lw(t). \tag{17.42b}$$



State variables:
$z(t)$ = vertical position
$v_z(t) = dz(t)/dt$ = vertical velocity
$x(t)$ = horizontal position
$v_x(t) = dx(t)/dt$ = horizontal velocity
$\theta(t)$ = angle (clockwise from vertical)
$\omega(t) = d\theta(t)/dt$ = angular velocity

Auxiliary variables:
$f_d(t) = 10\,|\mathbf{v}(t)|^2$ = aerodynamic drag
$\alpha(t)$ = angle of attack [see (17.39)]

Control input:
$u(t)$ = angle of thruster

Disturbance input:
$w(t)$ = wind + aerodynamic lift

Constants:
$\ell = 110$ = length
$m = 3 \times 10^6$ = mass
$J = m\ell^2/20$ = moment of inertia
$L = 10 \pm 5\%$ = distance C_p is ahead of C_m
$D = 40$ = distance from nozzle pivot to C_m
$f_t = 34 \times 10^6$ = thrust
$f_g = mg$ = weight ($g = 9.8$)

Figure 17.10: Coordinate system for a rocket stabilization problem. There are four forces acting on the rocket, directed as indicated: thrust f_t, weight f_g, drag $f_d(t)$, and "disturbances" (lift + wind) $w(t)$; the control $u(t)$ is the angle of the thruster. The rocket is assumed to be not spinning, and all angles indicated are assumed to be small, which decouples the control problem in the x-z plane (shown) from that in the y-z plane. All variables in SI units. [To clarify the diagram, only one of the five thrusters is shown in this sketch of the **Saturn V**.]

Example 17.11 Linearized dynamic models of aircraft

The state of an aircraft in flight may be defined by twelve variables: three to identify the position $\{X,Y,Z\}$, three to identify the velocity $\{U,V,W\}$, three to identify the orientation $\{\phi,\theta,\psi\}$, and three to identify the rate of change of the orientation, $\{p,q,r\}$. To identify the orientation, we first denote

- the **body-fitted coördinates** of the aircraft as three orthogonal vectors from the center of mass out the nose, out the right wingtip, and out the bottom of the aircraft for the x, y, and z axes, respectively, and
- a reference set of **inertial coördinates** (that is, a non-accelerating and non-rotating reference frame) as north, east, and down (NED) from the aircraft center of mass for the x_1, x_2, and x_3 axes, respectively.

Starting from the reference configuration of the aircraft, with its body-fitted coördinates aligned with the inertial (NED) coördinates, the orientation of the aircraft may then be identified unambiguously by three successive rotations[12] about its body-fitted coördinates, the most common choice in the aerodynamics literature being the $3-2-1$ **Tait-Bryan rotation sequence**[13] (a.k.a. $3-2-1$ **Euler rotation sequence**[14]) given by:

3 **yaw** the aircraft by an angle ψ about the z (down) axis (positive ψ yaws the nose to the right),
2 **pitch** the aircraft by an angle θ about the y (out-the-right-wing) axis (positive θ pitches the nose up),
1 **roll** the aircraft by an angle ϕ about the x (out-the-nose) axis (positive ϕ rolls the right wing down).

In the reference frame of the aircraft, three convenient auxiliary variables used to describe the dynamics are

- the **airspeed** v_T, which is the magnitude of the **relative wind** past the aircraft,
- the **angle of attack** (**AOA**) α, which is the angle between the x axis and the component of the relative wind in the $x-y$ plane, and
- the **sideslip angle** β, which is the angle between the x axis and the component of the relative wind in the $x-z$ plane.

The airspeed, angle of attack, and sideslip angle, $\{v_T,\alpha,\beta\}$, may be determined from the absolute velocity of the aircraft, $\{U,V,W\}$, together with the local wind velocity and the aircraft orientation as defined by the roll, pitch, and yaw variables, $\{\phi,\theta,\psi\}$, of the $3-2-1$ rotation sequence described above (alternatively, $\{U,V,W\}$ may be determined from $\{v_T,\alpha,\beta\}$ and $\{\phi,\theta,\psi\}$ and the local wind velocity). For the purpose of describing the dynamics of flight, of course, $\{v_T,\alpha,\beta\}$ are the natural variables to consider.

Next, an ODE model for how the state of the aircraft evolves in time must be developed. The process of developing accurate linearized dynamic models of an aircraft in flight is quite involved; this process may be started using simplified aerodynamic models and small-scale wind-tunnel tests, but generally must be subsequently refined using high-fidelity computational fluid dynamics simulations, large-scale wind-tunnel tests, and flight tests. Almost all models of aircraft dynamics today are based on **static stability derivatives**; that is, the forces and moments on the aircraft and the effectiveness of the control surfaces for any given state of the aircraft within its **flight envelope**[15] are determined assuming the aircraft is maintained in equilibrium in this configuration; that is, a *dynamic* model accounting for the unsteadiness of the flow itself is not accounted for with this approach. Certain dynamic maneuvers, such as the so-called **dynamic lift** available right before vortex separation and stall of a rapidly pitching airfoil moving at low speed (e.g., during spot landings with a flapping wing) are thus not accounted for well with such static models of the flow evolution. Nonetheless, a static model of the flow is in fact quite adequate for most fixed wing aircraft throughout most of their flight envelope.

[12] Even though these three rotations are usually not the actual rotations that brought the aircraft into this configuration!

[13] Note that *order matters* (that is, such rotations are **noncommutative**), as the latter steps rotate the aircraft about the body-fitted coördinates only after the former steps are complete. Various alternative rotation sequences may also be used to unambiguously identify the orientation of an aircraft, spacecraft, or other solid body; which rotation sequence is most convenient depends on the application.

[14] Note that the $3-2-1$ rotation sequence used here is often casually referred to as an **Euler rotation sequence** though, strictly speaking, an Euler rotation sequence repeats a rotation around one of the axes, a common choice being the $3-1-3$ **Euler rotation sequence** (in the present setting, yaw, then roll, then yaw again).

[15] A **flight envelope** is the set of states of an aircraft deemed safe for flight.

Linearized dynamic models governing the time evolution of the 12 variables describing an aircraft in cruise approximately decouple[16] into three essentially independent subsystems:

- the **lateral/directional dynamics** of the aircraft model, which relates the **yaw** (a.k.a. **heading angle**) ψ, the **roll** ϕ, the **yaw rate** $r = d\psi/dt$, the **body-axis roll rate** $p = d\phi/dt$, and the **sideslip angle** β,
- the **longitudinal dynamics** of the aircraft model, which relates the **pitch** θ, the **pitch rate** $q = d\theta/dt$, the **angle of attack** α, and the **airspeed** v_T, and
- the **navigation equations** $dX/dt = U$, $dY/dt = V$, $dZ/dt = W$; as mentioned previously, $\{U, V, W\}$ may, via simple geometry, be determined from the orientation angles $\{\phi, \theta, \psi\}$ together with knowledge of the local wind velocity and measurements of the relative wind past the aircraft, $\{v_T, \alpha, \beta\}$.

The navigation equations are straightforward to integrate in time (see §10) to track changes in the vehicle's absolute position in order to navigate; we thus focus our attention below on the more complex problems of the lateral/directional dynamics and the longitudinal dynamics of some representative aircraft.

Defining the deflection of the elevator, aileron, rudder, and throttle from their **trimmed** flight positions as δ_e, δ_a, δ_r, and δ_{th}, respectively, a representative linearized model of the lateral/directional dynamics of a large transport aircraft on approach to landing (see Minto, Chow, & Beseler 1989) is

$$\begin{matrix}\text{yaw:}\\ \text{roll:}\\ \text{yaw rate:}\\ \text{roll rate:}\\ \text{sideslip:}\end{matrix} \quad \frac{d}{dt}\begin{pmatrix}\psi\\ \phi\\ p\\ r\\ \beta\end{pmatrix} = \underbrace{\begin{pmatrix}0 & 0 & 1 & 0 & 0\\ 0 & 0 & .199 & 1 & 0\\ 0 & -.002 & -.194 & -.167 & .748\\ 0 & -.003 & .636 & -2.02 & -5.37\\ 0 & .136 & -.970 & .198 & -.148\end{pmatrix}}_{A_1}\begin{pmatrix}\psi\\ \phi\\ p\\ r\\ \beta\end{pmatrix} + \underbrace{\begin{pmatrix}0 & 0\\ 0 & 0\\ .053 & -.74\\ .865 & .904\\ .002 & .047\end{pmatrix}}_{B_1}\begin{pmatrix}\delta_a\\ \delta_r\end{pmatrix}. \tag{17.43}$$

A representative linearized model of the longitudinal dynamics of a large transport aircraft on approach to landing (see Stevens & Lewis 2003, Example 4.6-4) is

$$\begin{matrix}\text{airspeed:}\\ \text{AOA:}\\ \text{pitch:}\\ \text{pitch rate:}\end{matrix} \quad \frac{d}{dt}\begin{pmatrix}v_T\\ \alpha\\ \theta\\ q\end{pmatrix} = \underbrace{\begin{pmatrix}-.0386 & 19.0 & -32.1 & 0\\ -.00103 & -.633 & .0056 & 1\\ 0 & 0 & 0 & 1\\ -.00008 & -.76 & -.0008 & -.52\end{pmatrix}}_{A_2}\begin{pmatrix}v_T\\ \alpha\\ \theta\\ q\end{pmatrix} + \underbrace{\begin{pmatrix}10 & 0\\ -.00015 & 0\\ 0 & 0\\ .025 & -.011\end{pmatrix}}_{B_2}\begin{pmatrix}\delta_{th}\\ \delta_e\end{pmatrix}. \tag{17.44}$$

A representative linearized model of the longitudinal dynamics of an F-16 in cruise (300 knots at sea level; see Stevens & Lewis 2003, Example 4.4-1) is

$$\begin{matrix}\text{airspeed:}\\ \text{AOA:}\\ \text{pitch:}\\ \text{pitch rate:}\end{matrix} \quad \frac{d}{dt}\begin{pmatrix}v_T\\ \alpha\\ \theta\\ q\end{pmatrix} = \underbrace{\begin{pmatrix}-.0193 & 8.82 & -32.2 & -.575\\ -.000254 & -1.02 & 0 & .905\\ 0 & 0 & 0 & 1\\ 0 & .822 & 0 & -1.08\end{pmatrix}}_{A_3}\begin{pmatrix}v_T\\ \alpha\\ \theta\\ q\end{pmatrix} + \underbrace{\begin{pmatrix}.174\\ -.00215\\ 0\\ -.176\end{pmatrix}}_{B_3}\begin{pmatrix}\delta_e\end{pmatrix}. \tag{17.45}$$

The systems given above are written in the ubiquitous **state-space** form, $d\mathbf{x}/dt = A\mathbf{x} + B\mathbf{u}$, the characterization of which is studied in §20, and the control of which is considered in §21. We will also develop a variety of convenient ways to convert back and forth between first-order state-space forms and **single input, single output** (**SISO**) higher-order ODE forms; note that state space forms have the significant advantage of easily handling **multiple input, multiple output** (**MIMO**) systems. Further, state-space models reveal the inherent *coupling* present as the several states of a system (e.g., yaw, roll, yaw rate, roll rate, and sideslip) evolve in time, which often leads to significant practical insight regarding the physical system (see Exercise 17.1).

[16] That is, if the linearized dynamics of these 12 variables is written in the **state-space** form $d\mathbf{x}/dt = A\mathbf{x} + B\mathbf{u}$ with the components of $\mathbf{x}$ appropriately ordered, A may written in a 3×3 block upper-triangular form.

17.3 Laplace transform methods

The (one-sided) **Laplace transform** $F(s)$ of a **continuous-time** (**CT**) signal $f(t)$ is, in general, defined as

$$F(s) = \lim_{\varepsilon\to 0}\int_{-\varepsilon}^{\infty} f(t)e^{-st}dt \triangleq \int_{0^-}^{\infty} f(t)e^{-st}dt. \tag{17.46}$$

In the present text, we restrict all functions $f(t)$ to which we apply the Laplace transform to be at least **left-continuous** at the origin [that is, $f(-\varepsilon)\to f(0)$ as $\varepsilon\to 0$ with $\varepsilon>0$; see (B.69a)]. Accordingly, when the need for a CT unit impulse arises, we construct it as the large λ limit of the *one*-sided function[17] $\delta^{\lambda,m}(t)$ described in §5.3.4 for $m\geq 1$. In this restricted[18] setting, the Laplace transform may be defined (see LePage 1961) more simply as

$$F(s) = \int_{0}^{\infty} f(t)e^{-st}dt. \tag{17.47a}$$

Thus, given a left-continuous $f(t)$ for $t\geq 0$, we will define $F(s)$ via (17.47a). The **inverse Laplace transform** is given by

$$f(t) = \frac{1}{2\pi\mathrm{i}}\int_{\gamma-\mathrm{i}\infty}^{\gamma+\mathrm{i}\infty} F(s)e^{st}ds, \tag{17.47b}$$

where the real number γ is chosen such that the vertical line of the contour given in the above integral is to the right of all of the singularities[19] of $F(s)$ in the complex plane s.

Verification that (17.47b) in fact represents the inverse of the relationship expressed in (17.47a) is straightforward, by substituting (17.47a) into the RHS of (17.47b), substituting $s=\gamma+\mathrm{i}k$, applying Fubini's theorem (see footnote 2 on page 144), and noting that, for sufficiently large γ, $f(t)$ is indeed recovered:

$$\begin{aligned}\frac{1}{2\pi\mathrm{i}}\int_{\gamma-\mathrm{i}\infty}^{\gamma+\mathrm{i}\infty}\Big[\int_0^\infty f(t')\,e^{-st'}dt'\Big]e^{st}\,ds &= \lim_{K\to\infty}\frac{1}{2\pi\mathrm{i}}\int_{-K}^{K}\Big[\int_0^\infty f(t')\,e^{\gamma(t-t')}e^{\mathrm{i}k(t-t')}dt'\Big]\,\mathrm{i}\,dk\\ &= \lim_{K\to\infty}\int_0^\infty f(t')e^{\gamma(t-t')}\Big[\int_{-K}^{K}\frac{1}{2\pi}e^{\mathrm{i}k(t-t')}dk\Big]dt' = \int_0^\infty [f(t')\,e^{\gamma(t-t')}]\,\delta(t-t')\,dt' = f(t),\end{aligned}$$

where the definition of the Dirac delta given in (5.16a)-(5.17c) has been applied in the second line. The reason that the $e^{\gamma(t-t')}$ factor, for sufficiently large positive γ, is required by this formula is to ensure that the term $g(t')=[f(t')\,e^{\gamma(t-t')}]$ decays to zero exponentially as $t'\to\infty$, which allows us to swap the order of the integrals using Fubini's theorem and obtain the result that $\int_0^\infty g(t')\,\delta(t-t')\,dt' = g(t)$.

As discussed further below, the forward and inverse transforms expressed by (17.47) are immensely useful when solving differential equations (in CT). By (17.47a), knowing $f(t)$ for $t\geq 0$, one can define $F(s)$ on an appropriate contour. Conversely, by (17.47b), knowing $F(s)$ on an appropriate contour, one can determine $f(t)$ for $t\geq 0$. Before demonstrating further *why* such a transformation is useful, we first mention that, in practice, you don't actually need to compute the somewhat involved integrals given in (17.47) in order to use the Laplace transform effectively. Rather, it is sufficient to reference a table listing some Laplace transform pairs in a few special cases, as shown in Table 17.1a. Note also the following:

Fact 17.1 *The* **Laplace transform is linear***; that is,* **superposition holds***, and thus if the Laplace transforms of $a(t)$ and $b(t)$ are $A(s)$ and $B(s)$, then the Laplace transform of $c(t)=a(t)+b(t)$ is $C(s)=A(s)+B(s)$.*

Fact 17.2 *If the Laplace transform of $f(t)$ is $F(s)$, then the Laplace transform of the* **exponentially scaled** *function $g(t)=e^{-at}f(t)$ is $G(s)=\int_0^\infty f(t)e^{-(s+a)t}dt = F(s+a)$, and the Laplace transform of the* **delayed** *function $g(t)=f(t-d)$ is $G(s)=\int_0^\infty f(t-d)e^{-st}dt = \int_{-d^-}^\infty f(t)e^{-s(t+d)}dt = e^{-ds}F(s)$.*

[17] That is, rather than the small σ limit of the *two*-sided function $\delta^\sigma(t)$ described in §5.3.3.

[18] This "restriction" is said to be **technical**; that is, it narrows the precise mathematical setting in which the transform definition may be used, but in application does not limit the practical problems to which the transform may, when used correctly, be applied.

[19] That is, the contour of integration in (17.47b) is chosen to the right of all points $\overline{s}$ for which $|F(s)|\to\infty$ as $s\to\overline{s}$ in (17.47a).

$f(t)$ (for $t>0$)	$F(s)$
e^{at}	$1/(s-a)$
$t\,e^{at}$	$1/(s-a)^2$
$t^2\,e^{at}$	$2/(s-a)^3$
$t^p\,e^{at}$ (for integer $p\geq 0$)	$p!/(s-a)^{p+1}$
1 [i.e., $f(t)=h_0(t)$]	$1/s$
t	$1/s^2$
t^p (for integer $p\geq 0$)	$p!/s^{p+1}$
$\delta^{\lambda,m}(t)$	$\xrightarrow[\lambda\to\infty]{} 1$
$\delta^{\lambda,m}(t-d)$ (for $d\geq 0$)	$\xrightarrow[\lambda\to\infty]{} e^{-ds}$
$d[\delta^{\lambda,m}(t)]/dt \triangleq \delta'(t)$	$\xrightarrow[\lambda\to\infty]{} s$
$d^2[\delta^{\lambda,m}(t)]/dt^2 \triangleq \delta''(t)$	$\xrightarrow[\lambda\to\infty]{} s^2$
$e^{at}\cos(bt)$	$\dfrac{s-a}{(s-a)^2+b^2}$
$e^{at}\sin(bt)$	$\dfrac{b}{(s-a)^2+b^2}$
$\cos(bt)$	$s/(s^2+b^2)$
$\sin(bt)$	$b/(s^2+b^2)$
$\cosh(bt)$	$s/(s^2-b^2)$
$\sinh(bt)$	$b/(s^2-b^2)$

f_k (for $k=0,1,\ldots$)	$F(z)$
c^k	$z/(z-c)$
$k\,c^k$	$cz/(z-c)^2$
k^2c^k	$cz(z+c)/(z-c)^3$
k^3c^k	$\dfrac{cz(z^2+4cz+c^2)}{(z-c)^4}$
k^pc^k (for $p>0$)	$Li_{-p}(c/z)$
1 [i.e., $f_k=h_{0k}$]	$z/(z-1)$
k	$z/(z-1)^2$
k^p (for integer $p>0$)	$Li_{-p}(1/z)$
δ_{0k}	1
δ_{dk} (for integer d)	$1/z^d$
$c^k\cos(\theta k)$	$\dfrac{z[z-c\cos(\theta)]}{z^2-2cz\cos(\theta)+c^2}$
$c^k\sin(\theta k)$	$\dfrac{zc\sin(\theta)}{z^2-2cz\cos(\theta)+c^2}$
$c^k h_{1k}$	$c/(z-c)$
$(k-1)\,c^k h_{2k}$	$c^2/(z-c)^2$
$(k-2)(k-1)\,c^k h_{3k}$	$2c^3/(z-c)^3$
$(k-3)(k-2)(k-1)\,c^k h_{4k}$	$6c^4/(z-c)^4$
$(k-p)\cdots(k-1)\,c^k h_{p+1,k}$	$p!\,c^{p+1}/(z-c)^{p+1}$

Table 17.1: Tables of (left) some Laplace transform pairs, as considered in §17.3, and (right) some Z transform pairs, as considered in §17.4. Note that the (left-continuous) CT functions $f(t)=0$ for $t\leq 0$, and that the DT functions $f_k=0$ for integer $k<0$. The **CT unit impulse** in this work is taken as the large λ limit (under the integral sign!) of the *one*-sided function $\delta^{\lambda,m}(t)$ (see §5.3.4) for some integer $m\geq 1$. The **polylogarithm** $Li_n(z)$ is defined in (B.68), and the **DT Heaviside step function** h_{dk} is defined in (B.69b).

Note in Table 17.1a that the Laplace transform of the **delay function**, $f(t)=\delta^{\lambda,m}(t-d)$ in the limit of large λ for $d>0$, is $F(s)=e^{-ds}$; this is not a rational function[20] of s, which turns out to be inconvenient. The following **Padé approximation** of the Laplace transform of a delay, valid for small values of $|ds|$, is thus convenient to use in its stead

$$e^{-ds}\approx F_n(s)\triangleq\frac{\sum_{k=0}^n(-1)^k c_k(ds)^k}{\sum_{k=0}^n c_k(ds)^k},\qquad c_k=\frac{(2n-k)!\,n!}{(2n)!\,k!\,(n-k)!}. \tag{17.48}$$

The formula for the coefficients in the above approximation may be verified by considering the expression

$$[\textstyle\sum_{k=0}^n a_k(ds)^k]e^{-ds}\approx[\sum_{k=0}^n b_k(ds)^k],$$

[20] A **rational function** of s is a polynomial in s divided by a polynomial in s.

inserting the Taylor-series expansion for e^{-ds}, expanding, and matching as many coefficients of like powers of (ds) as possible. The resulting rational approximations of the delay function e^{-ds}, for $n = 1$, 2, and 4, are

$$F_1(s) = \frac{1 - ds/2}{1 + ds/2}, \quad F_2(s) = \frac{1 - ds/2 + (ds)^2/12}{1 + ds/2 + (ds)^2/12}, \quad F_4(s) = \frac{1 - ds/2 + 3(ds)^2/28 - (ds)^3/84 + (ds)^4/1680}{1 + ds/2 + 3(ds)^2/28 + (ds)^3/84 + (ds)^4/1680}.$$

17.3.1 The Laplace Transform of derivatives and integrals of functions

Assume $f(t)$ is smooth and bounded and define $f^{(1)}(t) = df(t)/dt = f'(t)$. Then, by integration by parts, the Laplace transform of $f^{(1)}(t)$ is given by

$$\begin{aligned} F^{(1)}(s) &= \int_0^\infty f^{(1)}(t) e^{-st} dt = \lim_{a\to\infty} \int_0^a f^{(1)}(t) e^{-st} dt \\ &= \lim_{a\to\infty} \left[e^{-sa} f(a) - f(0) + s \int_0^a e^{-st} f(t) dt \right] = sF(s) - f(0) \end{aligned} \tag{17.49a}$$

for $\Re(s) > 0$. Similarly, if $f^{(2)}(t) = d^2 f(t)/dt^2 = f''(t)$ and $f^{(n)}(t) = d^n f(t)/dt^n$, then

$$F^{(2)}(s) = \int_0^a f^{(2)}(t) e^{-st} dt = \ldots = s^2 F(s) - sf(0) - f^{(1)}(0), \tag{17.49b}$$

$$F^{(n)}(s) = \int_0^a f^{(n)}(t) e^{-st} dt = \ldots = s^n F(s) - s^{n-1} f(0) - s^{n-2} f^{(1)}(0) - \ldots - f^{(n-1)}(0). \tag{17.49c}$$

Thus, if $f^{(1)}(t) = df(t)/dt$, then $F^{(1)}(s) = sF(s) - f(0)$. Conversely, by integration, it therefore follows that, if $f(t) = \int_0^t f^{(1)}(t')dt'$, and thus $f(0) = 0$, then $F(s) = \frac{1}{s} F^{(1)}(s)$. We thus arrive at the most useful interpretation of the s variable:

Fact 17.3 *Multiplication of the Laplace transform of a CT signal by $1/s$ corresponds to integration of this signal (from $t = 0$) in the time domain. Similarly, multiplication by $1/s^2$ corresponds to double integration, etc.*

Note that, with $f^{(1)}(t) = df(t)/dt$,

$$\lim_{s\to 0} \left[\int_0^\infty f^{(1)}(t) e^{-st} dt \right] = \int_0^\infty \lim_{s\to 0} \left[f^{(1)}(t) e^{-st} \right] dt = \int_0^\infty f^{(1)}(t) dt = f(\infty) - f(0).$$

It follows by taking the limit of (17.49a) as $s \to 0$ that

Fact 17.4 (The CT final value theorem) $\lim_{s\to 0} sF(s) = \lim_{t\to\infty} f(t)$.

If we now consider the limit as $s \to \infty$ instead of $s \to 0$, we have to be a bit more careful. In the case in which $f(t)$ is a scalar $c = \lim_{\varepsilon\to 0} f(\varepsilon) - f(0)$ times a (left-continuous) unit step [see (B.69a)] plus other terms which are continuous near the origin, by Fact 5.6 we define $f^{(1)}(t)$ (kept under the integral sign!) as the scalar c times the Dirac delta[21] plus other terms which are bounded near the origin. From the sifting property of the Dirac delta [see (5.21c)], it follows by taking the limit of (17.49a) as $s \to \infty$ that $c = \lim_{s\to\infty} sF(s) - f(0)$, and thus

Fact 17.5 (The CT initial value theorem) $\lim_{s\to\infty} sF(s) = \lim_{t\to 0^+} f(t)$.

[21] Recall again that the Dirac delta is defined (under the integral sign!) via the effect, in the large λ limit, of the one-sided function $\delta^{\lambda,m}(t)$, as developed in §5.3.4, for some integer $m \geq 1$.

17.3.2 Using the Laplace Transform to solve unforced linear differential equations

Consider the unforced linear constant-coefficient second-order differential equation given by

$$f''(t)+a_1f'(t)+a_0f(t)=0 \quad \text{with} \quad f(0),\ f'(0) \text{ given.} \tag{17.50}$$

Taking the Laplace transform of this equation and applying the above relations gives

$$\int_0^\infty \{f''(t)+a_1f'(t)+a_0f(t)=0\}e^{-st}dt \quad \Rightarrow \quad [s^2F(s)-sf(0)-f'(0)]+a_1[sF(s)-f(0)]+a_0[F(s)]=0$$

$$\Rightarrow \quad F(s)=\frac{c_1s+c_0}{s^2+a_1s+a_0} \quad \text{where } c_1=f(0),\ c_0=f'(0)+a_1f(0).$$

Defining the roots of the denominator $p_\pm=\left(-a_1\pm\sqrt{a_1^2-4a_0}\right)/2$, known as the **poles** of this second-order equation, and performing a **partial fraction expansion** (see §B.6.3), it follows that

$$F(s)=\frac{c_1s+c_0}{(s-p_+)(s-p_-)}=\frac{d_+}{s-p_+}+\frac{d_-}{s-p_-} \quad \Rightarrow \quad \left\{\begin{array}{r} d_++d_-=c_1 \\ -d_+p_--d_-p_+=c_0 \end{array}\right\} \quad \Rightarrow \quad \left\{\begin{array}{l} d_+=\dfrac{c_1p_++c_0}{p_+-p_-}, \\ d_-=\dfrac{c_1p_-+c_0}{p_--p_+}. \end{array}\right.$$

Thus, by Table 17.1a and the linearity of the Laplace transform (Fact 17.1, from which the superposition principle follows immediately), we deduce that

$$f(t)=d_+e^{p_+t}+d_-e^{p_-t}, \tag{17.51}$$

thus solving the original differential equation (17.50). It is seen that, if the real parts of the poles $p_\pm$ are negative, the magnitude of the solution decays with time, whereas if the real parts of $p_\pm$ are positive, the magnitude of the solution grows with time. Also note that, if the coefficients $\{a_0,a_1\}$ and initial conditions $\{f(0),f'(0)\}$ defining the system in (17.50) are real, then the roots $p_\pm$ are either real or a complex conjugate pair, and thus the solution $f(t)$ given by (17.51) is real even though the roots $p_\pm$ might be complex.

Higher-order unforced constant-coefficient CT linear differential equations of the form

$$f^{(n)}(t)+a_{n-1}f^{(n-1)}(t)+\ldots+a_1f'(t)+a_0f(t)=0,$$

may be solved in an analogous manner, leveraging partial fraction expansions (again, see §B.6.3) to split up $F(s)$ into simple terms whose inverse Laplace transforms may be found in Table 17.1a. In such cases, as in the second-order case discussed above, the speed of oscillation and the rate of decay or growth of the various components of the solution are characterized solely by the poles [that is, the roots of a polynomial (in s) with coefficients a_i], whereas *how much* of each of these components this solution actually contains, in addition to their relative phase, is a function of the initial conditions on $f(t)$ and its derivatives.

17.3.3 Continuous-time (CT) transfer functions

Now consider the forced, CT, **linear time invariant** (**LTI**; that is, constant-coefficient), **single input, single output** (**SISO**) second-order ODE for $y(t)$ (the **output**) given by

$$y''(t)+a_1y'(t)+a_0y(t)=b_0u(t), \tag{17.52}$$

where $u(t)$ (the **input**) is specified, assuming $y(t)$ and $y'(t)$ are zero at $t=0$. Taking the Laplace transform now gives

$$\int_0^\infty \{y''(t)+a_1y'(t)+a_0y(t)=b_0u(t)\}e^{-st}dt \quad \Rightarrow \quad [s^2+a_1s+a_0]Y(s)=b_0U(s)$$

$$\Rightarrow \quad G(s)\triangleq\frac{Y(s)}{U(s)}=\frac{b_0}{s^2+a_1s+a_0}=\frac{b_0}{(s-p_+)(s-p_-)}, \tag{17.53}$$

Algorithm 17.1: Compute response of a CT system in transfer function form to a simple input.

View
Test

```
function [r,y,t]=ResponseTF(gs,fs,type,g)
% Using its partial fraction expansion, compute the response Y(s)=T(s)*R(s) of a
% CT SISO linear system T(s)=gs(s)/fs(s) to an impulse (type=0), step (type=1),
% or quadratic (type=2) input.  The derived type g groups together convenient
% plotting parameters: g.T is the interval over which response is plotted,
% g.N is the number of timesteps, and {g.styleu,g.styley} are the linestyles used.
numR=Fac(type-1); denR=1; for i=1:type, denR=[denR 0]; end,  gs=gs/fs(1);
[rp,rd,rk]=PartialFractionExpansion(numR,denR);              fs=fs/fs(1);
[yp,yd,yk]=PartialFractionExpansion(PolyConv(numR,gs),PolyConv(denR,fs));
h=g.T/g.N; t=[0:g.N]*h; for k=1:g.N+1
  if type>0, r(k)=real(sum(rd.*(t(k).^(rk-1)).*exp(rp*t(k))));  else, r(k)=0; end
             y(k)=real(sum(yd.*(t(k).^(yk-1)).*exp(yp*t(k))));
end
if nargout==0,
  plot(t,y,g.styley), axis tight, if type>0, hold on; plot(t,r,g.styler), hold off; end
end
end % function ResponseTF
```

where, again, the poles $p_\pm = \left(-a_1 \pm \sqrt{a_1^2 - 4a_0}\right)/2$. The quantity $G(s)$ given above is known as the **transfer function** of the linear system (17.52).

Higher-order forced SISO constant-coefficient CT linear systems of the form

$$y^{(n)}(t) + a_{n-1}y^{(n-1)}(t) + \ldots + a_1 y'(t) + a_0 y(t) = b_m u^{(m)}(t) + b_{m-1}u^{(m-1)}(t) + \ldots + b_1 u'(t) + b_0 u(t), \quad (17.54)$$

with $b_m \neq 0$ [and, normally, $m \leq n$; see §17.3.3.1], may be manipulated in an analogous manner, leading to a transfer function of the rational form

$$G(s) = \frac{Y(s)}{U(s)} = \frac{b_m s^m + b_{m-1}s^{m-1} + \ldots + b_1 s + b_0}{s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0} = K\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}. \quad (17.55)$$

The m roots of the numerator, z_i, are referred to as the **zeros** of the system, the n roots of the denomenator, p_i, are referred to as the **poles** of the system, and the coefficient K is referred to as the **gain** of the system.

Note that a differential equation governing a CT system, taken on its own, simply relates linear combinations of two or more variables describing the system and their derivatives; such an equation does *not* itself indicate one variable as a "cause" and another as an "effect" in a cause-effect relationship. However, the definition of a transfer function inherently identifies, or defines, a **cause-effect** or **input-output** relationship; in the examples discussed above, $u(t)$ is identified as the input, and $y(t)$ is identified as the output. This further distinction between input and output is signficant. Almost all systems encountered are **causal**, meaning that any variable identified as an "output" only responds to the current and past "inputs", but not to future inputs. The assumption of causality is essentially ubiquitous, and it is often implied without being explicitly stated. Indeed, the present text will always assume that any CT system under consideration is causal unless specifically stated otherwise.

Once a (causal) CT linear system's transfer function $G(s)$ is known, its response to simple inputs is easy to compute. Noting Table 17.1a:

- if $u(t)$ is a **unit impulse** [$u(t) = \delta^{\lambda,m}(t)$ for large λ and integer $m \geq 1$; see §5.3.4], then $U(s) \approx 1$;
- if $u(t)$ is a **unit step** [$u(t) = h_0(t)$; see (B.69)], then $U(s) = 1/s$;
- if $u(t)$ is a **unit ramp** [$u(t) = t$ for $t > 0$], then $U(s) = 1/s^2$, etc.

In such cases, $Y(s) = G(s)U(s)$ is easy to compute, and thus $y(t)$ may be found by partial fraction expansion and subsequent inverse Laplace transform, as implemented in Algorithm 17.1. As in the unforced case discussed in §17.3.2, the speed of oscillation and the rate of decay or growth of the various components of the

system's response to a simple input is characterized solely by the poles of the system, whereas *how much* of each of these components this response actually contains, in addition to their relative phase, is a function of its zeros and gain.

It is important to keep clear the distinction between the Laplace transform (a.k.a. transfer function) of a *system*, such as $G(s)$ above, and the Laplace transform of a *signal*, such as $Y(s)$ above. To make clear the connection between them, note in the special case that the input to the system happens to be a unit impulse $u(t) = \delta^{\lambda,m}(t)$ for large λ and integer $m \geq 1$, it follows that $U(s) \approx 1$ and thus $Y(s) \approx G(s)$. In other words,

Fact 17.6 *The transfer function of a CT linear system is the Laplace transform of its impulse response.*

It follows from the relation $Y(s) = G(s)U(s)$, expanding $Y(s)$, $G(s)$, and $U(s)$ with the Laplace transform formula (17.47a), noting that the impulse response $g(t) = 0$ for $t < 0^-$ (that is, that the system is causal, as discussed above), and following an analogous derivation as that leading to (5.36b), that

$$\int_0^\infty \Big[y(t)\Big] e^{-st} dt = \int_0^\infty g(t) e^{-st} dt \int_0^\infty u(t') e^{-st'} dt' = \int_0^\infty u(t') \left(\int_{-(t')}^\infty g(t) e^{-st} dt \right) e^{-st'} dt'$$
$$= \int_0^\infty u(t') \left(\int_0^\infty g(t-t') e^{-s(t-t')} dt \right) e^{-st'} dt' = \int_0^\infty \left[\int_0^t u(t') g(t-t') \, dt' \right] e^{-st} dt,$$

from which we deduce that, for $t \geq 0$,

$$y(t) = \int_0^t u(t') g(t-t') dt'; \tag{17.56}$$

note in particular that $y(t) \approx g(t)$ when $u(t) = \delta^{\lambda,m}(t)$ for large λ. Thus, as similarly noted for the Fourier transform in Fact 5.4,

Fact 17.7 *The product $Y(s) = G(s)U(s)$ in Laplace transform space corresponds to a convolution integral [of the input $u(t)$ with the impulse response $g(t)$] in the untransformed space.*

Products are generally much easier to work with than convolution integrals, thus highlighting the utility of the Laplace transform when solving constant-coefficient CT linear systems.

17.3.3.1 Proper, strictly proper, and improper CT systems

We now revisit the differential equation in (17.54) and its corresponding transfer function in (17.55), where the degree of the polynomial in the numerator is m, and the degree of the polynomial in the denominator is n. Define the **relative degree** of such a transfer function as $n_r = n - m$. In CT, such systems are said to be **improper** if $n_r < 0$. In §18 we will further distinguish the CT systems of interest as "plants" $G(s)$ and "controllers" $D(s)$. Most real plants $G(s)$ are **strictly proper**, with $n_r > 0$ [or at least **proper**, with $n_r \geq 0$], as most plants have some sort of **inertia**, **capacitance**, or **storage** which attenuates [or at least bounds] their response at high frequencies, as characterized precisely by their Bode plots (see §17.5.1). Further, to avoid amplifying high-frequency measurement noise which might be present as the measured signal is fed back to the actuator via control feedback (see, e.g., §18.3.3), it is strongly advised to use a strictly proper [or, at least, proper], controller $D(s)$. Thus, except for a brief mention in §18.3.1, we will focus our attention in this study almost exclusively on the case with $n_r \geq 0$. Note also that a transfer function with $n_r = 0$, which is proper but not strictly proper, is occasionally said to be **semi-proper**.

Example 17.12 The step response of second-order CT linear systems

We now focus further specifically on the forced second-order case (17.52), with $b_0 = a_0$, when forced by a unit step $u(t) = h_0(t)$; that is,

$$G(s) = \frac{a_0}{s^2 + a_1 s + a_0} = \frac{a_0}{(s - p_+)(s - p_-)} \quad \text{and} \quad U(s) = \frac{1}{s}.$$

If the poles $p_\pm = \left(-a_1 \pm \sqrt{a_1^2 - 4a_0}\right)/2$ are complex with negative real part, the solution of this system may be written in terms of sines and cosines modulated by a decaying exponential, as implied by (17.51). To illustrate this more clearly, assume first that $a_0 > 0$ and $0 \le a_1 < 2\sqrt{a_0}$. Defining in this case the **undamped natural frequency** ω_n and the **damping ratio** ζ such that

$$a_0 = \omega_n^2 \quad \text{and} \quad a_1 = 2\zeta\omega_n,$$

noting that $\omega_n > 0$ and $0 \le \zeta < 1$, it is seen (see Figure 17.11) that

$$p_\pm = -\sigma \pm \mathrm{i}\omega_d, \quad \text{where} \quad \sigma = \zeta\omega_n = a_1/2 \quad \text{and} \quad \omega_d = \omega_n\sqrt{1-\zeta^2} = \sqrt{a_0 - a_1^2/4},$$

in which case, via partial fraction expansion, we may write

$$Y(s) = G(s)U(s) = \frac{\omega_n^2}{(s-p_+)(s-p_-)} \cdot \frac{1}{s} = \frac{d_+}{s-p_+} + \frac{d_-}{s-p_-} + \frac{d_0}{s}, \quad \begin{cases} d_+ = -\mathrm{i}\omega_n^2/(2\omega_d p_+), \\ d_- = \mathrm{i}\omega_n^2/(2\omega_d p_-) = \overline{d_+}, \\ d_0 = 1. \end{cases}$$

Thus, by Table 17.1a, noting that $y(t) = 0$ for $t \le 0$, the closed-form solution of $y(t)$ for $t > 0$ is given by

$$y(t) = d_+ e^{p_+ t} + d_- e^{p_- t} + d_0 = e^{-\sigma t}\left[d_c \cos(\omega_d t) + d_s \sin(\omega_d t)\right] + 1, \quad \begin{cases} d_c = d_+ + d_- = -1, \\ d_s = \mathrm{i}(d_+ - d_-) = -\zeta/\sqrt{1-\zeta^2}, \end{cases}$$

as plotted in Figure 17.12. Since the system $G(s)$ considered in this example is real, the complex poles p_+ and p_- come as a conjugate pair. In addition, as consequence of the fact that the input $u(t)$ to this system is also real, the coefficients d_+ and d_- also work out to be a complex conjugate pair, and thus d_c and d_s, and $y(t)$ itself, are real. Again, note that the speed of oscillation, ω_d, and the rate of decay, σ, of this response are a function of the location of the poles of the transfer function $p_\pm = -\sigma \pm \mathrm{i}\omega_d$.

As indicated in Figure 17.12, there are three commonly-used characterizations of the step response: the **rise time** t_r, defined as the time it takes the response to increase from 0.1 to 0.9 of the steady state value of the step response, the **settling time** t_s, defined as the total time it takes the response to settle to within ± 5 percent of the steady state value of the step response, and the **overshoot** M_p, defined as the maximum percentage by which the output of the system exceeds its steady-state value when the system responds to a step input. By performing least-squares fits (see §4.7.2) of the rise time, settling time, and overshoot of several such step responses of second-order systems (as plotted in Figure 17.12) as a function of ω_n, σ, and ζ, respectively, the following handy approximate relations are readily determined:

$$t_r \approx 1.8/\omega_n, \quad t_s \approx 4.6/\sigma, \quad M_p \approx e^{-\pi\zeta/\sqrt{1-\zeta^2}}.$$

If the maximum values of t_r, t_s, and/or M_p are specified, then, the following **approximate design guides** for the admissible pole locations of a second-order system follow immediately:

$$\omega_n \gtrsim 1.8/t_r, \quad \sigma \gtrsim 4.6/t_s, \quad \begin{cases} \zeta \gtrsim 0.5 & \text{for} \quad M_p \le 15\% \\ \zeta \gtrsim 0.7 & \text{for} \quad M_p \le 5\%. \end{cases} \tag{17.57}$$

The natural frequency ω_n is often referred to as the **speed** of such a system, as it is inversely proportional to the rise time. Typical approximate design guides of this sort are illustrated graphically in Figure 17.13. The response of many higher-order systems is dominated by the response due to a pair of **dominant second-order poles** [i.e., the slowest (smallest ω_n) poles of the system that are not approximately cancelled by nearby zeros]. Thus, these approximate design guides are often handy even if the system is not second order. Recall also that the response of higher-order systems to simple inputs is easily plotted using Algorithm 17.2.

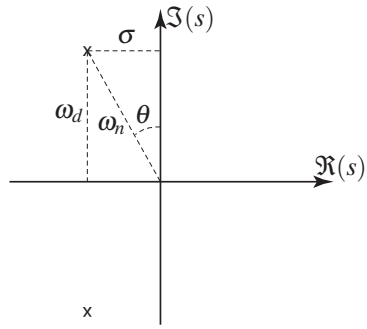


Figure 17.11: The poles $p_\pm$ of the system $y''(t)+2\zeta\omega_n y'(t)+\omega_n^2 y(t)=\omega_n^2 u(t)$ in the complex plane s in terms of ω_n, $\theta=\sin^{-1}\zeta$, $\sigma=\zeta\omega_n$, and $\omega_d=\omega_n\sqrt{1-\zeta^2}$. The response $y(t)$ to a step input $u(t)$ is plotted in Figure 17.12; note that ω_d sets the speed of oscillation and σ sets the exponential rate of decay.

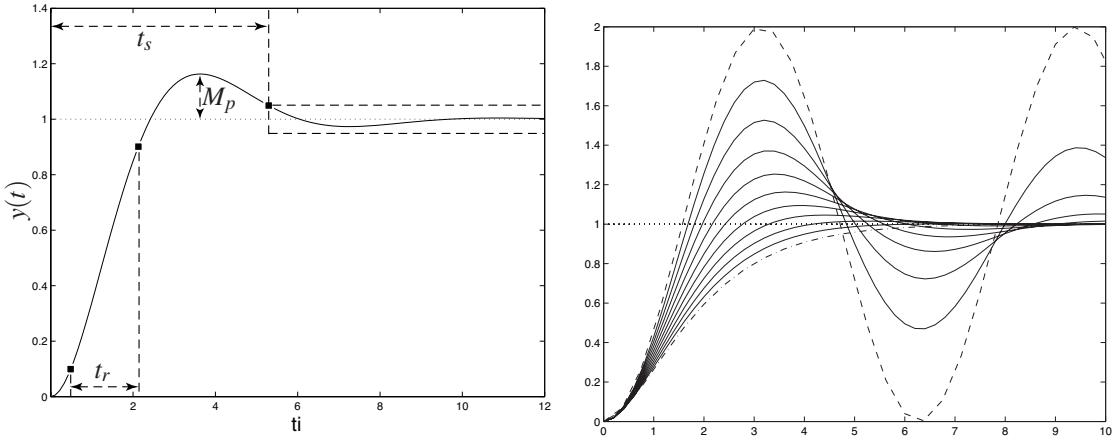


Figure 17.12: The unit step response of the system $y''(t)+2\zeta\omega_n y'(t)+\omega_n^2 y(t)=\omega_n^2 u(t)$, for $\omega_n=1$ and (left) taking $\zeta=0.5$, with the rise time, settling time, and overshoot indicated, and (right) taking $\zeta=0$ (dashed), $\zeta=0.1$, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, and 0.9 (solid), and $\zeta=1$ (dot-dashed).

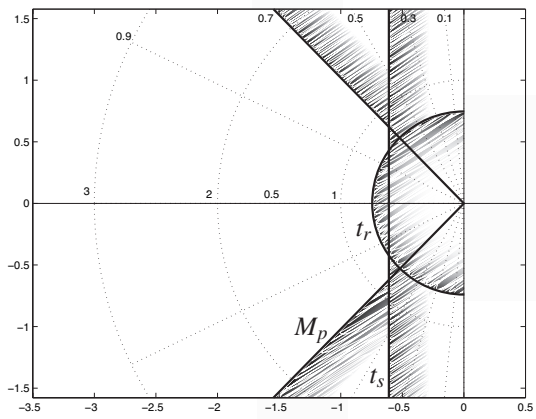


Figure 17.13: Approximate constraints, or **design guides**, on the admissible pole locations of a CT second-order system (or a higher-order system whose response is dominated by a pair of second-order poles) in the complex plane s in order to not exceed specified constraints on the rise time, settling time, and overshoot of the system's step response (see Figure 17.12), as specified in (17.57).

17.4 Z transform methods

The Z **transform** (a.k.a. the **unilateral** Z **transform**), $F(z)$, of a **discrete-time** (**DT**) signal f_k for $k = 0,1,2,\ldots$ is defined by

$$F(z) = \sum_{k=0}^{\infty} f_k z^{-k}. \tag{17.58a}$$

Given f_k for $k = 0,1,2,\ldots$, we define $F(z)$ via (17.58a). The **inverse** Z **transform** is given by

$$f_k = \frac{1}{2\pi \mathrm{i}} \oint_\Gamma F(z) z^{k-1}\, dz, \tag{17.58b}$$

where the complex contour Γ is a circle around the origin in complex plane z that is chosen to be of sufficiently small radius that it does not contain any singularities[22] of $F(z)$ in the complex plane z.

Verification that (17.58b) in fact represents the inverse of the relationship expressed in (17.58a) is straightforward, by substituting (17.58a) into the RHS of (17.58b) and noting (5.5a) and thus that, for a contour Γ given by $z = Re^{\mathrm{i}\theta}$ for $\theta = (-\pi,\pi)$ with sufficiently small, fixed R (and thus $dz = \mathrm{i}Re^{\mathrm{i}\theta}\, d\theta$), f_k is indeed recovered:

$$\begin{aligned}\frac{1}{2\pi\mathrm{i}} \oint_\Gamma \Big[\sum_{k'=0}^{\infty} f_{k'} z^{-k'}\Big] z^{k-1}\, dz &= \frac{1}{2\pi\mathrm{i}} \int_{-\pi}^{\pi} \sum_{k'=0}^{\infty} f_{k'} R^{-k'} e^{-\mathrm{i}\theta k'} R^{k-1} e^{\mathrm{i}\theta(k-1)} R\mathrm{i} e^{\mathrm{i}\theta}\, d\theta \\ &= \sum_{k'=0}^{\infty} f_{k'} R^{k-k'} \Big[\frac{1}{2\pi} \int_{-\pi}^{\pi} e^{\mathrm{i}\theta(k-k')}\, d\theta\Big] = \sum_{k'=0}^{\infty} f_{k'} R^{k-k'} \delta_{k,k'} = f_k,\end{aligned}$$

where the relation (5.5a) has been applied in the second line. The reason that the $R^{k-k'}$ factor, for sufficiently small positive R, is required by this formula is to ensure that the magnitude of the integrand decays to zero exponentially as $k' \to \infty$, which allows us to swap the order of the integral and the sum using Fubini's theorem.

As shown below, the forward and inverse transforms expressed by (17.58) are immensely useful when solving difference equations (in DT). By (17.58a), knowing f_k for $k = 0,1,2,\ldots$, one can define $F(z)$ on an appropriate contour. Conversely, by (17.58b), knowing $F(z)$ on an appropriate contour, one can determine f_k for $k = 0,1,2,\ldots$ As in §17.3, before demonstrating further *why* such a transformation is useful, we first mention that, in practice, you don't actually need to compute the somewhat involved integrals given in (17.58) in order to use the Z transform effectively. Rather, it is sufficient to reference a table listing some Z transform pairs in a few special cases, as shown in Table 17.1b. Note also the following:

Fact 17.8 *The* Z **transform is linear***; that is,* **superposition holds***, and thus if the* Z *transforms of the sequences* a_k *and* b_k *are* $A(z)$ *and* $B(z)$*, then the* Z *transform of the sequence* $c_k = a_k + b_k$ *is* $C(z) = A(z) + B(z)$.

Fact 17.9 *If the* Z *transform of the sequence* f_k *is* $F(z)$*, then the* Z *transform of the* **scaled** *sequence* $g_k = b^k f_k$ *is* $G(z) = \sum_{k=0}^{\infty} f_k (z/b)^{-k} = F(z/b)$*, and the* Z *transform of the* **delayed** *sequence* $g_k = f_{k-d}$ *is* $G(z) = \sum_{k=0}^{\infty} f_{k-d} z^{-k} = F(z)/z^d$.

17.4.1 The Z Transform of translated sequences

Define $f_k^{[1]} = f_{k+1}$ for $k = 0,1,2,\ldots$. Then the Z transform of $f_k^{[1]}$ is given by

$$F^{[1]}(z) = \sum_{k=0}^{\infty} f_k^{[1]} z^{-k} = z \sum_{k=0}^{\infty} f_{k+1} z^{-(k+1)} = z \sum_{k=1}^{\infty} f_k z^{-k} = zF(z) - zf_0. \tag{17.59}$$

[22] That is, the circular contour of integration in the (17.58b) is chosen to be of sufficiently small radius that it does not contain any points $\overline{z}$ for which $|F(z)| \to \infty$ as $z \to \overline{z}$ in (17.58a).

Similarly, if $f_k^{[2]} = f_{k+2}$ and $f_k^{[n]} = f_{k+n}$, then

$$F^{[2]}(z) = \sum_{k=0}^{\infty} f_k^{[2]} z^{-k} = z^2 F(z) - z^2 f_0 - z f_1, \qquad F^{[n]}(z) = \sum_{k=0}^{\infty} f_k^{[n]} z^{-k} = z^n F(z) - z^n f_0 - z^{n-1} f_1 - \ldots - z f_{n-1}.$$

Thus, if $f_k^{[1]} = f_{k+1}$, then $F^{[1]}(z) = zF(z) - zf_0$. Conversely, it follows that, if $f_{k+1} = f_k^{[1]}$ for $k = 0, 1, 2, \ldots$ with $f_0 = 0$, then $F(z) = \frac{1}{z} F^{[1]}(z)$. We thus arrive at the most useful interpretation of the z variable:

Fact 17.10 *Multiplication of the Z transform of a DT signal by $1/z$ corresponds to a delay of this signal in the time domain by one timestep. Similarly, $1/z^2$ corresponds to a delay of two timesteps, etc.*

Defining a new sequence $g_k = f_{k+1} - f_k$ for all k and taking the Z transform of g_k, applying (17.59), gives

$$G(z) = \sum_{k=0}^{\infty} g_k z^{-k} \quad \Rightarrow \quad [zF(z) - zf_0] - F(z) = \lim_{a \to \infty} \sum_{k=0}^{a-1} (f_{k+1} - f_k) z^{-k}.$$

Taking the limit of this expression as $z \to 1$, noting that the limit on the RHS approaches $f_\infty - f_0$ if the limit indicated in the above equation is bounded, thus gives

Fact 17.11 (The DT final value theorem) *If $\lim_{k \to \infty} f_k$ is bounded, then $\lim_{z \to 1} (z-1) F(z) = \lim_{k \to \infty} f_k$.*

On the other hand, it follows directly from the $z \to \infty$ limit of (17.58a) that

Fact 17.12 (The DT initial value theorem) $\lim_{z \to \infty} F(z) = f_0$.

17.4.2 Using the Z Transform to solve unforced linear difference equations

Now consider the unforced linear constant-coefficient second-order difference equation given by

$$f_{k+2} + a_1 f_{k+1} + a_0 f_k = 0 \quad \text{with} \quad f_0,\ f_1 \text{ given.} \tag{17.60}$$

Taking the Z transform of this equation and applying the above relations gives

$$\sum_{k=0}^{\infty} \{f_{k+2} + a_1 f_{k+1} + a_0 f_k = 0\} z^{-k} \quad \Rightarrow \quad [z^2 F(z) - z^2 f_0 - z f_1] + a_1 [zF(z) - z f_0] + a_0 [F(z)] = 0$$

$$\Rightarrow \quad F(z) = \frac{c_2 z^2 + c_1 z}{z^2 + a_1 z + a_0} \quad \text{where} \ \ c_2 = f_0, \ \ c_1 = f_1 + a_1 f_0.$$

Defining $p_\pm = (-a_1 \pm \sqrt{a_1^2 - 4a_0})/2$ and performing a partial fraction expansion, it follows that

$$F(z) = \frac{c_2 z^2 + c_1 z}{(z - p_+)(z - p_-)} = \frac{d_+ z}{z - p_+} + \frac{d_- z}{z - p_-} \quad \Rightarrow \quad \left\{ \begin{array}{r} d_+ + d_- = c_2 \\ -d_+ p_- - d_- p_+ = c_1 \end{array} \right\} \quad \Rightarrow \quad \left\{ \begin{array}{l} d_+ = \dfrac{c_2 p_+ + c_1}{p_+ - p_-} \\ d_- = \dfrac{c_2 p_- + c_1}{p_- - p_+}. \end{array} \right.$$

Thus, by Table 17.1b and the linearity of the Z transform (Fact 17.8), we deduce that

$$f_k = d_+ p_+^k + d_- p_-^k, \tag{17.61}$$

thus solving the original difference equation (17.60). It is seen that, if the magnitudes of $p_\pm$ are less than one, the magnitude of the solution decays with time, whereas if the magnitudes of $p_\pm$ are greater than one, the magnitude of the solution grows with time. A difference equation of precisely the form given in (17.60) leads to the well-known **Fibonacci's sequence**, as considered in Exercise 17.3.

Higher-order linear difference equations may be solved in an identical manner, leveraging partial fraction expansions to split up $F(z)$ into simple terms whose Z transforms may be found in Table 17.1b.

17.4.3 Discrete-time (DT) transfer functions

Now consider the forced linear constant-coefficient second-order difference equation, a.k.a. DT SISO LTI system, for u_k (the output[23]) given by

$$u_{k+2} + a_1 u_{k+1} + a_0 u_k = b_0 e_k, \tag{17.62}$$

where e_k (the input) is specified, assuming u_k and e_k are zero for $k < 0$. Taking the Z transform now gives

$$\sum_0^\infty \{u_{k+2} + a_1 u_{k+1} + a_0 u_k = b_0 e_k\} z^{-k} \quad \Rightarrow \quad [z^2 + a_1 z + a_0] U(z) = b_0 E(z)$$

$$\Rightarrow \quad D(z) \triangleq \frac{U(z)}{E(z)} = \frac{b_0}{z^2 + a_1 z + a_0} = \frac{b_0}{(z - p_+)(z - p_-)}, \tag{17.63}$$

where, again, the poles $p_\pm = \left(-a_1 \pm \sqrt{a_1^2 - 4a_0}\right)/2$. The quantity $D(z)$ is known as the transfer function of the linear system (17.62). Higher-order forced SISO constant-coefficient DT linear systems of the form

$$u_{k+n} + a_{n-1} u_{k+n-1} + \ldots + a_1 u_{k+1} + a_0 u_k = b_m e_{k+m} + b_{m-1} e_{k+m-1} + \ldots + b_1 e_{k+1} + b_0 e_k \tag{17.64}$$

with $b_m \neq 0$ [and, normally, $n \geq m$; see §17.4.3.2], may be manipulated in an analogous manner, leading to a transfer function of the form

$$D(z) = \frac{U(z)}{E(z)} = \frac{b_m z^m + b_{m-1} z^{m-1} + \ldots + b_1 z + b_0}{z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0} = K \frac{(z - z_1)(z - z_2) \cdots (z - z_m)}{(z - p_1)(z - p_2) \cdots (z - p_n)}. \tag{17.65}$$

By comparison, the parallels with the CT case in §17.3.3 are clear. Note that, in implementation, it is often more convenient to write (17.64) [in the case that $n \geq m$] as

$$u_k = -\overline{a}_1 u_{k-1} - \ldots - \overline{a}_{n-1} u_{k-(n-1)} - \overline{a}_n u_{k-n} + \overline{b}_0 e_k + \overline{b}_1 e_{k-1} + \ldots + \overline{b}_{n-1} e_{k-(n-1)} + \overline{b}_n e_{k-n}, \tag{17.66}$$

where, for convenience, we have renumbered the coefficients $\overline{a}_k = a_{n-k}$ and $\overline{b}_k = b_{n-k}$; this form is often referred to as a **finite impulse response** (**FIR**) filter if $\overline{a}_k = 0$ for $k > 0$, and an **infinite impulse response** (**IIR**) filter if not. Note that the FIR case is distinguished by an impulse response that vanishes after finite number of steps, whereas [due to the feedback built in to the difference equation (17.66)] the IIR case is not.

Note that a difference equation governing a DT system, taken on its own, simply relates linear combinations of two or more variables describing the system and their **tap delays**; such an equation does *not* itself indicate one variable as a "cause" and another as an "effect" in a cause-effect relationship. However, the definition of a transfer function inherently identifies, or defines, a cause-effect or input-output relationship; in the examples discussed above, e_k is identified as the input, and u_k is identified as the output. This further distinction between input and output is signficant. Almost all systems encountered are causal, meaning that any variable identified as an "output" only responds to the current and past "inputs", but not to future inputs. In the DT setting (in contrast with the CT setting discussed previously), this is facilitated *only* when $m \leq n$ in the general higher-order form given above; for further discussion, see §17.4.3.2.

Once a (causal) DT linear system's transfer function is known, its response to simple inputs is easy to compute. Noting Table 17.1b, if e_k is a unit impulse (that is, $e_k = \delta_{0,k}$), then $E(z) = 1$, and if e_k is a unit step [that is, $e_k = 1$ for $k \geq 0$], then $E(z) = z/(z-1)$. In both cases, $U(z)$ is easy to compute from (17.63), and thus u_k may be found by partial fraction expansion and subsequent inverse Z transform.

As in the CT case, it is important to keep clear the distinction between the Z transform (a.k.a. transfer function) of a *system*, such as $D(z)$ above, and the Z transform of a *signal*, such as $E(z)$ above. To make clear the connection between them, note in the special case that the input to the system happens to be a unit impulse $e_k = \delta_{0,k}$, it follows that $E(z) = 1$ and thus $U(z) = D(z)$. In other words,

[23] For the sake of later convenience (in §18), we have changed the letters associated with the inputs and output considered in §17.4.3, where we consider a **DT controller** $D(z) = U(z)/E(z)$, as compared with §17.3.3, where we considered a **CT plant** $G(s) = Y(s)/U(s)$.

Algorithm 17.2: Compute response of a DT system in transfer function form to a simple input.

View Test

```
function [r,y,t]=ResponseTFdt(gz,fz,type,g)
% Using its partial fraction expansion, compute the response Y(z)=T(z)*R(z) of a
% DT SISO linear system T(z)=gz(z)/fz(z) to an impulse (type=0), step (type=1),
% or quadratic (type=2) input.  The derived type g groups together convenient
% plotting parameters: g.T is the interval over which response is plotted,
% g.h is the timestep, and {g.styler,g.styley} are the linestyles used.
switch type, case 0, numR=1; denR=1; case 1, numR=[1 0]; denR=[1 -1];
             otherwise, [numR,denR]=PolylogarithmNegativeInverse(type-1,1); end
[ra,rd,rpp,rn]=PartialFractionExpansion(numR,denR);
[ya,yd,ypp,yn]=PartialFractionExpansion(PolyConv(numR,gz),PolyConv(denR,fz));
k=[0:g.T/g.h]; t=k*g.h; y=zeros(size(k)); r=zeros(size(k));
for i=1:yn, a=yd(i)/(Fac(ypp(i)-1)*ya(i)^ypp(i));
  b=a*ones(size(k)); for j=1:ypp(i)-1, b=b.*(k-j); end
  if ypp(i)>0, y(2:end)=y(2:end)+b(2:end).*ya(i).^k(2:end); else, y(1)=y(1)+yd(i); end
end, y=real(y); plot(t,y,g.styley)
for i=1:rn, a=rd(i)/(Fac(rpp(i)-1)*ra(i)^rpp(i));
  b=a*ones(size(k)); for j=1:rpp(i)-1, b=b.*(k-j); end
  if rpp(i)>0, r(2:end)=r(2:end)+b(2:end).*ra(i).^k(2:end); else, r(1)=r(1)+rd(i); end
end, r=real(r); if type>0, hold on; plot(t,r,g.styler), hold off; end
end % function ResponseTFdt
```

Fact 17.13 *The transfer function of a DT linear system is the Z transform of its impulse response.*

It follows from the relation $U(z) = D(z)E(z)$, expanding $U(z)$, $D(z)$, and $E(z)$ with the Z transform formula (17.58a), noting that the impulse response $d_k = 0$ for $k < 0$ (that is, that the DT system is causal), and following an analogous derivation as that leading to (17.56), that

$$\sum_{k=0}^{\infty}\Big[u_k\Big]z^{-k} = \sum_{j=0}^{\infty} e_j z^{-j} \sum_{k=0}^{\infty} d_k z^{-k} = \sum_{j=0}^{\infty} e_j \left(\sum_{k=-j}^{\infty} d_k z^{-k}\right) z^{-j}$$
$$= \sum_{j=0}^{\infty} e_j \left(\sum_{k=0}^{\infty} d_{k-j} z^{-(k-j)}\right) z^{-j} = \sum_{k=0}^{\infty}\left[\sum_{j=0}^{k} e_j d_{k-j}\right] z^{-k},$$

from which we deduce that, for $k \geq 0$,

$$u_k = \sum_{j=0}^{k} e_j\, d_{k-j}; \tag{17.67}$$

note in particular that $u_k = d_k$ when $e_j = \delta_{j,0}$. Thus, as similarly noted in the CT case,

Fact 17.14 *The product $U(z) = D(z)E(z)$ in Z transform space corresponds to a convolution sum [of the input e_k with the impulse response d_k] in the untransformed space.*

Products are generally much easier to work with than convolution sums, thus highlighting the utility of the Z transform when solving constant-coefficient DT linear systems.

17.4.3.1 The transfer function of a DAC – $G(s)$ – ADC cascade

By Fact 17.13, we may determine the transfer function of a DT system, $G(z)$, simply by computing the response of the system to an impulse input, $u_k = \delta_{0,k}$, then taking the Z transform of this response. Applying this experiment to a cascade of components given by (i) a **digital-to-analog converter** (**DAC**) implementing a **zero-order-hold**[24] (**ZOH**), (ii) a CT system $G(s)$, and (iii) an **analog-to-digital converter** (**ADC**) , noting

[24] That is, holding the value of the analog signal as constant between timesteps

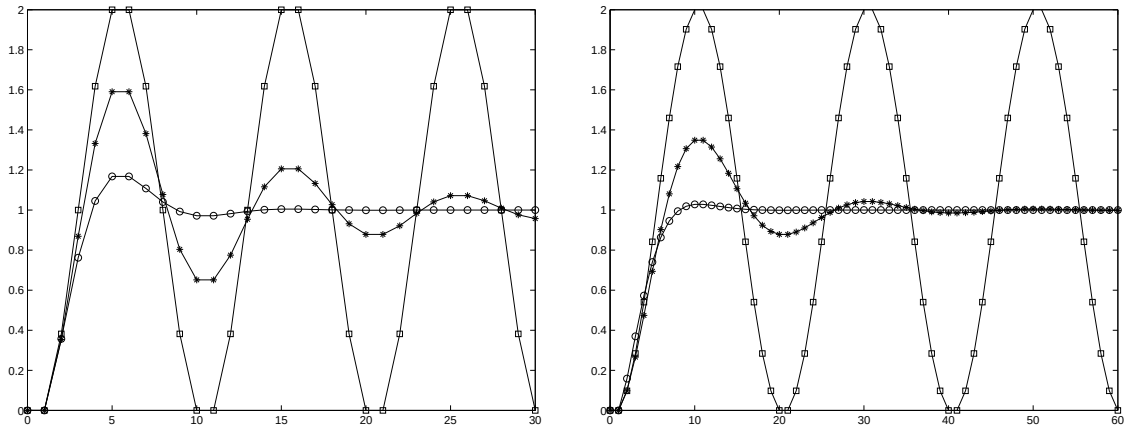


Figure 17.14: The unit step response of the system $u_{k+2} + a_1 u_{k+1} + a_0 u_k = b_0 e_k$, where $b_0 = 1 + a_1 + a_0$, with $r = 0.7$ (circles), $r = 0.9$ (asterisks), and $r = 1.0$ (squares) for $\theta = \pi/5$ (left) and $\theta = \pi/10$ (right), where $r = \sqrt{a_0}$ and $\theta = \cos^{-1}[-a_1/(2r)]$ (cf. Figure 17.12b). The lines in this figure are drawn to improve readability; the DT signals are defined only at each step, indicated by the symbols.

in this case that $u(t)$ [that is, the input to $G(s)$] is simply a unit step (with Laplace transform $1/s$ in CT) followed by a one-timestep-delayed negative unit step, it follows that

$$G(z) = \mathscr{Z}\left\{\frac{1-e^{-sh}}{s}\,G(s)\right\} = (1-z^{-1})\,\mathscr{Z}\left\{\frac{G(s)}{s}\right\} = \frac{z-1}{z}\,\mathscr{Z}\left\{\frac{G(s)}{s}\right\}, \tag{17.68}$$

where z^{-1} corresponds to the one-timestep delay, with Laplace transform e^{-sh}, and the shorthand $\mathscr{Z}\{G(s)/s\}$ means the Z transform of the discretization of the CT signal whose Laplace transform is $G(s)/s$. We will make use of this convenient (and exact!) expression in Figure 18.12 and §18.4.2.

17.4.3.2 Causal, strictly causal, and noncausal DT systems

We now revisit the difference equation in (17.64) and its corresponding transfer function in (17.65), where the degree of the polynomial in the numerator is m, and the degree of the polynomial in the denominator is n. Define the **relative degree** of such a transfer function as $n_r = n - m$. In DT, systems of this form with $n_r < 0$ are, by their very form, **noncausal** (that is, the output depends, in part, on future values of the input). In §18 we will further distinguish the DT systems of interest as "plants" $G(z)$ and "controllers" $D(z)$. All real DT plants $G(z)$, or DT analogs of CT proper (see §17.3.3.1) plants [formed, e.g., via the technique given in (17.68) of §18.4.2], are **causal**, with $n_r \geq 0$. Further, any controller $D(z)$ must only be based on available measurements, and thus must also be causal, with $n_r \geq 0$. If there is significant computation time necessary to compute the control (in digital electronics) before it can be applied back to the system, it is often most suitable to restrict the controller to be **strictly causal**, with $n_r > 0$. Thus, we will focus our attention in this study almost exclusively on the case with $n_r \geq 0$; that is, on difference equations that may be written in the form (17.66). Note also that a DT transfer function with $n_r = 0$ in (17.64) [that is, with $\overline{b}_0 \neq 0$ in (17.66)], which is causal but not strictly causal, is occasionaly said to be **semi-causal**. Two final catagories of difference equations are also occasionally encountered: if the output depends only on the current and *future* inputs, the transfer function is said to be **anti-causal**, and if the output depends strictly on future (but not current) inputs, the transfer function is said to be **strictly anti-causal**.

Example 17.13 The step response of second-order DT linear systems

We now focus further on the forced second-order case (17.62), written as $Y(z) = G(z)U(z)$, with $b_0 = 1 + a_1 + a_0$, when forced by a unit step $u_k = 1$ for $k \geq 0$; that is,

$$G(z) = \frac{1 + a_1 + a_0}{z^2 + a_1 z + a_0} = \frac{1 + a_1 + a_0}{(z - p_+)(z - p_-)} \quad \text{and} \quad U(z) = \frac{z}{z-1}.$$

Assuming that the poles are complex, $p_\pm = \left(-a_1 \pm \mathrm{i}\sqrt{4a_0 - a_1^2}\right)/2 = r e^{\pm \mathrm{i}\theta}$ where

$$r = \sqrt{a_0} \quad \text{and} \quad \theta = \cos^{-1}\left(-\frac{a_1}{2r}\right),$$

and have magnitude less than one (i.e., $a_1^2/4 < a_0 < 1$), the solution of this system may again be written in terms of sines and cosines modulated by a decaying exponential: writing the partial fraction expansion

$$Y(z) = G(z)U(z) = \frac{1 + a_1 + a_0}{(z - p_+)(z - p_-)} \cdot \frac{z}{z-1} = \frac{d_+ p_+}{z - p_+} + \frac{d_- p_-}{z - p_-} + \frac{d_0}{z-1}, \quad \begin{cases} d_+ = \frac{1+a_1+a_0}{(p_+ - p_-)(p_+ - 1)}, \\ d_- = \frac{-(1+a_1+a_0)}{(p_+ - p_-)(p_- - 1)} = \overline{d_+}, \\ d_0 = 1. \end{cases}$$

and computing the inverse Z transform of $Y(z)$ via Table 17.1b, the closed-form solution of y_k for $k > 0$ is

$$y_k = d_+ p_+^k + d_- p_-^k + d_0 = r^k \left[d_c \cos(\theta k) + d_s \sin(\theta k)\right] + 1, \quad \begin{cases} d_c = d_+ + d_- = -1, \\ d_s = \mathrm{i}(d_+ - d_-) = -\frac{a_1 + 2}{\sqrt{4a_0 - a_1^2}}, \end{cases}$$

as plotted in Figure 17.14. As in the CT case, since the system $G(z)$ considered in this example is real, the complex poles p_+ and p_- come as a conjugate pair. In addition, as consequence of the fact that the input u_k to this system is also real, the coefficients d_+ and d_- also work out to be a complex conjugate pair, and thus d_c and d_s, and y_k itself, are real. Again, the speed of oscillation θ and the rate of decay r of this response are a function of the location of the poles of the transfer function $p_\pm = r e^{\mathrm{i}\theta}$. Note also that $y_0 = y_1 = 0$; this follows directly from (17.62), noting the $k+2$ subscript on y on the LHS and the k subscript on u on the RHS.

As evident in Figure 17.14, rise time t_r, settling time t_s, and overshoot M_p characterizations, introduced in the CT case in Figure 17.12, may also be defined in the DT case. Appropriate design guides for the pole locations in the z plane in order to ensure specified maximum values of t_r, t_s, and M_p are presented in the following subsection.

17.4.4 Reconciling the Laplace and Z transforms

We now revisit the Laplace transform as defined in (17.47a) and the Z transform as defined in (17.58a):

$$F(s) = \int_0^\infty f(t) e^{-st} dt, \qquad F(z) = \sum_{k=0}^\infty f_k z^{-k}.$$

Note that, if we take $z = e^{sh}$ where h is the timestep [that is, $t_k = hk$ and $f_k = f(t_k)$], and if h is small as compared with the time scales of the variation of $f(t)$, then $F(z)$, scaled by h, is a rectangular-rule approximation of $F(s)$. Another way of making this connection between the CT analysis and the DT analysis is by comparing the closed-form solutions of the step responses of second-order CT and DT systems, as given in

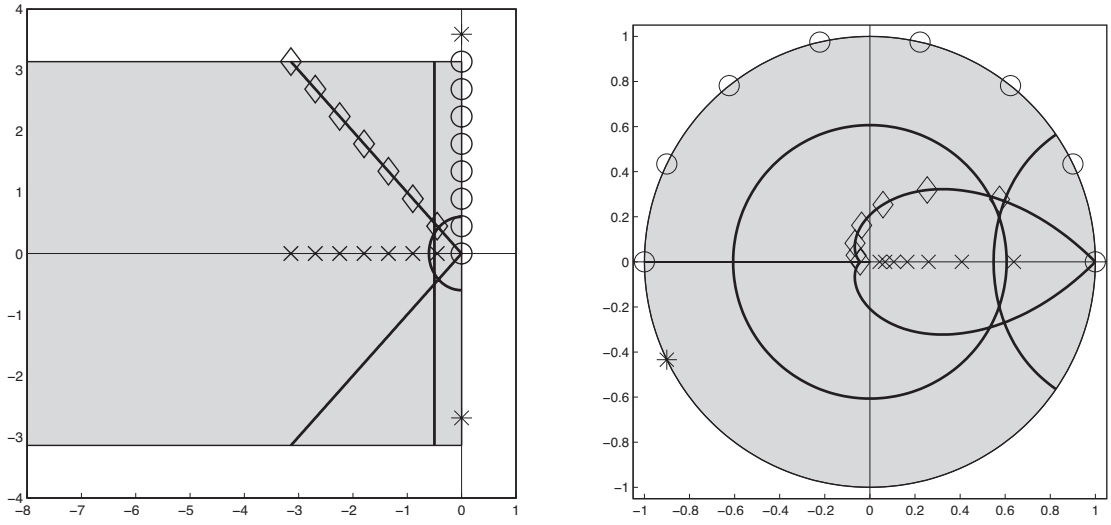


Figure 17.15: The mapping of several curves and points between the s plane (left) and the z plane (right) using (17.69). Taking $s = a + b\mathrm{i}$ and $z = re^{\mathrm{i}\theta}$, the shaded strip in the s plane with $-\infty < a \leq 0$ and $-\pi/h \leq b \leq \pi/h$ maps uniquely to the shaded disk in the z plane with $r \leq 1$. Points above and below this strip in the s-plane do not map uniquely to points in the z-plane; for example, both points marked by asterisks in the s plane map to the same point similarly marked in the z plane. This is a manifestation of the aliasing phenomenon depicted in Figure 5.4.

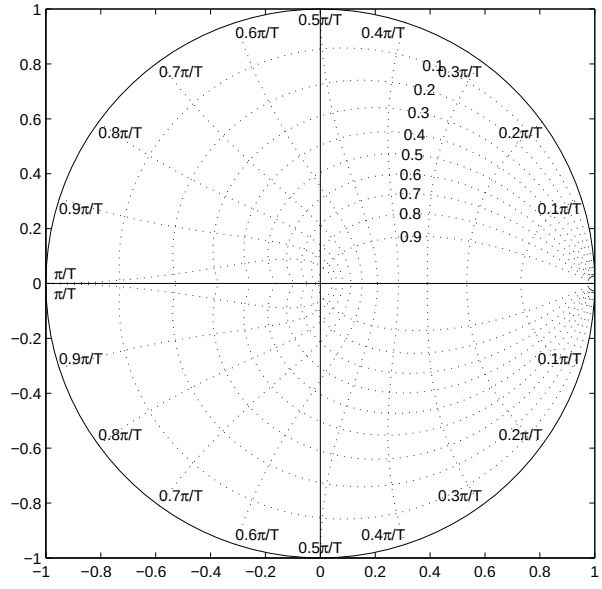


Figure 17.16: Approximate constraints, or design guides, on the admissible pole locations of a DT second-order system (or a higher-order system whose response is dominated by a pair of second-order poles) in the complex plane z in order to not exceed specified constraints on the rise time and overshoot of the system's step response (see Figure 17.14). These DT design guides are found simply by mapping the corresponding CT design guides (see Figure 17.13) using (17.69), and may be drawn with the command `zgrid` in Matlab syntax. Around the circumference are marked the values of ω_d, from $0.1\pi/T$ to π/T, and in the upper-right quadrant are marked the values of ζ, from 0.1 to 0.9, for the corresponding CT second-order design guides discussed in Example 17.12.

Examples 17.12 and 17.13. We see that the latter response is simply a discretization of the former if $r^k = e^{-\sigma t}$ and $\theta k = \omega_d t$; that is, if the CT second-order pole locations $s_\pm = -\sigma \pm i\omega_d$ and the DT second-order pole locations $z_\pm = re^{\pm i\theta}$ are related such that $r = e^{-\sigma h}$ and $\theta = \omega_d h$, and thus $z_\pm = re^{\pm i\theta} = e^{(-\sigma \pm i\omega_d)h} = e^{s_\pm h}$.

Thus, the pole locations in DT and CT are related by the mapping

$$z = e^{sh} = 1 + sh + \frac{s^2h^2}{2!} + \frac{s^3h^3}{3!} + \ldots, \tag{17.69}$$

as indicated in Figure 17.15. This connection is quite significant. For example, the approximate design guides for CT systems dominated by a pair of second-order poles, as illustrated in Figure 17.13, may be mapped immediately using this relation to obtain corresponding approximate design guides for DT second-order systems, as illustrated in Figure 17.16. It is seen that, for sinusoidal signals (that is, for $r = 1$), the number of timesteps per oscillation is $2\pi/\theta$. It is also seen that the settling time is related to r, with $r = 0.9$ corresponding to a settling time of 43 timesteps, $r = 0.8$ corresponding to a settling time of 21 timesteps, and $r = 0.6$ corresponding to a settling time of 9 timesteps.

For small h, (17.69) provides a simple connection between s-plane pole locations in the vicinity of $s = 0$ and the (scaled) z-plane pole locations in the vicinity of $z = 1$ via **Euler's approximation**

$$z \approx 1 + sh. \tag{17.70}$$

That is, for small h, the neighborhood of $z = 1$ in the z plane may be interpreted in a similar fashion as the (scaled) neighborhood of $s = 0$ in the s plane, and the three families of design guides (for t_r, t_s, and M_p) in these two regions indeed look quite similar.

17.4.4.1 Tustin's approximation

For larger h, Euler's approximation is not accurate. Motivated by the accuracy analysis of the CN method given in §10.3, the following rational approximation of (17.69), referred to in this setting as **Tustin's approximation**, is preferred for most applications:

$$z \approx \frac{1 + sh/2}{1 - sh/2} = 1 + sh + \frac{s^2h^2}{2} + \frac{s^3h^3}{4} + \ldots \quad \Leftrightarrow \quad s \approx \frac{2}{h}\frac{z-1}{z+1}. \tag{17.71}$$

Conveniently, both the exact mapping (17.69) and Tustin's approximation (17.71) map the left half plane of s to the interior of the unit circle in z; in particular, the stability boundary of s (the imaginary axis) maps to the stability boundary of z (the unit circle). This is why Tustin's approximation is strongly preferred over Euler's approximation (17.70), or other manners of truncating or approximating (17.69).

To see how to use Tustin's rule to approximate a general CT differential equation [interpreted in §18 as a controller] whose Laplace transform is $D(s)$ with a DT difference equation whose Z transform is $D(z)$, it is useful to consider first the transfer function of the following simple differential equation, with $u(t)$ and $e(t)$ taken to be zero for $t < 0$:

$$\frac{U(s)}{E(s)} = D(s) = \frac{s+a}{s+p} \quad \Rightarrow \quad (s+p)U(s) = (s+a)E(s) \quad \Rightarrow \quad \frac{du}{dt} + pu = \frac{de}{dt} + ae.$$

Approximating the derivatives in this differential equation with the CN method (see §10.1), we may write

$$\frac{u_k - u_{k-1}}{h} + p\frac{u_k + u_{k-1}}{2} = \frac{e_k - e_{k-1}}{h} + a\frac{e_k + e_{k-1}}{2}.$$

Taking the Z transform of this difference equation and reärranging leads immediately to

Algorithm 17.3: Compute the $D(z)$ that corresponds to $D(s)$ via Tustin's approximation with prewarping.

View
Test

```
function [bz,az]=C2DTustin(bs,as,h,omegac)
% Convert D(s)=bs(s)/as(s) to D(z)=bz(z)/as(z) using Tustin's method. If omegac is
% specified, prewarping is applied such that this critical frequency is mapped correctly.
if nargin==3, f=1; else, f=2*(1-cos(omegac*h))/(omegac*h*sin(omegac*h)); end
c=2/(f*h); m=length(bs)-1; n=length(as)-1; bz=zeros(1,n+1); az=bz;
for j=0:m; bz=bz+bs(m+1-j)*c^j*PolyConv(PolyPower([1 1],n-j),PolyPower([1 -1],j)); end
for j=0:n; az=az+as(n+1-j)*c^j*PolyConv(PolyPower([1 1],n-j),PolyPower([1 -1],j)); end
bz=bz/az(1); az=az/az(1);
end % function C2DTustin
```

$$D(z)=\frac{U(z)}{E(z)}=\frac{\frac{2}{h}\frac{z-1}{z+1}+a}{\frac{2}{h}\frac{z-1}{z+1}+p}=\frac{s+a}{s+p}\bigg|_{s=\frac{2}{h}\frac{z-1}{z+1}}=D(s)\bigg|_{s=\frac{2}{h}\frac{z-1}{z+1}}. \tag{17.72}$$

Using Tustin's rule (17.71), higher-order CT transfer functions $D(s)$ may similarly be approximated with a corresponding DT transfer functions $D(z)$, simply replacing each occurence of s in $D(s)$ with $\frac{2}{h}\frac{z-1}{z+1}$, then reducing to a rational expression in z, as illustrated in (17.72).

17.4.4.2 Tustin's approximation with prewarping

The exact mapping (17.69) maps the interval on the imaginary axis between $s=0$ and $s=\mathrm{i}\pi/h$ to the edge of the upper half of the unit circle (see Figure 17.15); in contrast, Tustin's rule (17.71) maps the entire upper half of the imaginary axis to the same region. Thus, though the stability boundaries of these two mappings coincide, the mapping due to Tustin's rule is **warped**, and is only accurate in the vicinity of $s=0$ and $z=1$. When designing controllers for mixed DT/CT systems (see §18.4), there is often a frequency ω of primary concern, such as a crossover frequency (see §18.2) or notch frequency (see §18.3.2). It is easy to adjust Tustin's rule via a **prewarping** strategy that *scales the s plane* by a factor $f>1$ prior to mapping it to the z plane, thus recovering the exact mapping (17.69) for the point $s=\mathrm{i}\omega$ (for some $\omega<\pi/h$) and providing a rational and accurate approximation of this mapping for points in the vicinity of $s=\mathrm{i}\omega$ without disrupting the correspondence of the two stability boundaries given by the exact mapping. To accomplish this, define

$$e^{\mathrm{i}\omega h}=\frac{1+\mathrm{i}f\omega h/2}{1-\mathrm{i}f\omega h/2}\quad\Rightarrow\quad f=\frac{2[1-\cos(\omega h)]}{\omega h\sin(\omega h)}.$$

Note that, when ω is in the range $0\le\omega h<\pi$, the factor f is in the range $1\le f<\infty$; note specifically that $f\to 1$ as $\omega\to 0$. We may then modify Tustin's rule (17.71) such that

$$z\approx\frac{1+fsh/2}{1-fsh/2}\quad\Leftrightarrow\quad s\approx\frac{2}{fh}\frac{z-1}{z+1}. \tag{17.73}$$

This is referred to as **Tustin's rule with prewarping** (see Algorithm 17.3); this rule is used in §18.4.1 to develop DT controllers which have the desired behavior near a particular frequency of interest, mimicking the behavior of effective CT controllers designed for CT plants[25].

[25] Though Tustin's rule is the method of choice for converting $D(s)$ into a DT $D(z)$, various simple alternatives to this method are sometimes enlightening to consider. For example, with the heuristic **pole-zero mapping** (a.k.a. **matched** z**-transform**) approach:
(i) All poles and finite zeros of $D(s)$ are mapped to $D(z)$ via $z=e^{sh}$.
(ii) All infinite zeros of $D(s)$ are mapped $z=-1$ in $D(z)$ (effectively, to the highest-frequency point on the stability boundary in the z plane). If a strictly causal $D(z)$ is required (see §17.4.3.2), one of the infinite zeros is instead mapped to $z=\infty$ in $D(z)$.
(iii) The gain of $D(z)$ at $z=e^{\mathrm{i}\omega h}$ is chosen to match the gain of $D(s)$ at $s=\mathrm{i}\omega$, either for $\omega=0$, or (better) for some critical ω of interest.

Algorithm 17.4: Code for drawing a Bode plot.

View Test

```
function Bode(num,den,g,h)
% The continuous-time Bode plot of G(s)=num(s)/den(s) if nargin=3, with s=(i omega), or
% the discrete-time   Bode plot of G(z)=num(z)/den(z) if nargin=4, with z=e^(i omega h).
% The derived type g groups together convenient plotting parameters: g.omega is the set of
% frequencies used, g.style is the linestyle, g.line turns on/off a line at -180 degrees,
% and, if nargin=4, h is the timestep (where the Nyquist frequency is N=pi/h).
if nargin==4, N=pi/h; g.omega=logspace(log10(g.omega(1)),log10(0.999*N),length(g.omega));
  arg=exp(i*g.omega*h); else arg=i*g.omega; end
subplot(2,1,1), loglog(g.omega,abs(PolyVal(num,arg)./PolyVal(den,arg)),g.style), hold on
a=axis; plot([a(1) a(2)],[1 1],'k:'), if nargin==4, plot([N N],[a(3) a(4)],'k--'), end
subplot(2,1,2), semilogx(g.omega,Phase(PolyVal(num,arg)./PolyVal(den,arg))*180/pi,g.style)
hold on, a=axis; if g.line==1, plot([a(1) a(2)],[-180 -180],'k:'), a=axis; end
if nargin==4, plot([N N],[a(3) a(4)],'k--'), end
end % function Bode
```

17.5 Frequency-domain analyses and filters

17.5.1 The Bode plot

The **Bode plot** (a.k.a. **open-loop Bode plot**) of a stable system is best introduced via a simple experiment: if a stable SISO linear system $G(s) = Y(s)/U(s)$, with all poles in the LHP, is excited with a sinusoidal input $u(t) = \sin(\omega t)$, then the output $y(t)$ (which, if $G(s)$ is known, may be determined via partial fraction expansion) will be composed of several components which decay exponentially in time, plus a sinusoidal component of the same frequency ω as the input but with a different magnitude and phase. The Bode plot shows the gain in magnitude and change in phase of this persistent component of the output over a range of sinusoidal input frequencies of interest; the plot of the gain versus frequency is represented in **loglog** form, and the plot of the phase change versus frequency is represented in **semilogx** form. If the system is MIMO, a Bode plot may be developed for every input/output combination. The requisite code is quite simple, as shown in Algorithm 17.4 and described further below. Bode plots of two simple systems are given in Figure 17.17, and those of two more complicated systems (with multiple breakpoints) are given in Figure 17.18.

The fact that any sinusoidal input in the experiment described above eventually leads to a sinusoidal output at the same frequency, but at a different magnitude and phase, is clearly seen if we consider first what happens if we put a *complex* input $u_1(t) = e^{i\omega t}$ into a SISO system. [This is not possible in a real physical experiment, of course, but can easily be done mathematically if we know the transfer function $G(s)$ of the (stable) system under consideration.] In this case, by Table 17.1, $U_1(s) = 1/(s-p_0)$ where $p_0 = i\omega$, and thus the partial fraction expansion of the output $Y_1(s)$ may be written [see §B.6.3] as

$$Y_1(s) = G(s)U_1(s) = d_0/(s-p_0) + \text{other terms} \qquad \Rightarrow \qquad y_1(t) = d_0 e^{i\omega t} + \text{other terms}.$$

The "other terms" in the partial fraction expansion of $Y_1(s)$ all have their poles in the LHP, because $G(s)$ is assumed to be stable, and thus the "other terms" in the corresponding inverse Laplace transform, $y_1(t)$, are all stable. Thus, the magnitude and phase of the persistent component of the output is given by the magnitude and phase of the complex coefficient d_0 which, by the discussion in §B.6.3, may be found simply as follows:

$$d_0 = \Big[Y_1(s)\cdot(s-p_0)\Big]_{s=i\omega} = \Big[G(s)\frac{1}{s-p_0}\cdot(s-p_0)\Big]_{s=i\omega} = G(i\omega).$$

The magnitude and phase shift of the persistent sinusoidal component $d_0 e^{i\omega t}$ of the complex output $y_1(t)$, as compared with the complex input $u_1(t) = e^{i\omega t}$, are thus simply the magnitude and phase of $G(i\omega)$.

Consider next what happens if we put the complex input $u_2(t) = e^{-i\omega t}$ into the system:

$$U_2(s) = 1/(s+p_0) \quad \text{where } p_0 = i\omega, \qquad Y_2(s) = G(s)U_2(s) = c_0/(s+p_0) + \text{other terms} \quad \Rightarrow$$
$$y_2(t) = c_0 e^{-i\omega t} + \text{other terms}, \quad c_0 = \Big[Y_2(s)\cdot(s+p_0)\Big]_{s=-i\omega} = \Big[G(s)\frac{1}{s+p_0}\cdot(s+p_0)\Big]_{s=-i\omega} = \overline{G(i\omega)}.$$

The magnitude and phase shift of the persistent sinusoidal component $c_0 e^{-i\omega t}$ of the complex output $y_2(t)$, as compared with the complex input $u_2(t) = e^{-i\omega t}$, are thus simply the magnitude of $G(i\omega)$ and the phase of $\overline{G(i\omega)}$, which equals the negative of the phase of $G(i\omega)$.

Finally, consider what happens if we put the *real* input $u_3(t) = [u_1(t)+u_2(t)]/2 = \cos(\omega t)$ into the system. Appealing to superposition and noting that $a\sin(x) + b\cos(x) = \sqrt{a^2+b^2}\sin(x+\psi)$ where $\psi = \text{atan2}(b,a)$,

$$\begin{aligned}
y_3(t) &= [y_1(t)+y_2(t)]/2 = (d_0 e^{i\omega t} + c_0 e^{-i\omega t})/2 + \text{other terms} \\
&= \{G(i\omega)[\cos(\omega t) + i\sin(\omega t)] + \overline{G(i\omega)}[\cos(\omega t) - i\sin(\omega t)]\}/2 + \text{other terms} \\
&= [G(i\omega) + \overline{G(i\omega)}]\cos(\omega t)/2 + [G(i\omega) - \overline{G(i\omega)}]i\sin(\omega t)/2 + \text{other terms} \\
&= \Re\{G(i\omega)\}\cos(\omega t) - \Im\{G(i\omega)\}\sin(\omega t) + \text{other terms} \\
&= |G(i\omega)|\sin[\omega t + \text{atan2}(\Re\{G(i\omega)\}, -\Im\{G(i\omega)\})] + \text{other terms} \\
&= |G(i\omega)|\sin[\omega t + \frac{\pi}{2} + \angle G(i\omega)] + \text{other terms} = |G(i\omega)|\cos[\omega t + \angle G(i\omega)] + \text{other terms}. \qquad (17.74)
\end{aligned}$$

The magnitude a and phase shift ϕ of the persistent sinusoidal component $a\cos(\omega t + \phi)$ of the real output $y_3(t)$, as compared with the real input $u_3(t) = \cos(\omega t)$, are thus, again, simply the magnitude and phase of $G(i\omega)$. An alternative derivation that leads to the same result is considered in Exercise 17.8.

Computing the Bode plot of DT systems

By (17.69), a Bode plot in DT may be drawn with the same code as that used in CT, taking $z = e^{i\omega h}$ rather than $s = i\omega$ when evaluating the response of the transfer function at various frequencies. Note that the frequency response of a DT system is only defined up to the Nyquist frequency, and thus a Bode plot in DT should only be drawn up to the Nyquist frequency.

Computing the Bode plot of unstable systems†

A Bode plot may also be developed for unstable systems. If the transfer function $G(s)$ of an unstable system is known, the process of computing its Bode plot is identical to that described above: simply calculate the magnitude and phase of $G(i\omega)$ for the relevant range of values of ω. Note that it doesn't matter that some of the components of the partial fraction expansion of $Y(s)$ have RHP poles in this case, because we need not actually perform the experiment described above, and thus we need not even consider $y(t)$.

If the transfer function $G(s)$ of an unstable system is not known, however, the computation described above can not be performed, and the experiment described in the first paragraph of §17.5.1 would be inconclusive, as the response would be dominated by one or more exponentially-growing component(s). However, if we can *guess* a simple $D(s)$ such that the closed-loop transfer function $H(s) = G(s)D(s)/[1+G(s)D(s)]$ is stable (see Figure 18.1 and the introduction to §18), then the Bode plot of $H(s)$ may be determined experimentally[26] and, since $D(s)$ is known, the magnitude and phase of $G(i\omega)$ for the corresponding range of ω may thus be deduced. Practically, it is often possible to guess a stabilizing controller for an unmodelled unstable system $G(s)$ that is adequate to determine its Bode plot; for example, if $D(s) = K$ stabilizes the closed-loop system for some value of K, then the Bode plot of $G(s)$ follows from that of $H(s)$ via the relation

[26] As described in the first paragraph of §17.5.1; again, see Figure 18.1 and the introduction to §18.

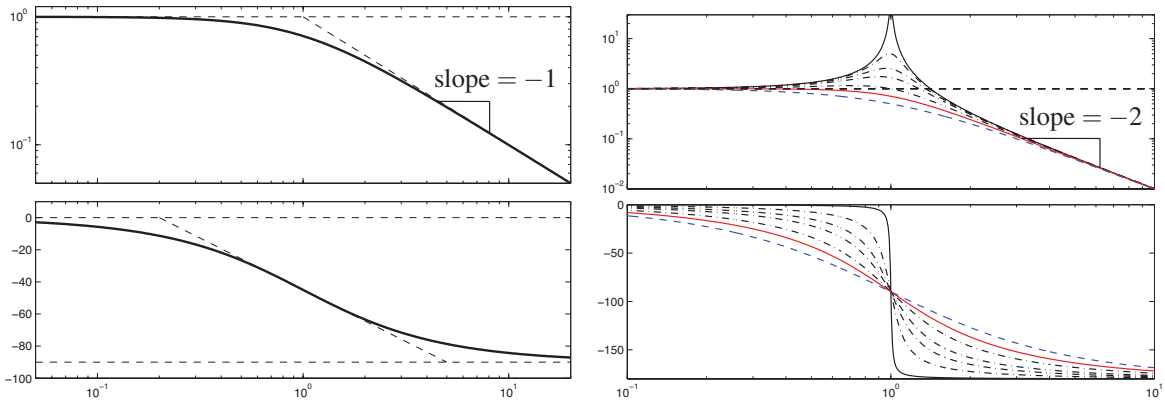


Figure 17.17: Bode plots of (left) a stable first-order low-pass filter $F_1(s) = \omega_c/[s+\omega_c]$ (with $\omega_c > 0$), and (right) a stable second-order low-pass filter $F_2(s) = \omega_c^2/[s^2 + 2\zeta\,\omega_c\,s + \omega_c^2]$ (with $0 < \zeta \leq 1$), illustrating (top) the gain, $|F(\mathrm{i}\omega)|$, and (bottom) the phase, $\angle F(\mathrm{i}\omega)$, of the filter response as a function of the normalized frequency, ω/ω_c, of a sinusoidal input. In the first-order filter, the asymptotes illustrated as dashed lines can be helpful to sketch the curve, noting that the gain at $\omega/\omega_c = 1$ is 0.707 and the phase at $\omega/\omega_c = 0.2$ is $-11°$; if the system were unstable (with $p > 0$), the phase would shift up by 90° instead of down by 90°. In the second-order filter, plotted are curves corresponding to (solid) $\zeta = 0.01$, (dot-dashed) $\zeta = 0.1, 0.2, 0.3, 0.5$, (solid) 0.707, and (dashed) $\zeta = 1$; if the system were unstable (with $-1 < \zeta < 0$), the phase would shift up by 180° instead of down by 180°. The Bode plots of a system with a first-order zero, $1/F_1(s)$, and a system with a second-order zero, $1/F_2(s)$, may be obtained by taking the reciprocals of the gain and inverting the sign of the phase in the plots shown above.

$G(\mathrm{i}\omega) = H(\mathrm{i}\omega)/\{K[1 - H(\mathrm{i}\omega)]\}$. Based on the Bode plot of $G(s)$ so determined, a much better controller $D(s)$ may then be developed using the techniques described in §18.

17.5.1.1 Sketching Bode plots of real systems by hand

The Bode plot, together with the root locus plot of §18.2.1 are two essential tools for classical feedback control design (§18). Though easily plotted using, e.g., Algorithm 17.4, it is important to know how to sketch a Bode plot by hand in order to anticipate how the Bode plot changes when a controller is modified, or to understand how to modify a controller to change a Bode plot in a desired manner.

To proceed, consider a stable or unstable real CT SISO LTI system with ℓ zeros [or $(-\ell)$ poles] at the origin, q real first-order zeros $z_i \neq 0$, r real first-order poles $p_i \neq 0$, Q pairs of complex-conjugate zeros $z^c_{i\pm}$, and R pairs of complex-conjugate poles $p^c_{i\pm}$, written in transfer function form

$$G(s) = K_o\,s^\ell \cdot \frac{(s-z_1)(s-z_2)\cdots(s-z_q)}{(s-p_1)(s-p_2)\cdots(s-p_r)} \cdot \frac{(s-z^c_{1+})(s-z^c_{1-})(s-z^c_{2+})(s-z^c_{2-})\cdots(s-z^c_{Q+})(s-z^c_{Q-})}{(s-p^c_{1+})(s-p^c_{1-})(s-p^c_{2+})(s-p^c_{2-})\cdots(s-p^c_{R+})(s-p^c_{R-})}.$$

Usually, $\ell \leq 0$; if $\ell > 0$, there are one or more *zeros*, rather than poles, at the origin. Multiplying together the factors corresponding to the pairs of complex-conjugate poles & zeros, we have

$$G(s) = K_o\,s^\ell \cdot \frac{(s-z_1)\cdots(s-z_q)}{(s-p_1)\cdots(s-p_r)} \cdot \frac{(s^2 + 2\,Z_1\,\Omega_1\,s + \Omega_1^2)\cdots(s^2 + 2\,Z_Q\,\Omega_Q\,s + \Omega_Q^2)}{(s^2 + 2\,\zeta_1\,\omega_1\,s + \omega_1^2)\cdots(s^2 + 2\,\zeta_R\,\omega_r\,s + \omega_R^2)},$$

where $-1 \leq Z_i \leq 1$, $-1 \leq \zeta_i \leq 1$, $\Omega_i > 0$, and $\omega_i > 0$. If all the $p_i < 0$ and all the $\zeta_i > 0$, then all of the poles are in the LHP and the system is stable; however, the following discussion is valid even for neutrally stable

or unstable systems. Evaluating at $s = \mathrm{i}\omega$ gives

$$G(\mathrm{i}\omega) = K_o(\mathrm{i}\omega)^\ell \cdot \frac{(\mathrm{i}\omega - z_1)\cdots(\mathrm{i}\omega - z_q)}{(\mathrm{i}\omega - p_1)\cdots(\mathrm{i}\omega - p_r)} \cdot \frac{(-\omega^2 + 2Z_1\Omega_1\,\mathrm{i}\omega + \Omega_1^2)\cdots(-\omega^2 + 2Z_Q\Omega_Q\,\mathrm{i}\omega + \Omega_Q^2)}{(-\omega^2 + 2\zeta_1\omega_1\,\mathrm{i}\omega + \omega_1^2)\cdots(-\omega^2 + 2\zeta_R\omega_R\,\mathrm{i}\omega + \omega_R^2)}, \quad (17.75)$$

Noting that $G(\mathrm{i}\omega)$ above is the product of three types of terms, the Bode plot may be sketched using the following handy rules[27] (Bode 1930):

1. For small ω, the gain and phase of the Bode plot approach the gain and phase of the following expression:

$$G(\mathrm{i}\omega) \approx (\mathrm{i}\omega)^\ell K_o[(-z_1)(-z_2)\cdots(-z_q)\cdot\Omega_1^2\Omega_2^2\cdots\Omega_Q^2]/[(-p_1)(-p_2)\cdots(-p_r)\cdot\omega_1^2\omega_2^2\cdots\omega_R^2].$$

2. The frequencies $\{|z_1|,\ldots,|z_q|;|p_1|,\ldots,|p_r|;\Omega_1,\ldots,\Omega_Q;\omega_1,\ldots,\omega_R\}$ are referred to as **breakpoints**. Starting from the asymptote at the far left of the gain and phase plots and working from left to right, the gain and phase components of the Bode plot change in an orderly fashion in the vicinity of each breakpoint:

2a. In the vicinity of each first-order pole [resp., zero] of multiplicity k, the slope of the gain curve decreases [resp., increases] by k. In the vicinity of each LHP first-order pole [resp., zero] of multiplicity k, the phase decreases [resp., increases] by $k\cdot 90^\circ$; in the vicinity of each RHP first-order pole [resp., zero] of multiplicity k, the phase increases [resp., decreases] by $k\cdot 90^\circ$. The slope of the gain curve and the value of the phase curve change gradually over a range of frequencies stretching from one order of magnitude below to one order of magnitude above the breakpoint, as illustrated in Figure 17.17a.

2b. In the vicinity of each pair of complex-conjugate poles [resp., zeros] of multiplicity k, the slope of the gain curve decreases [resp., increases] by $2k$. In the vicinity of each pair of LHP complex-conjugate poles [resp., zeros] of multiplicity k, the phase decreases [resp., increases] by $k\cdot 180^\circ$; in the vicinity of each pair of RHP complex-conjugate poles [resp., zeros] of multiplicity k, the phase increases [resp., decreases] by $k\cdot 180^\circ$. The slope of the gain curve and the value of the phase curve change gradually over a range of frequencies stretching from one order of magnitude below to one order of magnitude above the breakpoint, as illustrated in Figure 17.17b. The precise behavior of both curves in the vicinity of the breakpoint depends on the damping ζ_i [resp., Z_i], with small values of $|\zeta_i|$ [resp., $|Z_i|$] resulting in a **resonance** [resp., **anti-resonance**]; that is, a response with large [resp., small] gain in the immediate vicinity of the breakpoint.

When sketching a Bode plot, it is useful to ignore, at first, the fact that the slope of the gain curve and the value of the phase curve change gradually over two decades around the breakpoints, and simply plot straight-line asymptotes between each breakpoint. With these asymptotes as guides, the gain and phase curves may then be sketched by rounding out the corners of these asymptotes, using Figures 17.17a and b as guides.

Drawing the asymptotes between the breakpoints of a Bode plot, using rules 2a and 2b above while working from low frequencies to high frequencies, is in fact quite straightforward. The slope j of the asymptotes for any given ω between the breakpoints in the system $G(\mathrm{i}\omega)$ in (17.75) may be computed simply by assuming (even if its not true) that ω is much *smaller* than the higher-frequency breakpoints and that ω is much *larger* than the lower-frequency breakpoints, thus allowing each first-order and second-order factor in both the numerator and denominator of (17.75) to be reduced to either its first or last term as appropriate; the slope j is then given simply by the remaining power of ω in the numerator minus the remaining power of ω in the denominator. Further, in systems that are both *stable* (with no RHP poles) and *minimum phase* (with no RHP zeros; see §18.3.4.1), the corresponding phase is simply $j\cdot 90^\circ$ (mod 360); this useful rule of thumb is referred to as **Bode's gain/phase relationship**.

[27] Many texts cite the gain in terms of **decibels** (**dB**), defined as $20\cdot\log_{10}$ of the value. We avoid this convention, so that integer slopes of the gain curve on log-log plots are more readily recognized. If a gain in decibels is used, all slopes are multiplied by a factor of 20.

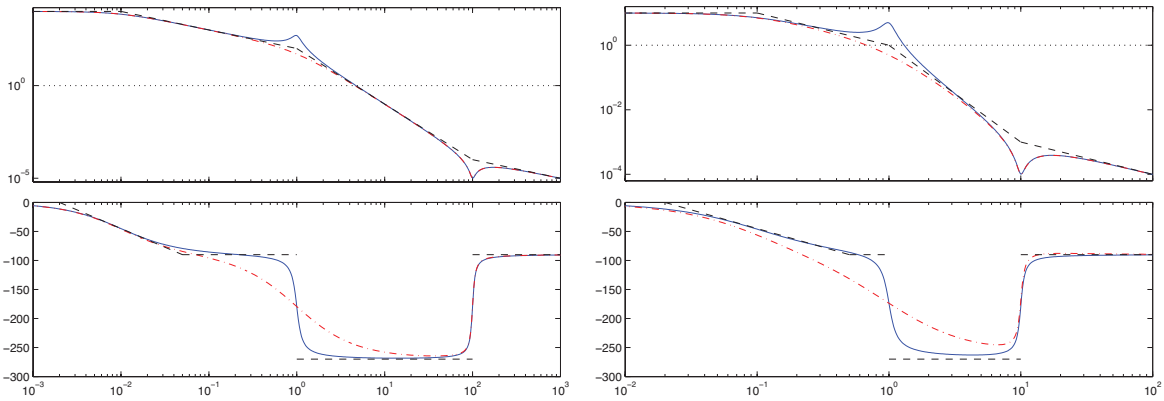


Figure 17.18: Bode plots of two more complicated systems: (left) $G_1(s)$ and (right) $G_2(s)$, as defined in 17.76.

Example 17.14 Sketching the Bode plot of representative systems.

To illustrate how to use the above-listed rules, Figure 17.18 shows the Bode plots of

$$G_1(s) = \frac{s^2 + 10s + 10000}{100\,(s+.01)(s^2 + 2\zeta s + 1)} \qquad \text{and} \qquad G_2(s) = \frac{s^2 + s + 100}{100\,(s+.1)(s^2 + 2\zeta s + 1)} \tag{17.76}$$

for $\zeta = 0.1$ (solid) and $\zeta = 1$ (dot-dashed), and effectively illustrate the process used for drawing the Bode plot of many other systems. Note that $G_1(\mathrm{i}\omega)$ has breakpoints at $\omega = .01$, 1, and 100, whereas $G_2(\mathrm{i}\omega)$ has breakpoints at $\omega = .1$, 1, and 10. In both cases, in order to sketch these Bode plots by hand, we draw the asymptotes (dashed) from left to right as if these breakpoints were far apart, where $G_1(s)$ and $G_2(s)$ act like

$$G_1(\mathrm{i}\omega) = \begin{cases} 10000 & \text{for } \omega \ll .01 \\ 100/(\mathrm{i}\omega) & \text{for } .01 \ll \omega \ll 1 \\ 100/(\mathrm{i}\omega)^3 & \text{for } 1 \ll \omega \ll 100 \\ 1/(100\,\mathrm{i}\omega) & \text{for } 100 \ll \omega \end{cases} \quad \text{and} \quad G_2(\mathrm{i}\omega) = \begin{cases} 10 & \text{for } \omega \ll .1 \\ 1/(\mathrm{i}\omega) & \text{for } .1 \ll \omega \ll 1 \\ 1/(\mathrm{i}\omega)^3 & \text{for } 1 \ll \omega \ll 10 \\ 1/(100\,\mathrm{i}\omega) & \text{for } 10 \ll \omega \end{cases} \tag{17.77}$$

The asymptotes for each of these regions are easily drawn. The Bode plot is then given by "smearing out" the corners of these asymptotes, using the behavior in the vicinity of simple first-order and second-order breakpoints illustrated in Figure 17.17 as guides. Note in particular the resonance (that is, the peak in the magnitude of the Bode plot) in Figures 17.18a-b in the case with $\zeta = 0.1$, and the lack of resonance (no peak) in the case with $\zeta = 1$. Note also that the approach described above (that is, sketching the asymptotes between the breakpoints, then "smearing out" the corners in accordance with Figure 17.17) is generally effective even if, as in the $G_2(s)$ case of Figure 17.18b, the breakpoints are so close together that ω is actually never simultaneously "far" from both of the neighboring breakpoints. The process of sketching other Bode plots is analogous, and becomes easy with practice; Exercise 17.11 gives several examples to try next.

A final important point to note about Bode plots is that:

Fact 17.15 *Bode plots are additive.*

In other words, if $G_3(s) = G_1(s)\,G_2(s)$, and if for some ω we have $G_1(\mathrm{i}\omega) = R_1\,e^{\mathrm{i}\phi_1}$ and $G_2(\mathrm{i}\omega) = R_2\,e^{\mathrm{i}\phi_2}$, then $G_3(\mathrm{i}\omega) = R_3\,e^{\mathrm{i}\phi_3} = R_1 R_2\,e^{\mathrm{i}(\phi_1+\phi_2)}$. That is, $\log(R_3) = \log(R_1) + \log(R_2)$ and $\phi_3 = \phi_1 + \phi_2$ for each ω. Thus, given log-log plots of R_1 and R_2 versus ω, and semilogx plots of ϕ_1 and ϕ_2 versus ω [i.e., given the Bode plots of $G_1(s)$ & $G_2(s)$], the corresponding log-log plot of R_3 versus ω and semilogx plot of ϕ_3 versus ω [i.e., the Bode plot of $G_3(s)$] is easily drawn. This useful fact is leveraged in Exercise 17.11c and §18.2.2.

Algorithm 17.5: Code for computing n'th-order Butterworth filters with cutoff frequency $\omega_c = 1$.

```
function [num,den]=ButterworthFilter(n)
p=exp(i*pi*(2*[1:n]-1+n)/(2*n)); num=1; den=Poly(p);
end % function ButterworthFilter
```

View
Test

17.5.2 Low-pass, high-pass, band-pass, and band-stop filters

We now explore the concept of **rational CT filters**; that is, of tunable systems (usually implemented as electric circuits, as discussed in §19) which selectively "accept" and "reject" the various frequency components of a signal. The goal an **ideal filter** is to have a gain of 1 and a phase of 0 over a specified **passband** of frequencies, and a gain of nearly 0 over the remaining frequencies (referred to as the **stopband**). Unfortunately, *no rational filters ever attain this ideal*; this section discusses a few of the common families of filters available which attempt to approximate this ideal behavior.

An ideal **low-pass filter** has a passband of all frequencies below a **cutoff frequency** ω_c, and a stopband of all frequencies above this cutoff frequency. The simplest realizable low-pass filters are the first-order filter $F_1(s) = 1/[1+(s/\omega_c)]$ depicted in Figure 17.17a and the second-order filter $F_2(s) = 1/[1+2\zeta(s/\omega_c)+(s/\omega_c)^2]$ depicted in Figure 17.17b (generally, $\zeta = 0.707$ is a good choice). For both filters, the gain approaches 1 and the phase approaches 0 for frequencies ω much smaller than ω_c, and the gain **rolls off** (on a log-log plot, at a slope of -1 in the first-order case and a slope of -2 in the second-order case) for frequencies much larger than ω_c. Neither filter has the ideal sharp cutoff at the boundary between the passband and the stopband as described above; *the higher-order filters discussed below provide a variety of ways of achieving a sharper cutoff at the boundary between the passband and the stopband, at the cost of sometimes significant phase loss, even at frequencies down to an order of magnitude below the cutoff frequency*[28].

Note that an ideal **high-pass filter** is a filter with a passband of all frequencies *above* the cutoff frequency ω_c, and a stopband of all frequencies *below* the cutoff frequency. *Any realizable low-pass filter may be converted into a high pass filter simply by replacing (s/ω_c) with (ω_c/s) in its transfer function and simplifying*; we thus focus exclusively on low-pass filter design in the discussion that follows.

Note also that an ideal **band-pass filter** is a filter with a passband of all frequencies between two critical frequencies, and a stopband at all other frequencies, whereas an ideal **band-stop filter** is a filter with a stopband of all frequencies between two critical frequencies, and a passband at all other frequencies. *A band-pass filter may be constructed from a low-pass filter and a high-pass filter connected in series, whereas a band-stop filter may be constructed from a low-pass filter and a high-pass filter connected in parallel.*

17.5.2.1 Maximal flatness filters: Butterworth and Bessel

Recalling Figure B.1b and defining r_k for $k = 1,\ldots,2n$ as the $(2n)$'th roots of -1, a **Butterworth filter** is defined by a transfer function with poles given by those values of $p_k \triangleq (\mathrm{i}r_k)$ which have a negative real part:

$$F_n^{Bu}(\overline{s}) = \frac{1}{B_n(\overline{s})} \quad \text{where} \quad \overline{s} = s/\omega_c, \quad B_n(x) = \prod_{k=1}^{n}(x-p_k), \quad \text{and} \quad p_k = e^{\mathrm{i}\pi(2k-1+n)/(2n)}. \tag{17.78}$$

Noting that the complex poles come in complex-conjugate pairs, $B_n(x)$ may be written with real coefficients by grouping together those factors in the above expression with complex-conjugate poles, such as p_1 and p_n;

[28] Such phase loss can have important negative consequences; as discussed in §18.2.2, loss of phase at crossover can lead to a significant loss of closed-loop system performance, and even closed-loop instability. Thus, if a low-pass filter is used to reject high-frequency measurement noise in a feedback control loop, the cutoff frequency of the low-pass filter should be placed at least an order of magnitude above the crossover frequency of the closed-loop system.

Algorithm 17.6: Code for computing n'th-order Bessel filters with cutoff frequency $\omega_c = 1$.

View Test

```
function [num,den]=BesselFilter(n)
k=[n:-1:0]; den=Fac(2*n-k)./Fac(n-k)./Fac(k)./2.^(n-k); num=PolyVal(den,0);
end % function BesselFilter
```

the first eight of the resulting **normalized Butterworth polynomials** $B_n(x)$ are:

$$
\begin{aligned}
B_1(x) &= (x+1),\\
B_2(x) &= (x^2+1.41421x+1),\\
B_3(x) &= (x+1)(x^2+x+1),\\
B_4(x) &= (x^2+0.76537x+1)(x^2+1.84776x+1),\\
B_5(x) &= (x+1)(x^2+0.61803x+1)(x^2+1.61803x+1),\\
B_6(x) &= (x^2+0.51764x+1)(x^2+1.41421x+1)(x^2+1.93185x+1),\\
B_7(x) &= (x+1)(x^2+0.44504x+1)(x^2+1.24698x+1)(x^2+1.80194x+1),\\
B_8(x) &= (x^2+0.39018x+1)(x^2+1.11114x+1)(x^2+1.66294x+1)(x^2+1.96157x+1).
\end{aligned}
$$

Bode plots of the first 6 Butterworth filters are given in Figure 17.19a. For sinusoidal inputs at normalized frequency $\overline{\omega} = \omega/\omega_c$, the gain of the n'th-order Butterworth filter is given by the square root of

$$
|F_n^{Bu}(\mathrm{i}\overline{\omega})|^2 = F_n^{Bu}(\mathrm{i}\overline{\omega})\,F_n(-\mathrm{i}\overline{\omega}) = \frac{1}{\prod_{k=1}^{n}[(\mathrm{i}\overline{\omega})-p_k]\prod_{k=1}^{n}[(-\mathrm{i}\overline{\omega})-p_k]} = \frac{1}{\overline{\omega}^{2n}+1},
$$

where the expression on the right follows simply because $\{\mathrm{i}p_1,\ldots,\mathrm{i}p_n,-\mathrm{i}p_1,\ldots,-\mathrm{i}p_n\}$ is the set of all roots of the equation $\overline{\omega}^{2n}+1=0$. Defining the gain $G_n^{Bu}(\overline{\omega}) = 1/(\overline{\omega}^{2n}+1)^{1/2}$, it follows that

$$
\frac{dG_n^{Bu}(\overline{\omega})}{d\overline{\omega}} = -n\,[G_n^{Bu}(\overline{\omega})]^3\,\overline{\omega}^{2n-1} < 0 \quad \text{and} \quad G_n^{Bu}(\overline{\omega}) = 1 - \frac{1}{2}\overline{\omega}^{2n} + \frac{3}{8}\overline{\omega}^{4n} + \ldots; \tag{17.79}
$$

that is, $G_n^{Bu}(\overline{\omega})$ decreases monotonically with $\overline{\omega}$ and the first $(2n-1)$ derivatives of $G_n^{Bu}(\overline{\omega})$ evaluated at $\overline{\omega}=0$ are zero; this property is referred to as **maximal flatness of the gain curve**, and is the central strength of the Butterworth filter. Unfortunately, as seen in Figure 17.19a, the higher-order Butterworth filters with sharp rolloff for $\overline{\omega} > 1$ suffer from significant phase loss over a large range of frequencies below ω_c.

Note that a **Linkwitz-Riley** (**L-R**) filter is simply two Butterworth filters applied in series. Linkwitz-Riley filters are particularly convenient in **audio crossovers**, as the magnitudes of a low-pass L-R filter (routed to a **woofer**) and a high-pass L-R filter (routed to a **tweeter**) add to unity across all frequencies[29].

A **Bessel filter** is defined by the transfer function

$$
F_n^{Be}(\overline{s}) = \frac{\theta_n(0)}{\theta_n(\overline{s})} \quad \text{where} \quad \overline{s} = s/\omega_c \quad \text{and} \quad \theta_n(x) = \sum_{k=0}^{n} \frac{(2n-k)!}{(n-k)!\,k!}\,\frac{x^k}{2^{n-k}}; \tag{17.80}
$$

[29] One or more **midrange** speakers may be added to an audio system using multiple matched pairs of low-pass/high-pass L-R filters.

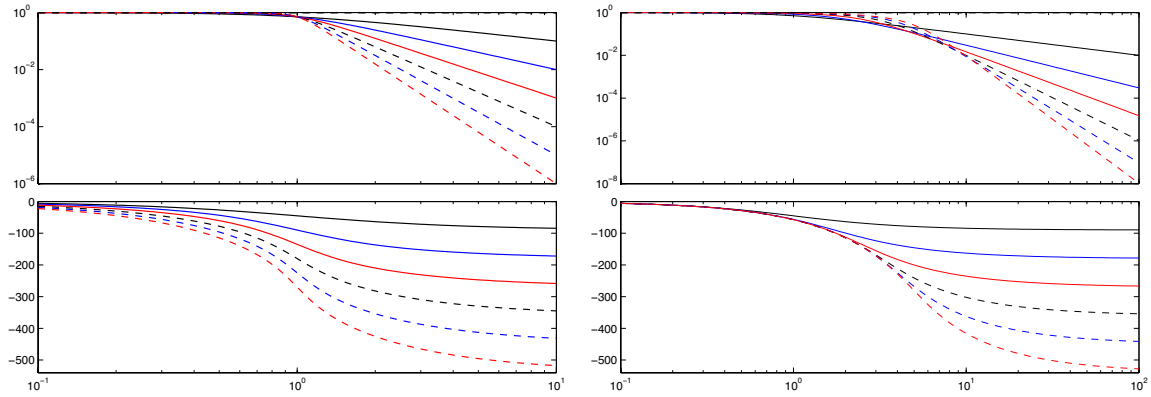


Figure 17.19: Bode plots of (solid) first-order to (dashed) sixth-order (left) Butterworth filters and (right) Bessel filters, illustrating (top) the magnitude, $|G(\mathrm{i}\overline{\omega})|$, and (bottom) the phase, $\angle G(\mathrm{i}\overline{\omega})$, of the system response as a function of the normalized frequency, $\overline{\omega} = \omega/\omega_c$, of a sinusoidal input. In the vicinity of ω_c, he Butterworth filter is optimal in terms of the flatness of the gain, whereas the Bessel filter is optimal in terms of the flatness of the group delay (that is, the phase). Note that the slope of the magnitude plot of the n'th-order filter in both cases is monotonic, and approaches $-n$ for $\overline{\omega} \gg 1$.

the $\theta_n(x)$ functions are known as **reverse Bessel polynomials**, the first eight of which are given by

$$\begin{aligned}
\theta_1(x) &= x+1,\\
\theta_2(x) &= x^2+3x+3,\\
\theta_3(x) &= x^3+6x^2+15x+15,\\
\theta_4(x) &= x^4+10x^3+45x^2+105x+105,\\
\theta_5(x) &= x^5+15x^4+105x^3+420x^2+945x+945,\\
\theta_6(x) &= x^6+21x^5+210x^4+1260x^3+4725x^2+10395x+10395,\\
\theta_7(x) &= x^7+28x^6+378x^5+3150x^4+17325x^3+62370x^2+135135x+135135,\\
\theta_8(x) &= x^8+36x^7+630x^6+6930x^5+51975x^4+270270x^3+945945x^2+2027025x+2027025.
\end{aligned}$$

Bode plots of the first 6 Bessel filters are given in Figure 17.19b. Defining $\overline{\omega} = \omega/\omega_c$ as before and (for F_6^{Be}) the phase $\phi_6^{Be}(\overline{\omega}) = -\mathrm{atan}\left[(21\overline{\omega}^5 - 1260\overline{\omega}^3 + 10395\overline{\omega})/(-\overline{\omega}^6 + 210\overline{\omega}^4 - 4725\overline{\omega}^2 + 10395)\right]$, it follows that

$$\frac{d\phi_6^{Be}(\overline{\omega})}{d\overline{\omega}} = -\frac{21\,\overline{\omega}^{10} + 630\,\overline{\omega}^8 + 18900\,\overline{\omega}^6 + 496125\,\overline{\omega}^4 + 9823275\,\overline{\omega}^2 + 108056025}{\overline{\omega}^{12} + 21\,\overline{\omega}^{10} + 630\,\overline{\omega}^8 + 18900\,\overline{\omega}^6 + 496125\,\overline{\omega}^4 + 9823275\,\overline{\omega}^2 + 108056025} < 0 \quad (17.81a)$$

$$= 1 - \frac{\overline{\omega}^{12}}{108056025} + \frac{\overline{\omega}^{14}}{1188616275} + O(\overline{\omega}^{16}); \quad (17.81b)$$

that is, $\phi_6^{Be}(\overline{\omega})$ decreases monotonically with $\overline{\omega}$ and the first 11 derivatives of the **group delay** $D_6^{Be}(\overline{\omega}) \triangleq -d\phi_6^{Be}/d\overline{\omega}$ evaluated at $\overline{\omega} = 0$ are zero; this property is referred to as **maximal flatness of the group delay curve**, and is the central strength of the Bessel filter. [To verify the correctness of (17.81), as well as to confirm that the group delay of Bessel filters at other orders are similarly flat, see Exercise 17.5.] Unfortunately, as seen in Figure 17.19b, Bessel filters have significantly less attenuation at any given frequency $\overline{\omega} > 1$ than do the corresponding Butterworth filters at the same order; although the both $|F_n^{Bu}(\mathrm{i}\overline{\omega})|$ and $|F_n^{Be}(\mathrm{i}\overline{\omega})|$ eventually roll-off at slope $-n$ on a log-log plot versus $\overline{\omega}$ for $\overline{\omega} \gg 1$, Bessel filters approach this asymptote at frequencies roughly an order of magnitude higher than do Butterworth filters at the same order.

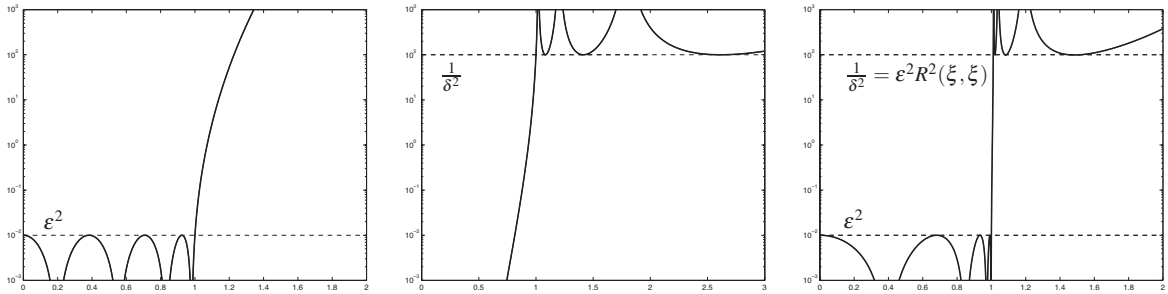


Figure 17.20: Plots of the squared and scaled (a) Chebyshev function $f_n^C(\overline{\omega}) = \varepsilon^2 T_n^2(\overline{\omega})$ [note Figure 5.9], (b) inverse Chebyshev function $f_n^I(\overline{\omega}) = 1/[\delta^2 T_n^2(1/\overline{\omega})]$, and (c) elliptic function $f_n^E(\overline{\omega}) = \varepsilon^2 R^2(\xi,\overline{\omega})$, with $n = 8$, $\varepsilon = \delta = 0.1$, and $\xi = 1.011$. The corresponding **Chebyshev**, **inverse Chebyshev**, and **elliptic filters**, known as **equiripple filters**, are characterized by the filter gain $|F_n(\overline{\omega})|^2 = 1/(1+f_n(\overline{\omega}))$ [see Figure 17.21].

17.5.2.2 Equiripple filters: Chebyshev, inverse Chebyshev, and elliptic†

The Butterworth and Bessel filter gains illustrated Figure 17.19 decrease monotonically with frequency. A comparison of these filters indicates an interesting tradeoff between the flatness of the gain in the passband, the flatness of the phase in the passband, and the roll-off of the gain above the passband. There have been a cornucopia of additional families of filters proposed over the years which seek to achieve different tradeoffs between such generally competing objectives; we indulge ourselves here with a brief discussion of one additional such family, known as **equiripple filters**.

Equiripple filters are based on the **Chebyshev function** (see Figures 5.9 and 17.20a,b) and a powerful generalization of the Chebyshev function known as the **elliptic function** (see Figure 17.20c).

For the remainder of this subsection (only), we focus our attention on the filter gain, plotting the square of this gain on a linear plot rather than a log-log plot as this convention illustrates well the criteria considered in equiripple filter design, as shown in Figures 17.21 and 17.22. We also define the **transition band** as the region between the passband and the stopband. In equiripple filter design (see Figures 17.21 and 17.22), one attempts to make this transition band as narrow as possible by sacrificing the monotonic behavior of the filter gain seen in Figure 17.19. That is, equiripple filters achieve rapid roll-off in the transition band by allowing the gain to **ripple** between minimum and maximum admissible values: in particular, **Chebyshev filters** allow ripples in the passband, **inverse Chebyshev filters** allow ripples in the stopband, and the (most general) **elliptic filters** allow ripples in both the passband and the stopband. The Chebyshev and inverse Chebyshev filters are both special cases of the elliptic filter, and the Butterworth filter is a special case of all three.

Chebyshev filters

For sinusoidal inputs at normalized frequency $\overline{\omega}$, the **Chebyshev filter** $F_n^C(\overline{s};\varepsilon)$ is characterized by the gain

$$|F_n^C(\mathrm{i}\overline{\omega};\varepsilon)| = \frac{1}{\sqrt{1+\varepsilon^2 T_n^2(\overline{\omega})}} \quad \text{where} \quad \overline{s} = s/\omega_c, \;\; \overline{\omega} = \omega/\omega_c; \tag{17.82}$$

note the tunable parameter ε in addition to the order parameter n and cutoff frequency ω_c.

In order to write the Chebyshev filter in transfer function form

$$F_n^C(\overline{s};\varepsilon) = c^C \frac{1}{(\overline{s}-p_1^C)(\overline{s}-p_2^C)\cdots(\overline{s}-p_n^C)},$$

we must identify the transfer function poles p_m^C and gain c^C (the zeros of the Chebyshev filter are all at

Algorithm 17.7: Code for computing the transfer function of a Chebyshev filter of order n.

View
Test

```
function [num,den]=ChebyshevFilter(n,epsilon)
% Computes an n'th order Chebyshev filter with cutoff frequency omega_c=1 and
% ripple in the passband between 1/(1+epsilon^2) and 1 (see Figures 17.20b, 17.21b).
p=i*cos(acos(i/epsilon)/n+[0:n-1]*pi/n); num=1/(epsilon*2^(n-1)); den=real(Poly(p));
end % function ChebyshevFilter
```

Algorithm 17.8: Code for computing the transfer functions of the inverse Chebyshev filter of order n.

View
Test

```
function [num,den]=InverseChebyshevFilter(n,delta)
% Computes an n'th order inverse Chebyshev filter with cutoff frequency omega_c=1 and
% ripple in the stopband between 0 and delta^2 (see Figures 17.20c, 17.21c).
p=-i./cos(acos(i/delta)/n+[0:n-1]*pi/n); z=i./cos((2*[1:n]-1)*pi/(2*n));
C=Prod(p)/Prod(z); num=real(C*Poly(z)); den=real(Poly(p));
end % function InverseChebyshevFilter
```

Algorithm 17.9: Complete code for computing the transfer function of an elliptic filter of order $n = 2^s$.

View
Test

```
function [num,den]=EllipticFilter(n,epsilon,delta)
% Computes an n'th order elliptic filter (FOR n=2^s ONLY) with cutoff frequency omega_c=1,
% ripple in the passband between 1/(1+epsilon^2) and 1, and
% ripple in the stopband between 0 and delta^2 (see Figures 17.20d, 17.21d).
s=log2(n); z=0; p.n=n; p.target=1/(epsilon*delta); xi=Bisection(1.0001,100,@Func,1e-6,0,p)
for r=s-1:-1:0, z=1./sqrt(1+sqrt(1-1./(CRF(2^r,xi,xi))^2)*(1-z)./(1+z)); z=[z;-z]; end
zeta=SN(n,xi,epsilon); a=-zeta*sqrt(1-zeta^2).*sqrt(1-z.^2).*sqrt(1-z.^2./xi^2);
b=z*sqrt(1-zeta^2*(1-1/xi^2)); c=1-zeta^2*(1-z.^2/xi^2); efz=i*xi./z; efp=(a+i*b)./c;
C=Prod(efp)/Prod(efz)/sqrt(1+epsilon^2); num=real(C*Poly(efz)); den=real(Poly(efp));
end % function EllipticFilter
%%%
```

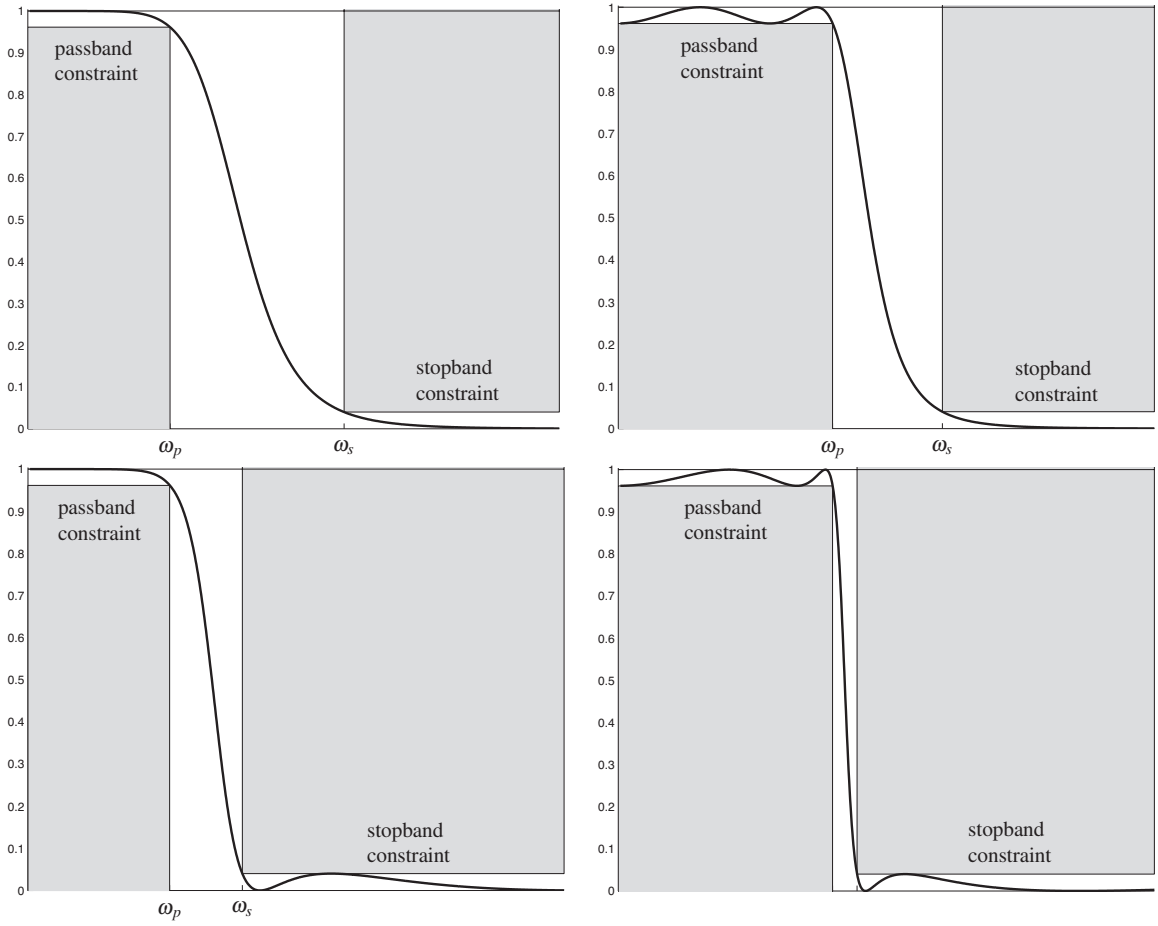


Figure 17.21: Linear plots of the square of the filter gain of the (a) Butterworth, (b) Chebyshev, (c) inverse Chebyshev, and (d) elliptic filters, along with the criteria used for the equiripple filter designs with $\varepsilon = \delta = 0.2$ and order $n = 4$. The interval $(0, \omega_p)$ is referred to as the **passband**, where the square of the filter gain is constrained to lie between $1/(1+\varepsilon^2)$ and 1, whereas the interval (ω_s, ∞) is referred to as the **stopband**, where the square of the filter gain is constrained to lie between 0 and δ^2. The interval (ω_p, ω_s) is referred to as the **transition band**. By allowing small ripples in the gain in the passband (Chebyshev), stopband (inverse Chebyshev), or both (elliptic), the width of the transition band is substantially reduced as compared with the nonrippled (Butterworth) case at a given order n.

Inverse Chebyshev filters

For sinusoidal inputs at normalized frequency $\overline{\omega}$, the **inverse Chebyshev filter** $F_n^I(\overline{s};\delta)$ is characterized by

$$|F_n^I(i\overline{\omega};\delta)| = \frac{1}{\sqrt{1+1/[\delta^2 T_n^2(1/\overline{\omega})]}} \quad \text{where} \quad \overline{s} = s/\omega_c, \;\; \overline{\omega} = \omega/\omega_c; \tag{17.83}$$

note the tunable parameter δ in addition to the order parameter n and cutoff frequency ω_c.

In order to write the inverse Chebyshev filter in transfer function form

$$F_n^I(\overline{s};\varepsilon) = c^I \frac{(\overline{s}-z_1^I)(\overline{s}-z_2^I)\cdots(\overline{s}-z_n^I)}{(\overline{s}-p_1^I)(\overline{s}-p_2^I)\cdots(\overline{s}-p_n^I)},$$

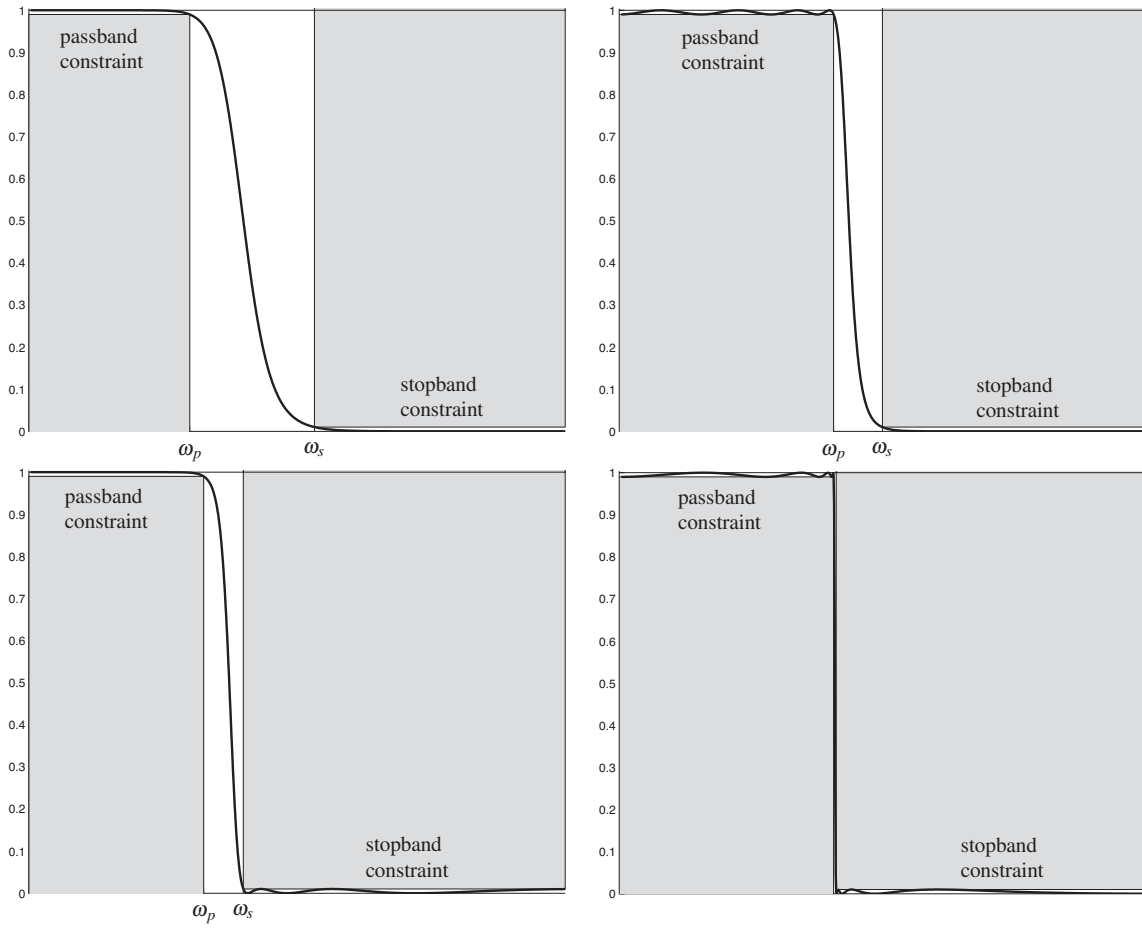


Figure 17.22: Linear plots of the square of the filter gain of the (a) Butterworth, (b) Chebyshev, (c) inverse Chebyshev, and (d) elliptic filters with $\varepsilon = \delta = 0.1$ and order $n = 8$ (cf. Figure 17.21).

we must identify the transfer function zeros z_m^I, poles p_m^I, and gain c^I. Noting (17.83) for $\overline{s} = -\mathrm{i}\overline{\omega}$, and additionally noting (5.56), the poles of $F_n^I(\overline{s};\varepsilon)$ are given by

$$1 + \frac{1}{\delta^2 T_n^2(1/\overline{\omega})} = 1 + \frac{1}{\delta^2 T_n^2(\cos\theta)} = 1 + \frac{1}{\delta^2 \cos^2(n\theta)} = 0 \quad \text{where} \quad \frac{1}{\overline{\omega}} = \frac{1}{\mathrm{i}\overline{s}} \triangleq \cos\theta,$$

and thus the (stable) transfer function poles (with negative real part) may be written

$$p_m^I = \frac{-\mathrm{i}}{\cos\theta_m} \quad \text{where} \quad \theta_m = \frac{1}{n}\operatorname{acos}\frac{\mathrm{i}}{\delta} + \frac{m\pi}{n} \quad \text{for} \quad m = 0,\ldots,n-1.$$

By (17.83), the transfer function zeros are simply the inverse of the zeros of the Chebyshev polynomial:

$$T_n\left(\frac{1}{\overline{\omega}}\right) = \cos(n\phi) = 0 \quad \text{where} \quad \frac{1}{\overline{\omega}} \triangleq \cos\phi \quad \Rightarrow \quad z_m^I = \frac{\mathrm{i}}{\cos\phi_m}, \quad \phi_m = \frac{(2m-1)\pi}{2n} \quad \text{for} \quad m = 1,\ldots,n.$$

The transfer function gain is given by $c^I = \prod p_m^I / \prod z_m^I$. These equations are implemented in Algorithm 17.8 and visualized in Figures 17.21c and 17.22c.

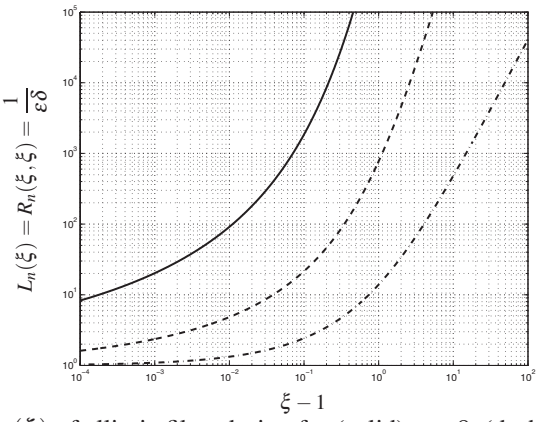


Figure 17.23: The factor $L_n(\xi)$ of elliptic filter design for (solid) $n = 8$, (dashed) $n = 4$, (dot-dashed) $n = 2$.

Elliptic filters

The **elliptic filter** (a.k.a. **Cauer filter**) $F_n^E(\overline{s};\varepsilon,\xi)$ is a remarkably flexible filter design characterized, for sinusoidal inputs at normalized frequency $\overline{\omega}$, by the gain function

$$|F_n^E(\mathrm{i}\overline{\omega};\varepsilon,\xi)| = \frac{1}{\sqrt{1+\varepsilon^2 R_n^2(\xi,\overline{\omega})}} \quad \text{where} \quad \overline{s} = s/\omega_c, \;\; \overline{\omega} = \omega/\omega_c;$$

with tunable parameters ε and ξ in addition to the order parameter n and cutoff frequency ω_c, where $R_n(\xi,x)$ is a special function known as the **elliptic rational function** (a.k.a. **Chebyshev rational function**), which is normalized such that $R_n(\xi,1) = 1$. A complete exposition of the elliptic rational function for all orders n is quite involved and a bit peripheral to the present discussion[30]; suffice it to note here that the elliptic rational function may be defined for order $n = 2^s$ for integer s via the recursive **nesting property**

$$R_{m\cdot p}(\xi,x) = R_m(R_p(\xi,\xi),R_p(\xi,x)) \quad \text{where} \quad R_2(\xi,x) = \frac{(t+1)x^2-1}{(t-1)x^2+1} \quad \text{and} \quad R_1(\xi,x) = x, \qquad (17.84a)$$

and, defining the **discrimination factor** $L_n(\xi) \triangleq R_n(\xi,\xi)$ [see Figure 17.23], the factor t defined according to

$$t_m(\xi) \triangleq \sqrt{1-1/L_m^2(\xi)} \quad \text{and} \quad t \triangleq t_1(\xi) = \sqrt{1-1/\xi^2}. \qquad (17.84b)$$

In order to write the elliptic filter in transfer function form

$$F_n^E(\overline{s};\varepsilon,\xi) = c^E \frac{(\overline{s}-z_1^E)(\overline{s}-z_2^E)\cdots(\overline{s}-z_n^E)}{(\overline{s}-p_1^E)(\overline{s}-p_2^E)\cdots(\overline{s}-p_n^E)},$$

we must identify the transfer function zeros z_m^E, poles p_m^E, and gain c^E. The zeros z_m^E of the elliptic filter $F_n^E(\overline{s};\varepsilon,\xi)$ are i times the poles p_m^R of the elliptic rational function, which may be written in the form

$$R_n(\xi,x) = c^R \frac{(x-z_1^R)(x-z_2^R)\cdots(x-z_n^R)}{(x-p_1^R)(x-p_2^R)\cdots(x-p_n^R)}.$$

The poles p_m^R of the elliptic rational function, in turn, are given by the reciprocal of the zeros z_m^R of the elliptic rational function, scaled by ξ, according to the **inversion relationship**

$$R_n(\xi,\xi/x) = \frac{R_n(\xi,\xi)}{R_n(\xi,x)} \quad \Rightarrow \quad p_m^R z_m^R = \xi \quad \Rightarrow \quad z_m^E = \mathrm{i}\xi/z_m^R. \qquad (17.84c)$$

[30] The interested reader is referred to Lutovac (2001) for a comprehensive discussion of elliptic rational functions at other orders.

For $n = 2^s$, the zeros of $R_n(\xi, x)$, $z_m^R \triangleq z_m^{R,1}$ for $m = 1, \ldots, n$, may be determined by initializing $z^{R,n} = 0$ and iterating

$$\left[z_m^{R,2^r} = 1 \Big/ \sqrt{1 + t_{2^r} \frac{1 - z_m^{R,2^{r+1}}}{1 + z_m^{R,2^{r+1}}}} \quad \text{and} \quad z_{m+2^{s-1-r}}^{R,2^r} = -z_m^{R,2^r} \quad \text{for } m = 1, \ldots, 2^{s-1-r} \right] \quad \text{for } r = s-1, \ldots, 0. \tag{17.84d}$$

The poles p_m^E of the elliptic filter $F_n^E(\bar{s}; \varepsilon, \xi)$ are given by $p_m^E = (a_m + \mathrm{i} b_m)/c_m$ for $m = 1, \ldots, n$ where

$$a_m = -\zeta_n \sqrt{1 - \zeta_n^2} \sqrt{1 - (z_m^R)^2} \sqrt{1 - (z_m^R)^2/\xi^2}, \quad b_m = z_m^R \sqrt{1 - \zeta_n^2 (1 - 1/\xi^2)}, \quad c_m = 1 - \zeta_n^2 \left(1 - \frac{(z_m^R)^2}{\xi^2}\right), \tag{17.84e}$$

where the ζ_n function may be found for $n = 2^s$ via the recursive formula

$$\zeta_n(\xi, \varepsilon) = \zeta_2\left(\xi, \sqrt{\frac{1}{\zeta_{n/2}^2(L_2(\xi), \varepsilon)} - 1}\right) \quad \text{with} \quad \zeta_2(\xi, \varepsilon) = \frac{2}{(1+t)\sqrt{1+\varepsilon^2} + \sqrt{(1-t)^2 + \varepsilon^2 (1+t)^2}}. \tag{17.84f}$$

The transfer function gain is given by

$$c^E = \frac{1}{\sqrt{1+\varepsilon^2}} \frac{\prod p_m^E}{\prod z_m^E}. \tag{17.84g}$$

These equations are implemented in Algorithm 17.9 and visualized in Figures 17.21d and 17.22d. Given constraints on ε and δ (see, e.g., Figures 17.21 and 17.22) and a choice for n, the necessary value for ξ may be calculated via the discrimination factor (see Figure 17.23), using a bisection search (see Algorithm 3.4) to find that value of ξ such that $L_n(\xi) - 1/(\varepsilon\delta) = 0$.

Exercises

Exercise 17.1 Compute the eigenvalues and eigenvectors of the system matrix A_1 in (17.43) (there will be three real eigenvalues and one complex conjugate pair of eigenvalues). Then, using Algorithm 10.1, perform a numerical simulation of each of these modes, and explain the results. [Hint: in each of the simulations performed corresponding to the real modes, simply initialize the state of the system in the shape of the (real) eigenvector, then plot how each of the states evolves in time. In the case of the pair of complex modes, initialize the state of the system in the shape of an appropriate linear combination of the two (complex) eigenvectors that results in a real perturbation, and again plot how each of the states evolves in time.]

Physically, the **dutch roll mode** of an aircraft corresponds to an oscillatory perturbation involving first a bit of roll, which creates adverse yaw towards the upward moving wing, which in turn causes a loss of lift on the upward perturbed wing, which then causes roll in the other direction, etc.; looking aft out the top of the cockpit using a **periscopic sextant** (which you could do in certain vintage transport aircraft, such as the **C124 Old Shaky**, in order to perform celestial navigation), the tail of the aircraft repeatedly draws an infinity sign on the horizon when the dutch roll mode is excited (which is, essentially, all the time); nonlinearities not discussed here ultimately limit the magnitude of the dutch roll mode of the C124 to a finite-amplitude **limit cycle**. The **roll subsidence mode** is an exponentially stable (and usually relatively fast) mode that quantifies how much the aircraft continues to roll once a slight roll is initiated then the ailerons neutralized. The **spiral mode** is an exponentially stable (and usually relatively slow) mode coupling the yaw rate and the roll (but typically not necessarily involving sideslip; it is often a nearly coördinated motion). Associate each of the simulations you perform in Exercise (17.1) to the modes characterized above, and determine the stability and damping of each. Also, there is one more (trivial) mode: physically interpret this mode as well.

Exercise 17.2 The step response of the system

$$\frac{d^2y}{dt^2} + 2\zeta\omega_n\frac{dy}{dt} + \omega_n^2 y = u, \tag{17.85}$$

determined analytically in Example 17.12, is plotted in Figure 17.12b for $\omega_n = 1$ and various values of ζ. Rescaling the axes as appropriate, draw the corresponding step response for the case with $\omega_n = 4$ and $\zeta = 0.1$.

Exercise 17.3 Determine the values of $\{a_0, a_1, f_0, f_1\}$ in (17.60) that lead to Fibonacci's sequence. Compute the corresponding values of $\{d_+, p_+, d_-, p_-\}$ in (17.61). Based on the magnitudes of $p_\pm$, do you expect the magnitudes of the elements of Fibonacci's sequence to grow or decay with k? Why? To quantify this answer more precisely, based on (17.61) and the computed values of $\{d_+, p_+, d_-, p_-\}$, what do you expect the ratio of f_{k+1}/f_k to approach for large k? Marching (17.60) numerically, determine the 100th and 101st terms of Fibonacci's sequence, compute f_{101}/f_{100}, and discuss.

Exercise 17.4 Recall that Algorithm 17.3 converts from CT to DT using Tustin's approximation, with or without prewarping. Note in (17.73) that this approximation is reversible; thus, write an analogous function, named `D2CTustin.m`, that uses the reverse of Tustin's rule to convert, with or without "postwarping", from a DT function $D(z)$ back to the corresponding function $D(s)$ in CT. Write and run an associated test script, `D2CTustinTest.m`, that calls `C2DTustin`, with prewarping, and then `D2CTustin`, with "postwarping", demonstrating that the original CT function is recovered.

Exercise 17.5 Verify (17.79). Then, verify (17.81a) and (17.81b), and determine similar expressions for the fourth- and eighth-order Bessel filters. Hint: use a symbolic differentiation package, such as Mathematica or the symbolic toolbox of Matlab, as this derivation is too tedious to perform by hand.

Exercise 17.6 Example 17.9 modeled the horizontal dynamics of a three-story building. Using Algorithm 17.4, compute the Bode plot of the (lightly damped) structure modeled in (17.36) with $k = 10000$ and $c = 10$ [see (17.37)], modeling the frequency response of the top floor, x_3, when

(a) the building is excited by an earthquake (modeled as horizontal movements of the ground, w), and
(b) the building is excited by vortex shedding caused by wind blowing past the top floor (modeled as horizontal forcing on the top floor, v).

By looking at the Bode plots so generated, identify the three resonant frequencies that the response of the structure to a sinusoidal input is greatest. Compute the poles of the transfer function in either forcing case (note that the poles in the two cases are identical—why?). Is there a correspondance between these pole locations and the resonant frequencies in the Bode plot? Explain. Which of the three resonances that appear in these Bode plots is the strongest? Is this consistent with the damping of the corresponding poles? Discuss.

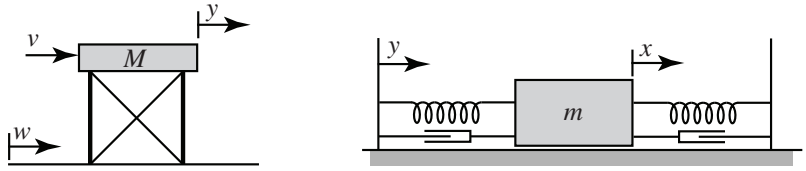


Figure 17.24: (a) A one-story structure, and (b) a mass-spring-damper system to be affixed to its top floor.

Exercise 17.7 Simplifying the linearized model considered in Example 17.9, a one-story building (Figure 17.24a) in which the top floor has horizontal position y, applied force v, and mass M is governed by

$$M\frac{d^2y}{dt^2} = v - K(y-w) - C\left(\frac{dy}{dt} - \frac{dw}{dt}\right), \tag{17.86a}$$

where K and C denote the effective spring and damping coefficients of the linearized structure, and w denotes the horizontal position of the ground. Extending the mass-spring-damper model considered in Example 17.1 to account for a moving wall, a single mass-spring-damper system (Figure 17.24b) in which the mass has horizontal position x and the wall has horizontal position y is governed by

$$m\frac{d^2x}{dt^2} = -k(x-y) - c\left(\frac{dx}{dt} - \frac{dy}{dt}\right), \tag{17.86b}$$

where $k/2$ and $c/2$ are the coefficients of the spring and damper mounted on each side of the mass.

We consider now the problem of mounting the mass-spring-damper system to the top of the one-story structure, so the y in (17.86a) corresponds precisely to the y in (17.86b), and the reaction force v on the top of the building from the mass-spring-damper system is precisely the negative of the RHS of (17.86b); that is,

$$v = k(x-y) + c\left(\frac{dx}{dt} - \frac{dy}{dt}\right). \tag{17.86c}$$

(a) Taking the Laplace transform of (17.86a)-(17.86c), assuming the system starts at rest, compute $\mathscr{L}_1$, $\mathscr{L}_2$, $\mathscr{L}_3$, and $\mathscr{L}_4$ as a function of M, K, C, m, k, c, and s in the following (transformed) representation these equations:

$$\mathscr{L}_1 Y(s) = V(s) + \mathscr{L}_2 W(s), \tag{17.87a}$$
$$\mathscr{L}_3 X(s) = \mathscr{L}_4 Y(s), \tag{17.87b}$$
$$V(s) = \mathscr{L}_4 X(s) - \mathscr{L}_4 Y(s). \tag{17.87c}$$

Now, premultiply (17.87b) by $\mathscr{L}_4$ and (17.87c) by $\mathscr{L}_3$ and, noting that $\mathscr{L}_3\mathscr{L}_4 = \mathscr{L}_4\mathscr{L}_3$, substitute the former into the latter to eliminate $X(s)$. Then substitute the resulting equation into $\mathscr{L}_3$ times (17.87a) to eliminate $V(s)$. The result may be written in transfer-function form as

$$\mathscr{L}_5 Y(s) = \mathscr{L}_6 W(s) \quad \Rightarrow \quad \frac{Y(s)}{W(s)} = \frac{\mathscr{L}_6}{\mathscr{L}_5} \tag{17.88}$$

(b) Compute $\mathscr{L}_5$ and $\mathscr{L}_6$ in terms of M, K, C, m, k, c, and the Laplace transform variable s. Is this system strictly proper, semiproper, or improper? How many poles and (finite) zeros are there in this system?
(c) Taking $m = 0$, simplify (17.88); is the result what you expect? Explain.
(d) In addition to taking $m = 0$, assume further that $M = K = 10$ and $C = 0$. Where are the poles and zeros of $G(s)$ in this case? Plot the Bode plot of this system.
(e) Taking $w(t) = \delta^{\lambda,m}(t)$ for $\lambda \to \infty$, and thus $W(s) = 1$, calculate the impulse response of the system considered in part (d) exactly, by hand, using partial fraction expansion and inverse Laplace transform, noting the Laplace transforms listed in Table 17.1. Accurately plot this impulse response. Then, taking $w(t)$ as a unit step, and thus $W(s) = 1/s$, calculate and plot the step response of this system in the same manner.
(f) Taking $C = 10\sqrt{2} = 14.14$, repeat parts (d) and (e) in their entirety.
(g) Now taking $M = K = 10$, $C = 0$, and $m = 1$, $k = 9$, $c = 0$ in (17.88), compute the poles of $G(s)$ by hand (hint: the denominator is now quadratic in s^2). Have the slowest poles, corresponding to the pole locations identified in part (d), without the spring/mass/damper system mounted atop the structure, moved? Discuss.
(h) Keeping $M = K = 10$, and $C = 0$ as in part (g), tuning k and c appropriately for any given m allows one to damp *all four* of the system poles. For example:

- Taking $m = 1$ and tuning c and k (to $c \approx 0.5$ and $k \approx .8$) gives poles at -0.130 ± 1.002i and -0.145 ± 0.873i.
- Taking $m = 5$ and tuning c and k (to $c \approx 3.5$ and $k \approx 2$) gives poles at -0.257 ± 0.890i and -0.268 ± 0.628i.

Which of these two cases is better damped? If a step input $w(t)$ is applied, what is the approximate settling time in each of these two cases? Discuss the limitations of the design guide you used to make these estimates, and why they might be somewhat off in this situation.

Exercise 17.8 Recalling the discussion in §17.5.1, if $Y(s) = G(s)U(s)$, then the magnitude and phase shift of the persistent sinusoidal component of the real output $y(t)$ corresponding to a real sinusoidal input $u(t)$ are simply the magnitude and phase of $G(\mathrm{i}\omega)$. Now consider the real input $u(t) = \sin(\omega t)$. *Without decomposing in terms of complex exponentials and appealing to superposition*, but instead using directly the entries in the Table 17.1 for $\sin(bt)$ and $\cos(bt)$, compute the coefficients f_0 and g_0 in the expression

$$Y(s) = G(s)U_4(s) = G(s) \cdot \frac{\omega}{s^2+\omega^2} = \frac{f_0}{s^2+\omega^2} + \frac{g_0\, s}{s^2+\omega^2} + \text{other terms}, \tag{17.89}$$

where f_0 and g_0 are both constrained to be real [hint: following an analogous approach as that used in (B.100), multiply (17.89) by $(s^2+\omega^2)$, simplify, then evaluate the result at both $s = \mathrm{i}\omega$ and $s = -\mathrm{i}\omega$, thus leading to two equations for the (real) unknowns f_0 and g_0]. Then, applying (B.46) appropriately, determine an expression for the magnitude a and phase shift ϕ of the persistent sinusoidal component of the output rewritten into the form $y(t) = a\sin(\omega t + \phi)$. Is this answer consistent with that in §17.5.1, as mentioned above?

Exercise 17.9 Redoing Examples 2.1 and 17.9, compute the statics and dynamics of the three-story structure considered in these examples when an $m = 400$ kg mass is placed on top of the structure. Taking $k = 10000$ and $c = 10$, the resulting dynamics will be of the same form as (17.36), with coefficients slightly modified from those in (17.37). Then, following the approach used in Exercise 17.7, optimize the stiffness and damping of a mass-spring-damper system with mass $m = 400$ kg mounted atop this three-story structure in order to minimize its resonance peak. Draw the modified Bode plot, and compare with the result of Exercise 17.6.

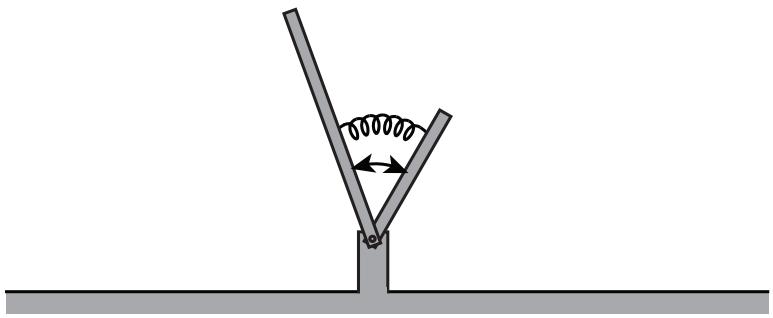


Figure 17.25: Two interconnected pendula pivotally mounted on a post.

Exercise 17.10 [Note: in this problem, all variables are taken in SI units, and all angles in radians.] Two pendula are mounted at their ends to a common pivot point on a post, as illustrated in Figure 17.25. The angles of the pendula, θ_1 and θ_2, are measured clockwise from vertical; in the configuration shown, $\theta_1 < 0$ and $\theta_2 > 0$. A linear spring, with spring constant k, and a motor, which can apply a torque u, are attached between the pendula around the pivot joint such that the equations of motion of the pendula are:

$$I_1 \frac{d^2\theta_1}{dt^2} = \frac{m_1\, g\, \ell_1}{2} \sin(\theta_1) + u + k(\theta_2 - \theta_1); \tag{17.90a}$$

$$I_2 \frac{d^2\theta_2}{dt^2} = \frac{m_2\, g\, \ell_2}{2} \sin(\theta_2) - u - k(\theta_2 - \theta_1). \tag{17.90b}$$

The pendula may be idealized as simple rods with masses $m_1 = 1$ and $m_2 = 3$ and lengths $\ell_1 = 2$ and $\ell_2 = 1$ and inertias $I_1 = m_1\ell_1^2/3$ and $I_2 = m_2\ell_2^2/3$, and we take $g \approx 10$ and $k = 6$.

(a) Linearize (17.90) about a possibly time-varying nominal state $\{\bar\theta_1(t), \bar\theta_2(t), \bar u(t)\}$, which itself solves (17.90), by taking $\theta_1 = \bar\theta_1 + \theta_1'$, $\theta_2 = \bar\theta_2 + \theta_2'$, and $u = \bar u + u'$ in (17.90), noting (B.46), (B.72), and (B.73), and, assuming the perturbation $\{\theta_1', \theta_2', u'\}$ is small, identifying a linear equation governing $\{\theta_1', \theta_2', u'\}$.

(b) Taking $\theta_1 = \bar{\theta}_1$, $\theta_2 = \bar{\theta}_2$, and $u = \bar{u}$ in (17.90) and assuming additionally that this nominal state is an *equilibrium* state (that is, $d\bar{\theta}_1/dt = d\bar{\theta}_2/dt = 0$), derive an equation that relates $\bar{\theta}_1$ and $\bar{\theta}_2$ (do not yet assume that $|\bar{\theta}_1|$ or $|\bar{\theta}_2|$ is small). Also, determine an expression for the corresponding $\bar{u}$ required to hold this equilibrium state as a function of $\bar{\theta}_1$ only. What is the largest value of $|\bar{\theta}_2|$ that can be held in equilibrium? If $|\bar{\theta}_1|$ and $|\bar{\theta}_2|$ are both small, what is the relationship between them in equilibrium?

(c) Linearizing (17.90a)-(17.90b) about the zero state $\bar{\theta}_1 = \bar{\theta}_2 = \bar{u} = 0$, compute the transfer function $G(s) = \Theta_2'(s)/U'(s)$. Where are the poles and zeros of this transfer function? (Note: we will consider the stabilization of this pathological unstable system in §18.3.6.) Using the Table 17.1a, compute the response of this linearized system, $\theta_2'(t)$, to a unit impulse $u'(t) = \delta(t)$ [and, thus, $U'(s) = 1$].

(d) Identify appropriate values of the physical parameters m_1, m_2, ℓ_1, and ℓ_2 such that the transfer function of this system has zeros at $s = \pm 2$ and poles at $s = \pm 1$ and ± 3.

Exercise 17.11 Sketch (using the rules of §17.5.1.1) and plot (using Matlab) the Bode plots of the following transfer functions, and identify whether each is strictly proper, semi proper, or improper:

(a) the (first-order) **low-pass filter** $G_1(s) = \dfrac{1}{s+1}$;

(b) the **PD filter** $G_2(s) = \dfrac{s-1}{1}$;

(c) the **all-pass filter** $G_3(s) = \dfrac{s-1}{s+1} = G_1(s)\,G_2(s)$ [appeal to Fact 17.15 together with (a) and (b) above];

(d) the **PID filter** $G_4(s) = \dfrac{s^2+101s+100}{s}$;

(e) the **bandpass filter** $G_4(s) = \dfrac{s}{s^2+101s+100}$;

(f) the (second-order) **low-pass filter** $G_6(s) = \dfrac{1}{s^2+s+1}$.

Exercise 17.12 Consider a DT system with transfer function

$$\frac{Y(z)}{U(z)} = G(z) = \frac{1}{z^2 - 0.1z} \tag{17.91}$$

(a) Compute the poles z_1 and z_2 of (17.91). Is this system stable, neutrally stable, or unstable? Explain.

(b) Compute the difference equation corresponding to (17.91). Assuming the system starts out at rest (that is, $y_k = 0$ for $k < 0$) and that $u_k = \delta_{0k}$, evaluate directly the response of this system for $k = 0, 1, 2, 3, 4$. Repeat for $u_k = h_{0k}$ [see (B.69b)].

(c) Taking an impulse input $u_k = \delta_{0k}$ [that is, taking $U(z) = 1$ in (17.91)], compute $\{c_1, c_2\}$ in the partial fraction expansion

$$Y(z) = G(z)U(z) = \frac{c_1}{z - z_1} + \frac{c_2}{z - z_2}. \tag{17.92}$$

Applying Table 17.1b to this result, compute the impulse response of (17.91) for all k. Evaluate the resulting expression for $k = 0, 1, 2, 3, 4$; does the result match the corresponding calculation in (b)? What does y_k approach as $k \to \infty$? Is this consistent with Fact 17.11 applied to $Y(z)$ in (17.91) when $U(z) = 1$?

(d) Taking a step input $u_k = h_{0k}$ [that is, taking, $U(z) = z/(z-1)$ in (17.91)], compute $\{d_1, d_2, d_3\}$ in the partial fraction expansion

$$Y(z) = G(z)U(z) = \frac{d_1}{z - z_1} + \frac{d_2}{z - z_2} + \frac{d_3}{z - 1}. \tag{17.93}$$

Applying Table 17.1b to this result, compute the step response of (17.91) for all k. Evaluate the resulting expression for $k = 0, 1, 2, 3, 4$; does the result match the corresponding calculation in (b)? What does y_k approach as $k \to \infty$? Is this consistent with Fact 17.11 applied to $Y(z)$ in (17.91) when $U(z) = z/(z-1)$?

References

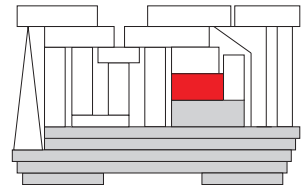
Franklin, G, Powell, JD, & Emami-Naeini, A (2006) *Feedback Control of Dynamic Systems*, Prentice Hall.

Gillespie, TD (1992) *Fundamentals of Vehicle Dynamics*, SAE.

LePage, WR (1961) *Complex Variables and the Laplace Transform for Engineers*, Dover.

Minto, KD, Chow, JH, Beseler, J (1989) Lateral Axis Autopilot Design For Large Transport Aircraft: An Explicit Model-Matching Approach *American Control Conference*, 585-591.

Stevens, BL, & Lewis, FL (2003) *Aircraft Control and Simulation*. Wiley.



Chapter 18

Classical control design

Contents

18.1 Closing the loop: an introduction to feedback control design . . . 528
18.1.1 Fundamental limitations† . . . 530
18.1.2 Simple parameterizations of all stabilizing controllers† . . . 531
18.2 Primary analysis tools used in classical control design . . . 533
18.2.1 The root locus with respect to K . . . 533
18.2.1.1 The simplified Routh and Bistritz tests† . . . 536
18.2.1.2 Approximate pole-zero cancellations in the LHP . . . 538
18.2.2 The Bode plot, revisited . . . 538
18.2.3 The Nyquist plot . . . 543
18.2.4 Final checks: the closed-loop step response and the closed-loop Bode plot . . . 546
18.2.4.1 System type and loop prefactors . . . 546
18.2.5 Extending the root locus, Bode, and Nyquist tools to DT systems . . . 547
18.3 Primary techniques used for classical control design . . . 548
18.3.1 PID (Proportional-Integral-Derivative) controllers . . . 548
18.3.1.1 Ad hoc PID tuning . . . 550
18.3.2 Lead, lag, and notch controllers . . . 553
18.3.3 Sensor dynamics, and noise suppression via low-pass and notch filtering . . . 558
18.3.4 Successive loop closure (SLC) leveraging frequency separation . . . 561
18.3.4.1 Nonminimum phase systems . . . 565
18.3.5 Stability and Control Augmentation Systems (SCAS) . . . 566
18.3.6 Unstable controllers for pathological SISO systems; pole placement . . . 569
18.3.7 Implementation of CT linear controllers in analog electronics† . . . 573
18.3.8 Extending the PID, lag, lead, low-pass, and notch techniques to DT systems . . . 573
18.3.9 Implementation of DT linear controllers in microcontrollers . . . 573
18.4 Classical control design of DT controllers for CT plants . . . 574
18.4.1 Discrete equivalent design . . . 575
18.4.2 Direct digital design . . . 576
18.4.3 Deadbeat control: pole placement at the origin for DT settling in finite time . . . 578
Exercises . . . 582

18.1 Closing the loop: an introduction to feedback control design

As illustrated in Figure 18.1, the problem of **feedback control design** is the problem of designing a **controller** [denoted $D(s)$ or $D(z)$] that coördinates the **control input**(s) **u** of the **plant**[1] [denoted $G(s)$ or $G(z)$] with the **measurement**(s) **y** of the plant in such a way as to *change* the dynamics exhibited by the system, optimizing some balance of the closed-loop system's **performance** (that is, the ability of the closed-loop system to track a desired **reference input r** with sufficient accuracy) and its **robustness** (that is, the insensitivity of the closed-loop system response to **state disturbances w**, **measurement noise v**, and **modeling errors** Δ in the plant itself). Of course, the performance and robustness measures of interest, as well as the balance between these two generally competing objectives, must be defined carefully in any given application. The large variety of possible systems that one might consider (ODE, PDE, DAE, linear, nonlinear, etc.), the wide range of performance and robustness measures that one might be interested in, the numerous balances between these measures that one might attempt to achieve, and various practical restrictions on actuator authority (saturation and bandwidth limits) and limitations in the controller design (sample time, decentralized communication architecture, computational complexity, and varying degrees of uncertainty of the plant itself) give rise to a rich variety of possible control strategies that have been and will continue to be developed. The brief introduction to the feedback control problem presented in the next few chapters, which surveys some of the key issues and foundational ideas, intends to serve as a prologue to a more in-depth study of this rich field.

A typical **performance specification** for an LTI system is the prescription of the minimum **rise time** and **settling time**, and the maximum **overshoot** and **steady-state error**, of the closed-loop system's step response (see Figure 17.12): that is, the response **y** to a step reference input **r** to the system depicted in Figure 18.1. A typical **robustness specification** for such a system is the minimization of the response of the system to external **state disturbances w** and **measurement noise v** that might otherwise disrupt the system. A balance between these simple performance and robustness specifications on the closed-loop behavior of a SISO LTI system forms a starting point for the study of the feedback control problem to be presented in this chapter. As a rule of thumb, a bit loosely stated, the following is a good starting point:

Guideline 18.1 *Apply just enough control feedback to narrowly achieve the performance specification.*

Pushing a system harder than this with control excitation generally degrades robustness to a host of possible unmodeled effects, as described in detail in §18.1.1. Time delays are especially dangerous, as discussed further in Examples 18.1 and 18.10.

In the CT SISO case, the Laplace transform of the output, $Y(s)$, is related to the Laplace transform of the input, $R(s)$, in the absence of state disturbances **w** and measurement noise **v** as follows:

$$Y(s) = G(s)U(s) = G(s)D(s)E(s) = G(s)D(s)[R(s) - Y(s)] \quad \Rightarrow \quad T(s) \triangleq \frac{Y(s)}{R(s)} = \frac{G(s)D(s)}{1 + G(s)D(s)}. \quad (18.1)$$

The matrix $T(s)$ [often, $H(s)$] is often referred to as "the" **closed-loop transfer function** of the system, and may be analyzed in order to ensure the desired performance specifications, such as those mentioned above. Note that, if $G(s)$ and $D(s)$ are rational functions of s, then we may write

$$G(s) = \frac{b(s)}{a(s)}, \quad D(s) = \frac{y(s)}{x(s)}, \quad \Rightarrow \quad T(s) = \frac{b(s)y(s)}{a(s)x(s) + b(s)y(s)}, \quad (18.2)$$

where $\{a(a), b(s), x(s), y(s)\}$ are polynomials. That is, $T(s)$ is a rational function of s as well. Thus, once simplified appropriately, the closed-loop transfer function $T(s)$ may be analyzed with the various techniques

[1] The ubiquitous use of the name **plant** for the system to be controlled is historical, reflecting the initial applications of linear systems theory to chemical process control.

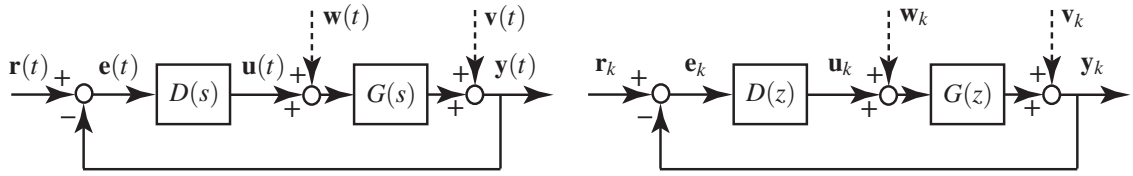


Figure 18.1: **Closed-loop** (a.k.a. **feedback**) connections of (left) a **continuous-time** (**CT**) LTI plant $G(s) = Y(s)/U(s)$ and controller $D(s) = U(s)/E(s)$, and (right) a **discrete-time** (**DT**) LTI plant $G(z) = Y(z)/U(z)$ and controller $D(z) = U(z)/E(z)$. In such **block diagrams**, for clarity, we denote the signals in the time domain and the systems in the transformed domain (that is, in **transfer function form**). The sign convention chosen at the leftmost summing junction is such that the **error signal** $\mathbf{e} = \mathbf{r} - \mathbf{y}$.

described in §17.3. It is, therefore, the roots of the *denominator* of $T(s)$ [that is, the **poles of the closed-loop transfer function**] which dictate the *nature* (that is, the speed of oscillation and the rate of exponential growth or decay) of each component of the response. The roots of the *numerator* of $T(s)$ [that is, the **zeros of the closed-loop transfer function**] dictate only the *coefficients* of each of these components. Via similar derivations, the following additional transfer functions may also be identified:

$$\frac{E(s)}{R(s)} = \frac{1}{1+G(s)D(s)} = S(s), \quad \frac{U(s)}{R(s)} = \frac{D(s)}{1+G(s)D(s)} = S_u(s), \quad \frac{Y(s)}{V(s)} = \frac{1}{1+G(s)D(s)} = S(s), \quad (18.3)$$
$$\frac{Y(s)}{W(s)} = \frac{G(s)}{1+G(s)D(s)} = S_i(s), \quad \frac{U(s)}{V(s)} = \frac{-D(s)}{1+G(s)D(s)} = -S_u(s), \quad \frac{U(s)}{W(s)} = \frac{-G(s)D(s)}{1+G(s)D(s)} = -T(s).$$

These transfer functions may be analyzed in order to ensure the desired robustness specifications, such as those mentioned above. As a mnemonic, the closed-loop transfer function of any feedback loop, including the several transfer functions listed above, is given by

$$\textbf{transfer function} = \frac{\textbf{forward gain}}{1 - (\textbf{loop gain})}.$$

Note also that the poles of the closed-loop transfer function $T(s)$ in (18.1) and (18.2) are given by

$$1 + G(s)D(s) = 0 \qquad \Leftrightarrow \qquad a(s)x(s) + b(s)y(s) = 0. \quad (18.4)$$

Once the transfer function of a CT controller, $D(s) = U(s)/E(s)$, is designed, it is easy to compute the inverse Laplace transform of $D(s)$ to find the **differential equation** [relating $\mathbf{u}(t)$ to $\mathbf{e}(t)$] that the controller must obey. It is also straightforward (see §19) to build an **electric circuit** that obeys this differential equation with a very inexpensive arrangement of resistors, capacitors, and operational amplifiers.

In the DT case, the closed-loop transfer functions are derived in an *identical* fashion as in the CT case, with the role of z replacing that of s. Thus, the closed-loop transfer function considered to analyze performance is

$$T(z) \triangleq \frac{Y(z)}{R(z)} = \frac{G(z)D(z)}{1+G(z)D(z)},$$

and the closed-loop transfer functions considered to characterize robustness again include all those defined in (18.3), with z replacing s. As in the CT case, once simplified appropriately, these closed-loop transfer functions are rational functions of z, and thus may be analyzed with the techniques described in §17.4. Furthermore, once the transfer function of a DT controller, $D(z)$, is designed, it is easy to compute the inverse Z transform of $D(z)$ to determine the **difference equation** [relating $\mathbf{u}_k$ to $\mathbf{e}_k$] that the corresponding controller must obey. It is also straightforward to implement this difference equation on an inexpensive **microcontroller**.

18.1.1 Fundamental limitations†

As discussed above, the following four **sensitivities** are inextricably linked:

$$\begin{aligned} S(s) &= \frac{1}{1+G(s)D(s)} = \frac{E(s)}{R(s)} = \frac{Y(s)}{V(s)}, & S_u(s) &= \frac{D(s)}{1+G(s)D(s)} = -\frac{U(s)}{V(s)}, \\ T(s) &= \frac{G(s)D(s)}{1+G(s)D(s)} = \frac{Y(s)}{R(s)} = -\frac{U(s)}{W(s)}, & S_i(s) &= \frac{G(s)}{1+G(s)D(s)} = \frac{Y(s)}{W(s)}; \end{aligned} \tag{18.5}$$

$S(s)$ is called the **sensitivity**, $T(s)$ is called the **complementary sensitivity** (or, at other times, the **closed-loop transfer function**), $S_u(s)$ is called the **control sensitivity**, and $S_i(s)$ is called the **output sensitivity**. Note in particular that, in SISO systems,

- $Y(s)/R(s) = -U(\mathrm{i}\omega)/W(\mathrm{i}\omega)$; thus, the response of the control $U(\mathrm{i}\omega)$ to state disturbances $W(\mathrm{i}\omega)$ is suppressed only at those frequencies ω for which the closed-loop tracking is poor.
- $S_u(s) = D(s)/[1+G(s)D(s)] = T(s)/G(s)$; thus, at frequencies characterized by good tracking ($T(\mathrm{i}\omega) \approx 1$) but low plant gain ($|G(\mathrm{i}\omega)| \ll 1$), large control gains $D(\mathrm{i}\omega) \gg 1$ are required.
- $S(s) = 1 - T(s)$; thus, the sensitivity $Y(\mathrm{i}\omega)/V(\mathrm{i}\omega)$ is suppressed only at those frequencies ω for which the complementary sensitivity $U(\mathrm{i}\omega)/W(\mathrm{i}\omega)$ is not.

The first two of these points directly motivate Guideline 18.1.

The sensitivities defined in (18.5) are related such that $S_i(s) = S(s)G(s)$ and $T(s) = S_u(s)G(s)$. Thus:

Fact 18.1 (Internal stability of SISO systems) *A closed-loop SISO system is said to be* **internally stable** *if the sensitivities* $\{T(s), S(s), S_u(s), S_i(s)\}$ *are all stable; in such systems,*

(a) the poles of $G(s)$ appear either as zeros of $S(s)$ or (if they are in the LHP) possibly as poles of $S_i(s)$;
(b) the zeros of $G(s)$ appear either as zeros of $S_i(s)$ or (if they are in the LHP) possibly as poles of $S(s)$;
(c) the poles of $G(s)$ appear either as zeros of $S_u(s)$ or (if they are in the LHP) possibly as poles of $T(s)$;
(d) the zeros of $G(s)$ appear either as zeros of $T(s)$ or (if they are in the LHP) possibly as poles of $S_u(s)$.

Taking a perturbed plant $G_\Delta(s) = G(s)\cdot\Delta(s)$ with (multiplicative) modeling errors $\Delta(s)$, and defining

$$\delta(s) = 1/[1+T(s)\Delta(s)], \tag{18.6}$$

it follows from their definitions in (18.5) that the sensitivities of the perturbed plant $G_\Delta(s)$ are given by

$$S_\Delta(s) = S(s)\,\delta(s), \tag{18.7a}$$
$$T_\Delta(s) = T(s)\,[1+\Delta(s)]\,\delta(s), \tag{18.7b}$$
$$S_{u\Delta}(s) = S_u(s)\,\delta(s), \tag{18.7c}$$
$$S_{i\Delta}(s) = S_i(s)\,[1+\Delta(s)]\,\delta(s). \tag{18.7d}$$

Noting (18.6), it is seen that good tracking [$T(\mathrm{i}\omega) \approx 1$] implies $O(1)$ susceptibility of all four of the sensitivity functions to destabilizing multiplicative modeling errors, risking instability. Modeling errors $\Delta(\mathrm{i}\omega)$ generally increase in magnitude with rising frequency ω; thus, to decrease the risk of closed-loop instability due to such modeling errors, the closed-loop bandwidth ω_{BW} [that is, the frequency ω above which $T(\mathrm{i}\omega)$ drops off] should be made as low as possible, again motivating Guideline 18.1.

Another class of fundamental limitation arise by integrating the log of the sensitivity over all frequencies. An example limitation of this class is given in Fact 18.2 below [recall from §17 that a stable CT transfer function has all of its poles in the LHP, that the **relative degree** n_r of a transfer function is the degree of the polynomial in its denominator minus the degree of the polynomial in its numerator, and that a transfer function is said to be **proper** if $n_r \geq 0$, and **strictly proper** if $n_r > 0$]:

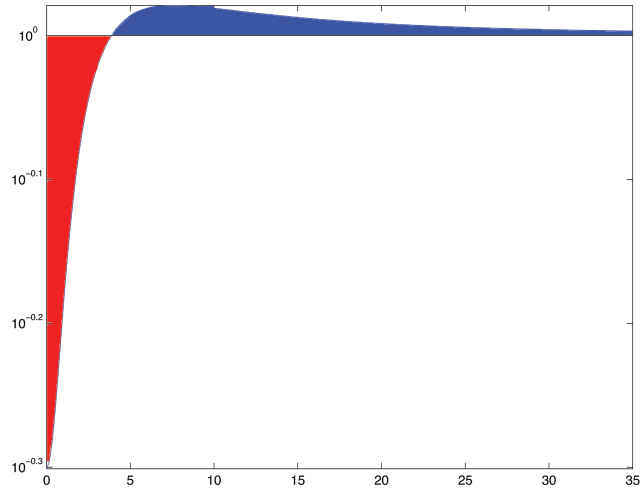


Figure 18.2: Semilogy plot of the log of the sensitivity $S(s) = 1/[1+L(s)]$ for $L(s) = K/[(s+1)(s+10)]$ with $K = 10$, illustrating Bode's Integral Theorem: that is, the (red) region below the $\ln|S(\mathrm{i}\omega)| = 0$ line has the same area as the (blue) region above it, and thus $\int_0^\infty \ln|S(\mathrm{i}\omega)|\,d\omega = 0$, independent of K.

Fact 18.2 (Bode's Integral Theorem) *Consider a stable strictly proper open-loop system $G(s)D(s) = L(s)$ with relative degree $n_r > 0$. Define κ as the initial value of the impulse response of $L(s)$ via the CT initial value theorem (Fact 17.5) such that $\kappa = \lim_{s\to\infty} sL(s)$; note that κ is finite if $n_r = 1$, and zero if $n_r > 1$. It follows that*

$$\int_0^\infty ln|S(i\omega)|\,d\omega = -\kappa\pi/2.$$

Proof of Bode's Integral Theorem is given in Example B.1. As shown in Figure 18.2, for a given value of κ [for example, if $n_r > 1$, then $\kappa = 0$, independent of $D(s)$], Bode's integral theorem may be understood geometrically as the "**waterbed effect**": if the magnitude of the sensitivity $S(\mathrm{i}\omega)$ is reduced over some frequencies, it is necessarily increased over other frequencies in such a manner that its integral over all frequencies is $-\kappa\pi/2$, independent of how you might adjust the controller $D(s)$, thus illustrating another fundamental tradeoff. In other words, attempts to reduce the magnitude of the sensitivity $S(\mathrm{i}\omega) = Y(\mathrm{i}\omega)/V(\mathrm{i}\omega)$ over all frequencies ω is, in a certain well-defined sense, a **zero-sum game**: if the sensitivity is reduced (improved) at some frequencies, it is increased (worsened) at other frequencies. However, the measurement noise $V(\mathrm{i}\omega)$ is generally most pronounced at higher frequencies, thus again motivating Guideline 18.1.

18.1.2 Simple parameterizations of all stabilizing controllers†

We will focus in §18 on the construction and tuning of stabilizing controllers for stable and unstable plants. In some cases (see in particular §18.9), identifying a controller $D(s)$ that results in a stable $T(s)$ is itself a somewhat difficult problem, and having a simple parameterization of all stabilizing controllers is handy. The difficulty is related to the fact that the relationship between $D(s)$ and $T(s)$ in (18.1) is nonlinear:

$$T(s) = \frac{G(s)D(s)}{1+G(s)D(s)} \quad \Leftrightarrow \quad D(s) = \frac{1}{G(s)} \cdot \frac{T(s)}{1-T(s)}.$$

If we instead write $T(s) = Q(s)G(s)$ and design $Q(s)$ corresponding to a stable $T(s)$, the control design problem becomes much easier. Once the desired $Q(s)$ is specified, the corresponding $D(s)$ is easily found:

$$T(s) = Q(s)G(s), \quad Q(s) = \frac{D(s)}{1+G(s)D(s)} \quad \Rightarrow \quad D(s) = \frac{Q(s)}{1-G(s)Q(s)}. \tag{18.8}$$

Further, $S(s) = 1 - Q(s)G(s)$, $S_u(s) = Q(s)$, and $S_i(s) = [1 - Q(s)G(s)]G(s)$. It follows (see Fact 18.1) that

Fact 18.3 *If $G(s)$ is stable and proper, the set of all proper controllers $D(s)$ that give an internally-stable closed loop may be written in the form given in (18.8) for all rational $Q(s)$ that are stable and proper.*

If $G(s) = b(s)/a(s)$ is **coprime** [that is, if $b(s)$ and $a(s)$ have no common factors] and proper but possibly has both RHP zeros and poles (and is thus possibly unstable), the construction in (18.8) is insufficient to assure an internally-stable closed loop. In this case, we instead consider a controller of the form:

$$D(s) = \frac{y(s) + a(s)\overline{Q}(s)}{x(s) - b(s)\overline{Q}(s)} \tag{18.9a}$$

where the polynomials $\{x(s), y(s)\}$ solve an associated **Diophantine equation** (see §B.2)

$$a(s)x(s) + b(s)y(s) = f(s), \tag{18.9b}$$

where $\{a(s), x(s), b(s), y(s), f(s)\}$ are polynomials, $\overline{Q}(s)$ is a rational transfer function that is stable and proper but otherwise arbitrary, and $f(s)$ has all of its roots in the LHP and is of sufficiently high order that a proper $y(s)/x(s)$ solving (18.9b) exists, but is otherwise arbitrary. The sensitivities of interest may now be written

$$T(s) = G(s)D(s)/[1+G(s)D(s)] = Q(s)G(s) = b(s)[y(s)+\overline{Q}(s)a(s)]/f(s), \tag{18.10a}$$

$$S(s) = 1/[1+G(s)D(s)] = 1-Q(s)G(s) = a(s)[x(s)-\overline{Q}(s)b(s)]/f(s), \tag{18.10b}$$

$$S_u(s) = D(s)/[1+G(s)D(s)] = Q(s) = a(s)[y(s)+\overline{Q}(s)a(s)]/f(s), \tag{18.10c}$$

$$S_i(s) = G(s)/[1+G(s)D(s)] = [1-Q(s)G(s)]G(s) = b(s)[x(s)-\overline{Q}(s)b(s)]/f(s); \tag{18.10d}$$

all four of these forms are stable as long as $\overline{Q}(s)$ is stable and $f(s)$ has its roots in the LHP. Note the relationship between $Q(s)$ and $\overline{Q}(s)$ in the third line above. Note also that

- the factor of $a(s)$ in the numerator of the expressions for $S(s)$ and $S_u(s)$, which implies that the poles of $G(s)$ appear as zeros of $S(s)$ and $S_u(s)$, thus satisfying parts (a) and (c) of Fact 18.1, and
- the factor of $b(s)$ in the numerator of the expressions for $T(s)$ and $S_i(s)$, which implies that the zeros of $G(s)$ appear as zeros of $T(s)$ and $S_i(s)$, thus satisfying parts (b) and (d) of Fact 18.1.

It follows that:

Fact 18.4 *If $G(s)$ is coprime and proper but possibly has RHP poles and zeros, the set of all proper controllers $D(s)$ that give an internally-stable closed loop may be written in the form given in (18.9a) for some $\{x(s), y(s)\}$ that solves (18.9b) [for some $f(s)$ with all of its roots in the LHP of sufficiently high order that (18.9b) is solvable] and for all rational $\overline{Q}(s)$ that are stable and proper.*

It is easily seen that (18.8) is a special case of (18.9) for $G(s)$ that are stable and proper; indeed, taking $f(s) = a(s)$ in this case results in $x(s) = 1$ and $y(s) = 0$ and $Q(s) = \overline{Q}(s)a(s)$, thus reducing (18.9a) to (18.8).

The discussion above extends immediately to DT problems simply by replacing s with z, replacing the phrase "LHP" with "inside the unit circle", and replacing the phrase "RHP" with "outside the unit circle".

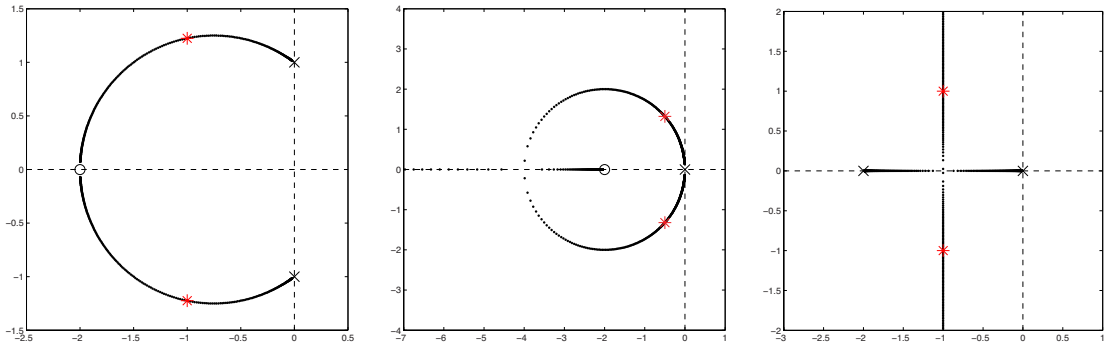


Figure 18.3: (dots) Root loci of three systems with proportional feedback $[D(s)=K]$ applied, with $\times$ marking the open-loop poles, $\circ$ marking the open-loop zeros, and $*$ marking the closed-loop poles for $K=1$. Systems considered are: (left) $G(s)=(s+2)^2/(s^2+1)$, (center) $G(s)=(s+2)/s^2$, and (right) $G(s)=2/(s^2+2s)$.

18.2 Primary analysis tools used in classical control design

We now consider the four essential tools of classical feedback control design: the **root locus** (§18.2.1), the **Bode plot** (§18.2.2), the **Nyquist plot** (§18.2.3), and the **closed-loop Bode plot** (§18.2.4); these four tools may be used in a deliberate fashion[2] along with the **step response** (see Example 17.12) for effective CT control design. To introduce these tools, we consider first (in §18.2) the closing of feedback loops around various simple CT plants with a **proportional control** strategy which sets the control input proportional to the error signal [that is, $u(t)=Ke(t)$, and thus $D(s)=K$]. In §18.2.5, we discuss how these tools may be extended to DT systems. In §18.3 and 18.3.8, we show how these tools may be used to tune more sophisticated control designs which more precisely target the dynamics of interest in the system under consideration.

18.2.1 The root locus with respect to K

Recall from (18.4) that, if $G(s)D(s)=L(s)=Kb(s)/a(s)$, then the poles of the closed-loop transfer function $T(s)$ are given by[3] $a(s)+Kb(s)=0 \;\Leftrightarrow\; a(s)/K+b(s)=0$. Thus,

- for small K, the poles of $T(s)$ are near the *poles* of $[G(s)D(s)]$ (i.e., the values of s with $a(s)=0$),
- for large K, the poles of $T(s)$ are near the *zeros* of $[G(s)D(s)]$ (i.e., the values of s with $b(s)=0$), and
- for intermediate K, the poles are in-between, moving continuously as K is increased (see §B.3.5).

A plot reflecting the movement of the closed-loop poles of a system as a parameter in the controller (in this case, K) is varied (in this case, from zero to infinity) is known as a **root locus**. A root locus with respect to the gain K is the most common type of root locus encountered, and is commonly referred to as "the" root locus of the system; root loci with respect to other controller parameters are considered in Exercise 18.2.

Root loci with respect to K of some simple plants $G(s)$ with proportional control $D(s)=K$ applied are illustrated in Figures 18.3 and 18.4. The simple code used to produce these plots is given in Algorithm 18.1. If $[G(s)D(s)]$ is strictly proper, with n poles and m finite zeros where $n>m$, it may be argued (see Figure 18.4) that the "missing zeros" that $(n-m)$ branches of the locus approach for large K are at the "point at infinity" in the **extended complex plane**, a notion which is most clearly understood when the extended complex plane is mapped onto the **Riemann sphere** (see Figure B.2).

[2] To ensure the controller excites the system as little as possible while meeting the control objectives (see Guideline 18.1), one is highly discouraged from applying the **random control design** (**RCD**) strategies facilitated by root locus **graphical user interfaces** (**GUIs**), which almost encourage one to plunk a controller pole here and zero there until, perhaps accidentally, stability is achieved.

[3] Thus, e.g., if $a(s)=s^2+1$ and $b(s)=(s+2)^2$ (see Figure 18.3a), then $(s^2+1)+K(s+2)^2=0$, and thus the closed-loop poles are located, as a function of K, at $s=[-4K\pm\sqrt{(4K)^2-4(K+1)(4K+1)}]/[2(K+1)]$.

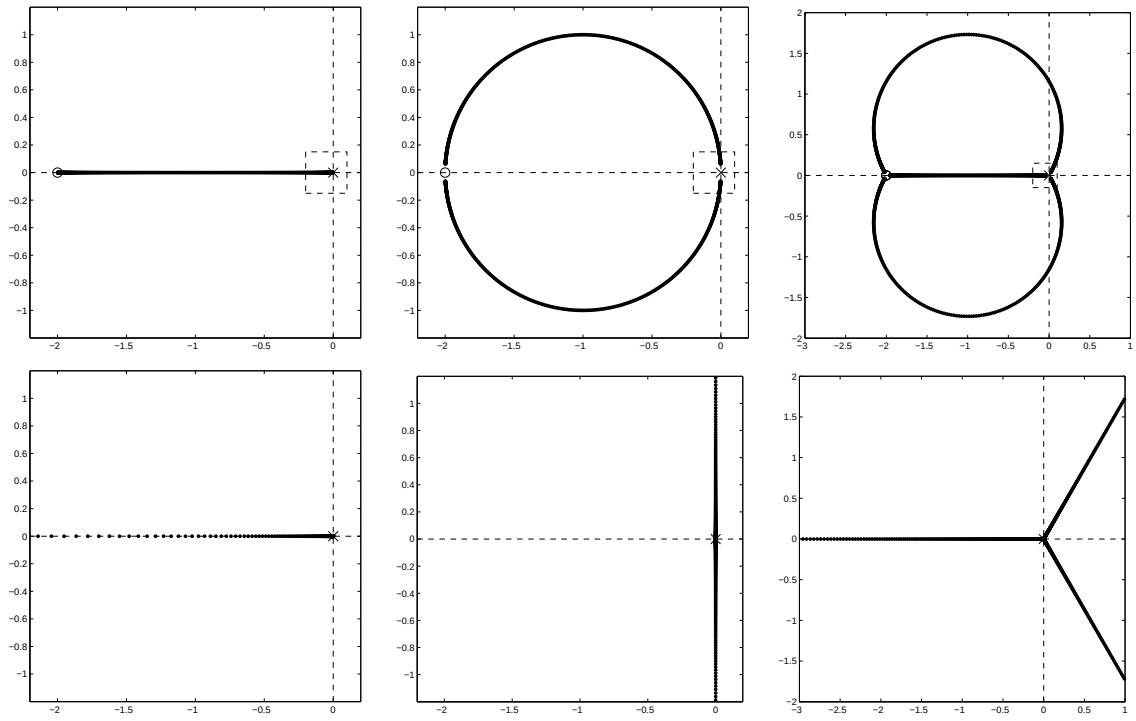


Figure 18.4: Root loci of (left) $L(s) = G(s)D(s) = K(s+c)/(cs)$, (center) $L(s) = K(s+c)^2/(cs)^2$, and (right) $L(s) = K(s+c)^3/(cs)^3$, taking (top) $c = 2$ and (bottom) the limit as $c \to \infty$. As c is increased, the location of the zero(s), towards which the closed-loop pole(s) converge for large K, moves off to the left towards infinity, and the root locus in the vicinity of the origin is dominated by the pattern outlined by the small dashed box. Thus, in the limit as $c \to \infty$, the three root loci approach those of $L(s) = K/s$, $L(s) = K/s^2$, and $L(s) = K/s^3$, respectively. This explanation helps one to visualize that the zeros at infinity are, in a sense, *all at the same place*—a place which one may refer to as the north pole if the s plane is conformally mapped onto a Riemann sphere, with the origin of the s plane mapped to the south pole (see Figure B.2).

Root loci are valuable for understanding parametric dependencies during feedback control design. Though easily plotted using, e.g., Algorithm 18.1 (and fairly easily parameterized analytically in the second-order case, as discussed in Footnote 3 on Page 533), it is sometimes useful to know how to quickly sketch a root locus with respect to the overall gain K by hand (even for systems of order higher than second) in order to anticipate how the closed-loop poles of a system change when the controller is modified, or (as discussed in §18.3 through §18.4) to understand how to modify a controller to change a root locus in a desired manner, thereby moving the poles of a closed-loop system into the region suggested by the approximate design guides illustrated in Figure 17.13. Some simple rules for sketching root loci by hand are thus outlined below.

Sketching root loci with respect to K by hand

To proceed, assume $L(s) = G(s)\,D(s)$ is a transfer function with m complex zeros z_k and n complex poles p_k, where $m \leq n$ (see §17.3.3.1), such that

$$L(s) = G(s)\,D(s) = K\,\frac{b(s)}{a(s)} = K\,\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}.$$

We also define the multiplicity of the k'th pole as q_k and the multiplicity of the k'th zero as r_k.

Algorithm 18.1: Code for plotting a root locus with respect to K.

View
Test

```
function RLocus(numG,denG,numD,denD,g)
% Plot root locus of K*D(s)*G(s) w.r.t. the extra gain K, where D(s)=numD(s)/denD(s) and
% G(s)=numG(s)/denG(s). The derived type g groups together convenient plotting parameters:
% g.K is the gains used, and g.axes is the axis limits. The roots for K=1 are marked (*).
[numL,denL]=RationalSimplify(PolyConv(numG,numD),PolyConv(denG,denD));
MS='MarkerSize'; clf, hold on
for j=1:length(g.K); denH=PolyAdd(g.K(j)*numL,denL);
  Hpoles=roots(denH); plot(real(Hpoles),imag(Hpoles),'k.',MS,6)
end,
Gpoles=roots(denG); plot(real(Gpoles),imag(Gpoles),'kx',MS,17)
Gzeros=roots(numG); plot(real(Gzeros),imag(Gzeros),'ko',MS,12)
Dpoles=roots(denD); plot(real(Dpoles),imag(Dpoles),'bx',MS,17)
Dzeros=roots(numD); plot(real(Dzeros),imag(Dzeros),'bo',MS,12)
Hpoles=roots(PolyAdd(numL,denL)); plot(real(Hpoles),imag(Hpoles),'r*',MS,17)
axis equal, a=g.axes; axis(a), plot([a(1) a(2)],[0 0],'k-'), plot([0 0],[a(3) a(4)],'k-')
end % function RLocus
```

The **180° root locus with respect to** K [i.e., noting (18.4), the locus of all points s such that $1+L(s)=0$ when $K>0$] may be sketched using the following handy rules (Evans 1950):

1. Mark the n poles p_i (the roots of $a(s)=0$) with an $\times$ and the m zeros z_i (the roots of $b(s)=0$) with an $\circ$.

2. Draw the locus on the real axis to the left of an odd number of real poles plus zeros counted from the right.

3. The branches of the locus **depart** (that is, start) from the open-loop poles and **arrive** (that is, end) at the open-loop zeros. If $n>m$, then $n-m$ branches extend to infinity, approaching $n-m$ asymptotes centered at $\alpha=\frac{\sum p_i-\sum z_i}{n-m}$ and departing at angles $\theta_\ell=\frac{180^\circ+(\ell-1)360^\circ}{n-m}$ for $\ell=1,\ldots,n-m$ (see, e.g., Figure 18.4d-f for $n-m=1$, 2, and 3).

4. For any open-loop pole p_k of multiplicity q_k, or zero z_k of multiplicity r_k, define ψ_i as the angle from the i'th pole to this point, and ϕ_i as the angle from the i'th zero to this point. Then one may calculate the

   a. **departure angles** $\psi_{\rm dep}^{k,\ell}$ from the open-loop pole p_k as $\psi_{\rm dep}^{k,\ell}=\frac{\sum\phi_i-\sum_{i\neq k}\psi_i+180^\circ+(\ell-1)360^\circ}{q_k}$ for $\ell=1,\ldots,q_k$.

   b. **arrival angles** $\phi_{\rm arr}^{k,\ell}$ to the open-loop zero z_k as $\phi_{\rm arr}^{k,\ell}=\frac{\sum_{i\neq k}\phi_i-\sum\psi_i+180^\circ+(\ell-1)360^\circ}{r_k}$ for $\ell=1,\ldots,r_k$.

5. In the case that two (resp., three) branches of the locus touch at a point, the angle between the branches of the locus at this junction is 90° (resp., 60°), and the branches into and out of the junction alternate.

The **0° root locus with respect to** K [i.e.,, the locus of all points s such that $1+L(s)=0$ with $K<0$] may be sketched by modifying the above rules as follows:

2. Draw the locus on the real axis to the left of an *even* number of real poles plus zeros counted from the right.

3. The asymptotes depart at angles $\theta_\ell=\frac{(\ell-1)360^\circ}{n-m}$ for $\ell=1,\ldots,n-m$.

4a. The departure angles from the open-loop pole p_k are $\psi_{\rm dep}^{k,\ell}=\frac{\sum\phi_i-\sum_{i\neq k}\psi_i+(\ell-1)360^\circ}{q_k}$ for $\ell=1,\ldots,q_k$.

4b. The arrival angles to the open-loop zero z_k are $\phi_{\rm arr}^{k,\ell}=\frac{\sum_{i\neq k}\phi_i-\sum\psi_i+(\ell-1)360^\circ}{r_k}$ for $\ell=1,\ldots,r_k$.

Note that, for any K, the closed-loop poles are characterized by $1+Kb(s)/a(s)=0$, and thus

$$K=-a(s)/b(s); \tag{18.11}$$

this useful formula gives the value of K corresponding to any given point s on the locus[4].

[4]Note that K must be real; if a point s on the locus is only known approximately, application of (18.11) might result in a complex value of K; taking the real part of this value results in a nearby point on the locus.

Further, in the vicinity of any **breakaway point** of the root locus (that is, any point where two or more branches of the locus connect to the real axis), the value of K given by (18.11) reaches either a local maximum or a local minimum with respect to the real value of s as you move to the left or right on the real axis. Thus, considering $-\infty < K < \infty$, the set of all breakaway points of the 180° *and* 0° root loci on the real axis are characterized by setting the derivative of (18.11) equal to zero:

$$\frac{dK}{ds} = 0 \quad \Rightarrow \quad -\frac{a'(s)}{b(s)} + \frac{a(s)\,b'(s)}{[b(s)]^2} = 0 \quad \Rightarrow \quad a'(s)\,b(s) - a(s)\,b'(s) = 0; \tag{18.12a}$$

the breakaway points are given by taking the roots of the polynomial at right in (18.12a). Dividing this formula by $[a(s)\,b(s)]$ and taking $b(s) = (s - z_1)\cdots(s - z_m)$ and $a(s) = (s - p_1)\cdots(s - p_n)$ gives the alternative form

$$\sum_{i=1}^{n} \frac{1}{s - p_i} - \sum_{i=1}^{m} \frac{1}{s - z_i} = 0. \tag{18.12b}$$

With a little practice, one can usually sketch a root locus quite accurately and quickly by hand by following the above several rules. Note that the connection of the various branches of a root locus sometimes change rather rapidly (but smoothly! see §B.3.5) as the open loop poles and zeros are moved, as seen in Figure 18.5. Such **rapid reconnections** of the root locus are somewhat difficult to anticipate when sketching the locus by hand; computation of the breakaway points is often helpful. Note, however, that these reconnections of the locus in fact have little impact on the closed-loop behavior of the system; the two systems depicted in Figure 18.5 have essentially identical step responses for all values of K.

18.2.1.1 The simplified Routh and Bistritz tests†

The **Routh test** is a relatively simple procedure for counting how many roots of a polynomial $p(s)$ are in the LHP, on the imaginary axis, and in the RHP [often referred to as the **inertia** of $p(s)$], without requiring the computation of the roots themselves, which can be computationally expensive. The test is developed in its entirety in §B.3.6, and a simple code implementing it is provided in Algorithm B.2.

When applying the Routh test to the polynomial in the denominator of a CT (open-loop or closed-loop) transfer function simply to determine whether or not all of the roots of this polynomial are in the LHP (and thus $p(s)$ is **Hurwitz**, and the corresponding CT system is stable), the **simplified Routh test** may be considered, as defined by the following three-term recurrence:

$$u_n(s) = u_{n,n}\,s^n + u_{n,n-2}\,s^{n-2} + \ldots \qquad = p_n\,s^n + p_{n-2}\,s^{n-2} + \ldots, \tag{18.13a}$$

$$u_{n-1}(s) = u_{n-1,n-1}\,s^{n-1} + u_{n-1,n-3}\,s^{n-3} + \ldots = p_{n-1}\,s^{n-1} + p_{n-3}\,s^{n-3} + \ldots, \tag{18.13b}$$

$$u_i(s) = u_{i+2}(s) - \alpha_i\,s\,u_{i+1}(s) \;\text{ where }\; \alpha_i = u_{i+2,i+2}/u_{i+1,i+1} \;\text{ for }\; i = n-2, n-3, \ldots, 0; \tag{18.13c}$$

all of the roots of $p(s)$ are in the open LHP iff $\{u_{n,n}, u_{n-1,n-1}, \ldots, u_{1,1}, u_{0,0}\}$ are all nonzero and the same sign. The coefficients of this recurrence may, for $n = 7$, be organized in the following table form:

p_n	p_{n-2}	p_{n-4}	p_{n-6}	odd (even) terms of original polynomial if n is odd (even),
p_{n-1}	p_{n-3}	p_{n-5}	p_{n-7}	even (odd) terms of original polynomial if n is odd (even),
$u_{n-2,n-2}$	$u_{n-2,n-4}$	$u_{n-2,n-6}$		$u_{n-2,i} = p_i - (p_n/p_{n-1})\,p_{i-1}$ for $i = n-2, n-4, n-6$,
$u_{n-3,n-3}$	$u_{n-3,n-5}$	$u_{n-3,n-7}$		$u_{n-3,i} = p_i - (p_{n-1}/u_{n-2,n-2})\,u_{n-2,i-1}$ for $i = n-3, n-5, n-7$,
$u_{n-4,n-4}$	$u_{n-4,n-6}$			$u_{n-4,i} = u_{n-2,i} - (u_{n-2,n-2}/u_{n-3,n-3})\,u_{n-3,i-1}$ for $i = n-4, n-6$,
$u_{n-5,n-5}$	$u_{n-5,n-7}$			$u_{n-5,i} = u_{n-3,i} - (u_{n-3,n-3}/u_{n-4,n-4})\,u_{n-4,i-1}$ for $i = n-5, n-7$,
$u_{n-6,n-6}$				$u_{n-6,i} = u_{n-4,i} - (u_{n-4,n-4}/u_{n-5,n-5})\,u_{n-5,i-1}$ for $i = n-6$,
$u_{n-7,n-7}$				$u_{n-7,i} = u_{n-5,i} - (u_{n-5,n-5}/u_{n-6,n-6})\,u_{n-6,i-1}$ for $i = n-7$;

all of the roots of $p(s)$ are in the open LHP iff the terms in the first column are all nonzero and the same sign. Lower-order cases (with $n < 7$) may be written in table form by removing all terms with negative indices above; higher-order cases (with $n > 7$) may be written in table form by adding the appropriate terms.

Algorithm 18.2: The Simplified Routh test.

View
Test

```
function RouthSimplified(p)
% Compute the simplified Routh table to determine if p(s) is Hurwitz. p(s) may be symbolic.
n=length(p)-1; s=strcmp(class(p),'sym'); R=0; f=1; disp(p(1:2:end));
for i=n:-1:1
  disp(p(2:2:end)), if p(2)==0, disp('Not Hurwitz.'), f=0; break, end
  a=p(1)/p(2); p(3:2:i)=p(3:2:i)-a*p(4:2:i+1); p=p(2:i+1); if ~s, if a<0; R=R+1; end, end
end
if f, if s, disp('Hurwitz iff all entries in first column are the same sign.'), else
if R==0, disp('Hurwitz'), else, disp(['Not Hurwitz: ',num2str(R),' RHP poles']),end,end,end
end % function RouthSimplified
```

A useful feature of the simplified Routh test, as implemented in Algorithm 18.2, is that one can carry (one or more) *variables* in a control design formulation, such as the controller gain K, all the way through the test, thereby determining necessary and sufficient conditions on these variables for closed-loop stability. This can sometimes assist in the drawing of an accurate root locus plot.

The simplified Routh test described above determines whether or not all of the poles of a transfer function $T(s)$ are in the open LHP, a condition sometimes referred to as **absolute stability**. A stricter condition that is sometimes useful to determine is whether or not all of the poles of a transfer function $T(s)$ have a real part to the left of $s=-\sigma$ for some $\sigma>0$, a condition referred to as **relative stability**. This is easy to determine by applying the simplified Routh test discussed above to the modified transfer function $T(s+\sigma)$.

The **Bistritz test**[5] is a relatively simple procedure for counting how many roots of a polynomial $p(z)$ are inside the unit circle, on the unit circle, and outside the unit circle [referred to as the **stationarity** of $p(z)$], without requiring the computation of the roots themselves, which can be computationally expensive. The test is developed in its entirety in §B.3.7, and a simple code implementing it is developed in Algorithm B.3.

When applying the Bistritz test to the polynomial in the denominator of a DT (open-loop or closed-loop) transfer function simply to determine whether or not all of the roots of this polynomial are inside the unit circle (and thus $p(z)$ is **Schur**, and the corresponding DT system is stable), the **simplified Bistritz test** may be considered, as defined by the following three-term recurrence [cf. (18.13)]:

$$u_n(z) = p(z) + p^r(z) \qquad \text{where } p^r(z) = z^n\, p(1/z) = p_0 z^n + p_1 z^{n-1} + \ldots + p_n, \tag{18.14a}$$

$$u_{n-1}(z) = [p(z) - p^r(z)]/(z-1) \qquad \text{polynomial division (note: can easily do by hand),} \tag{18.14b}$$

$$u_i(z) = [\alpha_i (z+1)\, u_{i+1}(z) - u_{i+2}(z)]/z \qquad \text{where } \alpha_i = u_{i+2}(0)/u_{i+1}(0) \text{ for } i = n-2, n-3, \ldots, 0; \tag{18.14c}$$

all of the roots of $p(z)$ are inside the unit circle iff $\{u_n(0), u_{n-1}(0), \ldots, u_1(0), u_0(0)\}$ are all nonzero and the same sign. Note in this test that each of the $u_i(z)$ are **symmetric** (that is, $u_{i,i-j} = u_{i,j}$ for for $i = 0, \ldots, n$ and $j = 1, \ldots, i$), so the work associated with this test is similar to that associated with the simplified Routh test described previously, in which each of the $u_i(s)$ considered is either **even** or **odd**.

A useful feature of the simplified Bistritz test, as implemented in Exercise 18.12, is that one can carry one or more *variables* in the control design formulation, such as the controller gain K, all the way through the test, thereby determining necessary and sufficient conditions on these variables for closed-loop stability.

The simplified Bistritz test described above determines whether or not all of the poles of a transfer function $T(z)$ are inside the unit circle, a condition sometimes referred to as **absolute stability**. A stricter condition that is sometimes useful to determine is whether or not all of the poles of a transfer function $T(z)$ are inside a circle of some radius $r<1$, a condition referred to as **relative stability**. This is easy to determine by applying the simplified Bistritz test discussed above to the modified transfer function $T(z/r)$.

[5] The **Schur-Cohn test** and **Jury test** are alternatives to the Bistritz test for determining whether or not $p(z)$ is Schur. We focus on the Bistritz test in this discussion due to its strong similarity to the Routh test.

18.2.1.2 Approximate pole-zero cancellations in the LHP

Consider a plant and controller given by

$$G(s)=\frac{1}{s(s+0.95)},\quad D(s)=10\frac{s+1}{s+4}\quad\Rightarrow\quad T(s)=\frac{G(s)D(s)}{1+G(s)D(s)}=\frac{10(s+1)}{(s+1.0219)(s-p_+)(s-p_-)}$$

where $p_\pm=-1.9641\pm 2.4348\,\mathrm{i}\triangleq -\sigma\pm\mathrm{i}\,\omega_d$. Note that there is a controller zero close to a plant pole in the LHP near $s=-1$; as a result, the closed-loop system has both a pole and a zero near $s=-1$. Via partial fraction expansion and inverse Laplace transform (see Example 17.12), the step response of this closed-loop system is

$$Y(s)=T(s)R(s)=\frac{T(s)}{s}=\frac{-1.0314s-4.0194}{(s+\sigma)^2+\omega_d^2}+\frac{0.0314}{s+1.0219}+\frac{1}{s}$$
$$\Rightarrow\quad y(t)=-1.0314\,e^{-\sigma t}\cos(\omega_d t)-0.8188\,e^{-\sigma t}\sin(\omega_d t)+0.0314\,e^{-1.0219t}+1.$$

Note in particular the third term of $y(t)$, which is the result of the closed-loop pole near $s=-1$ arising from the approximate pole/zero cancellation in the plant and the controller. This contribution to the step response is a decaying exponential with a very small coefficient, which is essentially negligible. Had we simply cancelled the plant pole and the nearby controller zero during the analysis,

$$G(s)D(s)\approx\frac{10}{s(s+4)}\quad\Rightarrow\quad T(s)\approx\frac{10}{s^2+4s+10}=\frac{10}{(s-\overline{p}_+)(s-\overline{p}_-)}$$

where $\overline{p}_\pm=-2\pm 2.4495\,\mathrm{i}\triangleq-\overline{\sigma}\pm\mathrm{i}\,\overline{\omega}_d$, then the step response computed would have been

$$Y(s)\approx\frac{T(s)}{s}=\frac{-1s-4}{(s+\overline{\sigma})^2+\overline{\omega}_d^2}+\frac{1}{s}\quad\Rightarrow\quad y(t)\approx-1\,e^{-\overline{\sigma}t}\cos(\overline{\omega}_d t)-0.8165\,e^{-\overline{\sigma}t}\sin(\overline{\omega}_d t)+1,$$

which is essentially identical. Note, however, that had the approximate pole/zero cancellation taken place in the RHP, then the step response $y(t)$ would have had a component with a *growing* exponential and a very small coefficient; this component would eventually dominate the system response. Note finally that all attempts to cancel a plant pole (or zero) with a controller zero (or pole) must be considered as approximate to some degree. The conclusion to be drawn is that *approximate pole/zero cancellations arising during controller design may simply be neglected in the LHP, but must never be attempted in the RHP*. The design of a controller to achieve an approximate pole/zero cancellation on or near the imaginary axis, called a **notch filter**, is delicate but doable, and is considered at length in §18.3.2.

18.2.2 The Bode plot, revisited

As described in §17.5.1, an (open-loop) Bode plot may be used to summarize the gain and phase of the response of the system $G(s)D(s)$, in *open loop*, to a sinusoidal input $u(t)$. As seen in (18.4), the poles of the *closed-loop* transfer function $T(s)$ are given by the roots of the equation $1+G(s)D(s)=0$ [and the motion of these roots as a parameter of the controller (usually, its gain) is varied is often plotted in a root locus].

A somewhat subtle connection between the open-loop and closed-loop problems makes the Bode plot especially useful for feedback control design. If for some s on the imaginary axis [that is, $s=\mathrm{i}\omega_r$ for some **resonant frequency** ω_r] the open-loop system $G(s)D(s)$ simultaneously has a gain of 1 and a phase of -180° [that is, $G(\mathrm{i}\omega_r)D(\mathrm{i}\omega_r)$ has the critical value of -1], then the closed-loop transfer function $T(s)=G(s)D(s)/[1+G(s)D(s)]$ has a pole on the imaginary axis, and is on the verge of instability, meaning that:

- if the system is given an impulse input, it will oscillate at the resonant frequency ω_r without decaying,

- if the closed-loop system is continuously excited (sinusoidally) at this resonant frequency ω_r, the magnitude of the response will be unbounded, and
- any tiny unmodeled error in either the plant or the controller could lead to closed-loop instability.

We might thus label the imaginary axis in the s-plane as an **axis of evil**; it is imperative to check any stable closed-loop system (that is, with its poles in the LHP) to ensure that its poles are in some sense "far" from being on this axis. Two valuable measures that may be read directly off the Bode plot (see, e.g., Figure 18.6) accomplish exactly this: the **phase margin** (**PM**) quantifies the amount that the phase of the open-loop system $G(i\omega)D(i\omega)$ is away from -180° at the frequency ω_c for which the open-loop system gain equals 1, whereas the **gain margin** (**GM**) quantifies the factor by which the gain of the open-loop system $G(i\omega)D(i\omega)$ is away from 1 at the frequency ω_p for which the open-loop system phase equals -180°. If the PM is large, then there may be correspondingly large errors in the modeling of the phase of the system (due, for example, to unmodeled delays in the system) before risking closed-loop instability, whereas if the GM is large, then there may be correspondingly large errors in the modeling of the gain of the system (due, for example, to uncertainty in the actuator authority or sensor sensitivity) before risking closed-loop instability.

The Bode plot illustrated in Figure 18.6 also depicts the typical constraints considered during the design of the controller $D(s)$. A *large open-loop gain* $|G(i\omega)D(i\omega)|$ *is generally sought at low frequencies* to ensure adequate tracking of the reference input, and a *small open-loop gain is generally sought at high frequencies* to ensure adequate attenuation of high-frequency disturbances[6]. Thus, at some intermediate frequency [dubbed the **crossover frequency** ω_c], $|G(i\omega_c)D(i\omega_c)| = 1$. As in (17.57), the following convenient **approximate design guides** may be identified by examining a range of step responses[7]:

$$\omega_c \approx \omega_n \approx \omega_{BW}/1.4, \quad \zeta \approx \mathrm{PM}/100. \tag{18.15}$$

Thus, a target value for ω_c may be determined from the rise time or tracking constraints on the system, and a target value for the PM may be determined from the overshoot constraint on the system. Noting Figure 18.6, when performing controller design using a Bode plot, one typically tunes first the phase of the controller $D(s)$ to achieve the desired PM at the target value of the crossover frequency ω_c, then tunes the overall gain of the controller to actually achieve crossover at this target frequency. Next, if necessary, the gain of the controller at the low and high frequencies are adjusted to meet the tracking and robustness constraints, in addition to ensuring an adequate GM, and the overall gain readjusted to maintain crossover at the target frequency. Finally, the step response of the closed-loop system is checked, and any required fine tuning applied to meet the design constraints (e.g., increasing the crossover frequency to reduce the rise time, and increasing the PM to reduce the overshoot). As shown in §18.3, this process can often be achieved by a straightforward and methodical combination of lead compensation, lag compensation, and low-pass filtering.

As noted in Fact 17.15, a remarkable and useful feature of the Bode plot is that it is *additive*; that is, $\log|G(i\omega)D(i\omega)| = \log|G(i\omega)| + \log|D(i\omega)|$, and $\angle G(i\omega)D(i\omega) = \angle G(i\omega) + \angle D(i\omega)$. Thus, when examining the Bode plot of the plant $G(s)$, it is usually clear what is needed in terms of the gain and phase characteristics of the controller $D(s)$ such that the cascade of the controller and plant together have the appropriate overall behavior to meet the design guides discussed above, and thus, e.g., the rise time and overshoot constraints on the closed-loop system. Control design leveraging the Bode plot in this manner is referred to as **loop shaping**.

As noted at the end of §17.5.1, in systems with no RHP zeros or poles, the gain and phase curves are related in a simple fashion: a gain slope of -2 over a particular range of frequencies corresponds to $\sim 1/(i\omega)^2$ behavior of the transfer function, and thus a phase of about -180°; similarly, a gain slope of -1 corresponds to a phase of about -90°, and a gain slope of 0 corresponds to a phase of about 0°. Thus, a rule of thumb for achieving a good PM (and, thus, good damping and low overshoot) in many systems is to *attempt to achieve*

[6] That is, for small ω, $\frac{Y(i\omega)}{R(i\omega)} = \frac{G(i\omega)D(i\omega)}{1+G(i\omega)D(i\omega)} \approx 1$ if $|G(i\omega)D(i\omega)| \gg 1$; for large ω, $\frac{U(i\omega)}{W(i\omega)} = \frac{-G(i\omega)D(i\omega)}{1+G(i\omega)D(i\omega)} \approx 0$ if $|G(i\omega)D(i\omega)| \ll 1$.

[7] For clarity of presentation, the definition of ω_{BW} is deferred to §18.2.4 and Figure 18.8b.

crossover at a gain slope of approximately -1; if crossover is attempted at a gain slope of closer to -2, the PM will often be unacceptably small (and, thus, the overshoot will be unacceptably high).

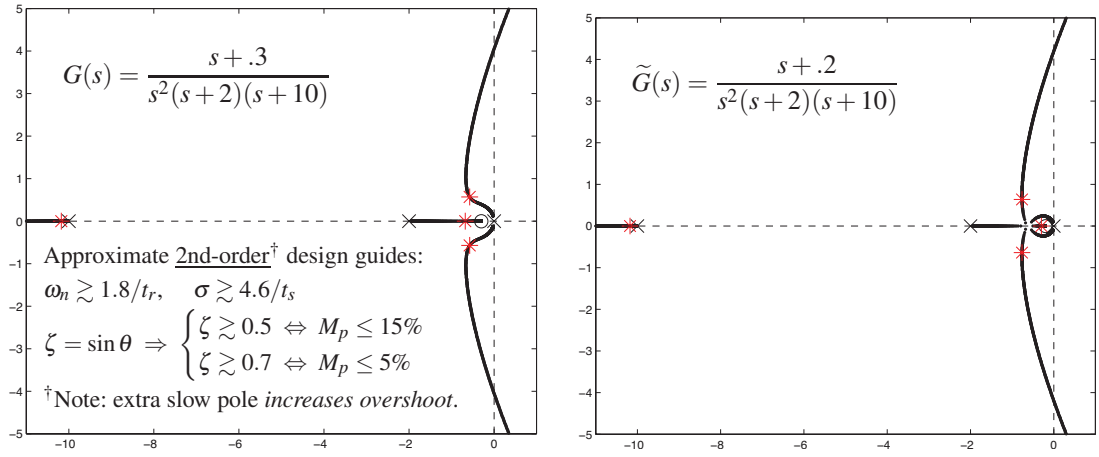


Figure 18.5: Root loci of (left) $G(s)$ and (right) the very slightly modified $\widetilde{G}(s)$, with proportional control $D(s) = K$ applied, and with $*$ marking the four closed-loop poles for $K = 15$. Though the locus makes a rather sudden reconnection, the step response of each of these systems is quite similar. If a system is dominated by second-order behavior, the closed-loop pole locations should generally lie in the region allowed by the approximate design guides specified by the rise time, settling time, and/or overshoot constraints (see Figure 17.13) on the closed-loop system. Note that the present (fourth-order) system has a complex pair of poles, plus two stable poles on the negative real axis. The pole near $s = -10$ is stable and (comparatively) fast, and thus has little effect on the closed-loop dynamics. The other pole on the negative real axis is of about the same speed as the dominant pair of complex poles; this generally results in significant extra overshoot of the step response, thus motivating increased damping on the complex poles than otherwise indicated by the approximate design guides (developed for second-order systems) to compensate appropriately.

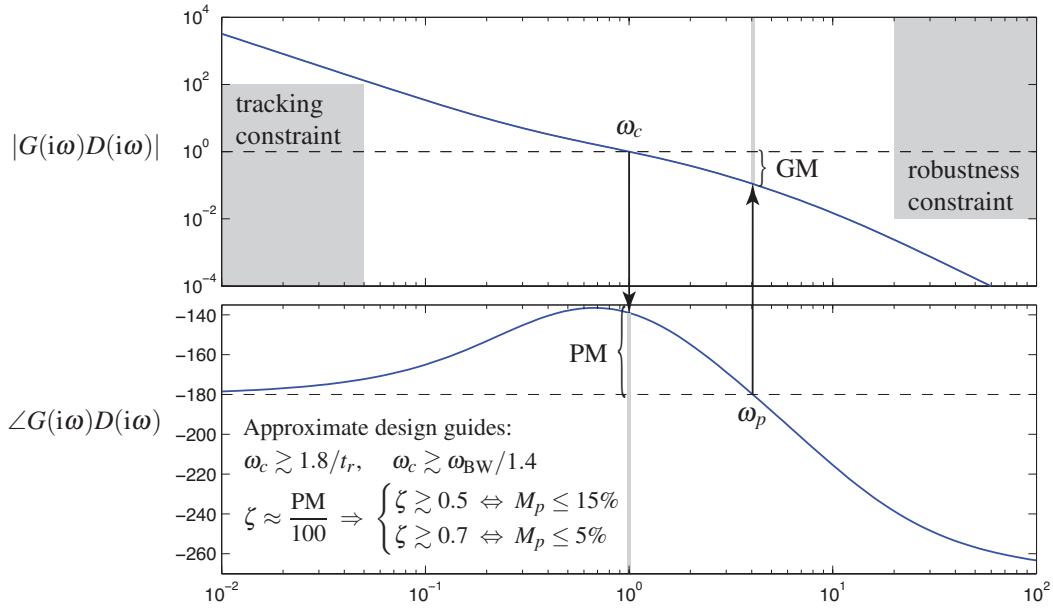


Figure 18.6: Open-loop Bode plot of $G(s)D(s) = K\frac{s+.3}{s^2(s+2)(s+10)}$ (see root locus in Figure 18.5a), with K adjusted to give crossover at $\omega_c = 1$. The PM is $180^\circ + \angle G(i\omega_c)D(i\omega_c)$, whereas the GM is $1/|G(i\omega_p)D(i\omega_p)|$, where ω_p is defined as that frequency where $\angle G(i\omega_p)D(i\omega_p) = -180^\circ$ (if it exists; otherwise, GM$= \infty$).

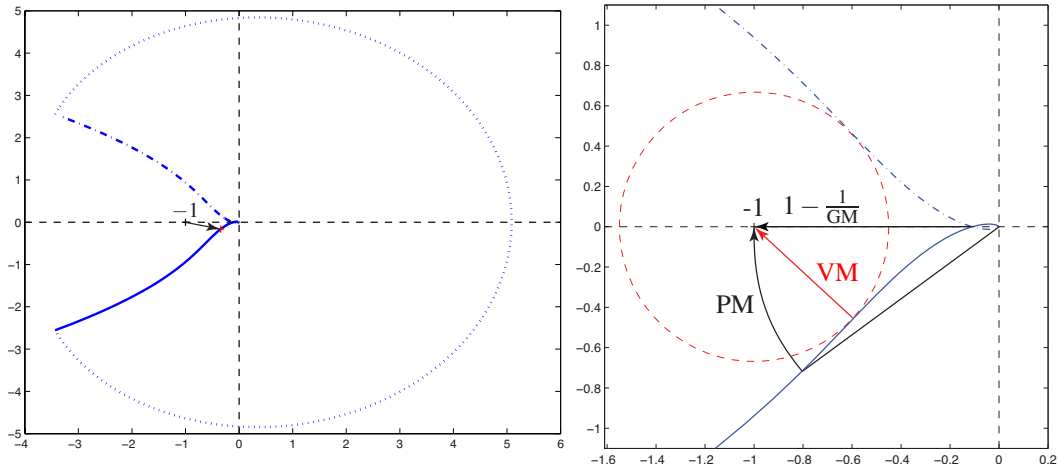


Figure 18.7: Nyquist plot of system considered in Figures 18.5a and 18.6. The GM and PM marked in Figure 18.6 are also apparent in the Nyquist plot; another measure, the VM, quantifies the distance of the point on the Nyquist contour closest (via a change in both gain *and* phase) to the critical $G(\mathrm{i}\omega)D(\mathrm{i}\omega) = -1$ condition.

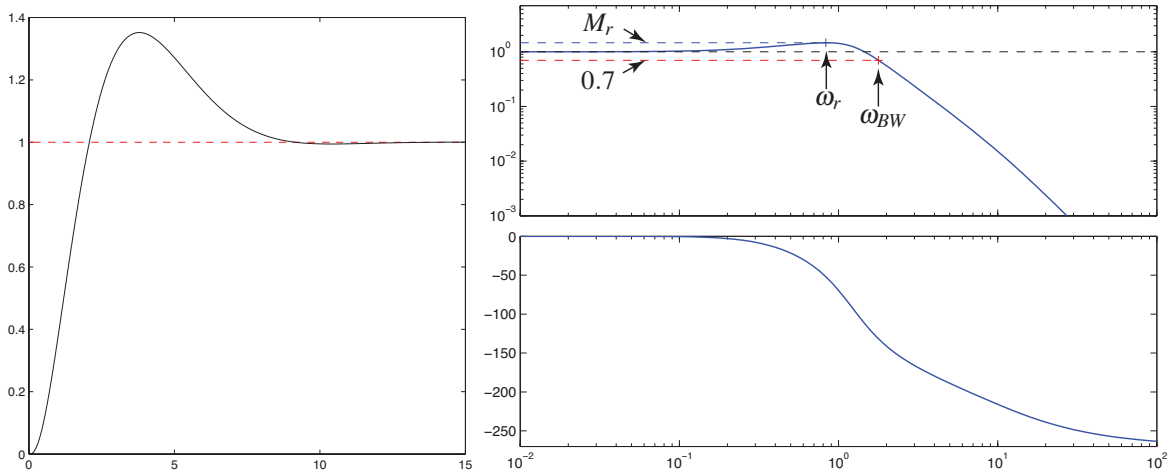


Figure 18.8: Final checks: (left) step response and (right) closed-loop Bode plot of system considered in Figures 18.5a, 18.6, and 18.7. The rise time, settling time, and overshoot of the step response are defined as in Figure 17.12; the rise time and overshoot are approximately related to the values of ω_c and PM in the corresponding Bode plot (Figure 18.6) as shown. The closed-loop Bode plot illustrates good tracking at low frequencies (that is, $|T(\mathrm{i}\omega)| \approx 1$ and $\angle T(\mathrm{i}\omega) \approx 0$) and good disturbance rejection at high frequencies (that is, $|T(\mathrm{i}\omega)| \ll 1$). Peaks of the gain curve of the closed-loop system, if present, occur at the **resonant frequencies** $\omega_{r,i}$, at which the gain is given by the **resonant peaks** $M_{r,i}$. The frequency at which the closed-loop gain falls below a value of 0.7, and thus the output ceases to **track** the reference input faithfully, is the **bandwidth**, ω_{BW}. The figures on pages 541 and 542 typify how classical tools are used in concert for targeted (see Guideline 18.1) feedback control design: a stabilizing controller is first found with the aid of a root locus (Figure 18.5) if $G(s)$ is known, or with the aid of a Nyquist plot (Figure 18.7) if $G(s)$ is unknown; the controller is then tuned as a function of frequency (a process known as **loop shaping**) using a Bode plot (Figure 18.6); finally, the closed-loop performance is checked by examining the rise time, settling time, and overshoot of the step response, and the tracking accuracy, resonant peaks, and bandwidth in the closed-loop Bode plot. Note that such a design process is often iterative.

Algorithm 18.3: Code for drawing a Nyquist plot.

View
Test

```
function Nyquist(num,den,g)
% Draw the Nyquist plot (i.e., a Bode plot in polar coordinates) of G(s)=num(s)/den(s).
% The derived type g groups together convenient plotting parameters: g.omega is the set of
% frequencies used, g.style is the linestyle, {g.figs,g.figL} are the figure numbers, and,
% in the s plane, g.eps is the (small) radius of the half-circles to the right of each
% pole on the imaginary axis, and g.R is the (large) radius of the D contour.
% Practical recommendation: do not make g.eps too small, or g.R too big, until you see
% where the corresponding curves are in both the s plane and the L plane!
L=PolyVal(num,0)./PolyVal(den,0); P=Roots(den); Z=Roots(num); tol=.0001;
figure(g.figs), clf, plot(real(P),imag(P),'kx'), hold on, plot(real(Z),imag(Z),'ko')
figure(g.figL), clf, plot(-1,0,'k+'),            hold on, plot(real(L),imag(L),'bo')
% First, find and sort the poles of D(s)*G(s) on the imaginary axis.
k=0; for j=1:length(P); if abs(real(P(j)))<tol, k=k+1; iP(k,1)=imag(P(j)); end, end
iP=QuickSort(iP,0,k); iPu(1)=iP(1); k=1;
for j=2:length(iP); if abs(iP(j)-iPu(k))>tol, k=k+1; iPu(k)=iP(j); end, end
% Draw small half circles in the s-plane to the right of each pole on the imaginary axis.
for j=1:k, w=i*iPu(j)+g.eps*exp(-i*[-pi/2:pi/50:pi/2]);
  if sign(iPu(j))<0, sym='r—'; else, sym='r-'; end, L=PolyVal(num,w)./PolyVal(den,w);
  figure(g.figs), plot(real(w),imag(w),sym), figure(g.figL), plot(real(L),imag(L),sym)
end
% Next, draw the (large) D contour in the s-plane.
w=g.R*exp(-i*[-pi/2:pi/50:pi/2]); L=PolyVal(num,w)./PolyVal(den,w); sym='k-.';
figure(g.figs), plot(real(w),imag(w),sym), figure(g.figL), plot(real(L),imag(L),sym)
% Finally, draw the line going up the imaginary axis in the s-plane (in several segments)
a(1)=-g.R; b=[];          % [note: segment j goes from a(j) to b(j)]
for j=1:ceil(k/2),        b=[b iPu(j)-g.eps]; a=[a iPu(j)+g.eps]; end
if floor(k/2)==ceil(k/2), b=[b -1e-12];       a=[a 1e-12];        end
for j=ceil(k/2)+1:k,      b=[b iPu(j)-g.eps]; a=[a iPu(j)+g.eps]; end, b=[b g.R];
for j=1:length(a), w=i*logspace(log10(abs(a(j))),log10(abs(b(j))),1000);
  if sign(b(j))<1, w=-w; sym='b—'; else, sym='b-'; end, L=PolyVal(num,w)./PolyVal(den,w);
  figure(g.figs), plot(real(w),imag(w),sym), figure(g.figL), plot(real(L),imag(L),sym)
end
end % function Nyquist
```

18.2.3 The Nyquist plot

As illustrated in Figure 18.7, *a Nyquist plot is just an (open-loop) Bode plot drawn in polar coördinates*. Defining $L(s) = G(s)D(s)$, the Nyquist contour is a curve in the L-plane comprised of points with modulus $|L(s)|$ and phase $\angle L(s)$, drawn for $s = \mathrm{i}\omega$ with the parameter ω varying from $-\infty$ to 0 to ∞ (see Algorithm 18.3). Note that the PM and GM indicated in the Bode plot in Figure 18.6 are both readily identified in the Nyquist plot in Figure 18.7. A third measure apparent in the Nyquist plot, the **vector margin** (**VM**), quantifies the minimum distance on the Bode plot (in polar coördinates), over all ω, to the critical response condition indicating marginal stability of the closed-loop system, $L(\mathrm{i}\omega) = -1$, via a modification of both the gain *and* the phase of the of the open-loop system $L(s)$.

The Nyquist stability criterion

Though it is easy to look at a root locus and discern whether or not a closed-loop system is stable [by ensuring all of the closed-loop poles are in the LHP], it is sometimes valuable[8] to discern whether or not a closed-loop system is stable by looking at the corresponding Nyquist plot[9]. A straightforward method for doing this follows from Cauchy's argument principle, Fact B.1, a review of which is advised before continuing.

[8] Specifically, when the frequency response (that is, the Bode plot) of the open-loop system is available, but a system model is not.
[9] It is often difficult to discern stability from a Bode plot though, as shown below, it is simple to discern stability from a Nyquist plot.

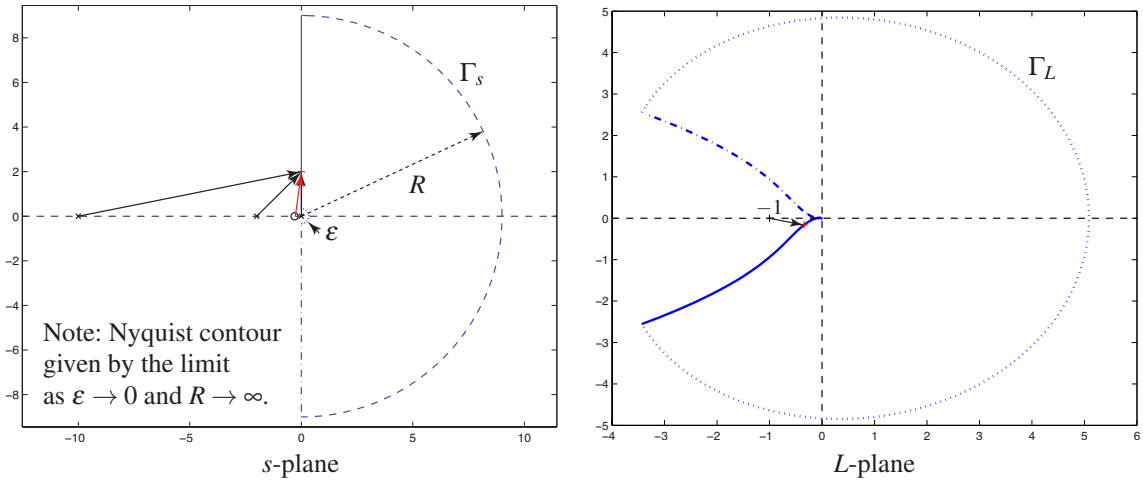


Figure 18.9: (a) Approximating contour Γ_s of the **Nyquist contour** in the s-plane, and (b) the corresponding contour Γ_L in the L-plane [for $L(s) = G(s)D(s)$ defined as in Figures 18.5a through 18.8]. The actual Nyquist contour is found by taking the limit as $R \to \infty$ and $\varepsilon \to 0$ of these approximating contours.

From Cauchy's argument principle (Fact B.1), it is a small step to the following important result.

Fact 18.5 (Nyquist stability criterion) *Define the* **Nyquist contour** Γ_s *as a D-shaped contour of radius R in the RHP of the s-plane as illustrated in Figure 18.9a (see also Figure 18.22), where a half circle of radius ε is taken into the RHP around every open-loop pole on the imaginary axis, in the limit that $R \to \infty$ and $\varepsilon \to 0$. Define also the corresponing contour Γ_L in the L-plane by applying the transform $L(s) = G(s)D(s)$ to all points $\overline{s}$ on the contour Γ_s in the s-plane (see, e.g., Figure 18.9b). It follows that:*
a. *If the number of poles of $L(s)$ with positive real part is 0, then the closed-loop system $T(s) = L(s)/[1+L(s)]$ is stable if and only if the contour Γ_L in the L-plane does not encicle the $L = -1$ point*[10].
b. *If the number of poles of $L(s)$ with positive real part is P, then the closed-loop system $T(s) = L(s)/[1+L(s)]$ is stable if and only if the contour Γ_L in the L-plane encicles the $L = -1$ point counterclockwise P times.*

Proof: As noted in (18.4), denoting the open-loop system $L(s) = G(s)D(s)$, the poles of the closed-loop system $T(s) = L(s)/[1+L(s)]$ are exactly the zeros of $F(s) = 1+L(s)$; thus, if $T(s)$ has Z RHP poles, then $F(s)$ will have Z RHP zeros. Now assume that $L(s)$ has P RHP poles; since $F(s) = 1+L(s)$, $F(s)$ has P RHP poles as well. By design, as illustrated in Figure 18.9a, the Nyquist contour (that is, Γ_s in the limit that $R \to \infty$ and $\varepsilon \to 0$) encloses the entire right half plane of s. It thus follows by Cauchy's argument principle (Fact B.1) that the contour Γ_F in the F-plane makes $(Z-P)$ clockwise encirclements of the origin, and thus (since $L(s) = F(s)-1$), Γ_L in the L-plane makes $(Z-P)$ clockwise encirclements of the point $L = -1$. Thus, if $Z = 0$ (that is, if $T(s)$ is stable), then the contour Γ_L in the L-plane makes exactly P counterclockwise encirclements of the point $L = -1$; if $Z > 0$ (that is, if $T(s)$ is unstable), then the contour Γ_L in the L-plane makes less than P counterclockwise encirclements of the point $L = -1$. □

Note that the Nyquist contour Γ_L and associated stability criterion (Fact 18.5) depend directly on the expression $L(s) = G(s)D(s)$; this is particularly convenient, because a rational factored form of $L(s)$ follows immediately from those of $G(s)$ and $D(s)$. In contrast, a formulation in terms of $T(s)$ (like the root locus) or in terms of $F(s)$ is at times less convenient, because a rational factored form for these expressions is not as easy to determine by hand. Note also that the contour Γ_s must take a curve of radius ε into the RHP to

[10] In the common case depicted in Figure 18.9, one might say that the closed-loop system is unstable if the **Pac-Man** engulfs the dot at $L = -1$. Note also that, mapping each plane onto its respective Riemann sphere (see Figure B.2), the contour near the south pole in s maps to the contour near the north pole in L, and the contour near the north pole in s maps to a contour near the south pole in L.

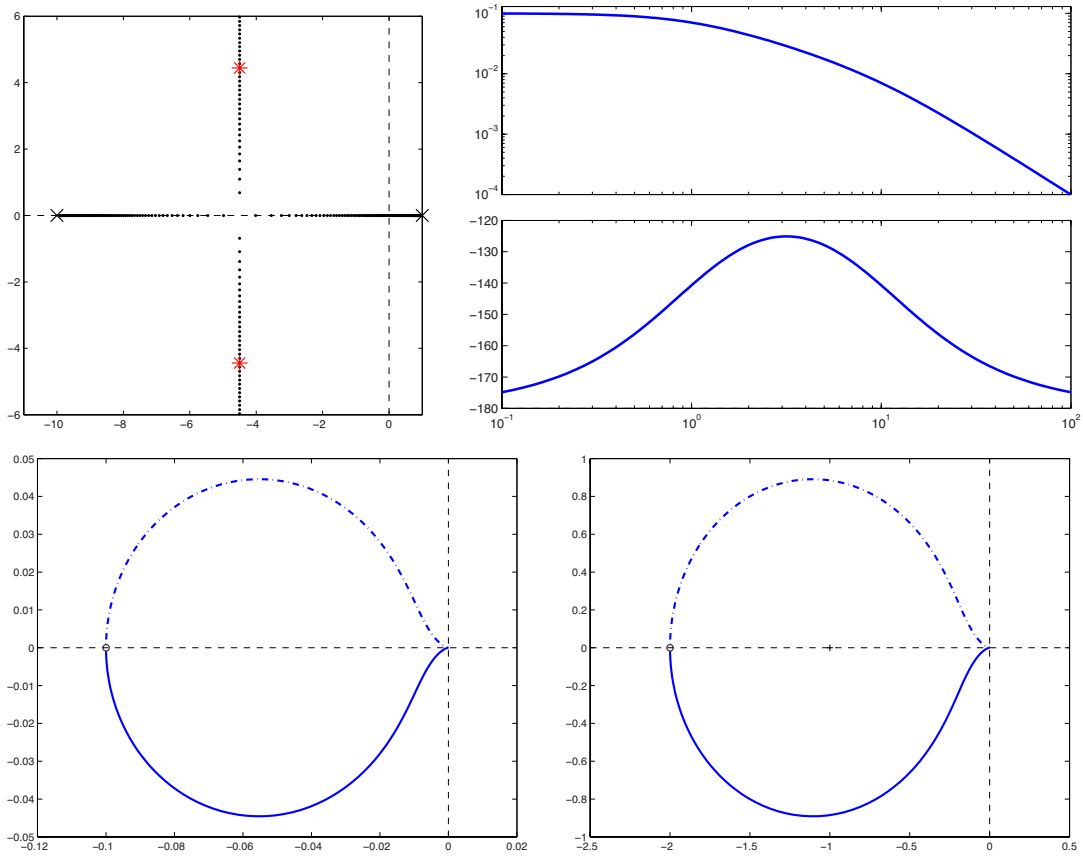


Figure 18.10: Classical control analysis for $G(s) = \frac{1}{(s-1)(s+10)}$: (a) Root locus plot of $G(s)D(s)$ with proportional control $D(s) = K$ applied with closed-loop poles for $D(s) = 20$ marked, (b) Bode plot of $G(s)$, (c) Nyquist plot of $L(s) = G(s)D(s)$ for $D(s) = 1$, (d) Nyquist plot of $G(s)D(s)$ for $D(s) = 20$.

avoid every pole of the open-loop system $L(s)$ which happens to lie on the imaginary axis, as indicated in the vicinity of the origin for the case depicted in Figure 18.9.

By part (a) of the Nyquist stability criterion, Figure 18.9b indicates *stability* of the closed-loop system for the controller gain K used, as the open-loop system $L(s) = G(s)D(s)$ has no RHP poles and the Nyquist contour does not encircle $L = -1$. A detail view illustrating this Nyquist contour near the origin is given in Figure 18.7; note that if the gain is increased by a factor larger than the GM, then the Nyquist contour would encircle the $L = -1$ point, in which case part (a) of the Nyquist stability criterion would indicate *instability* of the closed-loop system. This is consistent with the corresponding root locus in Figure 18.5a, which indicates closed-loop *stability* for small gain and closed-loop *instability* (2 poles in the RHP) for sufficiently large gain.

A case in which the open-loop system $L(s) = G(s)D(s)$ has one RHP pole is indicated in Figure 18.10. Note that, in this case, the root locus (Figure 18.10a) indicates closed-loop *instability* (1 pole in the RHP) for small gain and closed-loop *stability* for sufficiently large gain. Transforming the Bode plot of $G(s)$ (see Figure 18.10b) into polar coördinates to get the Nyquist plot for $D(s) = 1$ (see Figure 18.10c), it is seen that the Nyquist contour in this case does not encircle the -1 point, which by part (b) of the Nyquist stability criterion indicates *instability* of the closed-loop system. On the other hand, plotting the Nyquist plot for $D(s) = 20$ (see Figure 18.10d), it is seen that the Nyquist contour in this case encircles the origin counterclockwise exactly once, which by part (b) of the Nyquist stability criterion indicates *stability* of the closed-loop system. Thus, the conclusions drawn from the root locus and the Nyquest stability criterion are again consistent.

type	impulse	step	ramp	parabolic	cubic
$r=0$	0	finite	∞	∞	∞
$r=1$	0	0	finite	∞	∞
$r=2$	0	0	0	finite	∞
$r=3$	0	0	0	0	finite

Table 18.1: Steady-state error, $\lim_{t\to\infty} e(t)$, of a closed-loop system as a function of the type r, where $G(s)D(s) = b(s)/[s^r a_0(s)]$ with $[a_0(s)]_{s=0} \neq 0$ and $[b(s)]_{s=0} \neq 0$.

18.2.4 Final checks: the closed-loop step response and the closed-loop Bode plot

Once a controller is designed using the classical (that is, root-locus or Nyquist, and Bode) control design tools, final checks on its behavior may be performed by plotting the closed-loop system's **step response** (see Figure 18.8a) using Algorithm 17.1, and by plotting the **closed-loop Bode plot** (see Figure 18.8b) using Algorithm 17.4 applied to $T(s) = G(s)D(s)/[1+G(s)D(s)]$. The former indicates directly if the rise time, settling time, and overshoot constraints on the closed-loop system were indeed met, whereas the latter reveals the frequencies at which the closed-loop system accurately tracks the reference input (as well as the precision of this tracking), the peak magnitude of any **resonances** (that is, amplified response of the closed-loop system at certain frequencies; see, e.g., Exercises 17.6 and 18.2), and the **bandwidth frequency** ω_{BW} above which the gain of the closed-loop system **rolls off** and the system output no longer tracks the reference input. Based on these final checks, some final tweaking of the control design is sometimes required.

18.2.4.1 System type and loop prefactors

Returning to Figure 18.1a, note that we may write

$$E(s) = R(s) - Y(s) = R(s) - G(s)D(s)E(s) \quad \Rightarrow \quad \frac{E(s)}{R(s)} = \frac{1}{1+G(s)D(s)}.$$

Now assume that the controller $D(s)$ is chosen, based on the plant $G(s)$, such that **the closed-loop system is stable**, and further assume that we can write $G(s)D(s) = b(s)/[s^r a_0(s)]$, where $[a_0(s)]_{s=0} \neq 0$ and $[b(s)]_{s=0} \neq 0$, for some value of r (referred to as the **type** of the open-loop system), and thus

$$E(s) = \frac{s^r a_0(s)}{s^r a_0(s) + b(s)} R(s). \tag{18.16}$$

Noting the CT final value theorem (Fact 17.4), we may write the **steady state error** $\lim_{t\to\infty} e(t) = \lim_{s\to 0} sE(s)$. Considering impulse, step, ramp, parabolic, and cubic reference inputs[11], we may easily compute the behavior of the steady-state error as a function of type using (18.16), as listed in Table 18.1.

Focusing specifically on the case with unit step input (i.e., for $R(s) = 1/s$), you might notice in certain situations that, though you have designed a stabilizing controller $D(s)$ for a given plant $G(s)$, the step response $y(t)$ of the closed-loop system $T(s)$ does not approach unity as $t \to \infty$. By the above paragraph, if the system $G(s)$ is not accurately known, the change required to fix this problem is to build a sufficient number of integrators into $D(s)$ to make the open-loop system $G(s)D(s)$ type 1 [that is, $|G(i\omega)D(i\omega)| \to \infty$ as $\omega \to 0$ in the Bode plot] and thus, by the CT final value theorem,

$$\lim_{t\to\infty} y(t) = \lim_{s\to 0} sY(s) = \lim_{s\to 0} sT(s)U(s) = \lim_{s\to 0} T(s) = \lim_{s\to 0} \frac{G(s)D(s)}{1+G(s)D(s)} = 1,$$

regardless of any uncertainty in the overall gain or phase of $G(s)$.

[11] That is, $r(t) = \delta^\sigma(t)$, $h_1(t)$, $t\,h_1(t)$, $t^2 h_1(t)$, and $t^3 h_1(t)$, with Laplace transforms $R(s) = 1$, $1/s$, $1/s^2$, $2/s^3$, and $6/s^4$, respectively.

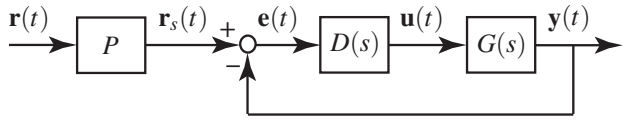


Figure 18.11: Introduction of a loop prefactor P [see (18.17)] to correct for a closed-loop with nonzero steady-state error in its unit step response [due to $G(\mathrm{i}\omega)D(\mathrm{i}\omega)$ being finite as $\omega \to 0$].

If, on the other hand, $|G(\mathrm{i}\omega)D(\mathrm{i}\omega)|$ is *finite* as $\omega \to 0$ and (importantly) the model of $G(s)$ is known to be relatively accurate at low frequencies, then simply selecting a prefactor

$$P = \frac{1}{T(s)}\Big|_{s=0} = \frac{1+G(s)D(s)}{G(s)D(s)}\Big|_{s=0} \tag{18.17}$$

and incorporating as in Figure 18.11 fixes the problem, bringing the step response back to $y(t) \to 1$ as $t \to \infty$.

18.2.5 Extending the root locus, Bode, and Nyquist tools to DT systems

As suggested by (17.69), when plotting frequency response (i.e., a Bode or Nyquist plot) in discrete time, one simply uses $z = e^{\mathrm{i}\omega h}$ in lieu of $s = \mathrm{i}\omega$. Algorithm 17.4 thus easily generates a DT Bode plot when called appropriately (see its associated test code for an example). A code for generating a DT Nyquist plot is considered in Exercise 18.3. Figures 18.12a-b give examples of both.

Further, as rational functions of s and z combine via identical rules when closing a feedback loop, a root locus in z for a DT system may be drawn with *exactly* the same code as is used to draw a root locus in s for a CT system. It is only the target region, specified by the approximate design guides, that changes (recall the mapping of the curves in Figure 17.13 into the z-plane, as indicated in Figure 17.15b).

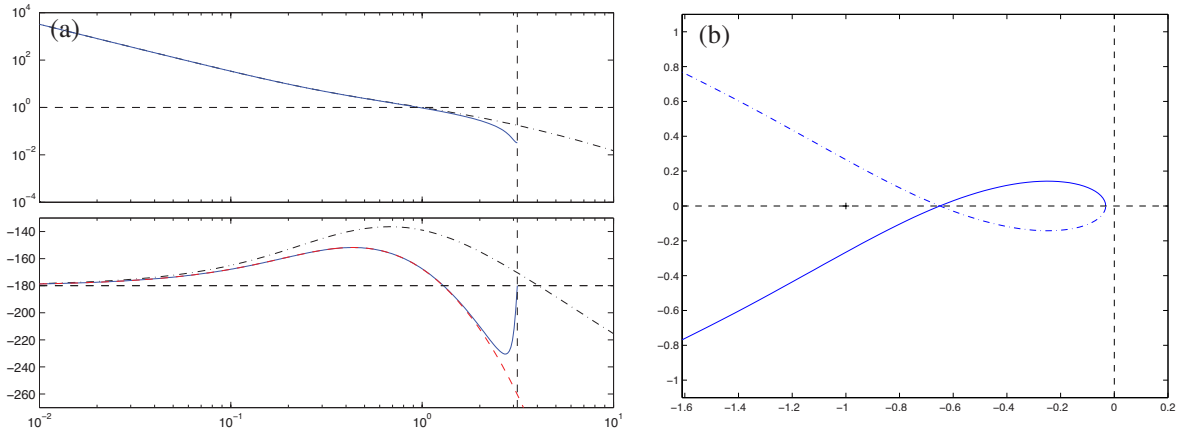


Figure 18.12: (a) Bode and (b) Nyquist plots of (solid) the DT system $G(z)$ given (see §17.4.3.1) by the cascade of a DAC with a ZOH, the system $G(s)$ analyzed in Figures 18.5a through 18.9, and an ADC, with $h = 1$. The highest frequency that can be represented uniquely in discrete time (see §5.4.2) is the Nyquist frequency $\omega_{\text{Nyquist}} = \pi/h$, indicated by the vertical dashed line in the Bode plot. The Bode plot of the corresponding CT system $G(s)$, (dot-dashed), is shown for comparison; as motivated by Figure 18.33, the effect of an $h/2$ delay on the phase of this CT Bode plot [see (18.32)] is also illustrated (dashed), and accounts for the bulk of the discrepancy between the Bode plots of $G(s)$ and $G(z)$.

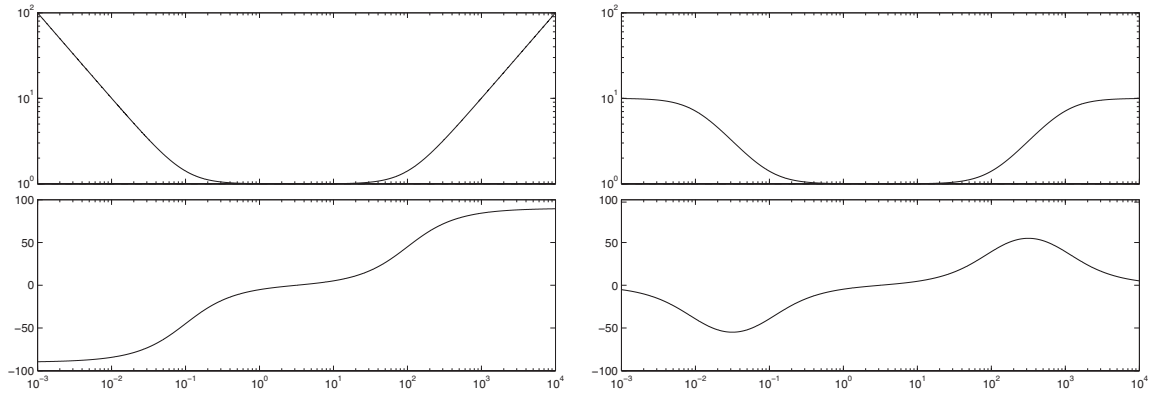


Figure 18.13: (a) Bode plot of the PID controller $D_{\text{PID}}(s)$ given in (18.18) with $K_p = 1$, $T_I = 10$, and $T_D = 0.01$. (b) Roll-off applied at low and high frequencies of the PID, applying the integral and derivative actions over only finite ranges of frequencies; this roll-off, which is built in (at prespecified frequencies) in commercial "black-box" PID controllers, can be adjusted precisely using the lead/lag techniques presented in §18.3.2.

18.3 Primary techniques used for classical control design

As exemplified in Figures 18.5–18.8, the process of **classical** (i.e., transform-based) control design includes

- identifying an appropriate family of stabilizing controllers using root locus or Nyquist plotting tools,
- tuning the control design via the (open-loop) Bode plot, a process known as **loop shaping**, and
- checking the resulting closed-loop performance via the step response and closed-loop Bode plot.

There are a number of prototype linear control designs that may be cascaded and tuned using these tools in order to implement Guideline 18.1. We now introduce, in continuous time, a few of the most common.

18.3.1 PID (Proportional-Integral-Derivative) controllers

The simple **PID** (**Proportional-Integral-Derivative**) controller is by far the most common controller implemented in industry. It is the ultimate "black box" controller, and is characterized by three simple "knobs":

- a constant of proportionality K_p (the gain of the controller at intermediate frequencies),
- a time constant T_I for the integral term (below which the gain of the controller rises $\propto 1/\omega$), and
- a time constant T_D for the derivative term (above which the gain of the controller rises $\propto \omega$).

The "ideal" PID controller may be written in transfer function form, for finite T_I and T_D, as

$$D_{\text{PID}}(s) = K_p\left(1 + \frac{1}{T_I s} + T_D s\right) = K_p T_D \frac{s^2 + s/T_D + 1/(T_I T_D)}{s} = K\frac{(s+z_+)(s+z_-)}{s}, \tag{18.18}$$

where, usually, $T_I > T_D$. Note that $K = K_p T_D$ and, if $T_I \gg T_D$, it follows that

$$z_\pm = \left[1 \pm \sqrt{1 - 4T_D/T_I}\right]/(2T_D) \approx \left[1 \pm (1 - 2T_D/T_I)\right]/(2T_D) = \left\{1/T_I,\, 1/T_D\right\}.$$

The Bode plot of an ideal PID controller $D_{\text{PID}}(s)$ is given in Figure 18.13a.

The reader is *strongly encouraged* to understand the impact of PID control on the closed-loop system of interest by examining the effect of the three knobs $\{K_p, T_I, T_D\}$ on the Bode plot [that is, on the magnitude and phase of $G(\mathrm{i}\omega)D(\mathrm{i}\omega)$ as a function of frequency ω], in order to construct a controller of this class that *just* meets the performance specification on the closed-loop system (and, thus, only minimally compromises robustness; see Guideline 18.1). Noting (18.18), taking $T_I \to \infty$ and $T_D \to 0$ reduces the PID controller to the

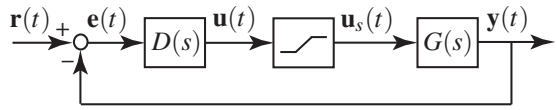


Figure 18.14: Schematic representation of the saturation nonlinearity caused by an actuator being driven to its limits. In this simple representation, the actuator response is modeled to be linear between the limits, but the control signal $u(t)$ is clipped to the min or max values outside these limits. Actuator saturation of this sort can cause integrator windup when PID control is applied, with potentially harmful consequences.

proportional (P) controller considered throughout §18.2; taking $T_I \to \infty$ or $T_D \to 0$ alone reduces PID to PD or PI, respectively. Now recall the typical Bode design constraints on $G(\mathrm{i}\omega)D(\mathrm{i}\omega)$ depicted in Figure 18.6, and consider again the Bode plot of $D_{\text{PID}}(s)$ in Figure 18.13a. Introducing the derivative term by dialing T_D up (from zero), so that $\omega_D = 1/T_D$ is in the vicinity of the crossover frequency ω_c, bumps up the phase at crossover, thereby improving the PM and reducing the overshoot of the closed-loop system. Introducing the integral term by dialing T_I down (from infinity), so that $\omega_I = 1/T_I$ is up to an order of magnitude below the crossover frequency ω_c, bumps up the low-frequency gain of the open-loop system without diminishing substantially the phase at the crossover frequency, thereby improving the tracking of the closed-loop system.

Note from (18.18) that the PID controller is governed by the differential equation

$$\frac{U(s)}{E(s)} = K_p\left(1 + \frac{1}{T_I s} + T_D s\right) \quad \Leftrightarrow \quad u(t) = K_p\left(e(t) + \frac{1}{T_I}\int_0^T e(t)\,dt + T_D\frac{de(t)}{dt}\right). \tag{18.19}$$

Given the prevalence of PID control in industry, it is important to identify the two primary and potentially catastrophic effects that simple PID control of this form can introduce in practice.

The first problem, associated with the derivative term of the PID, is its inherent **amplification of high-frequency noise**. Note on page 529 that the control sensitivity is $U(s)/V(s) = D(s)/[1 + G(s)D(s)]$. Even if the plant is characterized by a $G(s) \propto 1/s^2$ dependence at high frequencies (many are), a controller with a $D(s) \propto s$ dependence is problematical if the noise $v(t)$ has substantial high frequency components (it usually does), as such a controller amplifies this noise and sends this amplified high frequency garbage in the control signal $u(t)$ to the actuators, which then waste energy doing unnecessary work, or simply burn out.

The second issue (which is sometimes a problem), associated with the integral term of the PID, is **integrator windup** in the presence of **actuator saturation** (see Figure 18.14). Linear control theory is often found to be effective even on systems which are only "mostly linear". A nonlinearity that sometimes arises when applying control to physical systems is actuator saturation; that is, actuators used to apply a desired control input $u(t)$ to a physical system typically provide an actual control input $u_s(t)$ to the physical system (a force, torque, displacement, velocity, etc.) that varies between two bounds. In such a situation, proportional controllers usually suffer only a slight performance loss, exhibiting, effectively, reduced values of $K = u_s(t)/e(t)$ when the actuator is saturated. However, a controller with an integrator accumulating a (potentially, nonzero) $e(t)$ causes $u(t)$ to grow linearly (over, potentially, a long period of time), while the saturated value of $u_s(t)$ applied to the plant remains at its bound. This is generally not a problem until the controller needs to again reduce the control input applied to the plant. With $u(t)$ possibly driven to high values (that is, "wound up"), it can possibly take a correspondingly long time for $u(t)$ to decrease to the point that the actual control applied to the plant, $u_s(t)$, finally decreases. This delayed responsiveness of the actuation can lead quickly to closed-loop instability.

Considered in terms of the frequency response of the PID controller (see Figure 18.13a), the first problem is associated with the derivative part growing without bound as ω is increased, whereas the second issue is associated with the integral part growing without bound as ω is decreased. The cure to both is to **roll off** the magnitude of the controller response at both ends of the spectrum, as illustrated in Figure 18.13b. In fact, *black box PID controllers implement such roll-off at both ends of the spectrum in order to alleviate the two*

issues discussed above; however, they don't give the control designer the ability to specify the break points at which such roll-off sets in, or the degree of roll-off applied. The lead and lag controllers discussed in §18.3.2, together with the low-pass filtering discussed in §18.3.3, provide precisely this capability. That is:

Guideline 18.2 *Lead and lag controllers are the responsible way to apply derivative and integral control actions, respectively, over finite ranges of frequencies, thus enabling the implementation of Guideline 18.1.*

There is thus actually no compelling reason to use the restrictive PID control paradigm once the more flexible lead and lag controllers are well understood; however, due to their prevalence in industry, is is perhaps prudent to discuss the tuning of PID controllers a bit further below in order to familiarize the reader.

18.3.1.1 Ad hoc PID tuning

PID controllers may be tuned to be effective on a variety of simple plants; indeed, in spite of the two problems mentioned above, they are so simple and intuitive that many controls engineers hesitate to use anything else. PID controllers are sometimes tuned to achieve essentially the highest rise time possible given the parameters of the plant and the form of the PID controller while respecting the design constraints on the overshoot and tracking. While there is not a unique way of achiving this trade-off, one commonly used strategy is to first apply proportional feedback $D(s) = K$ to the system $G(s)$ of interest and then dial up the gain until a critical value $K = K_u$ is reached at which the system oscillates at constant amplitude, with a frequency which we denote $\omega_u = 1/T_u$. Knowledge of K_u and T_u is, in many cases, enough to tune an effective PID control stratetegy, which may be accomplished by setting $K_p = \alpha K_u$, $T_I = \beta T_u$, and $T_D = \gamma T_u$, for appropriate values of the parameters $\{\alpha, \beta, \gamma\}$. Various values for $\{\alpha, \beta, \gamma\}$ have been suggested in the literature; a few of the most popular are: Ziegler-Nichols P, $\{0.5, \infty, 0\}$; Ziegler-Nichols PI, $\{0.45, 0.83, 0\}$; Ziegler-Nichols PID, $\{0.6, 0.5, 0.125\}$; Pessen PID, $\{0.7, 0.4, 0.15\}$; Tyreus-Luyben PI, $\{0.31, 2.2, 0\}$; Tyreus-Luyben PID, $\{0.45, 2.2, 0.16\}$. All of these suggestions are nothing more than rules of thumb that were found to be effective on the particular problems that were of interest to the authors proposing them. An example of the effect they have in application is discussed below and considered further in Exercise 18.4.

Example 18.1 Control of a first-order system with a delay (cruise control of an automobile) with PID

As derived in Example 17.8, the linearized equations of motion of an automobile at cruise, neglecting the disturbance $w(t)$, may be written

$$\left(\frac{d}{dt} + \frac{a}{m}\right) v'(t) = \frac{1}{m} u'(t-d); \tag{18.20}$$

note that the model accounts for a slight delay d between the actuation of the throttle and its effect on the force applied to accelerate the vehicle. Noting (17.48), we may write the transfer function of the vehicle as

$$\frac{V'(s)}{U'(s)} = G(s) = C \frac{e^{-ds}}{s+a} \approx \frac{C}{s+a} \cdot \frac{1 - (ds)/2 + (ds)^2/12}{1 + (ds)/2 + (ds)^2/12}. \tag{18.21}$$

For the vehicle considered in Example 17.8 cruising at $\overline{v} = 34$ m/s $= 76$ mph, the constants in this transfer function are $C = 6.58 \times 10^{-4}$, $a = 39.3$, and $d = 0.04$; we use these values below.

The root locus of this system with proportional feedback applied is given in Figure 18.15a. Due to the delay[12], proportional feedback drives the system unstable at some critical gain $K = K_u$; the ad hoc tuning strategies above suggest how one can simply back off from this critical value of K a bit, then dial in some derivative compensation to improve the phase lead at crossover and some integral compensation to improve

[12] In fact, as can readily be identified by the RHP zeros in any Padé approximation of a delay [see (17.48)], it is true in general that *any system with a delay will be destabilized by sufficiently high gain feedback*, as exemplified in Figure 18.15a. As system delays often go unmodeled, this is yet another motivation for Guideline 18.1.

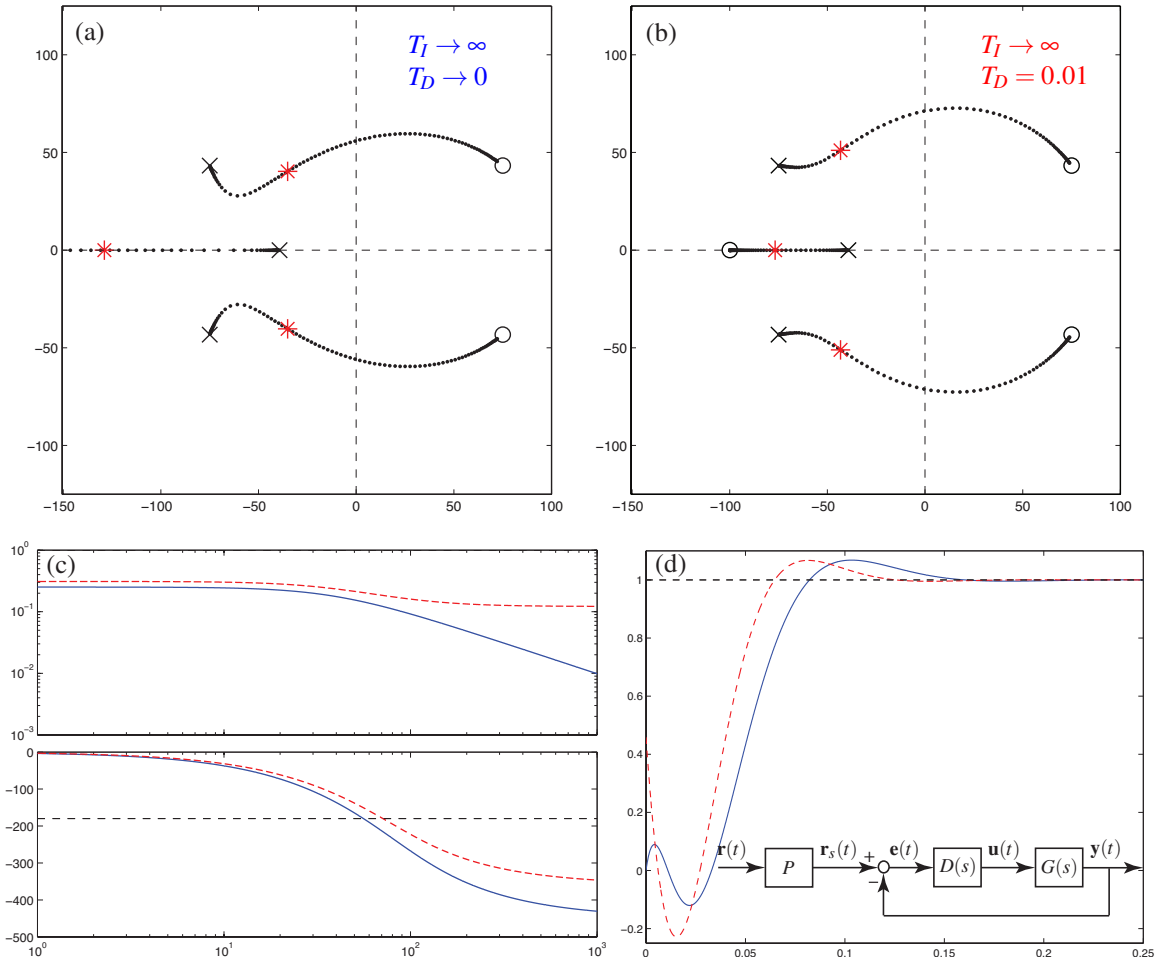


Figure 18.15: Application of P and PD control to the cruise control problem (18.21). (a) Root locus with respect to K when P control is applied. (b) Root locus with respect to K_p when PD control is applied, taking $T_D = 0.01$. (c) Bode plot and (d) step response in the (solid) P and (dashed) PD cases, with gains as marked in (a) and (b). In both cases, a loop prefactor P [see inset in (d), and (18.17)] has been used in order to achieve a zero steady-state error in the step response, assuming $G(s)$ has been modeled accurately.

the low-frequency tracking. A controller designed using such rules is by its nature aggressive, as the K is selected to give a rise time that is essentially as fast as possible given the modeled value of the delay d. Such an approach contradicts Guideline 18.1, and is sensitive to a variety of unmodeled effects, especially additional delays; a less aggressive approach to control design is thus generally preferred.

Figure 18.15 illustrates the effects of P and PD control applied to the automobile cruise control problem (18.21) with the delay approximated by its $n = 2$ Padé approximation (for refinement of this approximation, see Exercises 18.5 and 18.6). Note that adding bit of derivative compensation to the proportional controller bumps up the phase at high frequencies, thus facilitating a slightly faster rise time for the same overshoot (as shown) or, alternatively, a slightly reduced overshoot for the same rise time. The low-frequency gain in both systems depicted in Figure 18.15c is finite (in fact, less than one); in order to achieve a zero steady-state error in the step response, a loop prefactor is used [see inset in Figure 18.15d], assuming $G(s)$ is modeled accurately

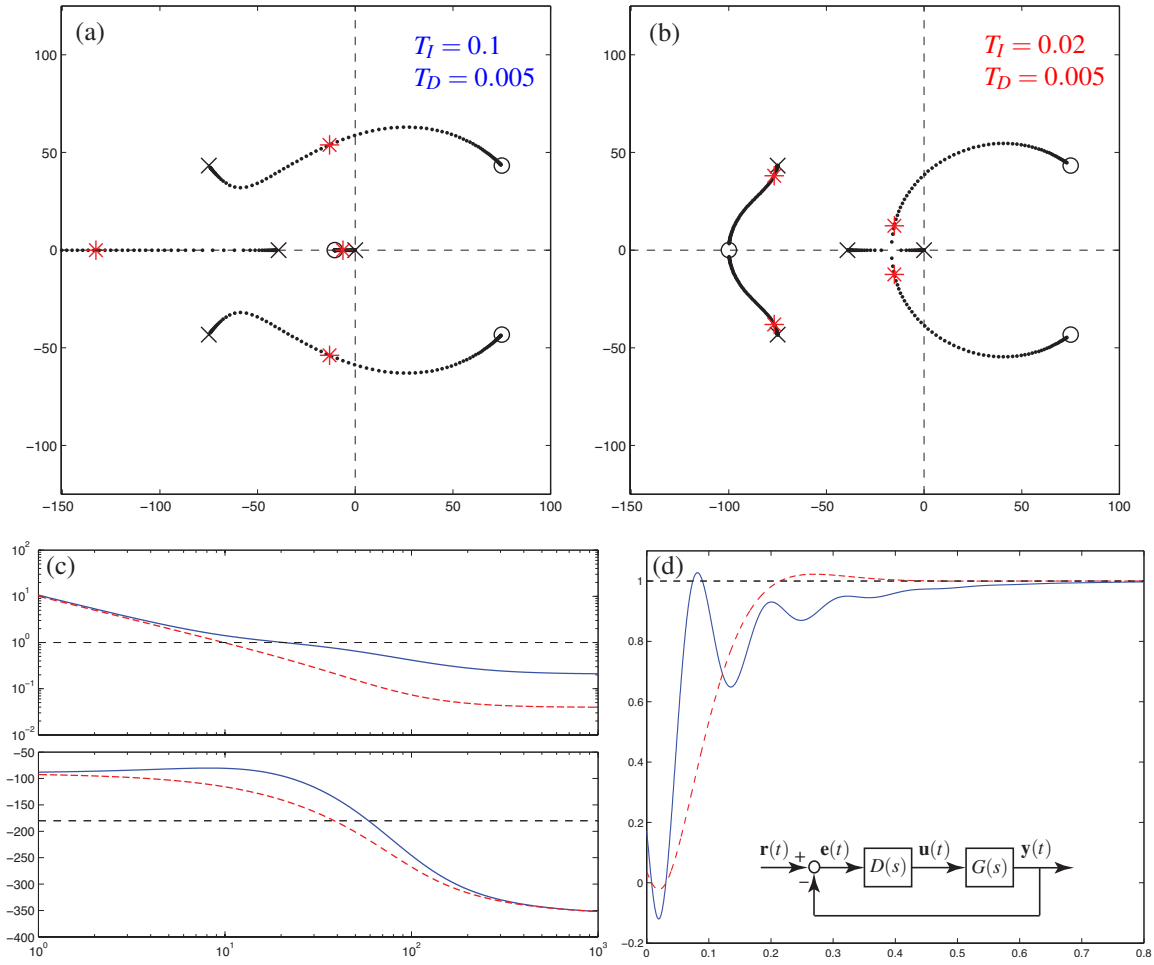


Figure 18.16: Application of PID control to the cruise control problem (18.21), taking $T_D = 0.005$. (a) Root locus with respect to K_p, taking $T_I = 0.1$. (b) Root locus with respect to K_p, taking $T_I = 0.02$. (c) Bode plot and (d) step response taking (solid) $T_I = 0.1$ and (dashed) $T_I = 0.02$, with gains as marked in (a) and (b). In both cases, the integral component of the controller $D(s)$ ensures a zero steady-state error in the step response even in the presence of significant uncertainty in the modeling of $G(s)$.

so that the appropriate value of P can be computed. If instead there were significant modeling uncertainty (e.g., variable road grade), it would be beneficial to add integral compensation (or at least some significant lag compensation) to increase the open-loop low-frequency gain, $G(\mathrm{i}\omega)D(\mathrm{i}\omega)$ for small ω, and thus reduce the steady-state error even in the presence of such modeling error, as illustrated in Figure 18.16; note that applying such integral compensation slows down the response time as compared with that achievable when using a loop prefactor to scale the step response (Figure 18.15). Finally, note that "performance" is not just about rise time, settling time, and overshoot, but there is also a less tangible question of the response "quality"; a cruise control with the solid response curve depicted in Figure 18.16d would lead to a very uncomfortable ride. Design engineers who tune the values of $\{K_p, T_I, T_D\}$ in the various PID loops in automobiles are thus to a large degree responsible for establishing the "feel" of a vehicle.

The transfer function $G(s)$ in (18.21) is a good starting point for modeling many systems of unknown structure. This model has three parameters, $\{C,d,a\}$, that can be tuned to match three critical features of a system that may be determined experimentally: the low-frequency gain $C_0 = C/a$ (including its sign), and both the gain $C_c = C/\sqrt{\omega_c^2 + a^2}$ and the phase $\phi_c = -d - \text{atan}(\omega_c/a)$ of this system at a desired crossover frequency ω_c. If the system being modeled is open-loop stable (and, thus, $a > 0$), the values of $\{C_0, C_c, \phi_c\}$ of the open-loop system may be measured directly, from which the model parameters $\{C,d,a\}$ may be determined immediately. If the system being modeled is unstable (that is, $a < 0$ in the model), appropriate model parameters $\{C,d,a\}$ may often be determined by applying stabilizing proportional feedback $D(s) = K$ and measuring the lower and upper bounds (K_l and K_u, respectively) on the gain K for closed-loop stability, in addition to the frequency of oscillation, $\omega_u = 1/T_u$, at $K = K_u$ (for details, see Exercise 18.7).

18.3.2 Lead, lag, and notch controllers

As illustrated in Figure 18.17, **lead** and **lag** controllers provide the responsible way of adding, respectively, derivative and integral control effects (see Figure 18.13) over targeted ranges of frequencies; as with the derivative and integral components of a PID, lead and lag are often used together; their cascade (see Figure 18.13b) is often referred to as a **lead-lag** controller. An alternative to lead control for oscillatory open-loop systems which are accurately modeled, called a **notch** controller, puts two complex controller zeros near the two oscillatory poles of the plant (in a sense, **knocking out** the oscillatory plant dynamics), replacing these two oscillatory poles with two stable poles on the negative real axis, far enough to the left to achieve a sufficiently fast settling time. The lead and lag controllers are so named because of the phase lead and phase lag which they provide (see Figures 18.17a-b); the notch controller is so named because of the shape of its gain response as a function of frequency (see Figure 18.17c).

The effective tuning of lead, lag, and notch controllers may all be understood, initially, by considering their effects on the root loci of the appropriate systems. However, all three of these controllers are more precisely tuned by considering the corresponding Bode plots. The methodology of tuning lead, lag, and notch controllers is best explained by considering the following examples.

Example 18.2 Control of a first-order system with a delay, revisited using lead-lag control

We now revisit the PID controller design for the first-order system with a delay (a cruise control of an automobile) as developed in Example 18.1. The final PID control design achieved there is of the form given in (18.18) with $T_I = 0.02$, $T_D = 0.005$, and $K_p = 1.2 \cdot 10^4$, and thus

$$D_{\text{PID}}(s) = K_p\left(1 + \frac{1}{T_I s} + T_D s\right) = 60\frac{(s+100)^2}{s}. \tag{18.22a}$$

The closed-loop performance of the system considered with this controller applied is denoted by the dashed curve in Figure 18.16d, and is actually quite good in many respects, with a fast rise time and settling time (given the inherent limitations of the system imposed by the delay), low overshoot, zero steady-state error to a step input, and a high "quality" step response (little oscillation). However, what is hidden in this figure is the two problems associated with PID control discussed in §18.3.1: specifically, the amplification of high-frequency measurement noise, and integrator wind-up in the presence of actuator saturation. To alleviate both of these problems, as suggested in §18.3.1, we can instead apply lead-lag control of the form

$$D_{\text{lead-lag}}(s) = K \cdot \frac{s+100}{s+1} \cdot \frac{s+100}{s+2000} \qquad \text{where} \qquad K = 60 \cdot 2000 \cdot 0.95. \tag{18.22b}$$

The (low-frequency) lag controller used here has $z/p = 100$, the (high-frequency) lead controller has $p/z = 20$, and the zeros of the lead-lag controller (18.22b) are in the same locations as the corresponding PID

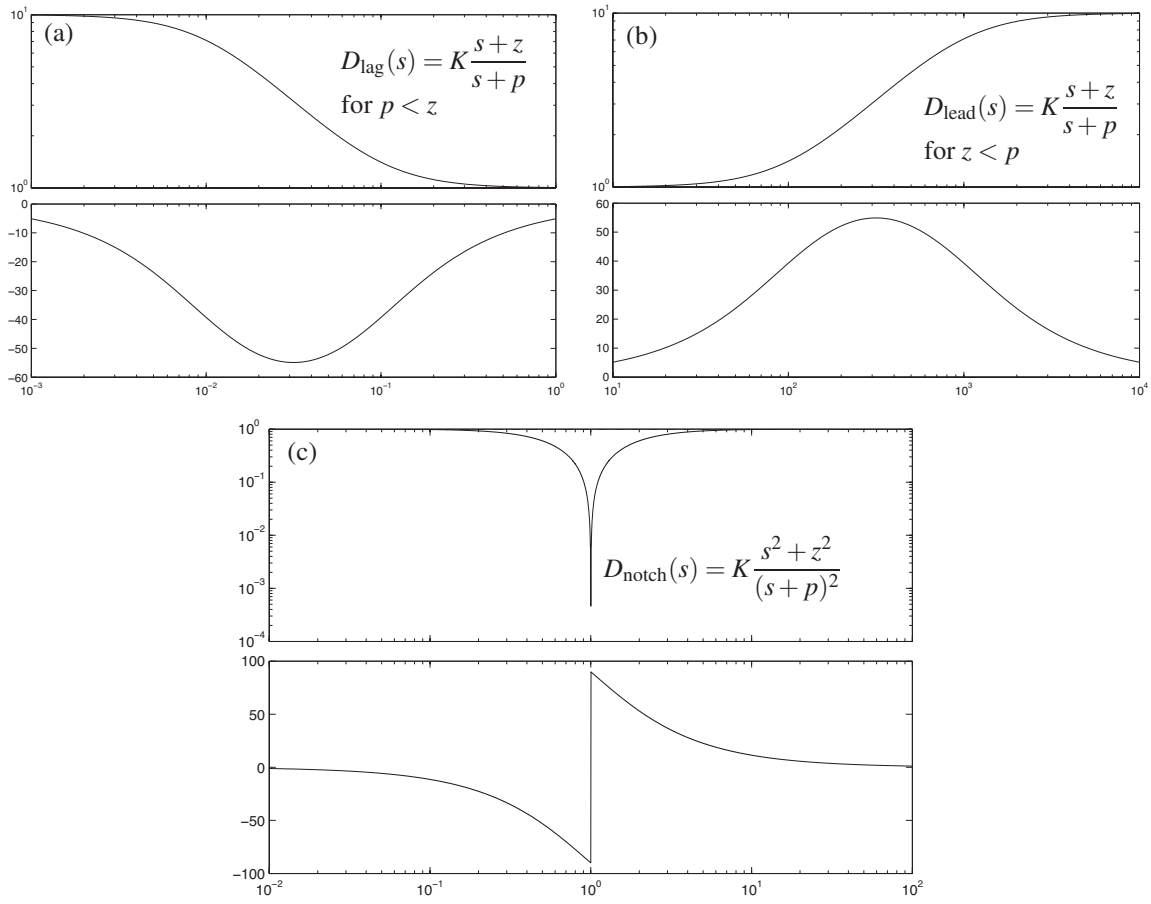


Figure 18.17: Bode plots of (a) a **lag** controller $D_{\text{lag}}(s) = K(s+z)/(s+p)$ for $p < z$ (with $K = 1$, $p = .01$, and $z = .1$), which is the responsible way of applying *integral* control action over a targeted range of frequencies, (b) a **lead** controller $D_{\text{lead}}(s) = K(s+z)/(s+p)$ for $z < p$ (with $K = p/z$, $z = 100$ and $p = 1000$), which is the responsible way of applying *derivative* control action over a targeted range of frequencies, and (c) a **notch** controller $D_{\text{notch}}(s) = K(s^2+z^2)/(s+p)^2$ (with $K = 1$, $z = 10$ and $p = 10$), which provides a high-performance alternative to lead control for oscillatory plants which are accurately modeled. The cascade of the first two of these controllers, $D_{\text{lead-lag}}(s) = D_{\text{lag}}(s) \cdot D_{\text{lead}}(s)$, gives the Bode plot in Figure 18.13b. For $p/z = 10$, the maximum phase lead of the lead controller, at $\omega = \sqrt{pz}$, is about 55°, with increasing values of p/z giving increasing (up to 90°) phase lead (see Exercise 18.8). The lag controller, on the other hand, gives a boost in the low-frequency gain by a factor of z/p. Adjusting the lead and/or lag portions of a controller on their own, or sometimes using a notch instead of a lead, allows one to design a controller that much more precisely meets ones needs across all frequencies in the Bode plot than typical PID tuning allows.

controller (18.22a). As seen in Figure 18.18, *the system with PID implemented and the system with lead-lag implemented have nearly identical step responses, though the lead-lag controller has reduced gain at both high frequencies and low frequencies, thereby alleviating the two key problems with PID mentioned previously*. Note that a prefactor $P = 1.1$ [see (18.17)] has been used in the case of the lead-lag controller; since the lead-lag controller's finite low-frequency open-loop gain, $G(i\omega)D(i\omega)$ for small ω, is relatively large (> 10), the tracking of the system will still be quite good even in the presence of modeling errors.

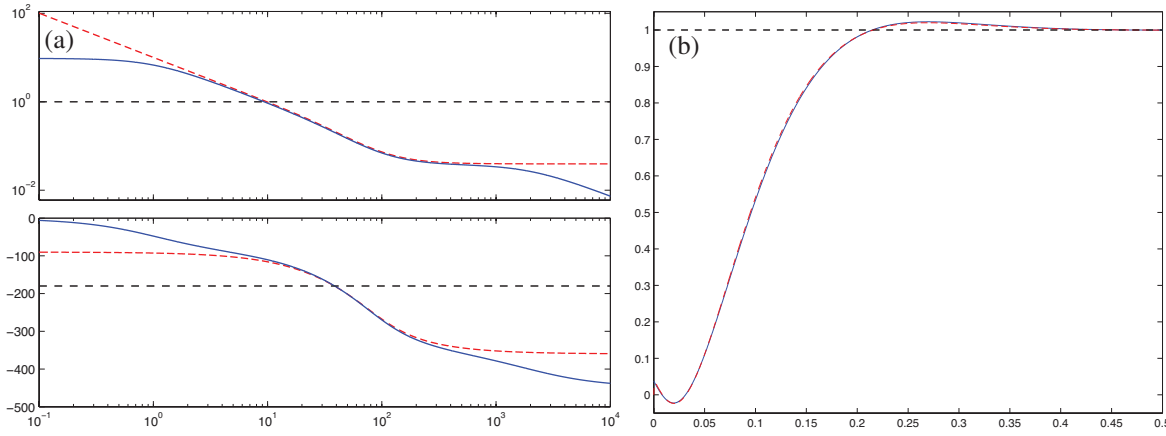


Figure 18.18: Comparison of (dashed) the best PID controller determined in Example 18.1 for the automobile cruise control problem [see (18.22a)] with (solid) the corresponding lead-lag controller of Example 18.2 [see (18.22b)] with $z/p = 100$ in the lag and $p/z = 20$ in the lead. (a) Bode plots, (b) step responses.

Example 18.3 Speeding up a (generic) first-order system with lag or lead control

The previous example highlighted the utility of lag compensation to boost the open-loop gain at frequencies well below the crossover frequency, thereby reducing the steady-state error of a step response. This is, perhaps, the primary use of lag control; however, lag control isn't always used below crossover.

Consider now the generic form of a linear first-order system

$$\frac{dy(t)}{dt} + ay(t) = C\left[\frac{du(t)}{dt} + bu(t)\right] \quad \Leftrightarrow \quad \frac{Y(s)}{U(s)} = G(s) = C\frac{s+b}{s+a},$$

where a and b are real, with $a \neq b$. If this system is controlled with proportional feedback $u = Ky$ then, by the discussion in the first paragraph of (18.2.1), the pole of the closed-loop system is given by

$$(s+a)/K + C(s+b) = 0 \quad \Rightarrow \quad s = -\frac{a+KCb}{1+KC}.$$

That is, for small values of K, the pole of the closed-loop system is near the pole of the open-loop system $s = -a$, and for large values of K, the pole of the closed-loop system is near the zero of the open-loop system $s = -b$. For intermediate values of K, the pole of the closed-loop system is somewhere in-between (on the real axis between $s = -a$ and $s = -b$). For some systems, this is adequate to move the closed-loop pole sufficiently far to the left to achieve the specified rise time and settling time criteria. For other systems, however, none of these possible closed-loop pole locations is acceptable. Rather than flip the sign of the gain and use a potentially high feedback amplitude (following the 0° root locus rules to determine a K that achieves stability and an adequate rise time and settling time), one can instead use (positive-gain) lag or lead control, as appropriate, and an approximate pole-zero cancellation in the LHP (see §18.2.1.2).

To illustrate, if the plant zero is to the left of the plant pole and the plant zero is stable (that is, $b > 0$), one may use a lag controller

$$D_{\text{lag}}(s) = K\frac{s+z}{s+p} \quad \text{with} \quad z > p \approx b$$

to "replace" the slow stable plant zero with a faster stable controller zero; tuning K sufficiently high then provides the requisite rise time and settling time in the closed-loop system, as well as providing low steady-state error in the step response even in the presence of significant uncertainty in the modeling of $G(s)$.

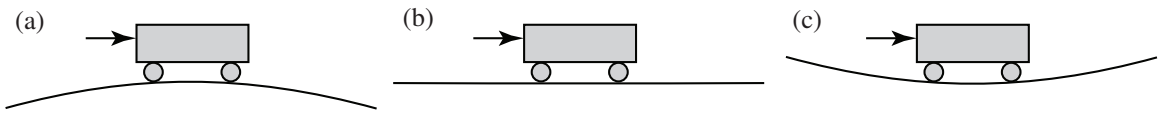


Figure 18.19: A second-order system governed by (18.23) in: (a) an unstable configuration with $\alpha < 0$, (b) a neutrally-stable configuration with $\alpha = 0$, and (c) an oscillatory configuration with $\alpha > 0$.

If, on the other hand, the plant pole is to the left of the plant zero and the plant pole is stable (that is, $a > 0$), one may use a lead controller

$$D_{\text{lead}}(s) = K\frac{s+z}{s+p} \quad \text{with} \quad p > z \approx a$$

to replace the slow stable plant pole with a faster stable controller pole; keeping K sufficiently low then again provides the requisite rise time and settling time in the closed-loop system. To achieve sufficiently low steady-state error in the step response even in the presence of significant uncertainty in the modeling of $G(s)$, we might need to boost up the low-frequency gain of the controller $D(s)$ beyond that provided by the lead controller described above; this may be accomplished via additional lag compensation, applied below the crossover frequency, as seen in Example 18.2.

Example 18.4 Stabilizing a second-order system (a rolling cart) with lead or notch control

Consider now the problem of controlling the position of a simple rolling cart (see Figure 18.19) governed by

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + mg\sin\left(\frac{dy}{dx}\right) = u, \quad y = \alpha x^2 \quad \xrightarrow[\text{linearization}]{} \quad \frac{d^2x}{dt^2} + \left(\frac{c}{m}\right)\frac{dx}{dt} + (2\alpha g)x = \left(\frac{1}{m}\right)u$$
$$\Rightarrow \quad \frac{X(s)}{U(s)} = \frac{C}{s^2 + a_1 s + a_0} \quad \text{where} \quad a_1 = \frac{c}{m}, \;\; a_0 = 2\alpha g, \;\; C = \frac{1}{m}. \tag{18.23}$$

Case (a): $\alpha = -1$, $c = 0$, $m = 1$. The plant in this case, $G(s) = C/[(s+\overline{p})(s-\overline{p})]$ with $\overline{p} = \sqrt{2|\alpha|g}$, is unstable. Lead control designed using the root locus with an approximate pole-zero cancellation in the LHP, $D_{\text{lead}}(s) = K(s+z)/(s+p)$ with $p > z \approx \overline{p}$ and the controller pole $s = -p$ taken sufficiently far into the LHP, stabilizes this system, as illustrated in Figure 18.20b. However, lead control designed using the Bode plot, with the pole/zero pair [and, thus, the boost in gain caused by $D_{\text{lead}}(s)$] centered at the desired crossover frequency as determined from the rise time specification (that is, with $\sqrt{pz} = \omega_c$, as shown in Figure 18.20d), improves the response (see Figure 18.20e) for a given value of p/z in the lead controller, even though the root locus in this case is a bit more complicated (Figure 18.20c). Note finally that a bit of damping in the plant (taking $c > 0$) changes the pole locations slightly, but leaves the control problem essentially unchanged.

Case (b): $\alpha = 0$, $c = 0$, $m = 1$. The plant in this case, $G(s) = C/s^2$, is known as a **double integrator**, and the lead control strategy using the Bode plot, as described in Case (a), works effectively (see Exercise 18.10).

Case (c): $\alpha = 1$, $c = 0$, $m = 1$. The plant in this case, $G(s) = C/[(s+i\overline{p})(s-i\overline{p})]$ with $\overline{p} = \sqrt{2\alpha g}$, is known as a **second-order oscillator**. Lead control can be designed for this system using the Bode plot as in the previous two cases, leading to the step response illustrated in Figure 18.21. Note the increased overshoot in this case due to the extra closed-loop pole on the negative real axis; this increased overshoot is generally the weakness of using lead control on an oscillatory plant. The strength of this control design is that it generally works adequately for a broad range of $\overline{p}$; that is, the control design is *robust* to uncertainty in the plant parameters.

If the plant is known accurately, a notch controller $D_{\text{notch}}(s) = K(s^2+z^2)/(s+p)^2$ (see Figure 18.17) can be used instead in case (c), approximately "knocking out" the neutrally-stable plant poles with nearby

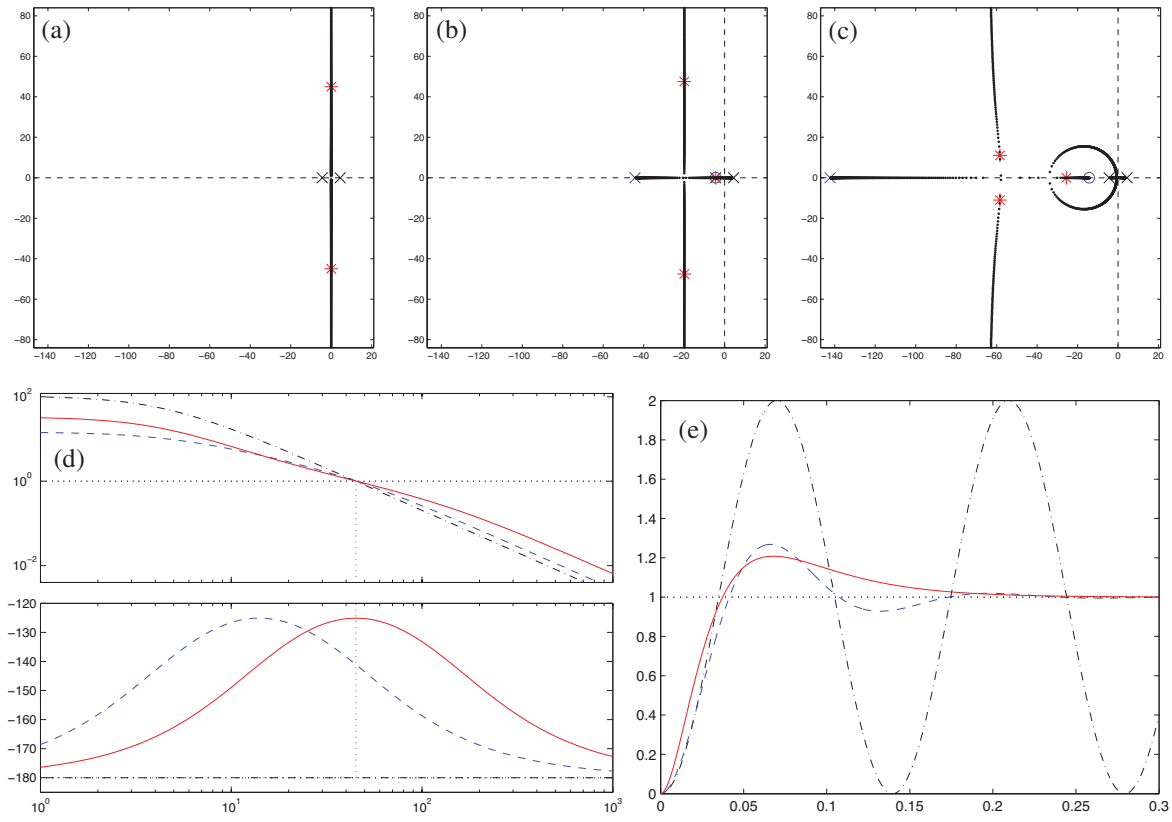


Figure 18.20: Control of the unstable second-order system of Figure 18.19a. (a) Root locus using proportional control. (b) Root locus using lead control designed for simple pole/zero cancellation. (c) Root locus using lead control designed (using the Bode plot) for maximum performance. (d) Bode plot and (e) step response using (dot-dashed) proportional control, (dashed) lead control designed (using root locus) for simple pole/zero cancellation, taking $p/z = 10$, and (solid) lead control designed for maximum PM given $p/z = 10$ (by taking $\sqrt{pz} = \omega_c$), with gains as marked in (a), (b), and (c).

controller zeros, and supplanting these oscillatory poles with a pair of stable controller poles sufficiently far into the LHP to achieve the desired rise time and settling time, as illustrated in Figure 18.22. The strength of this control design is its performance: the closed-loop system in this case doesn't suffer from the increased overshoot experienced when using lead control. The weakness of this control design is that it is sensitive to uncertainty in the plant parameters. Recall that cancellation of plant poles/zeros with controller zeros/poles must always be considered as approximate, as plant parameters are never known precisely. If a pole/zero cancellation is well into the LHP, the fact that the cancellation is only approximate isn't a problem, as discussed in §18.2.1.2. However, if the approximate pole/zero cancellation is on the imaginary axis, then it is critical that the controller zero be placed *on the appropriate side* of the plant pole, so that the small branch of the root locus that results from the inexact nature of the cancellation swings into the LHP instead of the RHP. In other words, the controller zero should not be put at the expected *value* of the plant pole, but must instead be put, conservatively, on the appropriate side of the expected *range* of where the plant pole might lie. The smaller this range is (that is, the more certain you are about the plant parameters), the better the performance that can be obtained, as this cancellation can be made more precise without risking instability. If the range is too large, the more robust lead controller should be used instead.

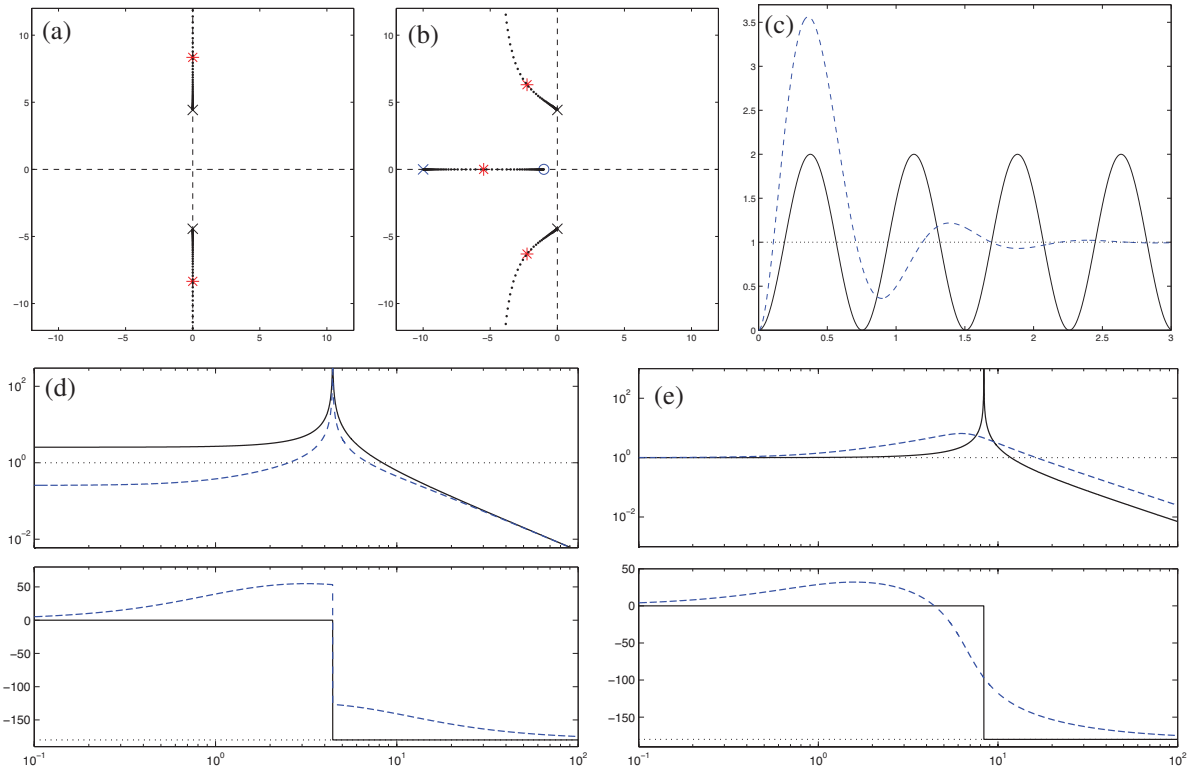


Figure 18.21: Control of the oscillatory second-order system of Figure 18.19c using proportional control and lead control. (a) Root locus using proportional control. (b) Root locus using lead control. (c) Step response, (d) open-loop Bode plot, and (e) closed-loop Bode plot using (solid) proportional control and (dashed) lead control taking $p/z = 10$, with gains as marked in (a) and (b). Note the peaks in the open-loop and (in the case of proportional control) closed-loop Bode plots due to the open-loop and (in the case of proportional control) closed-loop poles on the imaginary axis. Unfortunately, when using lead control in this case, there is significant overshoot in the step response due to the third pole on the negative real axis.

18.3.3 Sensor dynamics, and noise suppression via low-pass and notch filtering

All sensors have associated with them some level of noise, the intensity of which generally varies as a function of frequency. This noise intensity often does not diminish very quickly with increasing frequency[13], though the strength of the signal of interest usually does; thus, sensors ultimately become essentially useless at high frequencies, and feedback applied at such high frequencies is counterproductive. The typical Bode plot depicted in Figure 18.6 therefore has the important constraint of reduced open-loop gain at high frequencies.

To suppress the high-frequency gain, a low-pass filter (see §17.5.2) is sometimes necessary. Recall the Bode plots of the simple first-order and second-order low-pass filters given in Figure 17.17, and of the higher-order Butterworth and Bessel filters given in Figure 17.19. Unfortunately, such filters generally bring with them significant *phase delay*, even well below the cutoff frequency, as indicated in these Bode plots. Recall also that one generally designs a feedback system with a certain minimum phase at the crossover frequency, given by $-180°+$ PM where, as suggested by (18.15), PM $\approx \zeta \cdot 100$, and ζ is the desired damping selected

[13] In many sensors (e.g., thermocouples) the noise intensity can actually be well approximated as **white** (that is, with intensity nearly constant across a broad range of frequencies, like that of white light; for further discussion, see §5.3.1).

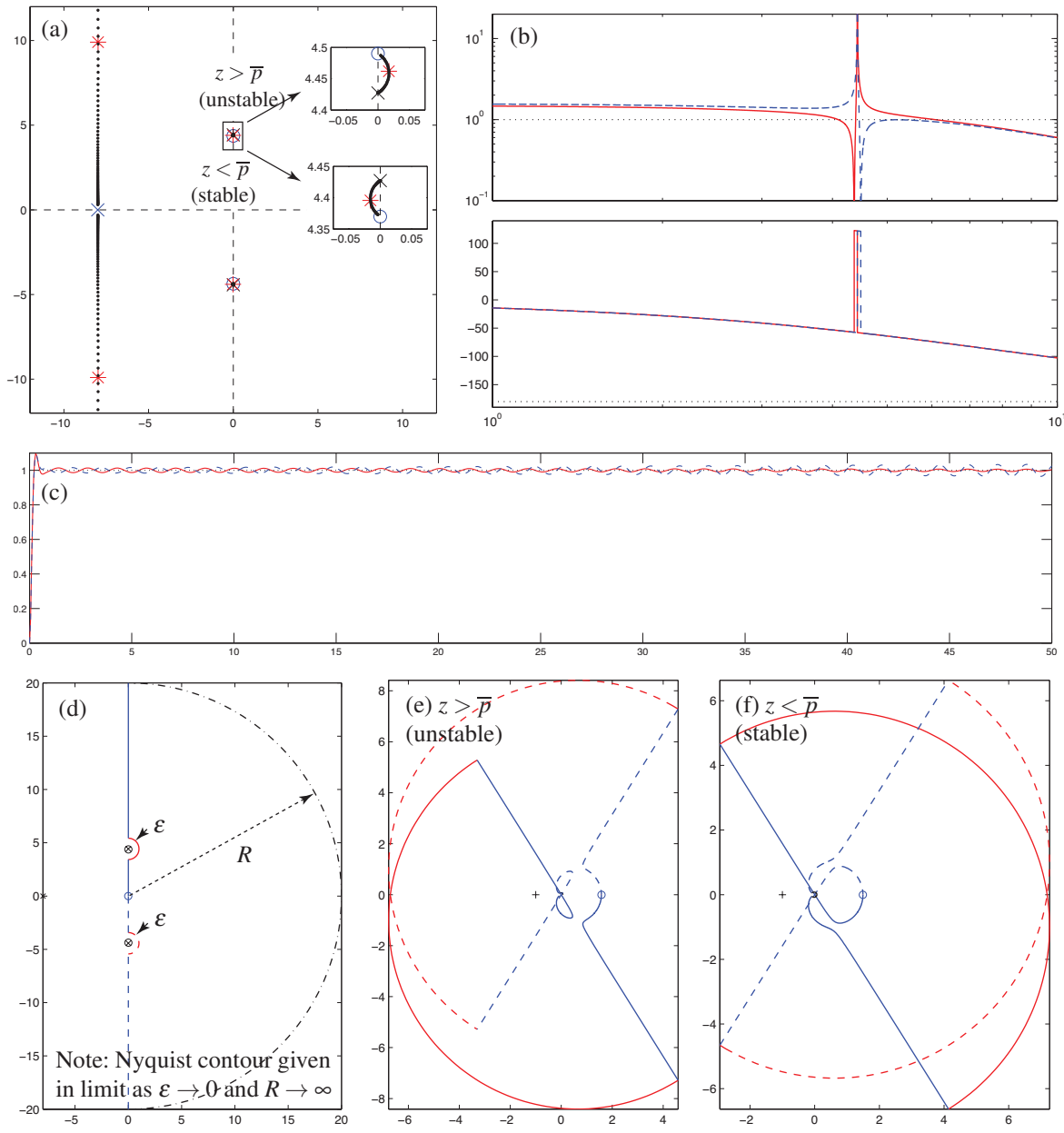


Figure 18.22: Control of the oscillatory second-order system of Figure 18.19c using notch control for two possible values for the controller zeros. (a) Root locus. (b) Bode plot and (c) step response taking (solid) $z < \overline{p}$ and (dashed) $z > \overline{p}$, with gains as marked in (a). The inexact cancellation of the plant pole $\overline{p}$ by the controller zero z leads to a stable closed-loop system if $z < \overline{p}$, and instability if $z > \overline{p}$. This is seen in the root locus very near the imaginary axis (see insets); it is see in the step response only by looking over a long period of time, as the (initially, small) oscillations grow or decay slowly. It is difficult to determine stability from the Bode plot. Designing an appropriate Nyquist contour in the s plane, (d), and mapping via $L(s) = G(s)D(s)$ to obtain the corresponding Nyquist plots [see (e) for $z > \overline{p}$, and (f) for $z < \overline{p}$], it is seen in (e) that the contour encircles the $L = -1$ point twice when $z > \overline{p}$, thus [by Fact 18.5] indicating two unstable closed-loop poles, whereas in (f) the contour does not encircle the $L = -1$ point when $z < \overline{p}$, indicating a stable closed loop.

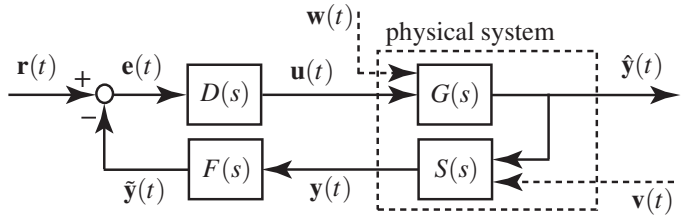


Figure 18.23: Closed-loop system with a **sensor dynamics block** $S(s)$ and a **filter block** $F(s)$ in the return portion of the feedback loop, where $\mathbf{y}(t)$ denotes the actual measurement, $\hat{\mathbf{y}}(t)$ denotes the ideal measurement (that is, the quantity of interest), and $\tilde{\mathbf{y}}(t)$ denotes the filtered measurement. The **closed-loop transfer function** is $T(s) = \hat{Y}(s)/R(s) = G(s)D(s)/[1 + G(s)D(s)F(s)S(s)]$, whereas the **sensitivity** [of $\hat{\mathbf{y}}(t)$ to the measurement noise $\mathbf{v}(t)$] is $\hat{Y}(s)/V(s) = G(s)D(s)F(s)S(s)/[1 + G(s)D(s)F(s)S(s)]$. A low-pass filter $F(s)$ [with $|F(i\omega)| \to 0$ as $\omega \to \infty$] in the return portion of the feedback loop reduces the sensitivity of the quantity of interest $\hat{\mathbf{y}}(t)$ to the high-frequency components of the measurement noise $\mathbf{v}(t)$, but not to the high-frequency components of the reference input $\mathbf{r}(t)$. Similarly, a notch filter $F(s)$ (see Figure 18.17c) reduces the response of the system to a dominant frequency in the spectrum of the measurement noise $\mathbf{w}(t)$.

to meet the overshoot specification, as suggested by (17.57). If a low-pass filter is used with cutoff frequency within an order of magnitude or so of the crossover frequency of the system, this phase loss should be accounted for during the controller design.

Thus, the selection and tuning of a low-pass filter to suppress the response of a system to the high-frequency noise picked up by the sensors is nontrivial, and reflects a compromise between the loss of phase and the rate of roll-off of the gain near the cutoff frequency of the low-pass filter. Many sensors already have low-pass filters built in, or superciliously added by the experimentalist constructing the sensor. As the responsible controls engineer, if one is designing for maximum performance, it is useful to identify exactly what kind of filter is already implemented in such settings, so the phase loss associated with this filter at crossover can be accounted for during the controller design.

Some sensors have inherent dynamics that may be significant near the target crossover frequency. For example, a **MEMS**[14] accelerometer is essentially a small floating mass balanced by a spring, and therefore has a response magnitude that is inherently a function of the forcing frequency[15]. Some **COTS** (**commercial off-the-shelf**) sensors have appropriate compensation circuits already built in which attempt to counteract the effects of such sensor dynamics, while others allow the controls engineer access to the raw measured signal, allowing the appropriate compensation to be developed as a part of the closed-loop system design.

In order to maximize performance [that is, the fidelity of the system response to a reference input $\mathbf{r}(t)$] while minimizing sensitivity [that is, the response of the system to measurement noise $\mathbf{v}(t)$], control in the presence of sensor dynamics and/or low-pass filtering is often implemented in the manner indicated in Figure 18.23. Finally, note that measurement noise sometimes has a dominant frequency component at a certain frequency (often, 50 or 60 Hz); this **buzz** can be mitigated with a notch filter in the return portion of the feedback loop in a similar fashion (see Example 19.13).

[14] A **Micro-Electro-Mechanical-System** is a very small physical system made using the same mask/expose/etch technology used to manufacture silicon chips. Today, this is a very mature technology, and several types of MEMS sensors are mass produced on a large scale. For example, MEMS accelerometers are used in airbag deployment systems in automobiles, video game controllers & smartphones, and laptop hard disk drives (to park the read head, before impact, if the laptop is dropped, thus preventing damage to the disk). MEMS gyros are also mass produced, albeit on a smaller scale, for use in video game controllers & smartphones.

[15] An expanded dynamic range of such devices may generally be obtained by active electrostatic **force rebalancing**; that is, by closing a control loop around the sensor itself, applying an electrostatic force that is just sufficient to keep the floating mass from moving, then measuring the electrostatic force required to "rebalance" the floating mass within the device in order to determine the acceleration applied to the entire system. This essentially supplants the mechanical time constant of the device, $\sqrt{m/k}$, with the electrical time constants of the sensor control circuit, RC and L/R, which are generally much faster.

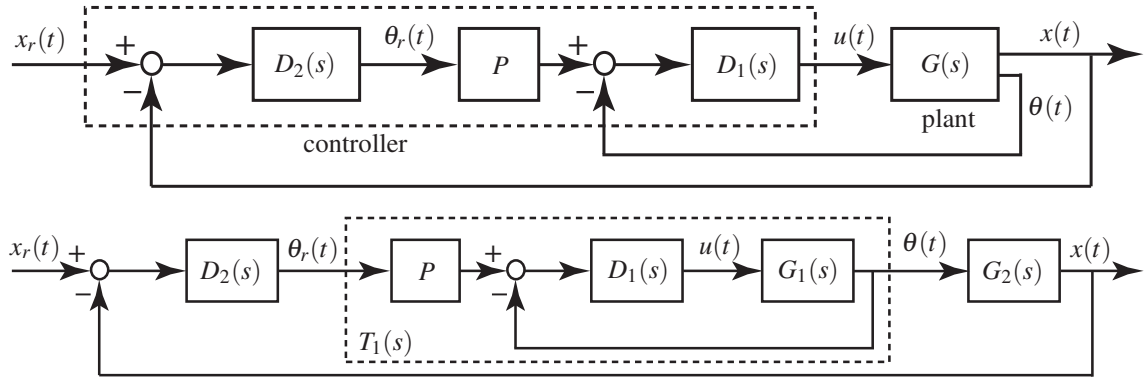


Figure 18.24: Successive loop closure of a SIMO plant $G(s)$ designed with "fast" $\theta(t)$ dynamics and "slow" $x(t)$ dynamics: (top) block diagram of physical realization; (bottom) idealization for the purpose of control design. Example 18.5 (see Figure 17.5a), Exercise 18.15, and Exercise 18.16 are all of this general form.

18.3.4 Successive loop closure (SLC) leveraging frequency separation

Classical control design techniques are used primarily for SISO systems. However, many systems are characterized by a substantial **frequency separation** of the modes of interest; that is, the system modes are characterized by natural frequencies which are an order of magnitude or so apart, or the frequencies of such modes can at least be made this way with control feedback. In such systems, it is straightforward to nest a "fast" feedback loop within one or more outer, "slower" feedback loops in order to string multiple SISO controllers together to stabilize a SIMO system with multiple unstable modes, leveraging the observation that a slow outer loop doesn't significantly alter the dynamics of a fast inner loop. As with the other controls problems considered previously, this control strategy is best introduced by example.

Example 18.5 Stabilization of an unstable fourth-order system (an inverted pendulum on a cart)

We now consider the control of the classic linearized inverted pendulum problem illustrated in Figure 17.5a and depicted in block diagram form in Figure 18.24, taking typical laboratory values for the constants: $m_c = 2$ kg for the mass of the cart, $m_p = 1$ kg for the mass of the pendulum (a uniform rod), $\ell = 1$ m for the distance from the cart to the center of mass of the pendulum (the half-length of the rod), $I_p = m_p \ell^2/3$ for the moment of inertia of the pendulum about its center of mass, $g = 9.8$ m/s^2 for the acceleration due to gravity, and $c_1 \approx 0.01$ and $c_2 \approx 0.05$ for the friction coefficients. As derived in Example 17.5, small perturbations of this system from the inverted state are governed by the coupled equations

$$\text{pendulum dynamics:} \quad (I_p + m_p\ell^2)\frac{d^2\theta}{dt^2} + c_1\frac{d\theta}{dt} - m_p g\ell\,\theta = m_p\ell\frac{d^2x}{dt^2}, \tag{18.24a}$$

$$\text{cart dynamics:} \quad (m_c + m_p)\frac{d^2x}{dt^2} + c_2\frac{dx}{dt} = m_p\ell\frac{d^2\theta}{dt^2} + u, \tag{18.24b}$$

Taking the Laplace transform of both (18.24a) and (18.24b) and combining appropriately, we may write

$$G_1(s) = \frac{\Theta(s)}{U(s)} = \frac{b_1 s}{a_3 s^3 + a_2 s^2 - a_1 s - a_0}, \quad G_2(s) = \frac{X(s)}{\Theta(s)} = \frac{\overline{b}_2 s^2 + \overline{b}_1 s - \overline{b}_0}{s^2} \tag{18.25}$$

where $b_1 = m_p\ell$, $a_3 = (m_c + m_p)I_p + m_c m_p \ell^2$, $a_2 = (m_c + m_p)c_1 + (I + m_p\ell^2)c_2$, $a_1 = (m_c + m_p)m_p g\ell + c_1 c_2$, $a_0 = m_p g\ell c_2$, $\overline{b}_2 = (I + m_p\ell^2)/(m_p\ell)$, $\overline{b}_1 = c_1$, and $\overline{b}_0 = g$, thereby facilitating interpretation of the

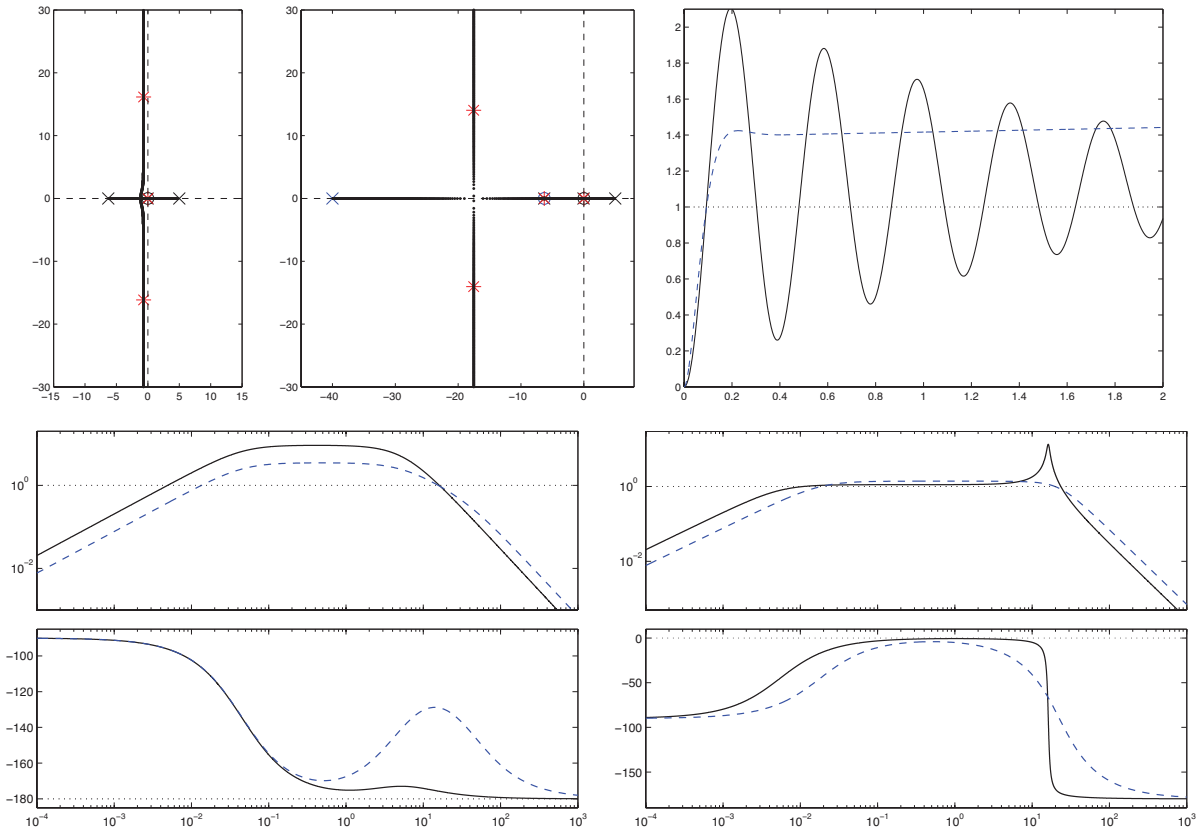


Figure 18.25: Classical control design techniques applied to the *inner loop* of the successive loop closure problem considered in Example 18.5. (a) Root locus using proportional control. (b) Root locus using lead control. (c) Step response, (d) Bode plot, and (e) closed-loop Bode plot using (solid) proportional control and (dashed) lead control, with gains as marked in (a) and (b).

SIMO system (18.24) in the **successive loop closure** configuration depicted in Figure 18.24. The related (oscillatory) **hanging pendulum** problem is considered in Exercise 18.15.

The secret to extending the essentially SISO control techniques discussed thus far to this problem is to first ignore the state $x(t)$ associated with what we have assigned as the "slow" outer loop, and focus our attention at first simply on designing an inner-loop controller $D_1(s)$ to stabilize (relatively quickly; say, with a rise time of $t_r \approx 0.1$ sec) the variable $\theta(t)$ back to a reference state $\theta_r(t)$, nominally taken to be zero.

Taking $c_1 = c_2 = 0$ reduces the plant considered in the inner loop to $G_1(s) = b_1/(a_3 s^2 - a_1)$, and thus the inner loop control problem reduces to the problem solved in Figure 18.20. In the problem considered now, neither of these friction coefficients is zero; however, as shown in Figure 18.25, using lead control and the root locus and Bode design techniques presented previously, stabilization of the high-speed dynamics of the inner loop is again quite straightforward. The subsystem to be controlled in the inner loop, $G_1(s)$, has an open-loop zero at $s = 0$ and open-loop poles at $s \approx -6.3$, $s \approx -0.045$, and $s \approx 5$. Thus, proportional control results in a pair of lightly damped, relatively fast closed-loop poles, and one very slow unstable closed-loop pole on the positive real axis close to $s = 0$. As before, the dominant fast modes of the system are easily stabilized with lead compensation. Due to the proximity of the open-loop zero at $s = 0$ to the open-loop pole at $s \approx -0.045$, the coefficient of the (exponentially growing) component of the response associated with the very slow unstable closed-loop pole is nonzero but small, as illustrated in the (dashed) step response in

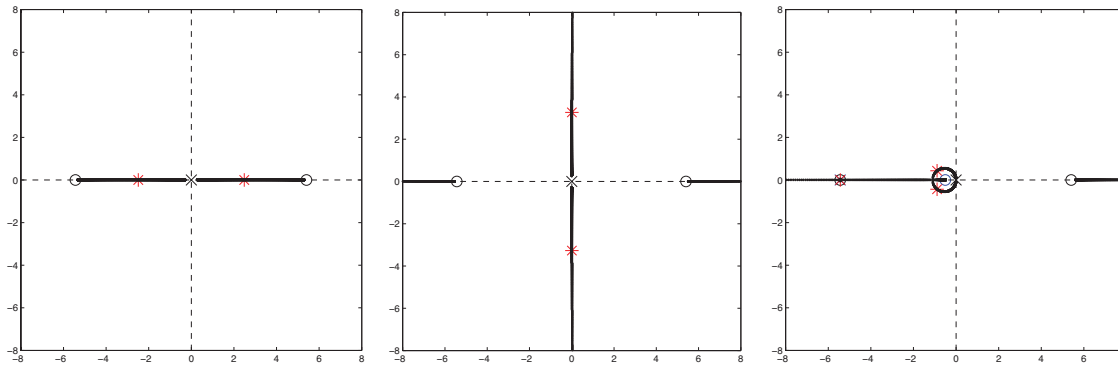


Figure 18.26: Classical control design techniques applied to the *idealized outer loop* [taking $T_1(s) = 1$] of the successive loop closure problem considered in Example 18.5. (a) Root locus using proportional control. (b) Root locus using proportional control with negative gain. (c) Root locus using lead control with negative gain. The corresponding Bode plot and step response are given as the (dashed) curves of Figures 18.27b-c.

Figure 18.25c. Again, it is difficult to determine stability versus instability directly from a Bode plot, though this would be straightforward to determine from a Nyquist plot. This is unnecessary in the present situation, as we already know that the inner loop is unstable by looking at the root locus.

Note in Figure 18.25c that the step response of the controlled inner loop rises quickly to 1.4, and begins to drift from there due to the slow unstable inner-loop pole. Thus, as suggested by Figure 18.11, we take $P = 1/1.4$ in Figure 18.24. For the time being, we simply ignore the unstable slow pole of the inner loop, as the outer-loop control design, considered below, changes significantly the low-speed dynamics of the system.

Assuming the closed inner loop has a transfer function of $T_1(s) \approx 1$ at the frequencies of interest (see dashed curve in Figure 18.25e, scaled by $P = 1/1.4$), we next design an outer-loop controller $D_2(s)$ to stabilize the outer loop relatively slowly (say, with $t_r \approx 1$ sec). Noting $G_2(s)$ in (18.25), it is seen in Figure 18.26 that a $D_2(s)$ with negative gain and some lead compensation effectively stabilizes the idealized outer-loop system.

Finally, applying the outer-loop controller $D_2(s)$ to the *actual* inner loop, taking $T_1(s) = P \cdot G_1(s)D_1(s)/[1 + G_1(s)D_1(s)]$, results in the root locus, Bode plot, and step response in Figure 18.27. The root locus of the full outer-loop system (Figure 18.27a) indicates essentially the same behavior at low frequencies (small $|s|$) as does the root locus of the idealized outer-loop system (Figure 18.26c), as well as a pair of well-damped closed-loop poles near $s \approx -16.3 \pm 3$i, as also seen in the root locus of the inner-loop system (Figure 18.25b), and a couple of approximate pole/zero cancellations at $s \approx -6.3$ and $s \approx -5.4$. The Bode plot of the full outer-loop system (solid curve in Figure 18.27b) is similar to that of the idealized outer-loop system (dashed curve in Figure 18.27b), but exhibits a significant loss of phase at high frequencies. It is thus clearly evident in this Bode plot why frequency separation is needed in order to apply the SLC approach: the loss of phase due to the (fast) inner loop dynamics erodes the PM of the outer loop if the crossover frequency ω_c of the outer loop is increased to be too close to the bandwidth frequency ω_{BW} of the closed inner loop, as evident in the closed-loop Bode plot of the inner loop in Figure 18.25e. Due to the only slight loss of phase from the inner loop dynamics for the rise times selected in the present control solution ($t_r \approx 0.1$ sec for the inner loop, and $t_r \approx 0.5$ sec for the outer loop), the step response of the full outer-loop system has only slightly higher overshoot than the step response of the idealized outer-loop system, as indicated in Figure 18.27c.

We now reëxamine the slow outer-loop dynamics of the full system, revisiting the interplay between the slow dynamics of the closed inner loop and the outer loop. As mentioned above, and illustrated clearly by the root loci of Figure 18.26 and the Bode plot and step response given as the dashed curves of Figures 18.27b-c, control of the idealized outer-loop system $G_2(s)$ [with two poles at the origin and two zeros at $s \approx \pm 5.4$] is straightforward using negative gain and lead compensation. The closed inner loop $T_1(s)$ has poles at $s \approx 0.018$ and $s \approx -17.5 \pm 14$i, a zero at $s = 0$, and an approximate pole/zero cancellation at $s \approx -6.3$. When forming

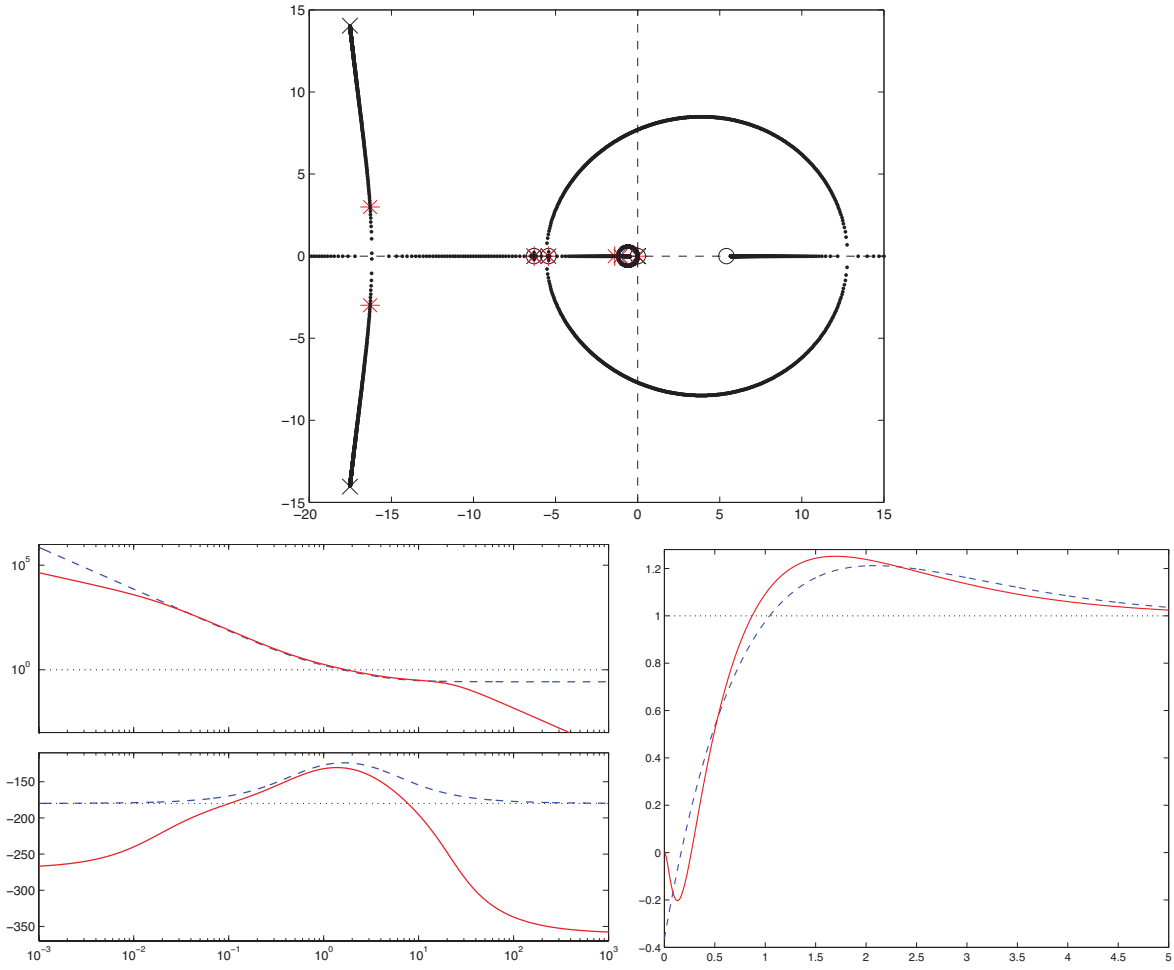


Figure 18.27: (a) Root locus of the *full outer loop* [taking the actual closed inner loop $T_1(s)$] of the successive loop closure problem considered in Example 18.5. (b) Bode plot, and (c) step response of (dashed) the idealized outer loop corresponding to the root locus in Figure 18.26c, and (solid) the actual outer loop corresponding to the root locus in (a).

the product $G_2(s)T_1(s)$, one of the poles at the origin of $G_2(s)$ cancels (*exactly*[16]) the zero at the origin of $T_1(s)$, leaving two poles remaining in the vicinity of the origin (at $s = 0$ and $s \approx 0.018$), thus leading to a root locus in the vicinity of the origin in Figure 18.27a which is quite similar to that seen in Figure 18.26c. Finally, note in the Bode plot of Figure 18.27b that, since there is only one pole exactly at the origin in the full $G_2(s)T_1(s)$ system rather than two poles at the origin as when considering the control of $G_2(s)$ on its own, the slope of the Bode plot for small ω is -1 instead of -2; this fact presents no significant problems.

[16]Recall from §18.2.1.2 that *approximate* cancellations of controller zeros/poles with plant poles/zeros are problematical in the RHP, and must never be attempted; similarly, as seen in Figure 18.22, approximate pole/zero cancellations on the imaginary axis (including near the origin) must be handled with extreme care to ensure stability. However, the pole/zero cancellation when forming the $G_2(s)T_1(s)$ product in this case comes from our *representation* of the SIMO plant as the casecade of two separate SISO blocks, $G_1(s)$ and $G_2(s)$. The pole/zero cancellation arising from this representation of a single plant as a two-block cascade is exact. Note that some software packages (e.g., Matlab) must be used with care in order to realize this exact pole/zero cancellation in the simulation of the outer loop, as small numerical errors in the calculation of $T_1(s)$ can sometimes conceal (numerically) an exact pole/zero cancellation of this sort.

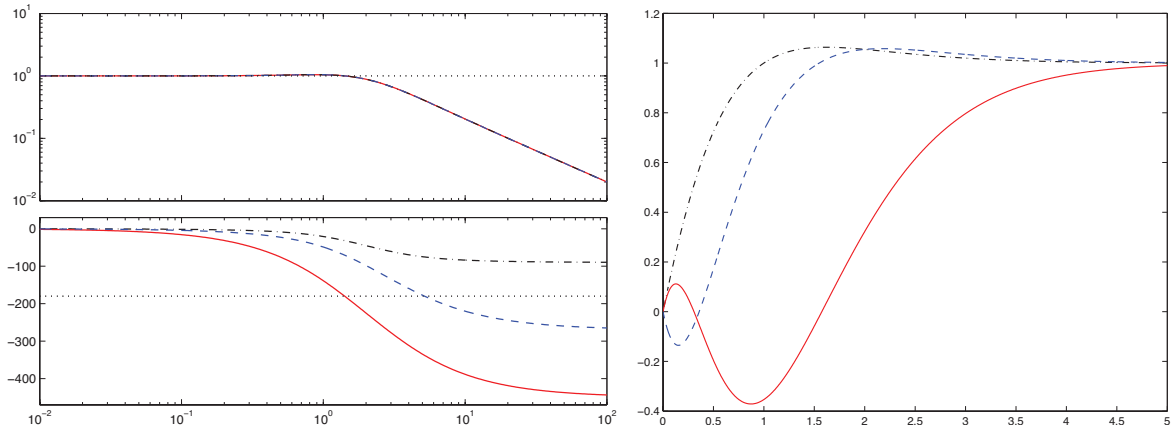


Figure 18.28: (a) Bode plot and (b) step response of three stable systems: (dot-dashed) $T_1(s)$, (dashed) $T_2(s)$, and (solid) $T_3(s)$ [see (18.26)].

18.3.4.1 Nonminimum phase systems

We conclude this section with a brief discussion of a characteristic behavior exhibited by the overall response (given by the solid curve in Figure 18.27c) to a step reference input, x_r, to the inverted pendulum system, for which the control feedback was determined using SLC in Example 18.5. In particular, note that this step response goes the wrong way (negative) before it goes the right way (positive). This behavior is not a fluke, and is characteristic of the step response in any stable (or stabilized) system with a single RHP zero. Any child who has balanced a meterstick (or, in the States, a yardstick...) on his hand is intuitively familiar with the need to take a small step backward before walking forward with such a system.

To consider this effect in greater detail, consider the three stable systems

$$T_1(s) = 2\frac{(s+1)(s+4)}{(s+2)^3}, \quad T_2(s) = -2\frac{(s+1)(s-4)}{(s+2)^3}, \quad T_3(s) = 2\frac{(s-1)(s-4)}{(s+2)^3}. \tag{18.26}$$

All three of these systems have identical (stable) poles; it is only the sign of the zeros, and the sign of the overall gain, which distinguishes them. The Bode plots and step responses of these three systems are given in Figure 18.28. The magnitude of the Bode plot is identical in all three cases. The phase change between the left end of the Bode plot and the right end of the Bode plot is minimized by $T_1(s)$ [with no RHP zeros, called a **minimum-phase system**], and is maximized by $T_3(s)$ [with only RHP zeros, called a **maximum-phase system**]; this phase change takes some intermediate value for $T_2(s)$ [with both LHP and RHP zeros, sometimes called a **mixed-phase system**]; note also that maximum-phase systems and mixed-phase systems are commonly referred to simply as **nonminimum phase systems**[17]. Note in Figure 18.28b that the step response in the presence of one RHP zero goes the wrong way before going the right way, and the step response in the presence of two RHP zeros goes the right way, then the wrong way, then the right way[18].

The characterization given above—specifically, that a stable transfer function with stable zeros is characterized by the minimum change in phase from the left side of the Bode plot to the right side of the Bode plot—extends to any transfer function for which the degree of the denominator is greater than or equal to the degree of the numerator [that is, for a **proper** CT transfer function (see §17.3.3.1), or a **causal** DT transfer function (see §17.4.3.2)].

[17] Thus, the next time you are in an elevator at a conference that goes the wrong way before it goes the right way, say "oh! nonminimum phase!" and see who gets the joke, thus identifying immediately those of your colleagues who know classical control theory...

[18] If your elevator does this, you should probably consider changing hotels!

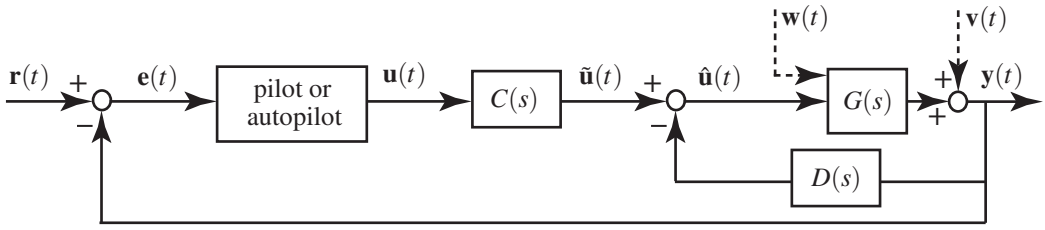


Figure 18.29: Block diagram of a **Stability and Control Augmentation System** (**SCAS**), in which the control input $\mathbf{u}(t)$ from the pilot or autopilot is augmented by small amounts of both **feedforward modification** $C(s)$ and **feedback stabilization** $D(s)$ to improve the handling qualities of an aircraft (cf. Figures 18.1 and 18.24).

18.3.5 Stability and Control Augmentation Systems (SCAS)

As introduced in Figure 18.1, we've concentrated thus far mainly on the design of controllers to be implemented in the forward part of a feedback loop in order to change substantially the input-output transfer function of a dynamic system, focusing on one or both of the following design objectives:

- tracking a reference input, $\mathbf{r}(t)$, with the output of the plant, $\mathbf{y}(t)$, for the sinusoidal components $\sin(\omega t)$ of the reference input $\mathbf{r}(t)$ up to a given bandwidth frequency ω_{BW}, and
- meeting specifications on rise time, settling time, and overshoot of the response $\mathbf{y}(t)$ to a step in $\mathbf{r}(t)$.

As emphasized by Guideline 18.1, one generally seeks to achieve these objectives with minimum excitation by control feedback to reduce the sensitivity of the closed-loop system to both external disturbances and internal modeling errors, especially unmodeled delays. We have also explored (in §18.3.3) the use of filters in the return portion of a feedback loop to reduce the sensitivity of the system to measurement noise.

There are certain situations, especially in the control of aircraft, in which the control objective is a bit more modest: in such situations, we don't want to change the input-output transfer function completely, but rather simply nudge the controls gently to dampen the unfavorable, oscillatory, or unstable modes of the vehicle to make it more easily or "naturally" controllable by its operator (e.g., a pilot or autopilot). Note that the various aircraft handling characteristics deemed "natural" are subjective, and have evolved over the years from the "feel" of revered classic (stable) transport and fighter aircraft, such as the **DC-3 Gooney Bird** and the **Supermarine Spitfire**. Preferred handling characteristics for military aircraft are described in detail in a number of military regulations, such as MIL-STD 1797A and MIL-SPEC 8785C; to achieve such handling qualities throughout the entire flight envelope, stability and control augmentation systems are almost always necessary in modern aircraft designs, most of which sacrifice the inherent aerodynamic stability of vintage aircraft (achieved via large horizontal and vertical stabilizers, and in some cases significant wing dihedral) for greatly improved agility, efficiency, or stealth, and are thus characterized by "poor" stability characteristics (from a handling perspective), at least in a portion of the flight envelope, before feedback is applied[19].

This section follows that on SLC, because the problem here is similar: based on an external objective or "reference input" on the system (e.g., maintain straight-and-level flight, initiate a climbing unaccelerated turn, etc.), the "outer-loop" controller (a pilot or an autopilot) gives appropriate control inputs $\mathbf{u}(t)$ to an inner loop, akin to the signal denoted $\theta_r(t)$ in the SLC paradigm as implemented in Figure 18.24. This control input from the pilot or autopilot is then augmented and applied to the system, as depicted in Figure 18.29; such a strategy is referred to as a **Stability and Control Augmentation System** (**SCAS**[20]). The use of an SCAS is, again, best illustrated by a few examples; *the examples below are based on the linearized dynamic models of aircraft developed in Example 17.2, a review of which is suggested before proceeding.*

[19] Indeed, the handling characteristics of early prototypes of the **F117 stealth fighter** were so poor, it earned the nickname **Wobblin' Goblin**; these characteristics were largely cured by appropriately-implemented SCAS systems in the production version of the aircraft.

[20] Some authors distinguish between a Stability Augmentation System (SAS) and a Control Augmentation System (CAS), though this distinction is, perhaps, a bit superfluous.

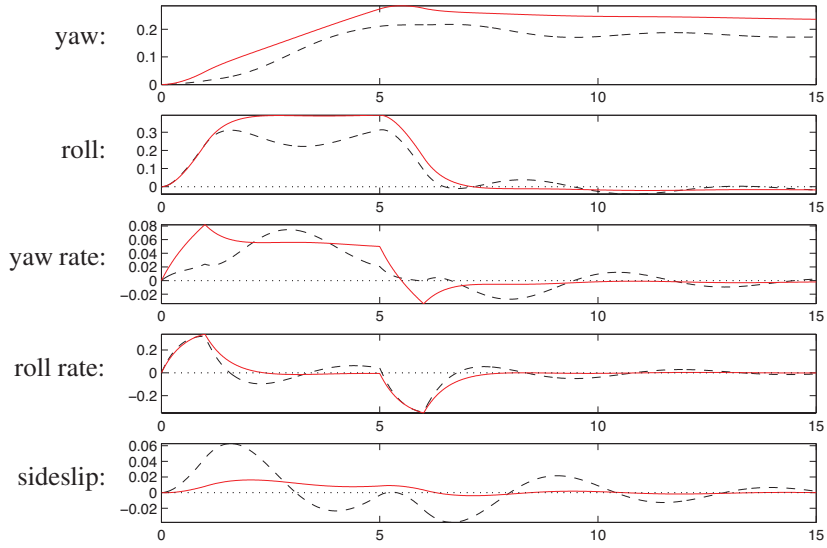


Figure 18.30: A turn with (solid) and without (dashed) opposite rudder applied to counteract **adverse yaw**.

Example 18.6 *Counteracting adverse yaw*

Normally, an aircraft turns by banking left or right, pitching up, then banking back to straight-and-level flight. When banked, a component of the lift force generated by the wings counters gravity, and another component of the lift force acts to turn the aircraft. The primary use of the aircraft rudder in non-emergency situations is simply to counteract the undesired sideslip (a.k.a. **adverse yaw**) generated when banking: when entering and exiting a bank, one wing produces more lift than the other, thus causing the aircraft to roll—the wing that generates more lift unfortunately also generates more drag, thus causing the aircraft to yaw towards the side generating the extra lift. In some aircraft, this effect is so strong that the pilot actually has to step on the rudder pedal every time a bank is initiated in order to maintain coördinated flight.

To illustrate, the linearized lateral/directional dynamics of a large transport aircraft on approach (at sea level, 137 knots, a flap angle of 1 degree, and a nominal cg location) may be modeled [see (17.43)] as

$$\begin{matrix}\text{yaw:}\\ \text{roll:}\\ \text{yaw rate:}\\ \text{roll rate:}\\ \text{sideslip:}\end{matrix} \quad \frac{d}{dt}\begin{pmatrix}\psi\\ \phi\\ p\\ r\\ \beta\end{pmatrix} = \begin{pmatrix}0 & 0 & 1 & 0 & 0\\ 0 & 0 & .199 & 1 & 0\\ 0 & -.002 & -.194 & -.167 & .748\\ 0 & -.003 & .636 & -2.020 & -5.374\\ 0 & .136 & -.970 & .198 & -.148\end{pmatrix}\begin{pmatrix}\psi\\ \phi\\ p\\ r\\ \beta\end{pmatrix} + \begin{pmatrix}0 & 0\\ 0 & 0\\ .053 & -.74\\ .865 & .904\\ .002 & .047\end{pmatrix}\begin{pmatrix}\delta_a\\ \delta_r\end{pmatrix}.$$

We now perform a numerical simulation of this system with $\delta_r(t) = 0$ and $\delta_a(t)$ executing a **doublet**:

$$\delta_a(t) = \begin{cases} 1 & t_0 < t < t_1, \\ -1 & t_2 < t < t_3, \\ 0 & \text{otherwise}, \end{cases}$$

thus entering a roll from $t_0 = 0$ s to $t_1 = 1$ s, and leaving the roll from $t_2 = 5$ s to $t_3 = 6$ s. The resulting system behaves as shown (dashed) in Figure 18.30; note the significant sideslip that accompanies the roll.

Adverse yaw is a predictable dynamic effect. One can thus implement a compensator $C(s)$ [see Figure 18.29] to apply an appropriate **feedforward** rudder correction every time the pilot commands an aileron deflection, thus improving the qualitative **handling quality** of the aircraft. To illustrate, taking $\tilde{\delta}_r(t) = \delta_r(t) -$

$0.1\delta_a(t)$ to add a small rudder correction with each application of the ailerons mitigates the adverse yaw in this system, as shown (solid) in Figure 18.30, reducing the peak sideslip by a factor of five and, importantly, reducing the oscillation in all of the state variables, thereby improving handling quality (see also Example 18.1 and Figure 18.16d). This relieves the (busy) pilot from having to perform such feedforward corrections.

Example 18.7 *Yaw damping of a stable "dutch roll" lateral/directional mode*

Physically, the **dutch roll mode** corresponds to an oscillatory perturbation involving first a bit of roll, which creates adverse yaw towards the upward moving wing, which in turn causes a loss of lift on the upward perturbed wing, which then causes roll in the other direction, etc.; looking aft out the top of the cockpit using a **periscopic sextant** (which you could do in certain vintage transport aircraft, such as the **C124 Old Shaky**, in order to perform celestial navigation), the tail of the aircraft repeatedly draws an infinity sign on the horizon when in a dutch roll limit cycle. The **roll subsidence mode** is an exponentially stable (and usually relatively fast) mode that quantifies how much the aircraft continues to roll once a slight roll is initiated then the ailerons neutralized. The **spiral mode** is an exponentially stable (and usually relatively slow) mode coupling the yaw rate and the roll (but not necessarily involving sideslip, it is often a nearly coördinated motion).

Using Algorithm 20.3, the state-space form of a large transport aircraft on approach [see (17.44)] may be converted to transfer function form, from rudder deflection $\delta_r(t)$ [taking the second column of B in (17.44)] to yaw rate $p(t)$ [taking $C = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \end{pmatrix}$ and $D = 0$], as

$$\frac{p(s)}{\delta_r(s)} = \frac{1.20(s-.0528)(s-2.18)(s+1.94)}{(s+.0679)(s+.696)(s+.403+2.01\mathrm{i})(s+.403-2.01\mathrm{i})}$$

where $p(s)$ is the Laplace transform of $p(t)$ [the yaw rate of the vehicle, in deg/s] and $\delta_r(s)$ is the Laplace transform of $\delta_r(t)$ [the rudder deflection, in deg]. At these flight conditions, two of the zeros of this open-loop system are in the RHP, and thus its dynamics are *nonminimum phase* (see §18.3.4.1). Recalling Figure 17.11, it is seen that the oscillatory poles, corresponding to the so-called **dutch roll mode**, have a natural frequency of $\omega_n = \sqrt{.403^2 + 2.01^2} = 2.05$ rad/sec, a period of $2\pi/\omega_n = 3.1$ sec, and a damping ratio of $\zeta = .403/\omega_n = .20$. The fast exponentially stable mode, corresponding to the so-called **roll subsidence mode**, has a time constant of $2\pi/.696 = 9.0$ sec. The slow exponentially stable mode, corresponding to the so-called **spiral mode**, has a time constant of $2\pi/.0679 = 93$ sec.

The dutch roll mode is destabilized at cruise speed and altitude in swept-wing commercial aircraft, and without a functioning SCAS is extremely difficult for the pilot to stabilize manually at this flight condition. As a result, many aircraft, such as the Boeing 727, actually have redundant SCAS systems implemented.

Example 18.8 *Damping of the "short-period" longitudinal mode*

The linearized longitudinal dynamics of an F-16 in straight-and-level flight at 300 knots at sea level, when the center of mass is[21] $0.3\bar{c}$ ahead of the center of pressure (a stable configuration), is given in (17.45), and may be written in transfer function form, from elevator deflection to pitch rate, as

$$\frac{q(s)}{\delta_e(s)} = \frac{-10.5s(s+.987)(s+.0218)}{(s+.00765+.0781\mathrm{i})(s+.00765-.0781\mathrm{i})(s+1.20+1.49\mathrm{i})(s+1.20-1.49\mathrm{i})},$$

where $q(s)$ is the Laplace transform of $q(t)$ [the pitch rate of the vehicle, in deg/s] and $\delta_e(s)$ is the Laplace transform of $\delta_e(t)$ [the elevator deflection, in deg]. Note that, at these flight conditions, all the poles and zeros of this open-loop system are in the LHP. It is seen that the slow oscillatory poles, corresponding to the oscillatory **phugoid mode**, have a natural frequency of $\omega_n = \sqrt{.00765^2 + .0781^2} = .0785$ rad/sec, a period of $2\pi/\omega_n = 80$ sec, and a damping ratio of $\zeta = .00765/\omega_n = .097$. The fast oscillatory poles, corresponding

[21] The **mean aerodynamic chord** $\bar{c}$ is the average distance from the leading edge to the trailing edge of the wing.

to the so-called **short period mode**, have a natural frequency of $\omega_n = \sqrt{1.20^2 + 1.49^2} = 1.91$ rad/sec, a period of $2\pi/\omega_n = 3.3$ sec, and a damping ratio of $\zeta = 1.20/\omega_n = .63$. That is, at this flight condition, the phugoid mode is relatively slow and lightly damped, whereas the short period mode is relatively fast and well damped[22]; neither necessitates modification in order to be easily controllable by the pilot at this flight condition. However, as flight conditions change (specifically, as the speed of the aircraft decreases, and/or the center of mass is shifted further aft), the short-period mode is destabilized. In such situations (for example, when a modern fighter aircraft performs a carrier landing), lead compensation implemented in the $D(s)$ block in Figure 18.29 can be used to stabilize the short period mode, thus resulting in an outer-loop system that is much more easily controlled by the pilot.

The above examples illustrate how classical control may be applied to dampen dutch roll and the short-period longitudinal mode, two common undesired modes of high-speed aircraft. Since modern aircraft have many control surfaces and many states of interest, state-space (MIMO) control design tools, developed in §21, are often preferred in such applications over the SISO control approaches developed here.

18.3.6 Unstable controllers for pathological SISO systems; pole placement

Up to now, we have developed just a few fundamental types of classical control components for SISO systems:

- **lead** and **lag**: $D(s) = K(s+z)/(s+p)$ with $p > z > 0$ (lead) or $z > p > 0$ (lag),
- **low-pass**: e.g., $D(s) = \omega_c^2/(s^2 + 1.414s\omega_c + \omega_c^2)$ for some "cutoff" frequency ω_c.
- **notch**: $D(s) = K(s^2 + z^2)/(s+p)^2$ for $z > 0$ and $p > 0$, and

Note that all of these components have their poles and zeros in the LHP. Such components may be replicated and cascaded as appropriate, with the overall gain magnitude and sign selected as necessary to construct a controller with the desired closed-loop characteristics. In particular,

- a **lead/lag** controller is a cascade of lead and lag components,
- a **PD** controller is a special case of a lead with no rolloff of the derivative action at high frequencies,
- a **PI** controller is a special case of a lag with no rolloff of the integral action at low frequencies, and
- a **PID** controller is a cascade of PD and PI controllers.

The SLC control design paradigm extends classical control tools developed in the basic SISO setting to the SIMO case by wrapping slower SISO outer loops around faster SISO inner loops. The SCAS setting illustrates how classical control concepts may be used to improve the stability of a system while still leaving the steering of the system up to the user. The venerable lead, lag, notch, and low-pass filtering techniques summarized above may be used together to stabilize most simple systems of practical interest (that is, to shift their closed-loop poles into the LHP). Such stabilizing controllers may then be tuned using the Bode plot via the process of **loop shaping** described previously. In certain pathological problems, however, a controller with RHP poles and/or zeros is required[23]. This situation is best illustrated by example.

Example 18.9 Pathological pendula

Consider again the SISO linear system analyzed in Exercise 17.10 and its associated transfer function which, for appropriate values of the physical parameters (see Exercise 17.10d), may be written as

$$G(s) = \frac{(s+2)(s-2)}{(s+1)(s-1)(s+3)(s-3)} = \frac{s^2-4}{s^4-10s^2+9}. \tag{18.27}$$

[22] Physically, the **phugoid mode** corresponds to a slow oscillatory exchange between potential and kinetic energy (climbing and slowing followed by diving and accelerating). The **short period mode** corresponds to a fast oscillation of the pitch θ and the angle of attack α (at almost constant airspeed and altitude).

[23] A controller with RHP poles is sometimes referred to as an **unstable controller**. This name, however, is something of a misnomer; a controller is designed to be used together with a plant *in closed loop*, so whether or not the controller itself has poles in the RHP is actually a matter of little practical consequence. It is, rather, the location of the *closed-loop poles* that ultimately matter.

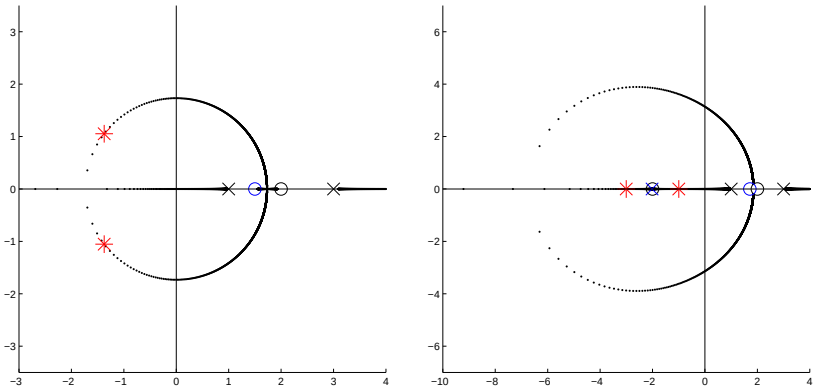


Figure 18.31: Root locus of the plant (18.27) with (a) the "lucky guess" controller design (18.28) applied, and (b) the minimal-order MESC design (18.29) applied, denoting: ($\times$, $\circ$) plant poles and zeros; ($\times$, $\circ$) controller poles and zeros; ($*$) closed-loop poles. For clarity, the pole/zero cancellations in the LHP are not marked.

Following the rules for plotting 180° and 0° root loci in §18.2.1, it is clear that, if all of the poles and zeros of $D(s)$ are in the LHP, then, *regardless of the precise form of* $D(s)$, there will be a closed-loop pole on the positive real axis, either somewhere in the range $1 < s < 2$, or somewhere in the range $2 < s < 3$. Further, as illustrated in §18.2.1.2, we must never attempt to "cancel" unstable plant poles or zeros with controller zeros or poles; in practice, such attempted cancellations always slightly miss their target, leaving closed-loop poles, with small coefficients, near these approximate cancellations. It is thus somewhat difficult to identify a stabilizing controller $D(s)$ for the plant given in (18.27) using the techniques presented thus far.

A "lucky guess". To identify a stabilizing controller $D(s)$ for the plant $G(s)$ given in (18.27), we first note that designing a controller that cancels *all* of the plant poles and zeros in the LHP is not a problem (it is pole/zero cancellations in the RHP that are problematic). If the controller additionally has a zero z_3 somewhere in the range $1 < z_3 < 3$, and a negative gain is used, then, following the rules for plotting the 0° root locus in §18.2.1, the root locus depicting the possible closed-loop pole locations will start at the (uncancelled) open-loop poles at $s = 1$ and $s = 3$ and move out towards infinity on the positive and negative real axes. Depending on the precise value of z_3, these two branches of the locus will meet on the real axis somewhere in the LHP (as depicted in Figure 18.31a), at the point at infinity, or on the real axis somewhere in the RHP. Wherever these two branches of the locus meet, the locus breaks off of the real axis and loops through the complex plane back to the pair of zeros on the positive real axis. It turns out that taking $1 < z_3 < 2$ leads to the two branches of the 0° root locus meeting in the LHP, and thus the possibility of achieving a stabilizing controller if the appropriate gain is used. Taking

$$D(s) = -1.08\frac{(s+1)(s+3)(s-1.5)}{s+2} \tag{18.28}$$

thus leads to closed-loop stability, as illustrated by the corresponding root locus in Figure 18.31a; note that, in addition to two LHP pole/zero cancellations, the remaining two closed-loop poles are also in the LHP.

The approach of requiring an "lucky guess" to achieve stability in feedback control design borders on the RCD approach mentioned previously (see Footnote 2 on Page 533), and is unsatisfactory. In harder pathological problems (see, e.g., Exercise 18.11), sufficient inspiration to achieve stability might not be readily available. Further, the $D(s)$ suggested above is improper, with infinite high-frequency gain, and is thus not implementable, and simply cascading a low-pass filter of sufficient order in series with this controller forfeits stability of the closed-loop system, as it introduces zeros at infinity. A more systematic approach of designing a stabilizing controller is thus required.

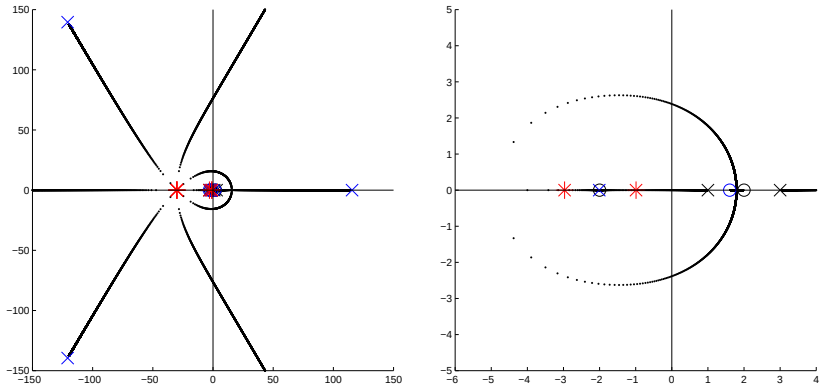


Figure 18.32: Root locus of the modified (strictly proper) MESC design (18.30) applied to (18.27): (left) complete locus, (right) close up around the origin. ($\times$, $\circ$) plant poles and zeros; ($\times$, $\circ$) controller poles and zeros; ($*$) closed-loop poles. For clarity, the pole/zero cancellations at $s=-3$ and $s=-1$ are not marked.

Systematic computation of the Minimum Energy Stabilizing Controller (MESC) via Pole Placement. In §21, we establish the following fact (proof is deferred to §21):

Fact 18.6 *A minimum-energy stabilizing feedback controller $D(s)$ places the poles of the closed-loop system $T(s)=G(s)D(s)/[1+G(s)D(s)]$ at the union of the stable open-loop poles of $G(s)$, and the reflection of the unstable open-loop poles of $G(s)$ into the LHP.*

We might thus follow a so-called **pole-placement** strategy, guided by Fact 18.6, to develop an initial stabilizing controller which can be further tuned in the frequency domain via the loop shaping process described previously. For the example system given in (18.27), we have

$$G(s)=\frac{b(s)}{a(s)}=\frac{s^2-4}{s^4-10s^2+9},\quad D(s)=\frac{y(s)}{x(s)},\quad T(s)=\frac{g(s)}{f(s)}=\frac{b(s)y(s)}{a(s)x(s)+b(s)y(s)}.$$

The problem at hand is the selection of $x(s)$ and $y(s)$ such that the target $f(s)$ satisfying Fact 18.6 is obtained:

$$f(s)=(s+1)^2(s+3)^2=s^4+8s^3+22s^2+24s+9=(s^4-10s^2+9)x(s)+(s^2-4)y(s);$$

this is known as a **Diophantine equation**[24], and an efficient general code for solving it is given in Algorithm B.1. The minimum order of $x(s)$ and $y(s)$ that lead to a solvable system of equations for the coefficients is

$$D(s)=\frac{y(s)}{x(s)}=\frac{b_3s^3+b_2s^2+b_1s+b_0}{a_1s+a_0}.$$

[After determining $D(s)$, we may rescale the numerator and denominator appropriately such that $x(s)$ is monic.] Written out completely, we have

$$\begin{aligned}
&s^4+8s^3+22s^2+24s+9=(s^4-10s^2+9)(a_1s+a_0)+(s^2-4)(b_3s^3+b_2s^2+b_1s+b_0)\\
&=(a_1+b_3)s^5+(a_0+b_2)s^4+(b_1-10a_1-4b_3)s^3+(b_0-10a_0-4b_2)s^2+(9a_1-4b_1)s+(9a_0-4b_0)\\
&\Rightarrow\begin{pmatrix}0&0&0&1&0&1\\0&0&1&0&1&0\\0&1&0&-4&0&-10\\1&0&-4&0&-10&0\\0&-4&0&0&0&9\\-4&0&0&0&9&0\end{pmatrix}\begin{pmatrix}b_0\\b_1\\b_2\\b_3\\a_0\\a_1\end{pmatrix}=\begin{pmatrix}0\\1\\8\\22\\24\\9\end{pmatrix}\quad\Rightarrow\quad\begin{pmatrix}b_0\\b_1\\b_2\\b_3\\a_0\\a_1\end{pmatrix}=\frac{1}{15}\begin{pmatrix}-288\\-216\\128\\56\\-113\\-56\end{pmatrix}.
\end{aligned}$$

[24] Placing the poles of a closed-loop system in certain desirable stable locations is a special case of the approach discussed in §18.1.2.

The minimal-order MESC following from Fact 18.6 in the present case is thus

$$D(s) = \frac{56s^3 + 128s^2 - 216s - 288}{-56s - 113} = -\frac{(s+1)(s+3)(s-1.714)}{s+2.018}, \tag{18.29}$$

which is remarkably close to the controller considered in (18.28). The corresponding root locus is shown in Figure 18.31b; in addition to the (stable) pole/zero cancellations, the remaining closed-loop poles are placed at precisely $s = -1$ and $s = -3$, in accordance with Fact 18.6.

Modifying the MESC design to achieve a strictly-proper form. A systematic procedure may be followed to determine a strictly-proper controller with performance that is in some sense close to that achieved by the controller given in (18.29). To accomplish this with a controller of minimal possible order, consider the strictly proper form

$$D(s) = \frac{y(s)}{x(s)} = \frac{b_3s^3 + b_2s^2 + b_1s + b_0}{a_4s^4 + a_3s^3 + a_2s^2 + a_1s + a_0}.$$

The denominator of $T(s)$ will now be eighth-order in s instead of fourth-order. We may thus simply add four fast stable poles (that is, a simple fourth-order low-pass filter) to our target for the closed-loop system,

$$\begin{aligned} f(s) &= (s+1)^2(s+3)^2(s+30)^4 \\ &= s^8 + 128s^7 + 6382s^6 + 153864s^5 + 1795689s^4 + 8986680s^3 + 20460600s^2 + 20412000s + 7290000. \end{aligned}$$

Following the same approach as before (or, more simply, using Algorithm B.1) leads to

$$\begin{aligned} f(s) &= (s^4 - 10s^2 + 9)(a_4s^4 + a_3s^3 + a_2s^2 + a_1s + a_0) + (s^2 - 4)(b_3s^3 + b_2s^2 + b_1s + b_0) \\ &= a_4s^8 + a_3s^7 + (a_2 - 10a_4)s^6 + (a_1 - 10a_3 + b_3)s^5 + (a_0 - 10a_2 + 9a_4 + b_2)s^4 + \\ &\quad (9a_3 - 10a_1 + b_1 - 4b_3)s^3 + (9a_2 - 10a_0 + b_0 - 4b_2)s^2 + (9a_1 - 4b_1)s + (9a_0 - 4b_0) \end{aligned}$$

$$\Rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -10 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & -10 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & -10 & 0 & 9 \\ 0 & 1 & 0 & -4 & 0 & -10 & 0 & 9 & 0 \\ 1 & 0 & -4 & 0 & -10 & 0 & 9 & 0 & 0 \\ 0 & -4 & 0 & 0 & 0 & 9 & 0 & 0 & 0 \\ -4 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 128 \\ 6382 \\ 153864 \\ 1795689 \\ 8986680 \\ 20460600 \\ 20412000 \\ 7290000 \end{pmatrix}$$

Thus, a modified controller that is strictly-proper, but is otherwise similar in closed-loop behavior to the Minimum Energy Stabilizing Controller, is

$$\begin{aligned} D(s) &= \frac{61163576s^3 + 146549888s^2 - 208926936s - 294313248}{15s^4 + 1920s^3 + 95880s^2 - 58836416s - 118655888} \\ &= K\frac{(s+1)(s+3)(s-1.604)}{(s+2.0104)(s-115.6)(s^2 + 2\zeta\omega_c + \omega_c^2)}. \end{aligned} \tag{18.30}$$

where $K = 4.08e6$, $\omega_c = 185$ and $\zeta = 0.654$. The corresponding root locus is shown in Figure 18.32; in addition to the (stable) pole/zero cancellations, the remaining closed-loop poles are placed at $s = -1$ and $s = -3$, and four more closed-loop poles are placed at $s = -30$, as expected. Without such a systematic procedure, it would be difficult to make the necessary "lucky guesses" that would lead to such a controller.

18.3.7 Implementation of CT linear controllers in analog electronics†

In order to achieve the **loop shaping** described previously (see §18.2.2 and Figure 18.6), it is straightforward to implement a CT controller $D(s)$ that cascades together a number of the individual filters discussed above and developed as simple analog circuits with low output impedance[25] in §19.2. In most cases[26], this involves

- lag filter(s) [see Figure 18.17a] to improve tracking, implemented at frequencies well *below* ω_c so as to not substantially erode the PM,
- lead filter(s) [see Figure 18.17b] to increase the PM and thus reduce overshoot, centered at the crossover frequency ω_c (that is, taking $\sqrt{p/z} = \omega_c$) so as to maximize their beneficial effect,
- low-pass filter(s) [see Figures 17.17a, 17.17b, 17.19a, and 17.19b] to improve robustness, implemented at frequencies well *above* ω_c so as to not substantially erode the PM, and/or
- notch filter(s) [see Figure 18.17c] to "knock-out" characteristic system oscillations, implemented carefully [see Figure 18.21] near the frequencies of the oscillatory open-loop poles of the plant so as to avoid open-loop instability.

[Note that one of the more delicate matters to attend to in classical control design is effectively "squeezing" the lag and low-pass filtering of $D(s)$ (situated on the Bode plot below and above ω_c, respectively) in as close as possible to ω_c while still achieving the required PM, in order to maximize the ranges of frequencies over which these filters have their beneficial effects.] The following results from §19.2 are of particular importance:

- a general adder/subtractor circuit is developed in Example 19.10 and illustrated in Figure 19.11c,
- lead, lag, P, I, D, PI, PD, and first-order low-pass filters are all special cases of the single op-amp circuit developed in Example 19.12 and illustrated in Figure 19.12a,
- second-order and fourth-order low-pass filter circuits are developed in Exercise 19.4, and
- a notch filter circuit is developed in Example 19.13 and illustrated in Figures 19.12b and 19.13.

Due to the issue of aliasing (see Figure 5.4 and the discussion in §18.4), low-pass filters used to reject disturbances generally need to be implemented in CT using analog electronics. [In other words, after you sample a signal with high-frequency noise, it is impossible to distinguish the low-frequency signal from the high-frequency noise.] However, due to their very low cost and ease of programming (and reprogramming when the system changes), DT microcontrollers are, today, often best suited for implementing the rest of the controller (as DT difference equations) even when controlling CT plants, as discussed in §18.4. We thus first very briefly review the tools for developing and implementing DT controllers.

18.3.8 Extending the PID, lag, lead, low-pass, and notch techniques to DT systems

As discussed in §18.2.5, the root locus, Bode, and Nyquist techniques extend immediately to DT systems. These tools may thus be used to tune DT linear controllers $D(z)$ formed as a cascade of lag, lead, low-pass, and notch filters in an essentially identical manner.

18.3.9 Implementation of DT linear controllers in microcontrollers

Via inverse Z transform of $D(z)$ and writing the resulting difference equation, such as (17.64), in a convenient form, such as (17.66), it is easy to see how the resulting difference equation may be implemented in DT in a microcontroller. As discussed in §17.4.3.2, for its corresponding difference equation to be implementable, the DT controller $D(z)$ must be **causal** (that is, $n \geq m$, where n is the order of the denominator and m is the order of the numerator). **Strictly causal** $D(z)$, with $n > m$, are somewhat easier to implement in practice than those arising from **semi-causal** $D(z)$, with $n = m$, as they give a full timestep to compute the next value of u_k.

[25] The low output impedance of each op amp circuit mentioned here (cf. the passive circuits of Example 19.4) simplifies the circuit design process significantly, as it effectively decouples each individual stage of the cascade, allowing them to be designed separately.

[26] Notable pathological exceptions requiring something different are discussed in §18.3.6.

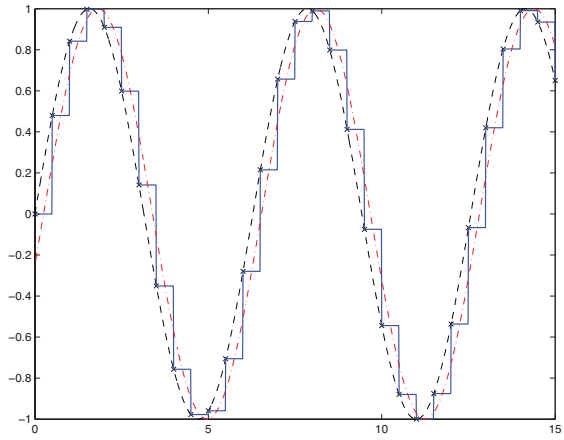


Figure 18.33: The principle effect of the cascade illustrated by the dashed box in Figure 18.34a, due primarily to the ZOH of the DAC, is the introduction of an $h/2$ delay in phase to the frequency response of $D(z)$, plus an (essentially negligible) high-frequency sawtooth wave. This is indicated by (solid) the response $u(t)$ of this cascade of components, for $D(z) = K = 1$, to (dashed) a sinusoidal input $e(t)$. The dot-dashed curve indicates the response $u(t)$ with the sawtooth removed, illustrating clearly this $h/2$ delay.

18.4 Classical control design of DT controllers for CT plants

When controlling a CT physical system $G(s)$ with a DT controller $D(z)$ implemented in the digital logic of a microcontroller, an interesting mix of CT and DT components arises. Such mixed systems may be analyzed using one of two methods, as discussed in the following two subsections. For most problems, the discrete equivalent design presented in §18.4.1 is entirely sufficient.

Note that the interconnection of CT plants with DT controllers requires both **digital-to-analog converters** (**DAC**) to convert the DT output from the digital electronics $D(z)$ [that is, from the microcontroller] into a CT input into the plant $G(s)$, and **analog-to-digital converters** (**ADC**) to convert the CT output from the plant $G(s)$ into a DT input into the controller $D(z)$.

To keep production costs down, nearly all commercial DACs incorporate a simple **zero-order hold** (**ZOH**): the output $u(t)$ of the DAC at any time t is taken as the input u_k of the DAC at the most recent timestep t_k. We thus assume that all DACs incorporate a ZOH strategy in the remainder of this text. The influence of the ZOH of the DAC is significant (see Figures 18.33 and 18.12a) and detrimental, as it can significantly reduce the PM (and, thus, increase overshoot) of a closed-loop system.

Nearly all ADCs incorporate an analog circuit implementing a CT low-pass filter to reduce components of the input signal above the **Nyquist frequency** π/h, to prevent **aliasing** when sampling (see Figure 5.4). If the sample period h is sufficiently small, the influence of this low-pass filter on the dynamics of the closed-loop system may be neglected. However, as no linear low-pass filter is ideal (see §17.5.2), and one is usually motivated economically to implement as large a sample period h as possible, it is generally best to account for the phase loss at crossover caused by this low-pass filter during the controller design.

In addition to discretizing CT signals in time, using a timestep h, ADCs convert real values (usually, voltages) to **finite-precision representations** in the microcontroller. If **fixed-point arithmetic** is being used (which is often required when implementing on an inexpensive PIC, AVR, or ARM microcontroller), the resulting **discretization errors** are well modeled as a bit of additive measurement noise[27].

[27] If **floating-point arithmetic** is being used, the discretization errors are better represented by a (more cumbersome) multiplicative noise model; however, all modern microcontrollers implementing floating-point arithmetic use single precision arithmetic or better, in which case discretization errors are generally negligible.

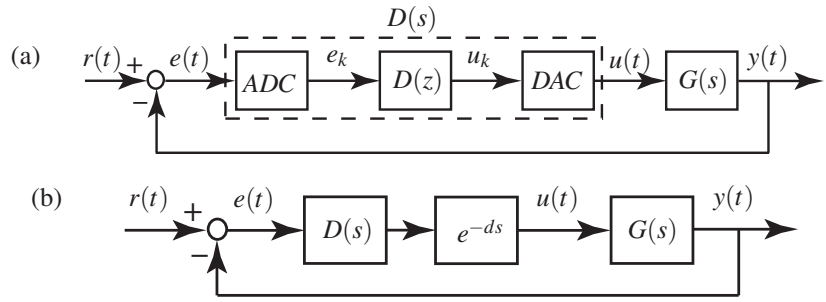


Figure 18.34: (a) A closed-loop configuration of a DT controller $D(z)$ and a CT plant $G(s)$ interpreted as a **discrete equivalent design**. When designing the CT transfer function $D(s)$ corresponding to the cascade of the ADC, the DT controller $D(z)$, and the DAC, the primary effect of the ADC & DAC is an $h/2$ delay due to the zero-order hold, as illustrated in Figure 18.33. (b) An equivalent circuit used for the design of $D(s)$, incorporating the $d = h/2$ delay using an appropriate Padé approximation [see (17.48)]. Once $D(s)$ is designed, it may be converted into a DT transfer function with similar dynamics, $D(z)$, using Tusin's approximation with prewarping, then inverse transformed into a difference equation that may easily be implemented in a microcontroller.

18.4.1 Discrete equivalent design

In the **discrete equivalent design** approach, we consider the inputs and outputs of the closed-loop system in continuous time, and represent the cascade of the ADC (with sample period h), the DT controller $D(z)$, and the DAC (with a ZOH) as a single CT system, which we denote $D(s)$, as highlighted by the dashed box in Figure 18.34a. To proceed following this approach, we first design an appropriate CT controller $D(s)$ directly for the CT system $G(s)$ using the techniques introduced in §18.2 and explored at length in §18.3, then approximate this CT controller $D(s)$ with a rational expression in DT, $D(z)$. This discrete approximation is best computed via Tustin's approximation with prewarping (see §17.4.4), determining $D(z)$ from $D(s)$ via the following simple substitution:

$$D(z) = D(s)\Big|_{s=\frac{2}{fh}\frac{z-1}{z+1}} \quad \text{where} \quad f = \frac{2[1-\cos(\overline{\omega}h)]}{\overline{\omega}h\sin(\overline{\omega}h)}, \tag{18.31}$$

and $\overline{\omega}$ denotes the frequency of primary interest in the controller for which an accurate mapping is desired—that is, the notch frequency if $D(s)$ has a notch (see §18.3.2), or the crossover frequency (see §18.2.2) of the closed-loop system if $D(s)$ does not have a notch. Once $D(z)$ is obtained via this approach, the corresponding difference equation is easily determined via the inverse Z transform techniques presented in §17.4. This difference equation may then be implemented in digital electronics.

The most significant detrimental effect encountered when implementing a CT controller, $D(s)$, in a DT fashion on a microcontroller, as illustrated by the dashed box in Figure 18.34a, is the *effective* $h/2$ *delay resulting from the ZOH of the DAC*, where h is the sample period, as illustrated in Figure 18.33. At frequencies well below the Nyquist frequency, the effect of this delay is negligible. At input frequencies within an order of magnitude of the Nyquist frequency, however, the effect of this delay is a significant *phase loss*:

$$\text{phase loss} = 2\pi\frac{\text{time delay}}{\text{wave period}} = 2\pi\frac{h/2}{2\pi/\omega} = h\omega/2 \ \text{ rad/s}, \tag{18.32}$$

as illustrated in Figure 18.12a. *A corresponding amount of extra phase lead at the crossover frequency* ω_c *should thus be built in to the CT control design* $D(s)$ *to compensate*. Alternatively, the effect of this $h/2$ delay in the ultimate DT implementation of the controller may be accounted for during the design of $D(s)$ by including a Padé approximation of the delay in series with $D(s)$, as illustrated in Figure 18.34b.

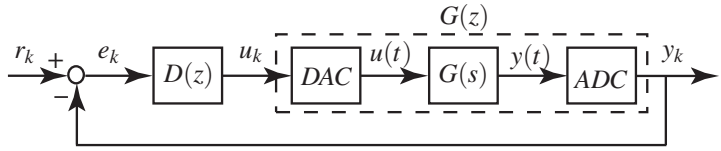


Figure 18.35: A closed-loop configuration of a DT controller $D(z)$ and a CT plant $G(s)$ interpreted as a **direct digital design**. The DT transfer function $G(z)$, given by the cascade of a DAC, a CT plant $G(s)$, and the ADC, is computed *exactly* in (17.68).

18.4.2 Direct digital design

In the **direct digital design** approach, we consider the inputs and outputs of the closed-loop system in DT, and represent the cascade of the DAC (with a ZOH), the CT plant $G(s)$, and the ADC (with sample period h) as a single DT system, which we denote $G(z)$, as highlighted by the dashed box in Figure 18.35, using the exact expression provided in (17.68). We then determine the DT controller $D(z)$ directly for this DT system $G(z)$, as introduced in §18.2.5 and discussed further in §18.3.8. If an initial CT controller $D(s)$ is known that is good for the CT plant $G(s)$, this controller may be approximated as an initial DT $D(z)$ with similar dynamics using Tustin's approximation with prewarping [see (17.73)], then tuned further directly in DT.

The primary benefit of the direct digital design approach is that it is *exact*. The primary drawback of this design approach is that it only considers the values of the output y_k *at* the timesteps, and thus doesn't detect whether or not there are actually significant oscillations in $y(t)$ *between* the timesteps, referred to as **intersample ripple**. *To avoid such intersample ripple, one should generally avoid placing the poles of the DT closed-loop system anywhere near the negative real axis during the controller design.*

Example 18.10 Evidence of the detrimental time delay from the ZOH when designing a DT controller for a CT plant

To illustrate that an $h/2$ time delay is in fact the leading-order (and detrimental) effect of the ZOH in the DAC when, following the **discrete equivalent design** paradigm, a CT controller is converted to DT and implemented in a microcontroller with sample period h, consider the following unstable CT plant and CT lead controller acting in closed loop:

$$G(s) = \frac{1}{(s+10)(s-10)}, \quad D(s) = K\frac{s+z}{s+p}.$$

Taking $z = 10$ (for a stable pole/zero cancellation) and $p = 20$ (for a little bit of phase lead) leads to a root locus for the CT problem as shown in Figure 18.36a; further, taking $K = 380$ leads to the (stable) closed-loop poles indicated by * in the root locus, gives crossover near the peak of the phase lead in the corresponding Bode plot, and gives about a 28% overshoot and a 0.1 second rise time in the corresponding step response. Looking at this root locus plot, it appears that turning up the control gain K to much larger values would lead to lightly damped poles, but would apparently *not* lead to closed-loop instability.

Now consider the implementation of this controller in DT taking $h = .02$ seconds (that is, taking a sample frequency of 50 Hz), using a ZOH in the DAC. We will approximate our well-behaved CT control design $D(s)$ as a DT difference equation with similar dynamics using Tustin's approximation with prewarping, as reviewed in (18.31). To analyze in discrete time how well this controller works (see Figure 18.35), we may convert the DAC – $G(s)$ – ADC cascade to the exact expression for the corresponding $G(z)$ via (17.68) and consider the problem in the **direct digital design** setting; the corresponding DT root locus plot is shown in Figure 18.36b. Note that the DT closed-loop system goes unstable if the gain K exceeds a critical value.

The reason why the DT root locus in Figure 18.36b goes unstable for large K, but the CT root locus in Figure 18.36a does not, is well explained by the effective $h/2$ delay caused by the ZOH in the DAC. Indeed,

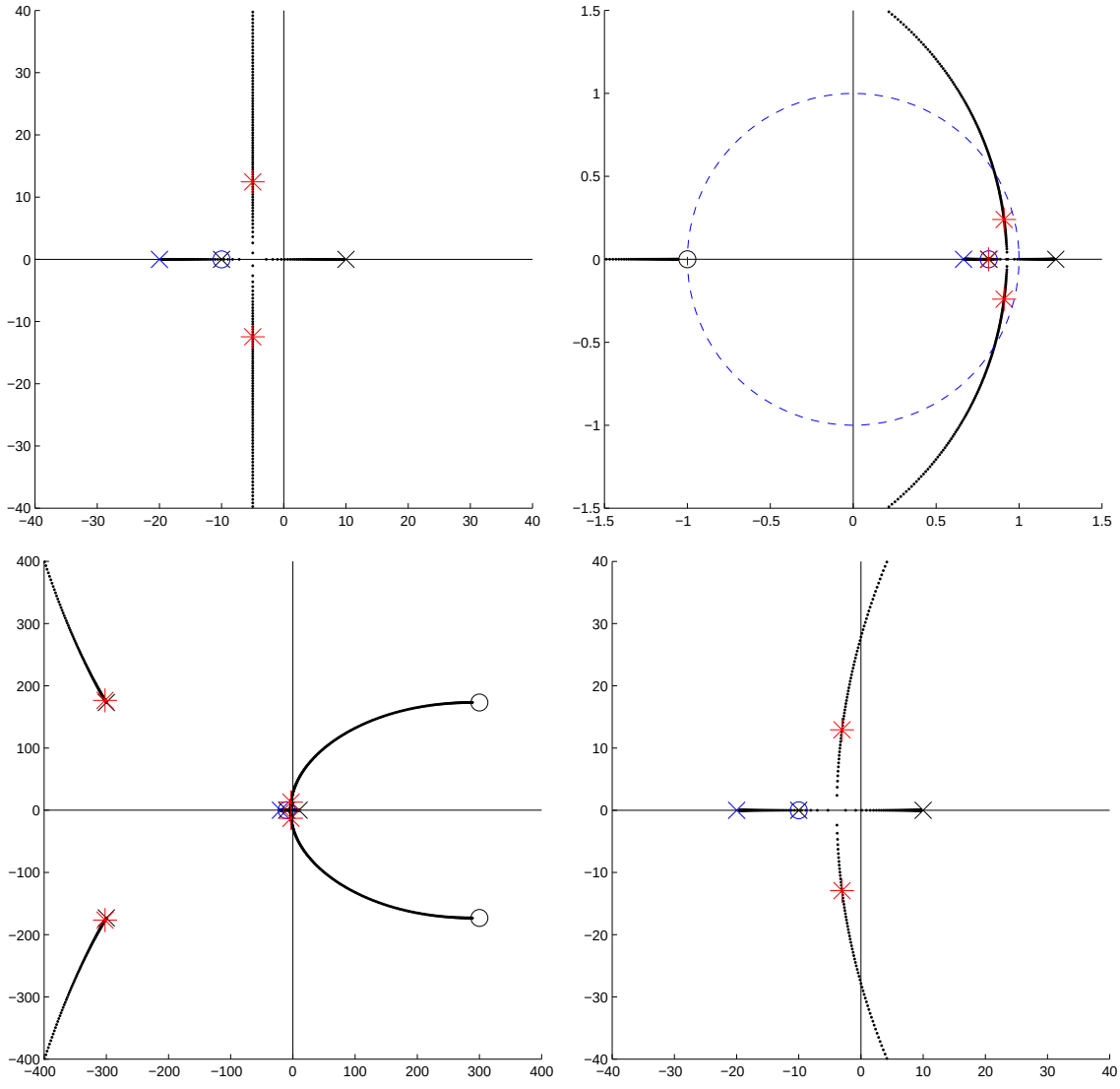


Figure 18.36: The root locus plots discussed in Example 18.10, illustrating the high-gain instability caused by the effective $h/2$ time delay caused by the ZOH of the DAC.

if we again analyze the system in CT, but now approximate the CT plant as $G(s) \cdot F_2(s)$, where $F_2(s)$ is an $n = 2$ rational approximation [see (17.48)] of the Laplace transform of a delay e^{-ds}, with $d = h/2$, the corresponding root locus is shown in Figure 18.36c, a closeup of which (near the origin) is given in Figure 18.36d. It is seen that the delay is approximated by n poles in the LHP and n zeros in the RHP, and that the speed of these LHP poles and RHP zeros is inversely proportional to the delay d [that is, for small d, these LHP poles and RHP zeros are **fast** (i.e., far from the origin)]. The CT root locus ultimately connects to the fast RHP zeros, so the CT closed-loop system now goes unstable if the gain K exceeds a certain critical value. As quantified in Exercise 18.13, the value of K for which this CT model of the problem goes unstable coincides accurately with the value of K for which the corresponding DT case goes unstable, as discussed above.

18.4.3 Deadbeat control: pole placement at the origin for DT settling in finite time

Deadbeat control is a special DT control technique that has no direct analog in CT. Recall from Figure 17.13 that the CT design guide for exponential settling of a step response with characteristic timescale t_s is given by placing all closed-loop poles in the s-plane such that their real parts are to the left of $s = -\sigma = -4.6/t_s$. Recall from Figure 17.16 that the corresponding DT design guide is given by placing all closed-loop poles in the z-plane inside a circle centered at the origin with radius $r = e^{-\sigma h} = e^{-4.6h/t_s}$. For rapid settling, then, we want closed-loop poles with large negative real parts in CT, and with radius much less than one in DT.

It is thus reasonable to consider a DT control design strategy that uses pole placement to design a causal $D(z)$ that *puts all of the closed-loop poles at the origin*. This results in a DT closed-loop system for which the *output settles completely after a finite number of timesteps*[28], and the closed-loop transfer function is

$$T(z) = \frac{g(z)}{z^\ell}. \tag{18.33a}$$

There is flexibility both in the choice of ℓ [that is, how many steps are taken until the output of the DT system settles completely; ℓ must be large enough that the resulting controller $D(z)$ is causal] and in the choice of $g(z)$ [that is, the (finite-time) dynamics expressed by the output of the closed-loop system as it settles].

Given a causal $G(z) = b(z)/a(z)$, the controller $D(z) = y(z)/x(z)$ required to achieve (18.33a) is

$$T(z) = \frac{G(z)D(z)}{1+G(z)D(z)} \quad \Leftrightarrow \quad D(z) = \frac{1}{G(z)} \cdot \frac{T(z)}{1-T(z)} = \frac{a(z)}{b(z)} \cdot \frac{g(z)}{z^\ell - g(z)} = \frac{y(z)}{x(z)}. \tag{18.33b}$$

Denote $deg\{p(z)\}$ the order of $p(z)$. For $D(z)$ to be causal, $deg\{x(z)\} \geq deg\{y(z)\}$, and thus

$$\ell \geq deg\{g(z)\} + deg\{a(z)\} - deg\{b(z)\}. \tag{18.34a}$$

For $D(z)$ to be strictly causal [which is easier to implement in a microcontroller running at a modest clock speed, as seen in (17.66)], $deg\{x(z)\} > deg\{y(z)\}$, and thus

$$\ell > deg\{g(z)\} + deg\{a(z)\} - deg\{b(z)\}. \tag{18.34b}$$

Further, if $T(z) = Y(z)/R(z)$ has zero steady-state error to a unit step [that is, if for $r_k = 1$ for $k = 0, 1, 2, \ldots$ and thus $R(z) = z/(z-1)$ it follows that $\lim_{k\to\infty} y_k = 1$], then by the DT final value theorem (Fact 17.4)

$$\lim_{k\to\infty} y_k = \lim_{z\to 1}(z-1)Y(z) = \lim_{z\to 1}(z-1)T(z)\frac{z}{z-1} = T(1) = 1 \quad \Rightarrow \quad g(1) = 1. \tag{18.34c}$$

It is thus clear from (18.33b) that either $D(z)$ or $G(z)$ has a pole at $z = 1$.

Minimal-time deadbeat controllers for stable minimum-phase systems

If $G(z) = b(z)/a(z)$ is strictly causal and both stable and minimum phase [that is, if all of the poles and zeros of $G(z)$ are inside the unit circle], we may simply take $g(z) = 1$ and $\ell = deg\{a(z)\} - deg\{b(z)\}$ in (18.33); this choice is referred to as the **minimal-time deadbeat controller**, as it results in the fastest-possible complete settling of the output of the DT system:

$$T(z) = \frac{1}{z^\ell} \quad \Leftrightarrow \quad D(z) = \frac{y(z)}{x(z)} = \frac{a(z)}{b(z)} \cdot \frac{1}{z^\ell - 1}. \tag{18.35a}$$

Note that $D(z)$ in this case cancels both the poles and zeros of $G(z)$. As pointed out (in the CT case) in §18.2.1.2, since the plant $G(z)$ is generally only known approximately, it must be emphasized that this strategy

[28] In other words, in the language of §17.4.3, the entire DT closed-loop system becomes an FIR filter instead of an IIR filter.

works only if $G(z)$ has all of its poles and zeros inside the unit circle. As discussed in §18.1, the transfer function from the reference r_k to the control u_k, referred to as the control sensitivity, is given in this case by

$$S_u(z) = \frac{U(z)}{R(z)} = \frac{D(z)}{1+G(z)D(z)} = \frac{y(z)a(z)}{a(z)x(z)+b(z)y(z)} = \frac{a(z)a(z)}{a(z)b(z)(z^\ell-1)+b(z)a(z)} = \frac{a(z)}{b(z)z^\ell}. \quad (18.35b)$$

Due to the pole/zero cancellations in $[G(z)D(z)]$, *the poles of $S_u(z)$ are different than the poles of $T(z)$*. [This is not normally the case; when pole/zero cancellations in $[G(z)D(z)]$ do not occur, the denominator of both $T(z)$ and $S_u(z)$ is simply $f(z) = a(z)x(z)+b(z)y(z)$.] As a result, *even though the output y_k settles completely after a finite number of timesteps, the control u_k does not*. This presents a significant problem: when the DT system considered arises as a result of the application of DT microcontroller to a CT system, the **control oscillations** in u_k caused by this approach often lead to a significant **intersample ripple** in the CT output $y(t)$, even well after the DT *samples* of the output, y_k, settle completely (see Example 18.11). This largely defeats the entire purpose of implementing the deadbeat control design methodology in the first place.

Ripple-free deadbeat controllers for stable systems

Designing a $D(z)$ that cancels the entire dynamics of the plant, as suggested in (18.35), is a heavy-handed approach. A much better approach, called a **ripple-free deadbeat controller**, strives to make both y_k *and* u_k settle after a finite number of timesteps. To see how to do this, assuming that $G(z) = b(z)/a(z)$ is stable (but not necessarily minimum-phase), we may take $g(z) = b(z)/b(1)$ and $\ell = deg\{a(z)\}$ in (18.33), and thus [cf. (18.35)]

$$T(z) = \frac{b(z)/b(1)}{z^\ell} \quad \Leftrightarrow \quad D(z) = \frac{y(z)}{x(z)} = \frac{a(z)/b(1)}{z^\ell - b(z)/b(1)}, \quad (18.36a)$$

from which it follows that

$$S_u(z) = \frac{U(z)}{R(z)} = \frac{D(z)}{1+G(z)D(z)} = \frac{a(z)/b(1)}{[z^\ell - b(z)/b(1)]+b(z)/b(1)} = \frac{a(z)/b(1)}{z^\ell}. \quad (18.36b)$$

In contrast with the minimal-time deadbeat controller (18.35), the ripple-free deadbeat controller (18.36) allows the zeros of $G(z)$ to appear in the numerator of $T(z)$, and by so doing they do *not* appear in the denominator of $S_u(z)$. Thus, *the output y_k and the control u_k both settle completely ℓ timesteps after a step input in r_k*, and the intersample ripple problem mentioned previously is eliminated (see Example 18.12).

As the timestep h is made small, deadbeat controllers demand large control inputs; one must thus exercise a certain amount of caution when following this approach. Guideline 18.1 still applies; to follow it, simply don't let the timestep h get too small.

Ripple-free deadbeat controllers for general (possibly unstable and nonminimum-phase) systems

All internally-stable (see Fact 18.1), causal, ripple-free deadbeat controllers for general (possibly unstable and nonminimum-phase) proper DT systems are given simply by the DT version of the remarkable construction in (18.9), taking $f(z) = z^\ell$:

$$D(z) = \frac{y(z)+a(z)\overline{Q}(z)}{x(z)-b(z)\overline{Q}(z)} \quad (18.37a)$$

where the polynomials $\{x(z), y(z)\}$ solve the associated **Diophantine equation** (see §B.2)

$$a(z)x(z)+b(z)y(z) = z^\ell \quad (18.37b)$$

with[29] $\overline{Q}(z) = q(z)/z^k$ where $k \geq 0$ and $deg\{q(z)\} \leq k$ [$q(z)$ is otherwise arbitrary], and, assuming no pole/

[29] To see clearly why this form for $\overline{Q}(z)$ works, see (18.10).

zero cancellations[30] in $[G(z)D(z)]$, $\ell = 2n-1$. Amongst these, the case with $c = 0$ is particularly simple (see Example 18.13), and results in

$$T(z) = \frac{G(z)\,D(z)}{1+G(z)\,D(z)} = \frac{b(z)\,y(z)}{z^{\ell}} \quad \text{and} \quad S_u(z) = \frac{D(z)}{1+G(z)\,D(z)} = \frac{a(z)\,y(z)}{z^{\ell}}.$$

Example 18.11 Minimal-time deadbeat control of a stable plant

Consider the direct digital design setting (Figure 18.35) applied to the stable minimum-phase CT system

$$G(s) = \frac{b(s)}{a(s)} = \frac{1}{(s+1)(s+5)}. \tag{18.38a}$$

Taking $h = 0.2$ and (exactly) converting the DAC – $G(s)$ – ADC cascade to DT [see (17.68)] gives the corresponding stable minimum-phase (with all poles and zeros inside the unit circle) DT transfer function

$$G(z) = \frac{b(z)}{a(z)} = \frac{0.0137\,z+0.0092}{z^2-1.1866\,z+0.3012}, \quad \text{with} \quad n = deg\{a(z)\} = 2 \quad \text{and} \quad m = deg\{b(z)\} = 1. \tag{18.38b}$$

Applying (18.35) for minimal-time deadbeat control of the stable minimum-phase system (18.38) results in $\ell = n-m = 1$ and

$$D(z) = \frac{a(z)}{b(z)} \cdot \frac{1}{z^{\ell}-1} = \frac{z^2-1.1866\,z+0.3012}{0.0137\,z+0.0092} \cdot \frac{1}{z-1}; \tag{18.39}$$

the DT and CT responses u_k, $u(t)$, y_k, and $y(t)$ resulting from a unit step input are illustrated in Figure 18.37a; note the significant intersample ripple.

Example 18.12 Ripple-free deadbeat control of a stable plant

Applying (18.36) for ripple-free deadbeat control of the system (18.38) results in $\ell = n = 2$ and

$$D(z) = \frac{a(z)}{z^{\ell}-b(z)} = \frac{z^2-1.1866\,z+0.3012}{z^2-[0.0137\,z+0.0092]}; \tag{18.40}$$

the DT and CT responses u_k, $u(t)$, y_k, and $y(t)$ resulting from a unit step input in this case are illustrated in Figure 18.37b; note that y_k takes one more timestep to settle than the minimal-time deadbeat controller considered in Example 18.11, but the significant intersample ripple is eliminated.

Example 18.13 Ripple-free deadbeat control of an unstable plant

Consider the direct digital design setting applied to the unstable CT system [cf. (18.38)]

$$G(s) = \frac{b(s)}{a(s)} = \frac{1}{(s-1)(s-5)}. \tag{18.41a}$$

Taking $h = 0.2$ and converting the DAC – $G(s)$ – ADC cascade to DT [see (17.68)] gives the corresponding unstable nonminimum-phase (with two poles and one zero outside the unit circle) DT transfer function

$$G(z) = \frac{b(z)}{a(z)} = \frac{0.0306\,z+0.0455}{z^2-3.9397\,z+3.3201}, \quad \text{with} \quad n = deg\{a(z)\} = 2 \quad \text{and} \quad m = deg\{b(z)\} = 1. \tag{18.41b}$$

Though the system considered in (18.41) has the same order as that considered in (18.38), neither (18.35) nor (18.36) may be applied in this case, as these approaches are based on pole-zero cancellations between the controller and the plant; since the DT plant, which is only known approximately, has both poles and zeros outside the unit circle, these approaches would certainly fail (see §18.2.1.2).

[30] Note that (18.37b) has a minimum of $\ell+1$ powers of z whose coefficients must be matched between the LHS and the RHS. Take $n = deg\{a(z)\}$ and $m = deg\{b(z)\}$; since $G(z) = b(z)/a(z)$ is causal, $n \geq m$. Take $i = deg\{x(z)\}$ and $j = deg\{y(z)\}$; since $D(z) = y(z)/x(z)$ is causal, $i \geq j$. It follows from (18.37b), assuming no pole/zero cancellations in $[G(z)D(z)]$, that $n+i = \ell$. There are at most $2(i+1)$ free coefficients in $D(z)$. Setting $2(i+1) \geq \ell+1$ for the resulting system to be solvable, it follows that $\ell \geq 2n-1$.

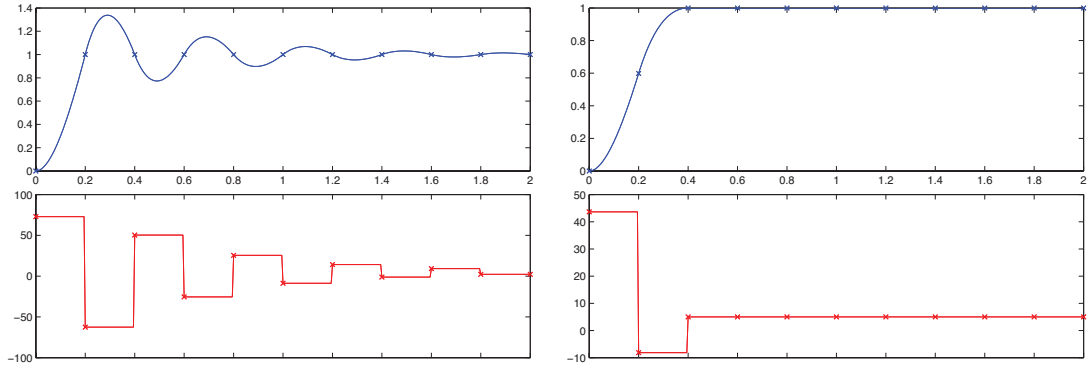


Figure 18.37: (a) Step response of the stable system (18.38) when the **minimal-time deadbeat controller** (18.35) is applied (see Example 18.11). Though the output of the DT system settles completely after $\ell = n - m = 1$ timestep, the output of the underlying CT system exhibits significant **intersample ripples** well after that, due to the control oscillations shown in the bottom subfigure. (b) Step response of the same system with the **ripple-free deadbeat controller** (18.36) applied (see Example 18.12). The output of both the DT and the underlying CT system settle completely after $\ell = n = 2$ timesteps, eliminating the intersample ripple seen previously.

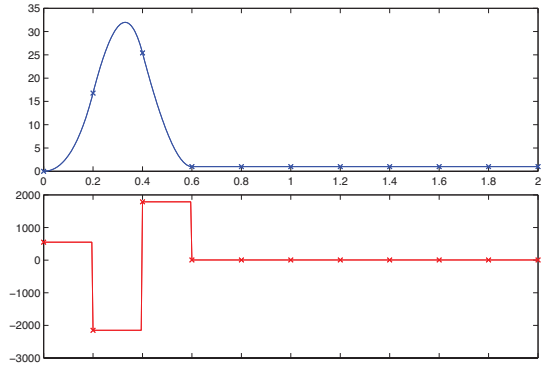


Figure 18.38: Step response of the unstable system (18.41) with the general ripple-free deadbeat controller (18.37) applied, taking $\overline{Q}(z) = 0$ (see Example 18.13). The output of both the DT and the underlying CT system settle completely after $\ell = 2n - 1 = 3$ timesteps.

Instead, applying (18.37) for ripple-free deadbeat control of the unstable nonminimum-phase system (18.41), solving $a(z)x(z) + b(z)y(z) = z^\ell$ with $\ell = 2n - 1 = 3$ using Algorithm B.1, results in

$$D(z) = \frac{y(z) + a(z)\overline{Q}(z)}{x(z) - b(z)\overline{Q}(z)} \quad \text{where} \quad \begin{aligned} y(z) &= 89.6468\,z - 87.5027, \\ x(z) &= z + 1.1998. \end{aligned} \tag{18.42}$$

Again, all causal ripple-free deadbeat controllers in this case are given by taking $\overline{Q}(z) = q(z)/z^k$ with $k \geq 0$ and $deg\{q(z)\} \leq k$ [$q(z)$ is otherwise arbitrary]. The DT and CT responses u_k, $u(t)$, y_k, and $y(t)$ resulting from a unit step input in this case, taking $\overline{Q}(z) = 0$, are illustrated in Figure 18.38; note that y_k takes one more timestep to settle than the ripple-free deadbeat controller considered in Example 18.12, but the deadbeat control strategy used in this case may be applied safely to general (unstable, nonminimum-phase) systems.

The controller derived above has closed-loop transfer function of $T(z) = g(z)/z^\ell$ where $g(z) = b(z)y(z)$. Note that $g(1) \neq 1$, and thus the closed-loop system has nonzero steady-state error [see (18.34c)]. The easiest way to fix this is with a prefactor of $P = 1/g(1)$ [see Figure 18.11], as used in Figure 18.38.

Exercises

Exercise 18.1 Consider a plant, which initially starts at rest, governed by the differential equation

$$y''(t) = u'(t) + 2u(t).$$

a. Take the Laplace transform of this differential equation. Determine the transfer function $G(s) = Y(s)/U(s)$.

b. Assuming proportional feedback $D(s) = K$, sketch the root locus of the closed-loop system. Confirm using Algorithm 18.1.

c. Assuming proportional feedback with $K = 1$, where are the closed-loop poles? Recalling the approximate design guides (17.57), what are the approximate rise time, settling time, and overshoot of the step response of the closed-loop system? Confirm using Algorithm 17.1.

d. Assuming proportional feedback with $K = 1$, compute the step response of this system analytically. Plot.

e. Sketch the (open-loop) Bode and Nyquist plots and, assuming proportional feedback with $K = 1$, determine the crossover frequency, gain margin, and phase margin. Confirm using Algorithms 17.4 and 18.3.

f. Assuming proportional feedback with $K = 1$, sketch the closed-loop Bode plot. Confirm using Algorithm 17.4. For what frequencies does the output accurately track the input?

g. Apply the CT final value theorem (Fact 17.4) to determine the steady-state error of this closed-loop system (assuming proportional feedback with $K = 1$) to a unit step input.

Exercise 18.2 Recall that Example 17.9 modeled the horizontal movements $y(t)$ of the third story of a three-story building to both ground motions $w(t)$ and forces applied to the top floor, $v(t)$. Exercise 17.6 examined the Bode plots of this system for both inputs, $w(t)$ and $v(t)$, quantifying its resonant responses to sinusoidal excitations and relating these resonances to the locations of the lightly-damped poles of the system transfer function. We now tune a large mass/spring/damper system that is to be attached to the top floor of this structure to suppress these resonant peaks. To begin, extend the single mass/spring/damper system model of Example 17.1 to account for motion of the surface, $y(t)$, to which the spring and damper will be attached (see, e.g., Figure 17.9b), denoting the deflection of the mass from its rest position as $x(t)$. Then combine equations appropriately to eliminate both the position of the mass, $x(t)$, and the horizontal force, $v(t)$, applied by the (moving) mass/spring/damper system onto the building, Takinginitially take the $m = 400$ kg an ...

Exercise 18.3 Following the suggestions in §18.2.5, write a code to generate a DT Nyquist plot.

Exercise 18.4 Figure 18.16 presented the results of a PID controller applied to the cruise control problem described in Example 18.1. Noting that $K_u = 1.04 \cdot 10^5$ and $\omega_u = 55.98$, test the Ziegler-Nichols P, Ziegler-Nichols PI, Ziegler-Nichols PID, Pessen PID, Tyreus-Luyben PI, and Tyreus-Luyben PID ad hoc tuning rules applied to this system, and comment on each.

Exercise 18.5 The system considered in Figures 18.15 and 18.16, as shown at the right in (18.21), is based on a $n = 2$ Padé approximation of the delay as given in (17.48). Repeat this analysis using $n = 4$ and $n = 8$ Padé approximations of the delay in $G(s)$, and replot all of the curves in Figures 18.15 and 18.16.

Exercise 18.6 Noting the results of Exercise 18.5, rewrite `Bode.m` and `ResponseTF.m` (Algorithms 17.4 and 17.1) in order to compute the precise step response of a system with a delay d. Applying these codes to (18.21), compute the exact Bode plots and step responses for the problems considered in Figures 18.15 and 18.16.

Exercise 18.7 Example 18.1 discussed how a simple first-order system with a delay [see (18.21)] can be used to model an unstable system of unknown structure which can be stabilized by proportional feedback. Derive the formulae to determine $\{C, d, a\}$ from $\{K_l, K_u, T_u\}$ following this approach.

Exercise 18.8 Compute (exactly) the phase of a lead controller $D_{\text{lead}}(s) = K(s+z)/(s+p)$, for $z < p$, at the frequency $\omega = \sqrt{pz}$, and plot this phase as a function of p/z for $1 < p/z < 50$.

Exercise 18.9 Consider a plant

$$G(s) = \frac{s+20}{s^3+40s^2}.$$

a. Assuming proportional feedback, draw the root locus of the closed-loop system.

b. If the plant is perfectly modeled, will the closed-loop system go unstable for sufficiently large K? Explain clearly, referring to the root locus.

c. If the system is controlled with a microcontroller (in discrete time) with a sampling frequency of 100 Hz, will the closed-loop system go unstable for sufficiently large K? Explain clearly, referring to the appropriate root loci in both the s plane and the z plane.

d. Assuming the gain of a proportional controller is adjusted such that crossover is achieved at $\omega = 100 rad/sec$, how much phase loss at crossover will there be due to the sampling at 100 Hz?

e. Convert $G(s)$ into a state space realization $\{A,B,C,D\}$, where A is in upper companion form.

Exercise 18.10 Design an effective controller for the double integrator system described in Example 18.4, Case (b).

Exercise 18.11 Using the systematic control design methods discussed in §18.3.6, design a Minimum Energy Stabilizing Controller for the system

$$G(s) = \frac{(s+2)(s-2)(s+4)(s-4)}{(s+1)(s-1)(s+3)(s-3)(s+5)(s-5)}. \tag{18.43}$$

Then, referencing the corresponding Bode plot, modify this MESC design by cascading it with an appropriate degree of low-pass filtering such that the resulting controller has similar performance, but is also strictly proper.

Exercise 18.12 Via slight modification of Algorithm 18.2, develop a code (and an associated test code) implementing the simplified Bistritz test (18.14).

Exercise 18.13 Using Algorithm 18.2, determine the critical value of K for which the CT root locus in Figures 18.36c-d goes unstable. Using the corresponding algorithm developed in Exercise 18.12 for the DT case, determine the critical value of K for which the DT root locus in Figures 18.36b goes unstable. Discuss.

Exercise 18.14 Consider a controller

$$D(s) = K\frac{s+50}{s+200}.$$

a. What is this kind of controller called? Draw its Bode plot by hand. For sinusoidal inputs near $\omega = 100 rad/sec$, does it act more like a differentiator or more like an integrator?

b. Convert the CT controller $D(s)$ to a discrete time controller $D(z)$ using Tustin's rule, using a sampling frequency of 100 Hz. Repeat this conversion using Tustin's rule with prewarping, designing for a crossover frequency of $\omega = 100 rad/sec$.

Exercise 18.15 Figure 17.5b illustrates an everyday example of a simple hanging pendulum on a cart; its linearized dynamics are governed by (17.18). Define the horizontal position of the load, $z'(t)$, as

$$z'(t) = x'(t) + \ell \sin\theta'(t) \approx x'(t) + \ell\,\theta'(t). \tag{18.44}$$

We consider a suspended point load, with $m_p = 1000$ kg and $I_p \approx 0$, and a cable of fixed length, $\ell = 20$ m.

(a) Combine (18.44) with (17.18)a-b to eliminate $x'(t)$ and $\theta'(t)$, and identify the (fourth-order) transfer function $G_1(s) = Z'(s)/U'(s)$ from the force $u'(t)$ to the position of the load $z'(t)$ [cf. (18.25)]. Based on the poles of this transfer function, characterize the dynamics of this system before control is applied. [Is it stable or unstable? If it is oscillatory, what are the oscillation period(s), and how much damping is there?] Then, compute a simple (force-based) controller $D_1(s)$ such that the closed-loop system has *zero overshoot* of the load (why is this sometimes important?) and a settling time of $t_s = 30$ s, and plot the system's step response.

(b) Compute the DT system $G_1(z)$ formed, using (17.68), by the cascade of a DAC with a ZOH, the plant $G_1(s)$ determined in (a), and an ADC, taking $h = 5$ s.

(c) Compute a (DT) *minimal-time* deadbeat controller $D_{1,\text{m-p}}(z)$ [see (18.35)] of the system computed in (b), such that z'_k settles exactly in the minimum number of timesteps possible. Plot the DT step response z'_k and, using Algorithm ?, overlay the plot the CT step response $z'(t)$. Discuss.

(d) Compute a (DT) *ripple-free* deadbeat controller $D_{1,\text{r-f}}(z)$ [see (18.36)] of the system computed in (b), such that both z'_k and u'_k settle exactly in the minimum number of timesteps possible. Plot the DT step response z'_k and, using Algorithm ?, overlay the plot the CT step response $z'(t)$. Discuss.

(e) Combine (18.44) with (17.18)a to eliminate $\theta'(t)$, and identify the (second-order) transfer function $G_2(s) = Z'(s)/X'(s)$ from the position of the cart $x'(t)$ to the position of the load $z'(t)$. Based on the poles of this transfer function, characterize the dynamics of this system before control is applied. Then, compute a simple (displacement-based) controller $D_2(s)$ such that the closed-loop system has *zero overshoot* of the load and a settling time of $t_s = 30$ s. In implementation, position-based control of the cart, as considered in (d), is more robust than force-based control of the cart, as considered in (a), as it is insensitive to modeling errors in (17.18)b [specifically, to errors in the modeling of the drag of the cart as it moves along the track].

Exercise 18.16 Stabilization of a Saturn V rocket with SLC

Consider the application of the stabilization of a rocket launch, as laid out in Example 17.10. Taking the Laplace transform of (17.42) leads immediately to

$$\left(ms^2 + \frac{\overline{f}_d\, s}{\overline{v}}\right) X(s) - f_t\, \Theta(s) = -f_t\, U(s) - W(s), \tag{18.45a}$$

$$\frac{\overline{f}_d\, Ls}{\overline{v}} X(s) + (Js^2 - \overline{f}_d\, L)\Theta(s) = f_t\, DU(s) - LW(s). \tag{18.45b}$$

Interpreting $\theta(t)$ as the control input and $f_w(t)$ as the disturbance input, we may again design the control via successive loop closure (see Figure 18.24) by taking $w(t) = 0$ and combining (18.45a)-(18.45b) to give

$$G_1(s) = \frac{\Theta(s)}{U(s)} = \frac{c_1 s + c_2}{c_3 s^3 + \hat{f} s^2 - c_4 s + c_5}, \quad G_2(s) = \frac{X(s)}{\Theta(s)} = -\frac{Js^2 - c_6}{c_7 s^2 + c_8 s}, \tag{18.46}$$

with $\hat{f} = J/L$, $\hat{D} = D/L$, $c_1 = f_t\, m\, \overline{v}\hat{D}/\overline{f}_d$, $c_2 = f_t(\hat{D}+1)$, $c_3 = m\overline{v}\hat{f}/\overline{f}_d$, $c_4 = m\overline{v}$, $c_5 = f_t - \overline{f}_d$, $c_6 = Df_t + L\overline{f}_d$, $c_7 = mD$, $c_8 = \overline{f}_d(L+D)/\overline{v}$. Following the examples given in §18.3.4, successive loop closure may be used as suggested in Figure 18.24 (coincidentally, taking again θ as controlled by the inner loop and x as controlled by the outer loop). For the design condition $\overline{v} = 100$ m/s, answer the following questions:

(a) For the rocket problem described above, assuming $L = 10$ with 20% uncertainty, design an SLC controller to stabilize the rocket using lead control to stabilize the inner loop. Plot the root locus, Bode, and step response of the complete system with $L = 8$, $L = 10$, and $L = 12$. Is this result sensitive to the precise value of L? Discuss.

(b) Repeat question (a), but now assuming $L = -10$ with 20% uncertainty. Plot the root locus, Bode, and step response of the complete system with $L = -8$, $L = -10$, and $L = -12$. Is this result sensitive to the precise

value of L? Discuss.

(c) Repeat question (b), but now use notch control to stabilize the inner loop. Make sure you design a single notch filter in a way that ensures that the notch will work for L anywhere within the specified range. Plot the root locus, Bode, and step response of the complete system with $L = -8$, $L = -10$, and $L = -12$. Is this result sensitive to the precise value of L? Discuss.

(d) Repeat exercise (c), assuming $L = 10$ with now only 5% uncertainty. As the range of L you are designed the notch filter for is now more precisely known, the performance should be improved. Plot the root locus, Bode, and step response of the complete system with $L = -9.5$, $L = -10$, and $L = -10.5$. Compare the nominal performance (that is, for $L = -10$) in this case versus that in cases (b) and (c). Discuss.

References

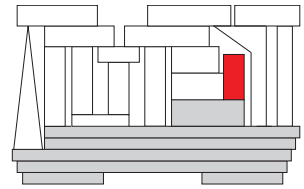
Evans, WR (1950) Control System Synthesis by Root Locus Method *AIEE Trans.* **69** part 2, 66-69.

Gillespie, TD (1992) *Fundamentals of Vehicle Dynamics*. Society of Automotive Engineers.

Lutovac, MD, Tosic, DV, & Evans, BL (2001) *Filter Design for Signal Processing Using MATLAB and Mathematica*. Prentice-Hall.

Nelson, R (1997) *Flight Stability and Automatic Control*. McGraw-Hill.

McCormick, BW (1995) *Aerodynamics, Aeronautics, and Flight Mechanics*. Wiley.



Chapter 19

Linear circuits

Contents

19.1 Introduction . . . 587
19.1.1 Electric charge, potential, and power . . . 587
19.1.2 Fundamental circuit elements . . . 589
19.1.3 Kirchoff's laws . . . 592
19.1.4 Laplace transform analysis of circuits and the definition of impedance . . . 595
19.2 Active analog circuits & filters . . . 597
19.2.1 Semiconductor circuit elements . . . 597
19.2.2 Operational amplifiers . . . 602
19.2.3 Design and analysis of a few useful op amp circuits . . . 603
Exercises . . . 612

19.1 Introduction

19.1.1 Electric charge, potential, and power

The SI units of the various quantities encountered in electric circuits is now summarized:

- **Charge** is denoted q. The fundamental unit of charge is that of an electron; the (negative) charge of 6.2415×10^{18} electrons is called a **coulomb** (C), which is the SI unit for charge.
- Electric charge passing a given point per unit time is called **current**. The current at any instant is denoted $I = dq/dt$. The SI unit for current is the **ampere** (A, a.k.a. **amp**), which is a flow of 1 C / s.
- **Energy** is denoted w, and the SI unit for (mechanical or electrical) energy is the **joule** (J). In mechanical terms, a Joule of energy is 1 kg m^2 / s^2, which may be interpreted as 1 N m when applying a force to a mass over a distance, or as 0.2390 calories of thermal energy, where 1 **calorie** (cal) is the amount of thermal energy it takes to warm 1 g (that is, 1 mL, or 1 cm^3) of water by 1°C at standard atmospheric conditions. Electric energy, also measured in Joules, is the electric equivalent, as electrical energy can easily be converted to mechanical energy (to apply a force over a distance) or to heat.

- **Power** is the rate of change of energy at any instant (that is, energy is the integral of power over time), and is denoted $P = dw/dt$; the SI unit for power is the **watt** (W), which is 1 J / s. In mechanical terms, a watt of power is 1 kg m^2 / s^3, which may be interpreted as 1 N m / s when applying a force to a mass moving at a certain speed, or as or as 0.2390 cal / s when warming a material.
- In an electric circuit, associated with any electron is its potential to do work[1] relative to some appropriately defined base state, called the **ground** state; this concept is analogous to the gravitational potential energy associated with any mass at any given height relative to the earth's surface. The **potential** of a charge to do work, also called the **voltage** of this charge, is denoted $V = dw/dq$, and is defined relative to this ground state; the SI unit for potential is the **volt** (V), which is 1 J / C.

Via the above definitions and the chain rule, it follows immediately that

$$P = \frac{dw}{dt} = \frac{dw}{dq}\frac{dq}{dt} \quad \Rightarrow \quad \boxed{P = VI} \tag{19.1}$$

Current may be envisioned as a flow of electrons, as described above; however, by convention, the (positive) direction of the current is defined as the direction *opposite* to the flow of electrons. This is known as the **passive sign convention**. Using this (at first, somewhat peculiar[2]) convention, when considering the voltage V across a device and the current I through a device, multiplying V times I as suggested by (19.1) results in

- *positive* power P if the device *absorbs* electric power from the rest of the circuit, as in a resistor[3], with current flowing from *higher* voltage to *lower* voltage, and
- *negative* power P if the device *delivers* electric power to the rest of the circuit, as in a battery, with current flowing from *lower* voltage to *higher* voltage.

Recall also the usual prefixes of the SI system:

prefix:	deci	centi	milli	micro	nano	pico	femto	atto	zepto	yocto
symbol:	d	c	m	μ	n	p	f	a	z	y
factor:	10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}	10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}

prefix:	deca	hecto	kilo	mega	giga	tera	peta	exa	zetta	yotta
symbol:	da	h	k	M	G	T	P	E	Z	Y
factor:	10^{1}	10^{2}	10^{3}	10^{6}	10^{9}	10^{12}	10^{15}	10^{18}	10^{21}	10^{24}

On the electric grid of a city, energy is usually billed in **kilowatt hours** (kW h) instead of megajoules (MJ); note that 1 kW h = 3.6 MJ. Similarly, battery charge is usually measured as **milliamp hours** (mA h) instead of coulombs (C); note that 1 mA h = 3.6 C.

The change of energy of a single electron if it is moved across a potential difference of one volt is defined as an **electron volt** (eV), and is given by $1/(6.2415 \times 10^{18}) = 1.6022 \times 10^{-19}$ J. Note that the energy E of a **photon** is given by $E = hc/\lambda$, where **Planck's constant** $h = 6.626 \times 10^{-34}$ J s and the **speed of light** $c = 2.99792 \times 10^{8}$ m / s; thus, if a single electron moves across a 1.91 V potential difference, then releases its excess energy as a photon, the resulting photon has wavelength $\lambda = 650$ nm, and is thus red in color.

[1] As an example, consider two identical metal spheres, one with an excess of electrons (said to be of lower voltage), and one with a depletion of electrons (said to be of higher voltage). If a resistor is connected between the two spheres, the excess repulsive force between the electrons on the first sphere tends to push electrons through the resistor and onto the second sphere until a balanced distribution of electrons is reached. In the process, the electrons being pushed through the resistor do work, generating heat.

[2] The reason for this peculiar convention is that the fundamental charge associated with an electron is defined as being *negative*; this perhaps unfortunate definition was made early on, and it stuck.

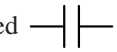
[3] The power absorbed may be converted it into **heat**, as in a resistor, a combination of **heat & electromagnetic radiation**, as in a lightbulb, laser, or RF transmitter, a combination of **heat & mechanical power**, as in a motor, fluid pump, or speaker coil, etc., or it may alternatively be *stored* (and, later, released), as in a capacitor or inductor (see §19.1.2), a rechargeable battery, a flywheel, etc.

19.1.2 Fundamental circuit elements

Resistors, capacitors, & inductors

Idealized current-voltage relationships for three common devices[4] used in analog circuits are

resistor (denoted ): $\boxed{V = RI} \quad \Rightarrow \quad P_{\mathrm{R}} = VI = I^2 R = V^2/R > 0,$ (19.2a)

capacitor (denoted ): $\boxed{I = C\,dV/dt} \quad \Rightarrow \quad P_{\mathrm{C}} = VI = (C/2)\,dV^2/dt = dw_C/dt,$ (19.2b)

inductor (denoted ): $\boxed{V = L\,dI/dt} \quad \Rightarrow \quad P_{\mathrm{L}} = VI = (L/2)\,dI^2/dt = dw_L/dt,$ (19.2c)

where $w_C = CV^2/2$ and $w_L = LI^2/2$. Approximate values of R, C, and L for such devices are identified with **color bands** or **numerical codes**, the interpretation of which are easy to find on the web. Note that:

- The SI unit for resistance R is the **ohm** (Ω) [thus, (19.2a) is known as **Ohm's law**]; an ohm is 1 V / A.
- The SI unit for capacitance C is the **farad** (F); a farad is 1 A s / V.
- The SI unit for inductance L is the **henry** (H); a henry is 1 V s / A.

Note also that power is always *absorbed* by a resistor, but is either *absorbed from* or *delivered to* the rest of the circuit by a capacitor (or inductor) based on the sign of the rate of change of the square of voltage (or current) over/through the device.

The idealized linear models listed above are accurate only for sufficiently small V and I inside what are known as the **rated limits** of the corresponding device; outside these limits, nonlinearities become significant (and far outside these limits, the corresponding device will fail).

The flow of electrons along a metal wire, like the flow of water through a pipe, is almost always[5] associated with some loss of potential per unit length (and, therefore, some resistance), as energy is lost as heat to sustain the flow when the magnetic fields generated by the flowing electrons interact with the electromagnetic fields of the atoms within the material. *Wires* are simply made from an appropriate metal, like copper, with relatively *low* (often, negligible) resistance per unit length, whereas *resistors* are made from an appropriate metal, like Nichrome (a non-magnetic alloy of nickel and chromium), that exhibits a relatively *high* amount of resistance per unit length, with a resistance that is fairly insensitive to the inevitable temperature fluctuations that are caused by driving current through the resistor. Note also that, when running a large current through a resistor beyond its rated limits, the metal warms up, and thus atoms within the metal start vibrating more energetically; this generally reduces the effective resistance R of the resistor when $|I|$ is large, thus leading to a nonlinear relationship between V and I [cf. (19.2a)].

Though they are packaged in a variety of different convenient geometries, *capacitors* are easily visualized in their most simple form as two parallel metal plates with a nonconducting material, called a **dielectric**, between them. If a voltage is suddenly applied across a capacitor, electrons flow in one wire and accumulate on one of the plates, electrically repelling the electrons on the other (nearby) plate, which then flow out the other wire, thus generating a current through the device. As more and more electrons accumulate on the first plate and are depleted from the second plate, however, an *electric potential difference* is built up between the two plates, thus inhibiting the further flow of electrons; at steady state, the current through the device reduces to zero. The resulting (linearized) relationship between between V and I is given in (19.2b). Note also that, as shown in (19.2b), the power absorbed by or released from a capacitor at any instant, P_C, is simply the rate of change of the energy, $w_C = CV^2/2$, stored in the capacitor.

Similarly, *inductors* are easily visualized in their most simple form as a tightly-wound (often, toroidal) copper wire coil wrapped around an air or ferromagnetic **core**; when current flows through the wire, a strong

[4] As suggested by Thomas & Rosa (2010), we will refer to the actual physical hardware components that a circuit is made from as **devices**, and the governing equations we use to model them as **elements**.

[5] The notable exception to this statement is the remarkable class of materials known as **superconductors**, which at temperatures below a material-dependent critical temperature exhibit **zero resistance** and the expulsion of magnetic fields from within the material.

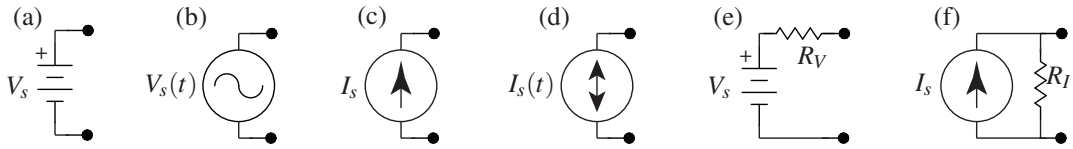


Figure 19.1: Various power sources and the symbols used for them in this text: (a) ideal constant voltage source, (b) ideal time-varying voltage source, (c) ideal constant current source, (d) ideal time-varying current source, (e) practical constant voltage source, (f) practical constant current source.

magnetic field is built up within the core. If a current is suddenly applied through the inductor, the existing magnetic field in the core, or lack thereof, opposes the newly applied current, thus generating a large voltage across the the device. As more and more current flows through the device and the magnetic field within the core builds to a strength consistent with the applied current and the number of windings of the coil, however, this opposing *electromotive force* on the moving electrons is reduced; at steady state, the voltage across the device reduces to (nearly) zero. The resulting (linearized) relationship between between V and I is given in (19.2c). Note also that, as shown in (19.2c), the power absorbed by or released from an inductor at any instant, P_L, is simply the rate of change of the energy, $w_L = LI^2/2$, stored in the inductor.

Note finally that the prepackaged resistors, capacitors, and inductors that are commercially available are manufactured with significant variation. For example, 1 kΩ resistors are available at the following variances in their actual resistance: $\{\pm 10\%, \pm 5\%, \pm 2\%, \pm 1\%, \pm .5\%, \pm .25\%, \pm .1\%, \pm .05\%\}$. Such devices are often produced as the result of a single manufacturing process, then tested to determine their precise resistance (using, for example, the Wheatstone bridge circuit analyzed in Example 19.2). They are then binned accordingly and, of course, those resistors most closely matching the target resistance are sold at a higher price. The result of this manufacturing/sorting process is that the distribution of the actual resistance of those resistors marked at higher variance values are often **bimodal**, as those units that more accurately match the target resistance value are not placed in the higher-variance bins. The manufacture of precision resistors is often accomplished by accurate **laser trimming** of resistors that are initially slightly below the target resistance.

Power sources

In order to make an electric circuit do something, of course, you need a source[6] of electric power. Such sources come in two types, voltage sources (which are most common) and current sources, either of which may produce constant or time-varying signals, and are denoted as indicated in Figure 19.1a-d.

The current-voltage relationships of ideal voltage and current sources may be written

$$\textbf{ideal voltage source}: \quad \boxed{V = V_s \ \text{(regardless of } I\text{)}} \tag{19.2d}$$

$$\textbf{ideal current source}: \quad \boxed{I = I_s \ \text{(regardless of } V\text{)}} \tag{19.2e}$$

Note that an ideal **voltage source** generates a specified *voltage* across its terminals[7] regardless of the current drawn by the rest of the circuit; an ideal voltage source can not function correctly if a wire (with zero resistance) is connected across its terminals (a.k.a. a **short circuit**), as that would cause the ideal voltage source to produce infinite current. Similarly, an ideal **current source** generates a specified *current* through the device(s) connected across its terminals regardless of the voltage required over the rest of the circuit in order to

[6] Note that some devices that normally act as *sources* of electric power, like rechargeable batteries, may also from time to time be used safely as *sinks* of electric power, like a capacitor. The practical distinction between a capacitor and a rechargeable battery is that a capacitor, which simply stores and releases electrons, typically loses its charge fairly quickly when not being used, whereas a battery, which stores and releases charge via internal chemical reactions, typically holds its charge for much longer periods of time.

[7] Note that a **terminal** of an electric circuit or individual circuit element is a point where other electric circuits are intended to be attached, as denoted by black dots in Figure 19.1.

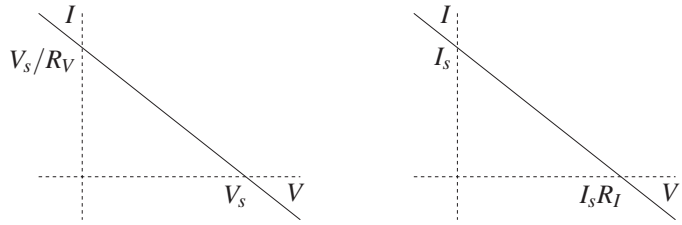


Figure 19.2: Current-voltage relationship of (left) the practical voltage source of Figure 19.1e, and (right) the practical current source of Figure 19.1f.

maintain it; an ideal current source can not function correctly if the circuit connected across its terminals is not closed (a.k.a. an **open circuit**), as that would cause the ideal current source to produce infinite voltage.

Despite the above-mentioned limitations, ideal voltage and current sources are reasonably good models in many situations when a circuit is properly configured. More accurate (yet still linear) models of real-world voltage and current sources are indicated in Figure 19.1e-f. Note that, in these more practical models,

- a resister is included in *series* with the voltage source, which thus generates a current of $I = V_s/R_V$ instead of an infinite current in the case of a short circuit across its terminals, and
- a resister is included in *parallel* with the current source, which thus generates a voltage of $V = I_s R_I$ instead of an infinite voltage in the case of a open circuit across its terminals.

The current-voltage relationship of the practical voltage and current sources indicated in Figures 19.1e-f are given in Figure 19.2. Note that, taking $I_s = V_s/R_V$ and $R_I = R_V$, these two relationships are identical, and thus these two sources are, consistent with the following definition, said to be **equivalent**[8].

Fact 19.1 (Equivalent circuit definition) *Two circuits are said to be* **equivalent** *at a specified pair of terminals if they exhibit an identical current-voltage relationship.*

Sensors & actuators for interfacing with the physical world

To connect an electric circuit to the physical world, sensors and actuators[9] are needed. Actuators are often built from some type of **electric motor**; others include **linear actuators** (like **voice coils**), **electroactive polymers**, etc. Common sensors include **accelerometers** (**accels**) to measure linear acceleration, **gyroscopes** (**gyros**) to measure angular acceleration, **encoders** to measure wheel rotation, **thermocouples** to measure temperature, etc. Note that some actuators which convert electrical energy to mechanical energy (like motors and piezoelectric actuators[10]) can also be used as sensors or **energy scavengers** to convert mechanical energy back into electrical energy (like generators and piezoelectric sensors[10]); this concept is central to the efficient operation of hybrid and fully electric cars like the Toyota Prius and the Chevy Volt, in which the motor normally used to drive the wheels may be operated as a generator during regenerative braking.

Though space constraints do not permit a review of the remarkable range of sensors and actuators available today, some issues regarding the use of brushed DC motors are considered further in Example 19.18.

[8] Note that, though these two practical source models are equivalent from the perspective of the current-voltage relationship at their terminals, they are *not at all equivalent* in terms of their internal operation:

- the practical voltage source expends no energy whatsoever if there is an *open* circuit across its terminals (since the current through the ideal voltage source component is zero in this case), but expends significant energy if there is a *closed* circuit across its terminals (since the current through the ideal voltage source component is V_s/R_V in this case);
- in contrast, the practical current source expends no energy whatsoever if there is a *closed* circuit across its terminals (since the voltage across the ideal current source component is zero in this case), but expends significant energy if there is an open circuit across its terminals (since the voltage across the ideal current source component is $I_s R_I$ in this case).

[9] More broadly, devices which convert one form of energy (electric, mechanical, thermal, etc.) to another are known as **transducers**.

[10] That is, actuators/sensors built on materials exhibing both the **piezoelectric effect**, generating an electric field in response to an applied mechanical strain, and the **reverse piezoelectric effect**, generating mechanical strain in response to an applied electric field.

19.1.3 Kirchoff's laws

A **node** of an electric circuit is defined as any point where two or more circuit elements (and, thus, two or more current paths) are connected. In a complex electric circuit with several circuit elements and several current paths, the following two simple rules facilitate analysis:

Fact 19.2 (Kirchoff's Current Law, or KCL) *The sum of the currents entering a node equals the sum of the currents leaving that node at any instant.*

Fact 19.3 (Kirchoff's Voltage Law, or KVL) *The sum of the voltages across the elements around any closed loop in a circuit is zero at any instant.*

Note that KVL may be satisfied *by construction* simply by keeping track of the voltage at each *node* of the circuit, rather than the voltage drop across each circuit element. Note also that, in a circuit with n nodes, there are only $(n-1)$ independent KCL equations for the currents between these nodes, as the KCL at the last node may be derived by combining appropriately the KCL relations at the other $(n-1)$ nodes.

Defining the voltage at each node (thus implicitly satisfying the KVL equations) and the current between each node, writing KCL at all but one of the nodes, and writing the current-voltage relationship across each circuit element [see, e.g., (19.2a)-(19.2e)] leads to a set of ODEs which, together with the initial conditions, completely describe the time evolution of the circuit. This is best illustrated by a few examples:

Example 19.1 Equivalent resistance, capacitance, and inductance. The concept of equivalent circuits, with identical current-voltage relationships at a pair of terminals, was defined in Fact 19.1. By the KCL and KVL given above, it follows that a set of n resistors, capacitors, or inductors in a **series connection** (see Figure 19.3a), in which the current I through the devices is equal and the voltages add[11], $\Delta V_1 + \Delta V_2 + \ldots \Delta V_n = \Delta V$, have the **equivalent resistance** R, **equivalent capacitance** C, or **equivalent inductance** L, respectively, of:

$$\begin{array}{lllllll}
\Delta V_1 = I R_1, & \Delta V_2 = I R_2, & \ldots & \Delta V_n = I R_n & \Rightarrow & \Delta V = I R & \text{where} \quad R = R_1 + R_2 + \ldots R_n, \\
\dfrac{d\,\Delta V_1}{dt} = \dfrac{I}{C_1}, & \dfrac{d\,\Delta V_2}{dt} = \dfrac{I}{C_2}, & \ldots & \dfrac{d\,\Delta V_n}{dt} = \dfrac{I}{C_n} & \Rightarrow & \dfrac{d\,\Delta V}{dt} = \dfrac{I}{C} & \text{where} \quad \dfrac{1}{C} = \dfrac{1}{C_1} + \dfrac{1}{C_2} + \ldots \dfrac{1}{C_n}, \\
\Delta V_1 = L_1 \dfrac{dI}{dt}, & \Delta V_2 = L_2 \dfrac{dI}{dt}, & \ldots & \Delta V_n = L_n \dfrac{dI}{dt} & \Rightarrow & \Delta V = L \dfrac{dI}{dt} & \text{where} \quad L = L_1 + L_2 + \ldots L_n.
\end{array}$$

Similarly, a set of n resistors, capacitors, or inductors in a **parallel connection** (see Figure 19.3b), in which the voltage ΔV across the devices is equal and the currents add, $I_1 + I_2 + \ldots I_n = I$, have the **equivalent resistance** R, **equivalent capacitance** C, or **equivalent inductance** L, respectively, of:

$$\begin{array}{lllllll}
I_1 = \dfrac{\Delta V}{R_1}, & I_2 = \dfrac{\Delta V}{R_2}, & \ldots & I_n = \dfrac{\Delta V}{R_n} & \Rightarrow & I = \dfrac{\Delta V}{R} & \text{where} \quad \dfrac{1}{R} = \dfrac{1}{R_1} + \dfrac{1}{R_2} + \ldots \dfrac{1}{R_n}, \\
I_1 = C_1 \dfrac{d\,\Delta V}{dt}, & I_2 = C_2 \dfrac{d\,\Delta V}{dt}, & \ldots & I_n = C_n \dfrac{d\,\Delta V}{dt} & \Rightarrow & I = C \dfrac{d\,\Delta V}{dt} & \text{where} \quad C = C_1 + C_2 + \ldots C_n, \\
\dfrac{dI_1}{dt} = \dfrac{\Delta V}{L_1}, & \dfrac{dI_2}{dt} = \dfrac{\Delta V}{L_2}, & \ldots & \dfrac{dI_n}{dt} = \dfrac{\Delta V}{L_n} & \Rightarrow & \dfrac{dI}{dt} = \dfrac{\Delta V}{L} & \text{where} \quad \dfrac{1}{L} = \dfrac{1}{L_1} + \dfrac{1}{L_2} + \ldots \dfrac{1}{L_n}.
\end{array}$$

In **reducible** connections of a single type of components, repeated application of the above rules is sufficient to determine the equivalent single component value. For example, if the Z_k in Figure 19.3c denote resistors,

- the equivalent resistance of the parallel connection of R_1 and R_2 is $R_a = R_1 R_2/(R_1 + R_2)$,
- the equivalent resistance of the parallel connection of R_4 and R_5 is $R_b = R_4 R_5/(R_4 + R_5)$, and
- the equivalent resistance of the entire series connection (of R_a, R_3, and R_b) is $R = R_a + R_3 + R_b$.

[11] In order to apply KVL as stated, visualize a battery connected across the terminals in each of the circuits in Figure 19.3.

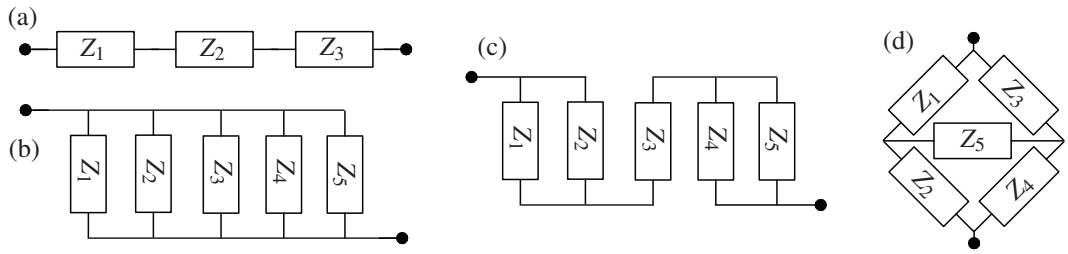


Figure 19.3: (a) Series, (b) parallel, (c) reducible, and (d) irreducible interconnections of a single type of component, with the Z_k denoting either resistors R_k, capacitors C_k, or inductors L_k.

On the other hand, repeated application of the above rules is *not* sufficient to simplify **irreducible** connections of a single type of components, such as that shown in Figure 19.3d, which must be treated directly with KCL/KVL. For example, if the Z_k in Figure 19.3d denote resistors, it follows from KCL, KVL, and Ohm's law [see (19.4) below, and Exercise 19.1] that the equivalent resistance is rather involved:

$$R = \frac{R_1R_2R_3 + R_1R_2R_4 + R_1R_3R_4 + R_1R_3R_5 + R_1R_4R_5 + R_2R_3R_4 + R_2R_3R_5 + R_2R_4R_5}{R_1R_2 + R_1R_4 + R_1R_5 + R_2R_3 + R_2R_5 + R_3R_4 + R_3R_5 + R_4R_5}. \tag{19.3}$$

Example 19.2 KCL/KVL analysis of Wheatstone bridges. Consider the **Wheatstone bridge** of Figure 19.4a. For convenience, we number the elements of a circuit sequentially, and denote by I_k the current through element k, with positive current indicated by the direction of the arrow (this keeps us from having to label each current component individually in the figure). The voltage at the top of the bridge is V_0 (as it is connected to the top of the battery), and the voltage at the bottom of the bridge is 0 (as it is connected to the bottom of the battery, which is defined as ground), thus leaving two undetermined nodal voltages, $\{V_a, V_b\}$. We may thus determine the eight unknowns $\{I_0, I_1, I_2, I_3, I_4, I_5, V_a, V_b\}$ given the six parameters $\{V_0, R_1, R_2, R_3, R_4, R_5\}$ via KCL at three of the four nodes and the current-voltage relationship across each of the five resistors:

$$I_0 = I_1 + I_3, \quad I_1 = I_2 + I_5, \quad I_3 + I_5 = I_4, \tag{19.4a}$$

$$V_o - V_a = I_1R_1, \quad V_o - V_b = I_3R_3, \quad V_a - V_b = I_5R_5, \quad V_a = I_2R_2, \quad V_b = I_4R_4. \tag{19.4b}$$

This amounts to eight linear equations for the eight unknowns, which may easily be solved.

The Wheatstone bridge is particularly useful for the precise measurement of a resistor value (taken here as R_4) given three other resister values (taken here as $\{R_1, R_2, R_3\}$). Indeed, if $R_1/R_2 = R_3/R_4$, then the bridge is said to be **balanced**, and the current through the resister in the center of the bridge, I_5 (which may be measured precisely using a **galvanometer**), will be exactly zero, as $V_a = V_b$ in this case. If $R_1/R_2 \neq R_3/R_4$, then it is straightforward to plug the above equations together by hand to determine I_5 as a function of R_4, given values of $\{V_0, R_1, R_2, R_3, R_5\}$. To save time and prevent algebra mistakes, we may instead use symbolic manipulation: assuming $R_1 = R_2 = R_3 = 1$ kΩ, $R_5 = 100$ kΩ, and $V_0 = 5$ V, enumerating in the **x** vector the eight unknowns in the order listed above, and writing (in order) the eight linear equations (19.4) in these unknowns as $A\mathbf{x} = \mathbf{b}$, the system may be solved via the following symbolic computation in Matlab:

```
syms R4;   V0=5; R1=1e3; R2=1e3; R3=1e3; R5=1e5;
A=[1 -1 0 -1 0 0 0 0; 0 1 -1 0 0 -1 0 0; 0 0 0 1 -1 1 0 0; 0 R1 0 0 0 0 1 0;  ...
     0 0 0 R3 0 0 0 1; 0 0 0 0 0 R5 -1 1; 0 0 R2 0 0 0 -1 0; 0 0 0 0 R4 0 0 -1];
b=[0; 0; 0; V0; V0; 0; 0; 0];  x=A\b
```

from which it follows immediately that $I_5 = (1000 - R_4)/(4.06 \times 10^4 R_4 + 4.02 \times 10^7)$ amps.

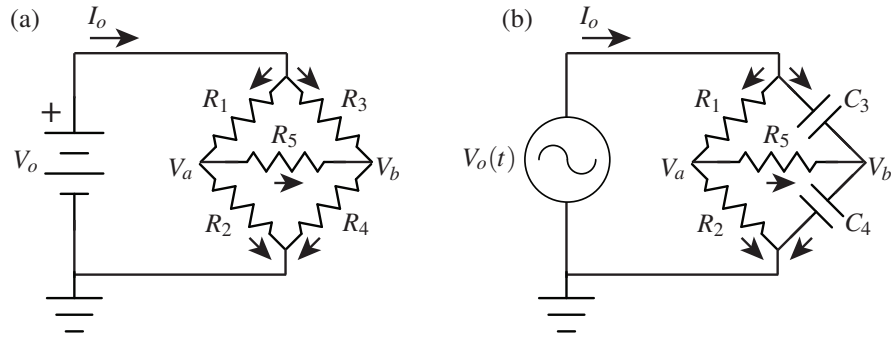


Figure 19.4: Wheatstone bridges for accurate measurement of (a) resistance, and (b) capacitance.

Now, if we simply replace the resistors R_3 and R_4 in Figure 19.4a with capacitors C_3 and C_4, where C_3 is known and C_4 is unknown, and observe the circuit at steady state, we run into a problem: setting the time derivatives equal to zero (that is, taking $I_3 = I_4 = 0$), it follows that $I_5 = 0$ and thus $V_a = V_b$ regardless of the value of C_4! Thus, C_4 can *not* be determined in this simplistic manner.

However, as indicated in Figure 19.4b, if we replace resistors R_3 and R_4 with capacitors C_3 and C_4, and we also replace the constant voltage source with a (sinusoidal) time-varying voltage source, then we can now easily determine C_4. Our eight equations for the eight unknowns $\{I_0, I_1, I_2, I_3, I_4, I_5, V_a, V_b\}$ now take the form

$$I_0 = I_1 + I_3, \quad I_1 = I_2 + I_5, \quad I_3 + I_5 = I_4,$$
$$V_o - V_a = I_1 R_1, \quad I_3 = C_3\, d(V_o - V_b)/dt, \quad V_a - V_b = I_5 R_5, \quad V_a = I_2 R_2, \quad I_4 = C_4\, d(V_b)/dt.$$

Assuming $R_1 = R_2 = 1$ kΩ, $C_3 = 10$ μF, and $R_5 = 100$ kΩ, that $\{I_0, I_1, I_2, I_3, I_4, I_5, V_a, V_b, V_o\}$ are all initially zero, taking the Laplace transform, and performing an analogous symbolic manipulation:

```
syms C4 s V0;   R1=1e3; R2=1e3; C3=1e-5; R5=1e5;
A=[1 -1 0 -1 0 0 0 0; 0 1 -1 0 0 -1 0 0; 0 0 0 1 -1 1 0 0; 0 R1 0 0 0 0 1 0;  ...
   0 0 0 1 0 0 0 C3*s; 0 0 0 0 0 R5 -1 1; 0 0 R2 0 0 0 -1 0; 0 0 0 0 1 0 0 -C4*s];
b=[0; 0; 0; V0; C3*s*V0; 0; 0; 0];  x=A\b
```

it follows that, in Laplace transform coordinates,

$$\frac{I_5(s)}{V_0(s)} = G(s) = \frac{(C_4 - C_3)s}{(2.01 + 2.01 \times 10^5\, C_4)s + 2}.$$

As in the case of drawing a Bode plot, we are interested in the magnitude and phase of the output, $I_5(s)$, when the input, $V_0(s)$, is sinusoidal. Rather than taking $V_0(s) = V\sin(\omega t)$ or $V_0(s) = V\cos(\omega t)$, we can simplify the math by taking $V_0(s) = Ve^{i\omega t}$, in which case it again follows that $I_5(t) = d_0 e^{i\omega t} +$ other terms that decay with time, where $|d_0| = |G(i\omega)|$ and $\angle d_0 = \angle G(i\omega)$. Writing the persistent component of the current through the center resistor as $I_5(t) = Ie^{i(\omega t + \phi)}$, it follows that

$$\frac{I}{V} = \frac{|C_4 - C_3|\,\omega}{\sqrt{(2.01 + 2.01 \times 10^5\, C_4)^2 \omega^2 + 4}}, \quad \phi = \begin{cases} 90^\circ - \text{atan2}[(2.01 + 2.01 \times 10^5\, C_4)\omega, 2] & \text{if } C_4 > C_3, \\ -90^\circ - \text{atan2}[(2.01 + 2.01 \times 10^5\, C_4)\omega, 2] & \text{if } C_4 < C_3. \end{cases}$$

If $\omega = 0$, then $I = 0$ regardless of C_4, consistent with the comments made in the previous paragraph. If $\omega > 0$ and $I = 0$, it follows immediately that the bridge is in balance and thus $C_4 = C_3 = 10$ μF. If $\omega > 0$ and $I \neq 0$, C_4 may be determined from I and ϕ according to the above expressions, noting the 180° shift in ϕ when C_4 goes from below C_3 to above C_3.

Inductance may be quantified with a Wheatstone bridge in an analogous fashion (see Exercise 19.2).

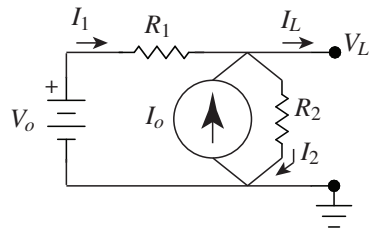


Figure 19.5: A circuit with both voltage and current sources as well as two resistors.

Example 19.3 Equivalent sources Any combination of voltage sources, current sources, and resistors leads to a linear current-voltage relationship like those in Figure 19.2; the following facts follow as consequence.

Fact 19.4 (Thévenin's theorem) *Any circuit containing only voltage sources, current sources, and resistors can be converted to a* **Thévenin equivalent circuit***, with one ideal voltage source and one resistor in series.*

Fact 19.5 (Norton's theorem) *Any circuit containing only voltage sources, current sources, and resistors can be converted to a* **Norton equivalent circuit***, with one ideal current source and one resistor in parallel.*

As an example, consider the circuit shown in Figure 19.5. Writing down KCL at the node at the top of the circuit and Ohm's law across each resistor, the current-voltage relationship at the terminals may be determined:

$$\left.\begin{aligned} I_1 + I_o &= I_L + I_2 \\ V_o - V_L &= I_1 R_1 \\ V_L &= I_2 R_2 \end{aligned}\right\} \quad \Rightarrow \quad I_L = I_1 + I_o - I_2 = \left(I_o + \frac{V_o}{R_1}\right) - \left(\frac{1}{R_1} + \frac{1}{R_2}\right) V_L. \tag{19.5}$$

It follows from this analysis of the circuit in Figure 19.5 that

- its Thévenin equivalent (Figure 19.1e) sets $R_V = R_1 R_2/(R_1 + R_2)$ and $V_s = (I_o + V_o/R_1)R_V$, and
- its Norton equivalent (Figure 19.1f) sets $R_I = R_1 R_2/(R_1 + R_2)$ and $I_s = I_o + V_o/R_1$;

note that all of these circuits have identical current-voltage relationships, as illustrated in Figure 19.2.

Of particular interest in this example is the question of how much power is actually provided by the two sources. To simplify, assume first that $R_1 = I_L = 0$; in this case, $V_L = V_o$, and

- the power absorbed by the current source is $-I_o V_o < 0$ (that is, power is *provided* by the current source regardless of the relative magnitudes of I_o and V_o/R_2), whereas
- the power absorbed by the voltage source is $-I_1 V_o = -(I_2 - I_o)V_o = -(V_o/R_2 - I_o)V_o$ (that is, power is *provided* by the voltage source if $V_o/R_2 > I_o$, and is *absorbed* by the voltage source if $V_o/R_2 < I_o$).

Taking $R_1 > 0$ and $I_L \neq 0$, similar conclusions may be drawn (see Exercise 19.3).

19.1.4 Laplace transform analysis of circuits and the definition of impedance

As seen in the first half of Example 19.2 and Example 19.3, in simple circuits without capacitors or inductors, combining KCL and KVL and the current-voltage relationship across each component leads to straightforward systems of *algebraic* equations which may be solved by hand or with symbolic numerical tools.

As seen in the second half of Example 19.2, when considering circuits which incorporate capacitors and/or inductors, combining KCL and KVL and the current-voltage relationship across each component leads more generally to sets of *linear constant-coefficient ODEs* together with algebraic constraints (jointly referred to as **descriptor systems**). Without the Laplace transform, as developed in §17, the analysis of such systems would be difficult. However, as seen in Example 19.2, application of the Laplace transform to such descriptor

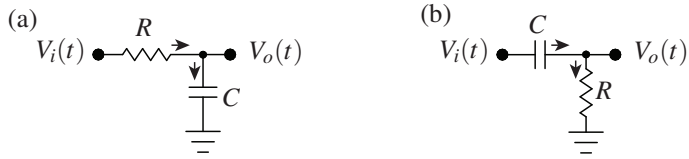


Figure 19.6: Two passive first-order filters: (a) a low-pass filter, (b) a high-pass filter. See also Exercise 19.4.

systems converts them back to straightforward systems of algebraic equations, incorporating the Laplace transform variable s, that are again easy to solve by hand or with symbolic numerical tools.

It is thus seen that, when analyzing electric circuits, *working in the Laplace domain is essential*. Further, one is often (but not always) interested in the *frequency response* of an electric circuit subject to sinusoidal excitation; as shown in §17.5.1, the magnitude and phase of the persistent sinusoidal component of the output of an input-output system $G(s)$ when excited by a sinusoidal input may be calculated simply by evaluating the magnitude and phase $G(\mathrm{i}\omega)$. Thus, taking the Laplace transform of the fundamental current-voltage relationships for resistors, capacitors, and inductors, as listed in (19.2a)-(19.2c), and evaluating at $s = \mathrm{i}\omega$ gives

$$G_{\text{resistor}}(\mathrm{i}\omega) = \frac{V(\mathrm{i}\omega)}{I(\mathrm{i}\omega)} = R \triangleq Z_R, \quad G_{\text{capacitor}}(\mathrm{i}\omega) = \frac{V(\mathrm{i}\omega)}{I(\mathrm{i}\omega)} = \frac{-\mathrm{i}}{\omega C} \triangleq Z_C, \quad G_{\text{inductor}}(\mathrm{i}\omega) = \frac{V(\mathrm{i}\omega)}{I(\mathrm{i}\omega)} = \mathrm{i}\omega L \triangleq Z_L.$$

The quantities Z_R, Z_C, and Z_L are used often, and are commonly referred as the **impedance** of a resistor, a capacitor, and an inductor, respectively. Note that, in other texts, the concept of impedance is often introduced loosely as a *complex generalization of resistance* even before Laplace transforms are properly introduced. This approach is unnecessarily convoluted; pedagogically, the author recommends instead mastering the Laplace transform (§17.3) and the Bode plot (§17.5.1) before reading the present discussion; the frequency response of the current-voltage relationships represented by the (complex) transfer functions $G_{\text{resistor}}(s)$, $G_{\text{capacitor}}(s)$, and $G_{\text{inductor}}(s)$, as listed above, are then quite easy to interpret[12]:

- the voltage across a capacitor *lags* the current through the capacitor by 1/4 cycle ($\phi = -90^\circ$), with the magnitude of the sinusoidal voltage across the capacitor divided by the magnitude of the sinusoidal current through the capacitor *decreasing* with frequency;
- the voltage across an inductor *leads* the current through the inductor by 1/4 cycle ($\phi = 90^\circ$), with the magnitude of the sinusoidal voltage across the inductor divided by the magnitude of the sinusoidal current through the inductor *increasing* with frequency.

Example 19.4 Passive first-order filters. Consider the circuits shown in Figures 19.6a-b and assume that

(a) the input voltage $V_i(t)$ is precisely specified regardless of the current drawn by the filter, and
(b) the current, if any, out the output terminal [marked $V_o(t)$ in the figure] is negligible.

It follows that the **passive first-order low-pass filter** in Figure 19.6a is governed by

$$I_R = I_C, \quad V_i - V_o = I_R R, \quad I_C = C\frac{dV_o}{dt} \quad \Rightarrow \quad V_i(s) - V_o(s) = RCsV_o(s) \quad \Rightarrow \quad \frac{V_o(s)}{V_i(s)} = \frac{1/RC}{s+1/RC},$$

whereas the **passive first-order high-pass filter** in Figure 19.6b is governed by

$$I_C = I_R, \quad I_C = C\frac{d[V_i - V_o]}{dt}, \quad V_o = I_R R \quad \Rightarrow \quad RCs[V_i(s) - V_o(s)] = V_o(s) \quad \Rightarrow \quad \frac{V_o(s)}{V_i(s)} = \frac{s}{s+1/RC}.$$

Assumptions (a) and (b) above are restrictive: if the inputs of such **passive filters** are attached to a real sensors, if they are cascaded, or if their outputs are attached to real actuators, one or both of these assumptions are generally invalid. We thus need **active filters** which relax these assumptions, as developed below.

[12] As a mnemonic, a *capacitor has low voltage across it at high frequencies*, as electric charge doesn't have enough time build up on it, whereas an *inductor has low current through it at high frequencies*, as a compatible magnetic field doesn't have time to form.

19.2 Active analog circuits & filters

19.2.1 Semiconductor circuit elements

A **semiconductor** is a material (often, a single crystal[13] of **silicon**, **germanium**, **gallium arsenide**, or **silicon carbide**) that has conduction properties that may be tuned during fabrication in various useful ways.

A single pure crystal of semiconductor material, such as silicon, is generally nonconductive, as all of the **valence** (outer-shell) electrons of the constituent atoms are tied up in the **covalent bonds** of the crystal[14].

However, if a semiconductor crystal is formed, or **doped**, with **n-type dopant** atoms, such as **phosphorus**, **arsenic**, or **antimony**, an extra valence electron is introduced in the crystal lattice for each atom of the n-type dopant present in the crystal. These extra valence electrons can move fairly easily when a voltage is applied across the material, thus making an n-doped semiconductor, such as phosphorus-doped silicon, a conductor.

Similarly, if a semiconductor crystal is formed with **p-type dopant** atoms, such as **boron**, **aluminum**, or **gallium**, a valence electron is missing from the crystal lattice for each atom of the p-type dopant present in the crystal, forming what is known as a "**hole**" in the electron structure of the crystal. These holes in the electron structure of the crystal can also move fairly easily[15] when a voltage is applied across the material, thus making a p-doped semiconductor, such as boron-doped silicon, also a conductor.

p-n junctions & diodes

When a semiconductor crystal has various neighboring sections, some that are p-doped and some that are n-doped, thus forming **p-n junctions** within the semiconductor, useful electrical characteristics arise. For example, a semiconductor crystal which has just two adjacent doping regions, with a single p-n junction in the middle, is called a **semiconductor diode**, which behaves in the ideal setting as follows:

- If a semiconductor diode is put under **forward bias**, with positive voltage applied to the p side of the semiconductor and negative voltage applied to the n side of the semiconductor, then electrons will flow into the n side of the semiconductor, pushing free valence electrons in the crystal lattice towards the p-n junction. These electrons, in turn, flow into the nearby holes on the p side of the semiconductor and creating, in effect, a flow holes on the p side of the semiconductor that is equal in magnitude and opposite in direction to the flow of electrons on the n side of the semiconductor, thus sustaining an electric current through the material with very little (ideally, zero) resistance[16].
- If, on the other hand, a semiconductor diode is put under **reverse bias**, with positive voltage applied to the n side of the semiconductor and negative voltage applied to the p side of the semiconductor, then some of the extra valence electrons on the n side of the semiconductor are pulled away from the p-n junction and out of the semiconductor, and some of the holes in the electron structure of the crystal on the p side of the semiconductor are pulled away from the p-n junction and, effectively, out of the semiconductor, creating a so-called **depletion layer** with neither holes nor free valence electrons to carry moving charge (that is, to sustain the current) in the vicinity of the p-n junction. As a result, an ideal diode under negative bias does not conduct.

[13] **Amorphous** semiconductors, which lack a long-range ordered crystal structure, can also be manufactured, and can be done so in especially thin layers over large areas. Such semiconductors may be doped in a manner similar to the single-crystal semiconductors discussed in §19.2.1, and can be switched from one physical state to another (e.g., from translucent to opaque), which makes them especially useful in a variety of applications, such as **CD**s/**DVD**s/**BD**s, **liquid-crystal displays** (**LCD**s), and **solar cells**.

[14] Note that both silicon and germanium, like carbon, crystallize in a **diamond crystal structure**; gallium arsenide crystallizes in a **zincblende crystal structure**, which is simply a diamond crystal structure with gallium and arsenic in alternating lattice sites.

[15] Note that it is actually a neighboring electron in the crystal lattice that moves, thereby changing the "hole" location in the crystal; several successive movements of electrons into neighboring hole locations give the appearance that it is the hole itself that is moving.

[16] As a loose physical analogy of current flow in a diode under forward bias, one might visualize the electron motion (on the n-doped side) towards the p-n junction as tiny drops of rain falling through air toward an air/water interface, and the corresponding hole motion (on the p-doped side) towards the p-n junction as equally tiny bubbles of air rising through water toward the air/water interface at precisely the same rate, thus resulting in zero net accumulation of negative or positive charge (raindrops or bubbles) at the interface.

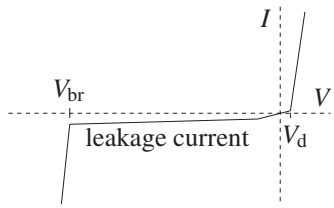


Figure 19.7: Typical current-voltage relationship of a diode, with forward bias given by $V > 0$ and reverse bias given by $V < 0$, indicating the **breakdown voltage** V_{br}, the leakage current, and the **cut-in voltage** V_d.

Though the ideal model of a semiconductor diode described above is adequate for many purposes, the deviations of actual semiconductor diodes from this ideal behavior are important to appreciate:

- Within the depletion layer described above in the reverse bias setting, the n-doped side, now lacking its extra valence electrons, is positively charged, and the p-doped side, now lacking its holes, is negatively charged by the same amount. This sets up an electric field across the p-n junction. When the applied voltage exceeds a certain **breakdown threshold** V_{br} (typically 5 to 20 volts), one of two phenomena occurs (which phenomena sets in first depends on various particular details of the diode).
    - In **Zener breakdown**, this electric field directly breaks some of the covalent bonds in the semiconductor crystal, thus allowing the resulting freed electrons to act as charge carriers.
    - In **avalanche breakdown**, on the other hand, the electric field accelerates free valence electrons near the edge of the depletion layer to sufficient energies that their subsequent collision with bound electrons can break covalent bonds within the depletion layer, resulting in the creation of additional charge carriers (pairs of free electrons and holes), which in turn collide with other bound electrons within the depletion layer to create still more charge carriers, etc.

  Note that avalanche breakdown is **hysteretic** (that is, after it sets in and the additional charge carriers are created within the depletion layer, the semiconductor continues to conduct even after the voltage is reduced below the breakdown threshold), whereas Zener breakdown is not. Diodes designed to undergo these types of breakdown at specific voltages without being damaged, called **Zener diodes** and **avalanche diodes**, are both useful in electric circuit design.
- **Diffusion** of charge carriers (electrons and holes) across the p-n junction in a diode sets up a small depletion zone and a corresponding **built-in voltage** even when the external voltage applied to the diode is zero. Thus, under forward bias, the applied voltage must exceed a certain **cut-in voltage** V_{d} ($\sim$0.65 volts for silicon diodes, $\sim$0.2 volts for germanium diodes) before current will begin to flow.
- Due to a weak thermodynamic process of **carrier generation and recombination** inside the depletion layer, a small **leakage current** always flows through a diode under reverse bias even when the applied voltage is below the breakdown threshold. Note in particular that
    - **carrier generation** due to the absorption of energy of incident photons, and the resulting current when under forward bias, is how **photodiodes** measure the intensity of incident light, whereas
    - energy release as photons during **carrier recombination** is how **light-emitting diodes** (**LED**s) work.
- Finally, under both forward bias (with the applied voltage exceeding the cut-in voltage described above) and reverse bias (with the applied voltage exceeding the breakdown threshold described above), a diode exhibits a nonzero but very small amount of electrical **resistance**.

The current-voltage relationship of a real diode is summed up in Figure 19.7. A regular diode is denoted by the symbol  , and a Zener diode by  , with the arrow pointing in the direction of the current when under forward bias. Real semiconductor diodes are usually small cylinders with a wire out each end, with the *n*-doped end marked with a single bar, consistent with the bar at the end of the diode symbol.

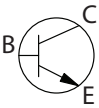
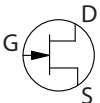
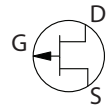
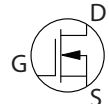
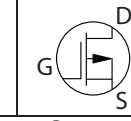
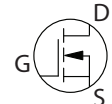
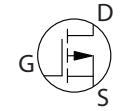
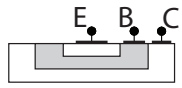
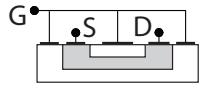
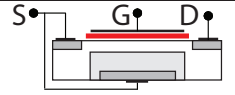
Bipolar Junction Transistor (BJT)		Field-Effect Transistor (FET)					
		Junction FET (JFET) (depletion mode)		Insulated-Gate FET (IGFET)			
				depletion mode		enhancement mode	
p-n-p type	n-p-n type	n-channel	p-channel	n-channel	p-channel	n-channel	p-channel
							
							

Table 19.1: The eight main types of transistors, their symbols, and their essential construction features. The three nodes of a BJT are denoted the **base** (B), **emitter** (E), and **collector** (C), whereas the corresponding nodes of an FET are denoted the **gate** (G), **source** (S), and **drain** (D).

Transistors

A semiconductor crystal designed to behave as an **amplifier** or an **electronically-activated switch** is called a **transistor**, and is built from three or more adjacent doping regions. As shown in Table 19.1, there are eight main types of transistors, which are all somewhat similar in their application, though they differ considerably in their physical construction and internal operation.

A **bipolar junction transistor** (**BJT**) is the most robust and common type of transistor. A BJT is, in effect, two oppositely-facing diodes placed back-to-back in a single semiconductor crystal. Thus, if the middle section of a BJT, called the **base**, is left unconnected, then (in the ideal setting, assuming no breakdown) there will be zero net current between the two ends of the BJT (called the **emitter** and the **collector**). However, if a small current is initiated between the base and the emitter, this populates the central region of the transistor (including the depletion zone in the p-n junction between the base and the collector, which is nominally under reverse bias) with charge carriers, thus causing a much larger current, proportional to the base current, to flow between the emitter and the collector. As suggested by their respective names and symbols,

- in a p-n-p type BJT, the emitter-base connection is a p-n diode nominally under forward bias, whereas
- in an n-p-n type BJT, the base-emitter connection is a p-n diode nominally under forward bias;

we also denote the voltages and the magnitude of the currents of the emitter, base, and collector as, respectively, $\{V_E, V_B, V_C\}$ and $\{I_E, I_B, I_C\}$; note that $I_E = I_B + I_C$ in both p-n-p and n-p-n type BJTs. Assuming the voltage differences are sufficiently small that avalanche breakdown does not set in, the four modes of operation of a p-n-p transistor are as follows (the n-p-n case is similar, with all polarities reversed):

- **Forward active** or "**linear**" mode: $V_E > V_B \geq V_C$. This is the nominal setting in which the transistor acts as a current amplifier. The **current gain** from I_B to I_C in this mode is denoted h_{FE} or β_F, and is typically about $h_{FE} = \beta_F = I_C/I_B \approx 100$, whereas the ratio I_C/I_E is denoted α_F, and is typically about $\alpha_F = I_C/I_E = \beta_F/(1+\beta_F) \approx 0.99$.
- **Saturation** or "**on**" mode: $V_E > V_B$ and $V_C > V_B$. Both p-n junctions are forward biased; current flows freely, limited by resisters elsewhere in the circuit.
- **Cutoff** mode: $V_B > V_E$ and $V_B > V_C$. Both p-n junctions are reverse biased; very little current flows.
- **Reverse active** or "**backwards**" mode: $V_C \geq V_B > V_E$. This is what happens when you install a transistor backwards. As indicated Table 19.1, the physical construction of a BJT is *not* symmetric (notwithstanding introductory explanations of how a BJT functions that might indicate to the contrary); in particular, the surface area of the p-n junction between the base and the collector is much larger than the surface area of the p-n junction between the base and the collector. As a consequence, the current gain of a BJT in reverse active mode is typically quite poor; this mode is thus to be avoided.

Noting that I_B is typically much smaller than I_C, the power dissipated by a BJT is nearly $P = |V_E - V_C| I_C$. A transistor operating in the linear region is not power efficient: typical values of $|V_E - V_C|$ in low-power applications might be several volts, and typical values of i_C might be hundreds of milliamps, leading to over a watt of power dissipated by the (typically, diminutive) BJT when operating in linear mode, thus sometimes necessitating a heat sink. On the other hand, when used as a switch (in saturation and cutoff modes), very little power is dissipated by the transistor (typical values are 0.2 V at 100 mA $\approx$ 20 mW in saturation mode, and 10V at 50nA $\approx 0.5\ \mu$W in cutoff mode). Thus,

Guideline 19.1 *For power-efficient operation of transistors, use them as fast switches rather than amplifiers.*

As mentioned above, the simplest model of a transistor in forward active mode is as a current amplifier,

$$I_C = \beta_F I_B, \quad I_C = \alpha_F I_E \quad \text{with} \quad \alpha_F = \beta_F/(1+\beta_F) \tag{19.6a}$$

and the **current gain** $\beta_F = h_{FE}$ taken as a (large) constant. The more accurate **Early model** of forward active mode models β_F as a function of the magnitude of the collector-emitter voltage $V_{CE} = |V_C - V_E|$ such that

$$\beta_F = \beta_{F0}\,(1 + V_{CE}/V_A) \tag{19.6b}$$

in (19.6a), where the constants β_{F0} and V_A are referred to as the **current gain at zero bias** and the **Early voltage**, respectively. This model may be extended to also incorporate the base-emitter voltage $V_{BE} = |V_B - V_E|$ such that

$$I_C = I_{ES}\,(e^{V_{BE}/V_T} - 1)\,(1 + V_{CE}/V_A), \tag{19.6c}$$

where the constants I_{ES} and V_T are referred to as the **reverse saturation current** and the **thermal voltage**, respectively. Typical constant values are $I_{ES} = 10^{-15}$ to 10^{-12} amps, $V_T = 26$ mV, and $V_A = 15$ V to 150 V.

In a manner analogous to BJTs, the main flow of current in a **Field-Effect Transistor** (**FET**), between the **source** and the **drain**, is regulated by the voltage of the **gate**. The main distinction between an FET and a BJT is that the drain-source current of an FET is regulated by the *voltage* of the gate, whereas the emitter-collector current of a BJT is regulated by the *current* through the base. FETs come in two main types, Junction FETs and Insulated-Gate Semiconductor FETs.

A **Junction Field-Effect Transistor** (**JFET**) is a (usually, symmetric) transistor design in which the source and drain are connected to the two ends of a single semiconductor channel that is a either n-doped or p-doped. As indicated Table 19.1, adjacent to the channel are oppositely-doped semiconductor regions connected to the gate. If the gate of the JFET is left disconnected, the JFET readily conducts current from the source to the drain or the drain to the source. However, if a voltage is applied to the gate of the appropriate sign such that the p-n junctions along the edge of the channel are reverse-biased, a depletion zone forms in the channel which diminishes the amount of current the JFET can conduct between the source and the drain. Increasing the magnitude of this voltage applied to the gate increases the size of this depletion zone, which further diminishes the current the JFET can conduct between the source and the drain, until a **pinch-off** level is reached, in which the current the JFET can conduct between the source and the drain is essentially zero.

In an **Insulated-Gate Field Effect Transistor** (**IGFET**), the gate is *electrically insulated* from the channel carrying the current between the source and the drain. The most common type of IGFET, in which the gate insulation (indicated in red in Table 19.1) is a metal oxide, is known as a **Metal Oxide Semiconductor Field-Effect Transistor** (**MOSFET**). Due to the insulation of the gate, an IGFET has a *very* high input impedance, and almost zero current flows through the gate. This makes IGFETs particularly efficient in logic circuits or as fast switches; however, it also makes them susceptible to damage from static electricity. IGFETs come in two classes, those that work in a **depletion mode** similar to that of a JFET as described above, and those that work in an **enhancement mode**, in which the channel between the source and the drain is generally nonconducting (it may even be undoped) until an appropriate voltage is applied at the gate, which populates the channel between the source and drain with charge carriers, thus allowing current to flow.

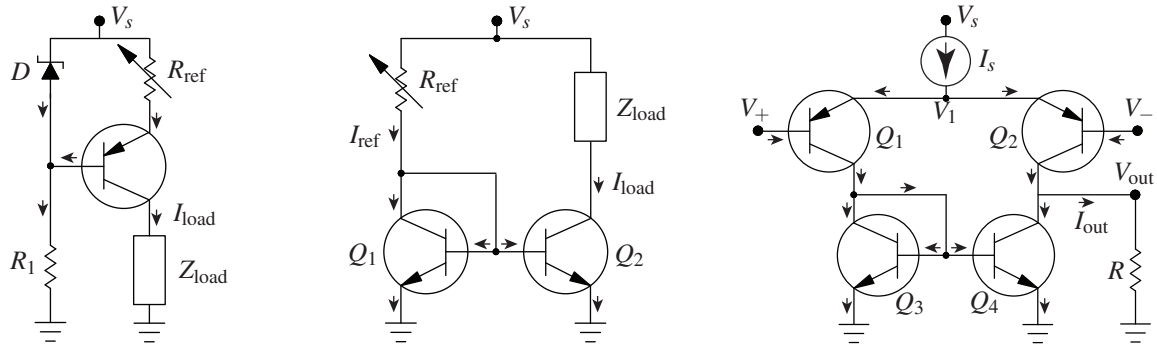


Figure 19.8: Simple transistor circuits. (a) A **current source** based on a p-n-p BJT and a Zener diode, with $I_{\text{load}} \approx (V_{\text{br}} - V_{\text{EB}})/R_{\text{ref}}$. (b) A **current mirror** based on two n-p-n BJTs, with $I_{\text{load}} \approx I_{\text{ref}} = (V_s - V_{\text{BE}})/R_{\text{ref}}$. (c) A **differential amplifier** based on four BJTs and a current source, as in (a), with $V_{\text{out}} \approx D(V_+ - V_-)$ where $D = I_{\text{ES}}\,R/V_{\text{T}}$; note that the lower half of this circuit is exactly the current mirror considered in (b).

For more operational details of JFETs and IGFETs/MOSFETs, the reader is referred to the literature. We now demonstrate the versatility of transistor-based circuits by presenting just a few (of the *many*!) useful circuits that may be built from BJTs and other fundamental circuit elements.

Example 19.5 Current source. Consider the circuit illustrated in Figure 19.8a. As discussed previously (see Figure 19.7), a Zener diode under a sufficiently large reverse bias has an essentially constant voltage drop across it, V_{br}, regardless of the current flowing through it (the resistor R_1 limits this Zener diode current). In this circuit, the Zener breakdown voltage is also applied across the series connection of the resistor R_{ref} and the emitter-base terminals of the transistor; the voltage across resister R_{ref} is thus given by $V_{\text{br}} - V_{\text{EB}}$, where V_{EB} is the voltage drop between the emitter and base of the transistor (about 0.65V for silicon). The emitter current of the transistor is thus given by $I_{\text{E}} = (V_{\text{br}} - V_{\text{EB}})/R_{\text{ref}}$; since V_{br} and V_{EB} are approximately constant, I_{E} is approximately constant. Finally, since a BJT acts as a current amplifier with $I_{\text{B}} = h_{\text{FE}}\,I_{\text{C}}$ where $h_{\text{FE}} \geq 100$, it follows that $I_{\text{E}} \approx I_{\text{C}} = I_{\text{load}}$ regardless of the precise values of V_s and h_{FE}, provided they are sufficiently large, and regardless of the precise values of R_1 and $|Z_{\text{load}}|$, provided they are sufficiently small.

Example 19.6 Current mirror. Consider the circuit illustrated in Figure 19.8b. Assuming $V_{\text{CE}} \ll V_{\text{A}}$ and thus $\beta_{\text{F}} \approx \beta_{\text{F0}}$ in (19.6b), which is often a good assumption, it follows from (19.6c) that I_{C} of transistor Q_1 is related (exponentially) to V_{BE} such that $I_{\text{C}} = I_{\text{S}}\,e^{V_{\text{BE}}/V_{\text{T}}}$. Since the base-emitter voltage of the (matched) transistors Q_1 and Q_2 are precisely equal in this circuit, their collector currents are equal as well. Finally, since the base currents are negligible compared to the collector currents, it follows from KCL that $I_{\text{load}} \approx I_{\text{ref}}$.

Example 19.7 Differential amplifier. Consider the circuit illustrated in Figure 19.8c. Let $\{V_{\text{E}_k}, V_{\text{B}_k}, V_{\text{C}_k}\}$ and $\{I_{\text{E}_k}, I_{\text{B}_k}, I_{\text{C}_k}\}$ denote the voltage and current of the emitter, base, and collector, respectively, of transistor Q_k, with positive current in the directions indicated in the figure. Due to the current mirror in the lower half of the circuit (see Example 19.6 and Figure 19.8b), $I_{\text{C}_1} \approx I_{\text{C}_4}$. Taking $V_{\text{CE}}/V_{\text{A}} \ll 1$ in (19.6c), it follows that

$$\left.\begin{aligned} I_{\text{C}_1} &= I_{\text{ES}}\,(e^{(V_1 - V_+)/V_{\text{T}}} - 1) = \alpha_F\,I_{\text{E}_1} \\ I_{\text{C}_2} &= I_{\text{ES}}\,(e^{(V_1 - V_-)/V_{\text{T}}} - 1) = \alpha_F\,I_{\text{E}_2} \\ I_s &= I_{\text{E}_1} + I_{\text{E}_2} \\ I_{\text{out}} &= I_{\text{C}_2} - I_{\text{C}_4} \approx I_{\text{C}_2} - I_{\text{C}_1} \end{aligned}\right\} \Rightarrow \begin{aligned} I_s &\approx \frac{I_{\text{ES}}}{\alpha_F}\left(\frac{V_1 - V_-}{V_{\text{T}}} + \frac{V_1 - V_+}{V_{\text{T}}}\right) \;\Rightarrow\; V_1 \approx \frac{1}{2}\left[\frac{V_{\text{T}}\,\alpha_F\,I_s}{I_{\text{ES}}} + V_+ + V_-\right], \\ I_{\text{out}} &\approx I_{\text{ES}}\left(\frac{V_1 - V_-}{V_{\text{T}}} - \frac{V_1 - V_+}{V_{\text{T}}}\right) = \frac{I_{\text{ES}}}{V_{\text{T}}}\Big(V_+ - V_-\Big), \\ V_{\text{out}} &= I_{\text{out}}\,R \approx D(V_+ - V_-) \quad \text{with} \quad D = I_{\text{ES}}\,R/V_{\text{T}}. \end{aligned}$$

Note that V_{out} responds primarily to the **differential voltage** $(V_+ - V_-)$, while $V_1 = V_{\text{E}_1} = V_{\text{E}_2}$ "floats" up and down in response to the **common voltage** $(V_+ + V_-)$.

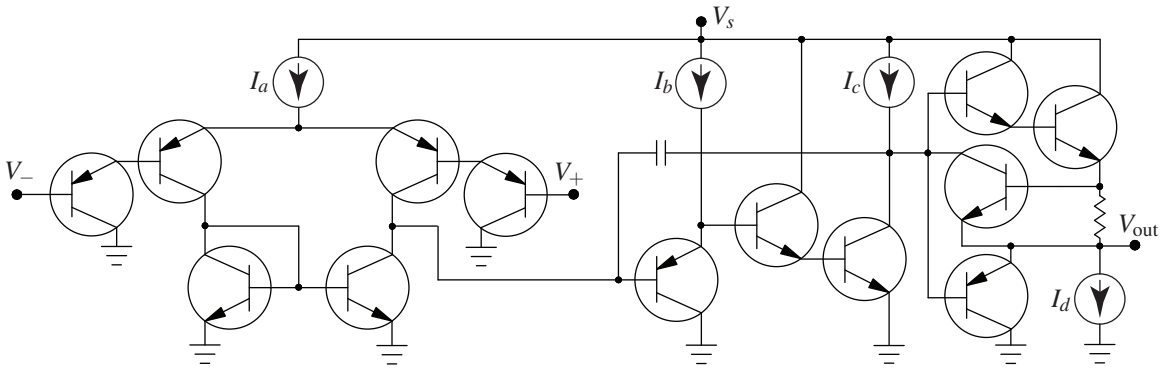


Figure 19.9: Internal construction of an LM324 op amp, with $I_a = I_b = 6\,\mu\text{A}$, $I_c = 100\,\mu\text{A}$, $I_d = 50\,\mu\text{A}$. The first stage of the op amp is the differential amplifier of Figure 19.8c, with **darlington transistors** (that is, a cascade of two transistors, interconnected as shown) used on each of the inputs to increase the gain, and the output resistor replaced by the op amp's second stage. The rest of the circuit amplifies the output from the first stage, implements a first-order low-pass filter to suppress high-frequency noise, and provides high current driving capability with low output impedance and short circuit protection. The LM324 quad op amp implements four such circuits together in a single, robust, and convenient 14-pin **dual in-line package** (**DIP**).

19.2.2 Operational amplifiers

An **operational amplifier** (a.k.a. **op amp**) is an **active** (powered) integrated circuit with two inputs, $V_+(t)$ and $V_-(t)$, and one output, $V_{\text{out}}(t)$, that functions as a differential amplifier with output

$$V_{\text{out}}(t) = \begin{cases} V_{s+} & \text{if } V_{s+} < V_o(t) \\ V_o(t) & \text{if } V_{s-} < V_o(t) < V_{s+} \\ V_{s-} & \text{if } V_o(t) < V_{s-} \end{cases} \quad \text{with} \quad V_o(t) \approx A\,[V_+(t) - V_-(t)], \tag{19.7a}$$

where the gain A is *very* large (indeed, it is often approximated as $A \to \infty$), and two additional properties:

a) very high input impedance (that is, the input terminals of the op amp draw negligible current), and
b) very low output impedance (that is, the output voltage of the op amp is set by the input voltages as specified above, essentially independent of the attached load).

The internal construction of an op amp is a fairly involved arrangement of transistors and other circuit elements, as typified by[17] the LM324 op amp illustrated[18] in Figure 19.9. A more accurate *dynamic* model of $V_o(t)$ in the (typical) LM324 op amp, which takes into account the fact that the magnitude of its frequency response rolls off at a couple hundred kilohertz [cf. (19.7a)], may be written in transfer-function form as

$$V_o(s) = G(s)\,[V_+(s) - V_-(s)] \quad \text{with} \quad G(s) = A\,\frac{a}{s+a}, \tag{19.7b}$$

where $A \approx 10^5$ and $a \approx 10^6$. Note that the low-pass-filter nature of an op amp is usually neglected [see (19.7a)]; that is, the cutoff frequency a is so large that the transfer function $G(s)$ of the op amp is usually approximated as a pure gain (and, further, the gain A of an op amp is so large that it is often considered to be essentially infinite when modeling the behavior of an op amp circuit). However, the more precise model of an op amp given in (19.7b) is the best starting point to understand op amp behavior, as it explains why an op amp with feedback is either stable or unstable (note that both modes have their uses), depending on which input terminal the feedback is connected to, as shown below.

[17] Note that there are many such op amp designs that lead to the same essential properties.
[18] Note that, in Figure 19.9, V_{s+} is denoted V_s, and V_{s-} is denoted by ground.

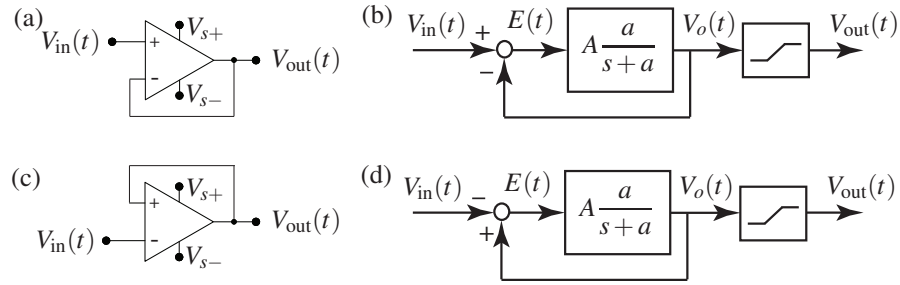


Figure 19.10: Two simple op amp circuits. (a) An op amp wired with negative feedback and (b) its corresponding block diagram. (c) An op amp wired with positive feedback and (d) its corresponding block diagram. As op amps are **active** devices, their connections to V_{s+} & V_{s-} are often shown explicitly in the op amp symbol, as in (a) and (c); these connections are often suppressed for notational simplicity in the remainder of this text.

Consider first the simple op amp circuit in Figure 19.10a, with **negative feedback**, and its corresponding block diagram in Figure 19.10b, with an input-output transfer function of

$$\left.\begin{aligned} V_o(s) &= G(s)E(s) \\ E(s) &= V_{\text{in}}(s) - V_o(s) \end{aligned}\right\} \quad \Rightarrow \quad H(s) = \frac{V_o(s)}{V_{\text{in}}(s)} = \frac{G(s)}{1+G(s)} = \frac{aA}{s+a+aA} \approx \frac{aA}{s+aA}.$$

The gain of this first-order low-pass filter is nearly unity over a wide range of frequencies; note the fast stable pole at $s \approx -aA$. With large A, this circuit behaves as a **voltage follower** or **buffer**, with $V_{\text{out}}(t) \approx V_{\text{in}}(t)$. This active circuit is useful because, due to its high input impedance and low output impedance, it **isolates** the circuits hooked to its input and output terminals; that is, it draws negligible current from the circuit connected to its input terminal, and maintains $V_{\text{out}}(t) \approx V_{\text{in}}(t)$ while providing as much current as required (within limits) by the circuit connected to its output terminal, thus allowing filters to be constructed and analyzed as independent stages then cascaded together, effectively relaxing the restrictive assumptions of Example 19.4.

Now consider the op amp circuit in Figure 19.10c, with **positive feedback**, and its corresponding block diagram in Figure 19.10d, with an input-output transfer function of

$$\left.\begin{aligned} V_o(s) &= G(s)E(s) \\ E(s) &= V_o(s) - V_{\text{in}}(s) \end{aligned}\right\} \quad \Rightarrow \quad H(s) = \frac{V_o(s)}{V_{\text{in}}(s)} = \frac{-G(s)}{1-G(s)} = \frac{-aA}{s+a-aA} \approx \frac{-aA}{s-aA}.$$

Due to the (fast) unstable pole at $s = aA$, *the equilibrium $V_o(t) \approx V_{in}(t)$ is unstable*, and is thus, in practice, never realized. Instead, $V_{\text{out}}(t)$ is driven to one of the limiting values of the op amp, V_{s+} or V_{s-}, and stays there; which limit it goes to depends on the initial values of $V_o(t)$ and $V_{\text{in}}(t)$ when the op amp is turned on.

19.2.3 Design and analysis of a few useful op amp circuits

We now show via several examples how the clever arrangement of transistors in an op amp is convenient in a wide variety of practical situations. Note that *almost all useful op amps circuits implement feedback*, with Example 19.8 being a notable exception. Further, *almost all useful op amps circuits implementing feedback use the (stable)* **negative feedback** *configuration* discussed previously, with Examples 19.11, 19.14, and 19.15 being notable exceptions.

Example 19.8 Voltage comparator. When implemented without feedback, a bare op amp (19.7a) in the large gain limit $A \to \infty$ functions simply as a **voltage comparator**:

$$V_{\text{out}}(t) = \begin{cases} V_{s+} & \text{if } V_+(t) > V_-(t), \\ V_{s-} & \text{if } V_+(t) < V_-(t). \end{cases}$$

Example 19.9 Inverting and noninverting amplifiers. Implementing (stabilizing) feedback to the inverting input of the op amp, an **inverting amplifier** may be implemented as shown in Figure 19.11a, in which

$$V_{\text{in}}(t) - V_-(t) = I_{\text{in}}(t)\,R/M, \quad V_-(t) - V_{\text{out}}(t) = I_R(t)\,R, \quad I_{\text{in}}(t) = I_R(t);$$

applying (19.7b) thus leads to

$$V_{\text{out}}(s) = \frac{aA}{s+a}\,[0 - V_-(s)] \quad \Rightarrow \quad \frac{V_{\text{out}}(s)}{V_{\text{in}}(s)} = \frac{-M\,aA}{(M+1)s + [aA + a(M+1)]} \underset{A\to\infty}{\Longrightarrow} V_{\text{out}}(t) \approx -M\,V_{\text{in}}(t).$$

Similarly, a **noninverting amplifier** may be implemented as shown in Figure 19.11b, in which

$$V_-(t) = I_O(t)\,R/f, \quad V_-(t) - V_{\text{out}}(t) = I_R(t)\,R, \quad I_O(t) = -I_R(t);$$

applying (19.7b) leads to

$$V_{\text{out}}(s) = \frac{aA}{s+a}\,[V_{\text{in}}(s) - V_-(s)] \quad \Rightarrow \quad \frac{V_{\text{out}}(s)}{V_{\text{in}}(s)} = \frac{aA}{s + a + aA/(1+f)} \underset{A\to\infty}{\Longrightarrow} V_{\text{out}}(t) \approx (1+f)\,V_{\text{in}}(t).$$

As illustrated by both of these examples, the (stabilizing) feedback to the inverting input of the op amp leads, in the $A \to \infty$ limit, to the condition that $V_+ = V_-$; note in both cases the very fast stable poles. It often simplifies the analysis of a stable op amp circuit to simply apply the condition $V_+ = V_-$ at the outset; if you have doubts whether or not the circuit considered is stable, implement (19.7b) instead, as done above.

Example 19.10 A general op amp circuit for adding and subtracting. Appropriate combination of the inverting and noninverting amplifiers of Example 19.9 leads to an op amp circuit such that

$$V_{\text{out}}(t) = \sum_{j=1}^{n} m_j\,v_j - \sum_{j=1}^{N} M_j\,V_j, \tag{19.8}$$

that is, to an op amp circuit that can perform an arbitrary linear combination of $n+N$ inputs, with n positive coefficients m_j and N negative coefficients $(-M_j)$. Defining $f = \sum m_j - \sum M_j - 1$, we will consider three cases: $f < 0$, $f = 0$, and $f > 0$. In the sample circuit we will consider, we take $n = N = 3$; the modifications required to handle a different numbers of inputs are trivial. Most op amp circuits used for adding and subtracting are special cases of the general circuit presented here.

The circuit required in the $f < 0$ case is illustrated in Figure 19.11c. For notational clarity, in this example only, we take the voltages, currents, and resistances in the upper half of the circuit as uppercase, and the voltages, currents, and resistances in the lower half of the circuit as lowercase. Ohm's law and KCL then give

$$\begin{array}{llll} V_1 - V_- = I_1\,R/M_1, & V_2 - V_- = I_2\,R/M_2, & V_3 - V_- = I_3\,R/M_3, & V_- - V_{\text{out}} = I_R\,R, \quad I_1 + I_2 + I_3 = I_R, \\ v_a - v_+ = i_a\,r/m_a, & v_b - v_+ = i_b\,r/m_b, & v_c - v_+ = i_c\,r/m_c, & v_+ = i_o\,r/|f|, \quad i_a + i_b + i_c = i_o. \end{array}$$

Since negative (stable) feedback is used, assuming $A \to \infty$, we take $V_- = v_+$; noting that $f = \sum m_j - \sum M_j - 1$ and solving then leads immediately to (19.8).

As $f \to 0$, the resistance of the connection between the noninverting input of the op amp and ground in Figure 19.11c goes to infinity. In the $f = 0$ case, this connection may thus be eliminated entirely; removing the equation $v_+ = i_o\,r/|f|$ from the above set of equations, taking $i_o = 0$, and solving leads again to (19.8).

Finally, in the $f > 0$ case, we replace the connection between the noninverting input of the op amp and ground with a connection between the inverting input of the op amp and ground, with resistance R/f. In this case, the equation $v_+ = i_o\,r/|f|$ in the above set of equations is replaced by $V_- = I_o\,R/f$, and the two KCL relations are now $I_1 + I_2 + I_3 = I_o + I_R$ and $i_a + i_b + i_c = 0$; solving again leads to (19.8).

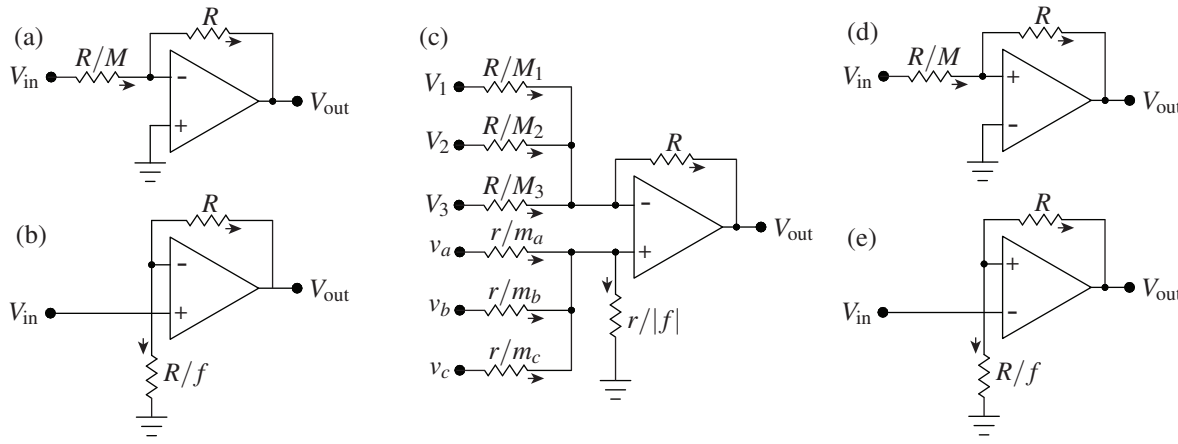


Figure 19.11: Some useful op amp circuits. (a) Inverting amplifier. (b) Noninverting amplifier. (c) A general adder/subtractor [note: the ground connection shown is for $f < 0$; if $f = 0$, this connection to ground is removed; if $f > 0$, the connection to ground is attached to V_- instead of V_+, through a resistance R/f]. (d) Inverting Schmitt trigger. (e) Noninverting Schmitt trigger. Note that (d) and (e) are hysteretic.

In all three cases, $f < 0$, $f = 0$, and $f > 0$, the resulting relation between the voltages, (19.8), is in fact *independent* of both R and r, which are typically selected so that all resistors used in the circuit are between 1 kΩ and 100 kΩ. Note that, in the case that $n = 0$ (see, e.g., Figure 19.11a), we may take $r = 0$, wiring the noninverting input of the op amp directly to ground. In the case that $n = 1$ and $f \geq 0$ (see, e.g., Figure 19.11b), we may also take $r = 0$, wiring the noninverting input of the op amp directly to v_a.

Example 19.11 Schmitt triggers. We now consider two **hysteretic** circuits that are simply the inverting and noninverting amplifiers of Example 19.9 with the inputs to the op amp swapped from the stable (negative-feedback) configuration to the unstable (positive-feedback) configuration. Assuming $V_{s+} = V_s$ and $V_{s-} = -V_s$,

- in the unstable circuit illustrated in Figure 19.11d, called a **noninverting Schmitt trigger**,
  - if $V_{\text{out}} = +V_s$, then it will stay there until V_{in} passes below $-V_s/M$ (that is, until V_+ passes below V_-), after which the output will switch to $V_{\text{out}} = -V_s$, whereas
  - if $V_{\text{out}} = -V_s$, then it will stay there until V_{in} passes above V_s/M (that is, until V_+ passes above V_-), after which the output will switch to $V_{\text{out}} = +V_s$;
- in the unstable circuit illustrated in Figure 19.11e, called a **inverting Schmitt trigger**,
  - if $V_{\text{out}} = +V_s$, then it will stay there until V_{in} passes above $V_s/(1+f)$ (that is, until V_- passes above V_+), after which the output will switch to $V_{\text{out}} = -V_s$, whereas
  - if $V_{\text{out}} = -V_s$, then it will stay there until V_{in} passes below $-V_s/(1+f)$ (that is, until V_- passes below V_+), after which the output will switch to $V_{\text{out}} = +V_s$.

A primary application of Schmitt triggers is **switch debouncing**: once a remotely-operated Schmitt trigger, acting as a switch, is flipped one way, it takes a large change in the input to flip the Schmitt trigger the other way, thus preventing "chatter" of the switch due to noise over the communication channel.

Example 19.12 A general-purpose inverting first-order filter. The circuit illustrated in Figure 19.12a is a remarkably flexible general-purpose inverting first-order filter design with transfer function[19]

$$F(s) = \frac{V_{\text{out}}(s)}{V_{\text{in}}(s)} = -\frac{C_1}{C_2}\frac{s+1/(R_1C_1)}{s+1/(R_2C_2)} = -\frac{R_2}{R_1}\frac{1+R_1C_1s}{1+R_2C_2s} = -R_2C_1\frac{s+1/(R_1C_1)}{1+R_2C_2s} = -\frac{1}{R_1C_2}\frac{1+R_1C_1s}{s+1/(R_2C_2)}.$$

[19] This transfer function is easily derived from the corresponding circuit via the techniques used in Examples 19.9 and 19.10.

That is, $F(s) = -K_0(s+z)/(s+p)$, where $K_0 = C_1/C_2$, $z = 1/(R_1C_1)$, and $p = 1/(R_2C_2)$; we also define $K_1 = R_2/R_1$, $K_2 = R_2C_1$, and $K_3 = 1/(R_1C_2)$. If the op amp is ideal, the circuit design in Figure 19.12a is actually nine circuits in one, reducing[20] in the appropriate limits to *all* of the inverting first-order filters:

- taking $R_1C_1 > R_2C_2$, it is an inverting **lead filter** with $F(s) = -K_0(s+z)/(s+p)$ where $z < p$;
- taking $R_2C_2 > R_1C_1$, it is an inverting **lag filter** with $F(s) = -K_0(s+z)/(s+p)$ where $p < z$;
- removing C_1, it is an inverting **first-order low-pass filter** with $F(s) = -K_3/(s+p)$;
- removing R_1, it is an inverting **first-order high-pass filter** with $F(s) = -K_0\,s/(s+p)$;
- removing[21] R_2, it is an inverting **PI filter** with $F(s) = -K_0\,(s+z)/s$;
- removing[21] R_2 and C_1, it is an inverting **pure integrator** $F(s) = -K_3/s$;
- removing C_2, it is an inverting **PD filter**[22] with $F(s) = -K_2\,(s+z)$;
- removing C_2 and R_1, it is an inverting **pure differentiator**[22] $F(s) = -K_2\,s$;
- removing C_1 and C_2, it is an inverting **amplifier** $F(s) = -K_1$.

The development of a corresponding general-purpose *noninverting* first-order filter is considered in Exercise 19.6. To build a second-order filter that incorporates both a first-order lag filter at low frequencies (to reduce steady-state error) and a first-order lead filter at high frequencies (to improve damping and reduce overshoot), creating what is called a **lead/lag filter** (see Figure 18.3.2b), one may simply cascade together the lead and lag filters described above as necessary. Note that a **PID filter** is simply a special case of a lead/lag filter with

- the roll-off of the integral action of the lag filter taken all the way down to $\omega \to 0$, and
- the roll-off of the derivative action of the lead filter taken all the way up to $\omega \to \infty$.

To build a PID filter, one could simply cascade together the PI and PD filters described above. However, note that lead/lag filters are strongly preferred over PID filters for the reasons discussed in §18.3.1: that is, the roll-off of the low-frequency gain and the high-frequency gain mentioned above almost never need to be taken all the way to zero and infinity respectively, and doing such generally causes significant problems (specifically, *integrator windup* and the *amplification of high-frequency noise*) in the closed-loop setting.

Example 19.13 Notch filter. The circuit illustrated in Figure 19.12a, called an **active twin-T filter**, is a convenient op amp circuit implementation of the **notch** transfer function[19]

$$F_{\text{notch}}(s) = \frac{V_{\text{out}}(s)}{V_{\text{in}}(s)} = K\,\frac{s^2+\omega_o^2}{s^2+(\omega_o/Q)s+\omega_o^2} \tag{19.9}$$

where[23] $K = 1 + R_2/R_1$, $Q = 1/[2(2-K)]$, and $\omega_o = 1/(RC)$. Note that the zeros of the notch are pure imaginary, and the so-called "**quality**" of the notch is given by $Q = 1/(2\zeta)$, where ζ is the damping of its poles; thus, $Q = 0.5$ (that is, $K = 1$, or $R_2 = 0$ and $R_1 \to \infty$) corresponds to two identical real poles, and $Q > 0.5$ corresponds to a pair of complex-conjugate poles with damping which decreases as Q is increased.

The Bode plot of $F_{\text{notch}}(s)$ for $\omega_o = 1$ and two different values of Q is illustrated in Figure 19.13; note that the gain of the notch filter is zero at ω_o, and that the width of the range of frequencies significantly affected by the notch decreases with increasing Q. When using a notch to eliminate, e.g., a 50 or 60 Hz "buzz" (that is, noise with a very narrow power spectrum) in a signal, a high Q value is used to minimize the impact of the notch on the signal of interest outside of the range of frequencies corrupted by the buzz. However, when using a notch in a feedback control setting to "knock out" the oscillatory dynamics of a plant (see §18.3.2),

[20] Note that removing a capacitor corresponds to taking its capacitance $C \to 0$, and removing a resistor corresponds to taking its resistance $R \to \infty$; in both cases, by (19.2), the current goes to zero through the (removed) component regardless of the applied voltage.

[21] Alternatively, R_2 may be replaced by a switch, allowing the integrator to be reset whenever desired.

[22] PD filters and pure differentiators must never be used in practice, because they amplify high-frequency noise without bound, which is a significant problem. *Lead filters* and *first-order high-pass filters* (a.k.a. **dirty differentiators**) should be used instead.

[23] Note that Q and K can not be set independently in this particular circuit; this usually does not create much of issue, however, because a notch filter is usually cascaded with other op amp circuits that can be used to set the gain to the desired value.

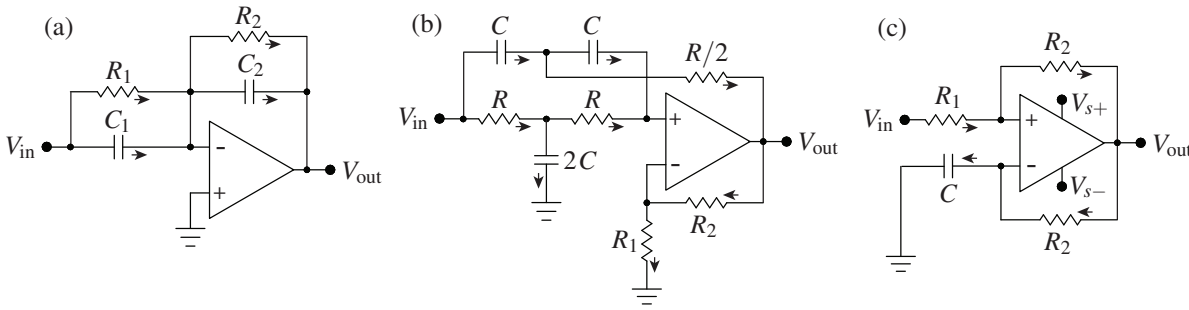


Figure 19.12: Some *dynamic* op amp circuits: (a) an **inverting first-order filter** $F(s) = -K(s+z)/(s+p)$, which may be simplified in various ways; (b) a **notch filter** $F_{\text{notch}}(s) = K(s^2+\omega_o^2)/(s^2+\omega_o s/Q+\omega_o^2)$; and (c) a circuit which, taking $V_{\text{in}} = 0$, $V_{s+} = +V_s$, $V_{s-} = -V_s$, and $R_1 = R_2 = R$, gives a square-wave **relaxation oscillator** with frequency $1/(2RC\ln 3)$ Hz and duty cycle $1/2$, whereas, taking $V_{s+} = V_s$, $V_{s-} = 0$, and $R_1 \ll R_2$, gives a square wave with duty cycle V_{in}/V_s and frequency $V_{\text{in}}(V_s - V_{\text{in}})/(R_1 C V_s^2)$ Hz, which may be used to drive a load in an efficient partial power setting via a **pulse width modulation** strategy.

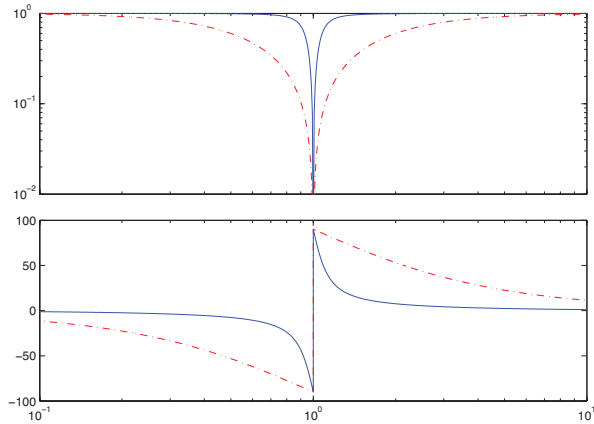


Figure 19.13: Bode plot of the **notch filter** of Example 19.13, rescaled such that $K = 1$, taking $\omega_o = 1$, (dot-dashed) $Q = 0.5$, and (solid) $Q = 5$. Note the "notch" shape of the magnitude part of the Bode plot.

one certainly does *not* want to introduce lightly damped poles with the notch, and values of Q in the range $0.5 \le Q \le 0.707$ are preferred[24]. Note finally that an active twin-T implementation of a notch filter may be cascaded with a doubled lead filter to move the poles resulting from the notch even further into the LHP.

Example 19.14 Relaxation oscillator. The operation of the **relaxation oscillator** circuit depicted in Figure 19.12c is analogous to the operation of the Schmitt triggers considered previously, with the successive charging and discharging of a capacitor leading to the periodic flipping of the switch. Assume that $V_{\text{in}} = 0$, $V_{s+} = +V_s$, $V_{s-} = -V_s$, $R_1 = R_2 = R$, and that V_+ is initially higher than V_-. Given this initial state,

(a) $V_{\text{out}} = V_s$, and thus $V_+ = V_s/2$, as the two upper resistors act as a voltage divider. Current thus flows from the output of the op amp through the lower resistor and the capacitor to ground, charging the capacitor until V_- just exceeds $V_+ = V_s/2$, and the switch flips to state (b).

(b) $V_{\text{out}} = -V_s$, and thus $V_+ = -V_s/2$. Current thus flows from ground through the capacitor and the lower resistor to the output of the op amp, charging the capacitor the other direction until V_- falls just below $V_+ = -V_s/2$, and the switch flips back to state (a).

[24] That is, when implementing a notch filter to stabilize, e.g., a Ford automobile, achieving high quality is *not* necessarily job one.

The period of this oscillation is constant, and may be calculated by determining how long it takes the capacitor of the relaxation oscillator to gather sufficient charge to flip the switch in each state. For example, taking $V_{\text{out}} = V_s$ and $V_-(0) = -V_s/2$, the dynamics of state (a) are governed by

$$C\frac{d}{dt}(V_- - 0) = \frac{V_s - V_-}{R} \quad \Rightarrow \quad \left(1 + RC\frac{d}{dt}\right)V_- = V_s.$$

The homogeneous solution of this system is $V_-(t) = Ae^{-t/RC}$, and a particular solution is $V_-(t) = V_s$. Combining and matching the initial condition $V_-(0) = -V_s/2$, the full solution is given by

$$V_-(t) = V_s - (3V_s/2)e^{-t/(RC)}.$$

Setting $V_-(t)$ equal to $V_s/2$ at $t = T/2$, when the switch flips to state (b), it is easy to compute the period T:

$$V_-(T/2) = V_s - (3V_s/2)\,e^{-T/(2RC)} = V_s/2 \quad \Rightarrow \quad e^{-T/(2RC)} = 1/3 \quad \Rightarrow \quad T = 2RC\ln 3.$$

Example 19.15 Efficient power control of purely resistive loads via pulse width modulation. Given a power supply (e.g., a battery) and a purely resistive load (e.g., a light bulb), a question arises as to the most effective way to run the load at partial power. An inefficient solution to this problem is simply to put a **variable resistor** (a.k.a. **rheostat**) in series with the load, thus reducing both the current through the load and the voltage across the load, thereby reducing the power consumed by the load. Unfortunately, the variable resistor used in such a setting itself consumes a lot of power that is rejected as waste heat, thereby wiping out any potential energy savings that might otherwise be realized.

A much more efficient way to regulate the power applied to a purely resistive load is to *repeatedly cycle the voltage applied to the load on and off very quickly*; the percentage of the time the switch is on, called the **duty cycle**, then regulates the (time-averaged) percentage of full power at which the load will operate. This solution, referred to as **pulse width modulation** (**PWM**), is facilitated by the fact that transistors are quite efficient when operated as fast switches (see Guideline 19.1).

It is instructive to note that the simple op amp circuit considered in Example 19.14 can in fact be put to task quite easily in the PWM setting. The component relations and KCL in the upper and lower portions of the circuit lead, respectively, to

$$\frac{V_{\text{in}} - V_+}{R_1} = \frac{V_+ - V_{\text{out}}}{R_2} \quad \Rightarrow \quad V_+ = \frac{R_2 V_{\text{in}} + R_1 V_{\text{out}}}{R_1 + R_2}, \tag{19.10a}$$

$$C\frac{d}{dt}(V_- - 0) = \frac{V_{\text{out}} - V_-}{R_2} \quad \Rightarrow \quad \left(1 + R_2 C\frac{d}{dt}\right)V_- = V_{\text{out}}. \tag{19.10b}$$

Taking $V_{s+} = V_s$, $V_{s-} = 0$, and $R_1 \ll R_2$ and defining the limits $V_{+,\max} = (R_2 V_{\text{in}} + R_1 V_s)/(R_1 + R_2)$ and $V_{+,\min} = R_2 V_{\text{in}}/(R_1 + R_2)$, it follows, as in Example 19.14, that there are two states to consider:

(a) $V_{\text{out}} = V_s$, and thus $V_+ = V_{+,\max}$. Starting (19.10b) from an initial condition of $V_- = V_{+,\min}$ at $t = 0$, current flows from the output of the op amp through the lower resistor and the capacitor to ground, charging the capacitor until V_- just exceeds V_+; the switch thus flips to state (b) at time $t = T_1$, which may be computed as follows:

$$V_-(t) = V_s + (V_{+,\min} - V_s)e^{-t/(R_2C)}, \quad V_-(T_1) = V_{+,\max} \quad \Rightarrow$$

$$e^{T_1/(R_2C)} = \frac{V_{+,\min} - V_s}{V_{+,\max} - V_s} = 1 + \frac{R_1}{R_2}\frac{V_s}{(V_s - V_{\text{in}})} \quad \Rightarrow \quad T_1 = R_2 C\ln\left(1 + \frac{R_1}{R_2}\frac{V_s}{V_s - V_{\text{in}}}\right) \approx R_1 C\frac{V_s}{V_s - V_{\text{in}}}.$$

(b) $V_{\text{out}} = 0$, and thus $V_+ = V_{+,\min}$. Starting (19.10b) from an initial condition of $V_- = V_{+,\max}$ at $t = 0$ (resetting the time variable t appropriately to simplify the analysis), current flows from ground through

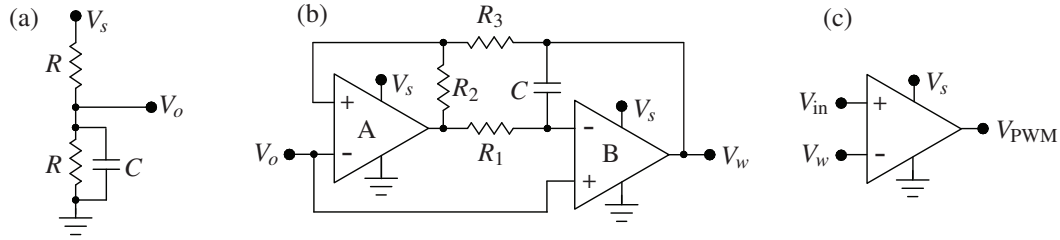


Figure 19.14: An improved PWM circuit with duty cycle V_{in}/V_s and frequency $1/(4R_1C)$ Hz, formed by the cascade of (a) a passive **stabilized voltage divider**, (b) a **triangle-wave generator**, and (c) a **comparator**.

the capacitor and the lower resistor to the output of the op amp, discharging the capacitor until V_- falls just below V_+; the switch thus flips back to state (a) at $t = T_2$, which may be computed as follows:

$$V_-(t) = 0 + (V_{+,\max} - 0)e^{-t/(R_2C)}, \quad V_-(T_2) = V_{+,\min} \quad \Rightarrow$$
$$e^{T_2/(R_2C)} = \frac{V_{+,\max}}{V_{+,\min}} = 1 + \frac{R_1}{R_2}\frac{V_s}{V_{\text{in}}} \quad \Rightarrow \quad T_2 = R_2C\ln\left(1 + \frac{R_1}{R_2}\frac{V_s}{V_{\text{in}}}\right) \approx R_1C\frac{V_s}{V_{\text{in}}}.$$

Thus, the square wave oscillation is cyclically "on" ($V_{\text{out}} = V_s$) for a period T_1 and then "off" ($V_{\text{out}} = 0$) for a period T_2, with duty cycle D and frequency ω given by

$$D \triangleq \frac{T_1}{T_1 + T_2} \approx \frac{V_{\text{in}}}{V_s}, \quad \omega = \frac{1}{T_1 + T_2} \approx \frac{V_{\text{in}}(V_s - V_{\text{in}})}{R_1CV_s^2} \text{ Hz}.$$

Unfortunately, the frequency of the simple oscillator described above varies with V_{in}, with $\omega \to 0$ as $V_{\text{in}}/V_s \to 0$ and as $V_{\text{in}}/V_s \to 1$. A significantly improved PWM circuit, which operates at a constant (independent of V_{in}) frequency $\omega = 1/(4R_3C)$ Hz, is given by cascading together the three stages shown in Figure 19.14. The first stage (Figure 19.14a) is a passive voltage divider with $V_o = V_s/2$, with a capacitor added to stabilize the output voltage in case the (small) load attached to its output fluctuates; values of $R \sim 10$ kΩ and $C \sim 100$ nF are typical. The second stage (Figure 19.14b) generates a triangle wave V_w between 0 and V_s and operating at a frequency of $\omega = 1/(4R_1C)$ Hz, as discussed below. Finally, the third stage (Figure 19.14c) compares the triangle wave V_w to V_{in}, outputting V_s whenever V_{in} is greater than V_w, and outputting 0 whenever V_{in} is less than V_w, thus resulting in a duty cycle of V_{in}/V_s.

To analyze the operation of the triangle-wave generator of Figure 19.14b, denote the inputs and outputs of op amp A (configured in an unstable configuration with positive feedback) as $\{V_{\text{A},+}, V_{\text{A},-} = V_o, V_{\text{A,out}}\}$, and denote the inputs and outputs of op amp B (configured in a stable configuration with negative feedback) as $\{V_{\text{B},+} = V_o, V_{\text{B},-}, V_{\text{B,out}} = V_w\}$. Note that, since op amp B is wired with (stabilizing) negative feedback, $V_{\text{B,out}}$ adjusts so that $V_{\text{B},-} = V_{\text{B},+} = V_o = V_s/2$ at all times. Note also that resistors R_2 and R_3 form another voltage divider so that, taking[25] $R_2 \approx R_3$, it follows that $V_{\text{A},+} = (V_{\text{A,out}} + V_{\text{B,out}})/2$ at all times. As in the oscillator circuits considered previously, there are two states to consider:

(a) $V_{\text{A,out}} = 0$. Assuming the capacitor is initially charged such that $V_{\text{B,out}} = 0$, current flows from the output of op amp B through C and R_1 to the output of op amp A, charging the capacitor until $V_{\text{B,out}} = V_s$ and thus $V_{\text{A},+}$ just[25] exceeds $V_{\text{A},-} = V_s/2$, and op amp A flips to state (b).

(b) $V_{\text{A,out}} = V_s$. Assuming the capacitor is initially charged such that $V_{\text{B,out}} = V_s$, current flows from the output of op amp A through R_1 and C to the output of op amp B, charging the capacitor until $V_{\text{B,out}} = 0$ and thus $V_{\text{A},+}$ falls just[25] below $V_{\text{A},-} = V_s/2$, and op amp A flips back to state (a).

[25] Note that R_3 should actually be chosen to be just slightly smaller than R_2, so that $V_{\text{A},+} = (0.5 - \varepsilon)V_{\text{A,out}} + (0.5 + \varepsilon)V_{\text{B,out}}$, and the states indeed flip as described. This can be achieved by selecting two resistors of the same rated resistance and, say, 5% variance, then carefully measuring the resistance of the two and and putting the one with smaller resistance in the R_3 location.

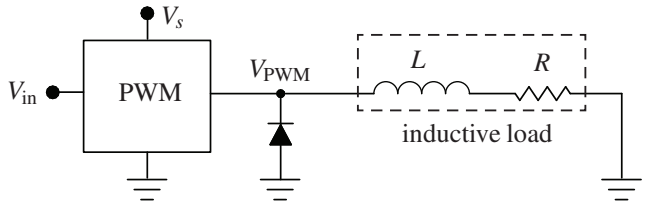


Figure 19.15: Application of a PWM circuit, formed by cascading the three stages of Figure 19.14, to an inductive load, incorporating a protective **flyback diode** (as suggested in Example 19.16) to prevent large negative voltage spikes forming at V_{PWM} (and, possibly, an arc between exposed wires or damage to one of the op amps) when the PWM circuit acts to quickly turn off the power to the energized inductive load.

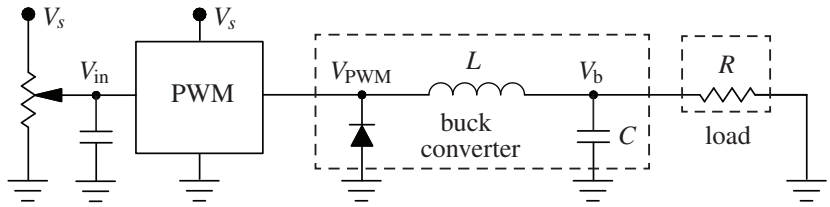


Figure 19.16: Application of a **buck regulator** for DC/DC conversion. A variable voltage divider (stabilized by a capacitor) may be used to generate an input V_{in} to a PWM circuit, such as that given by cascading the three stages of Figure 19.14. The output of the PWM is then buffered with an inductor, capacitor, and flyback diode, which together act as a **second-order low-pass filter** which passively and efficiently removes the square wave from the PWM signal *without the losses associated with extra resistive elements*.

The period of this oscillation is constant, and may be calculated by determining how long it takes the capacitor of the relaxation oscillator to gather sufficient charge to flip the switch in each state. For example, taking $V_{\text{A,out}} = 0$ and $V_{\text{B,out}}(0) = 0$, the dynamics of state (a) are governed by

$$C\frac{d}{dt}(V_{\text{B,out}} - V_{\text{B,}-}) = \frac{V_{\text{B,}-} - V_{\text{A,out}}}{R_1} \quad \Rightarrow \quad R_1 C\frac{d}{dt}V_{\text{B,out}} = V_{\text{B,}-} - V_{\text{A,out}} \quad \Rightarrow$$

$$V_{\text{B,out}}(t) = \frac{1}{R_1 C}(V_{\text{B,}-} - V_{\text{A,out}})t, \quad V_{\text{B,out}}(T/2) = V_s \quad \Rightarrow \quad T = 4R_1 C.$$

Example 19.16 Efficient power control of inductive loads via pulse width modulation. The PWM strategy for driving loads at partial power, as described above, is highly efficient and remarkably inexpensive to implement with modern electronics. If applied to a load with inductance, however, a problem is encountered. The PWM effectively acts as a switch, quickly turning on and off the power to (and, thus, the current through) the attached load. If the load contains an inductor governed by $V = L\,dI/dt$ [see (19.2c)], a rapid reduction in current through the inductor would tend to induce a large negative voltage spike at V_{PWM}. The diode used in the circuit illustrated in Figure 19.15, called a **flyback** (a.k.a. **snubber**, **freewheeling**, or **suppressor**) diode, ensures (by providing current from ground when necessary) that V_{PWM} does not drop below $-V_{\text{d}}$, where V_{d} is the (small) cut-in voltage of the diode. *A good PWM circuit should always have such a flyback diode incorporated, just in case the load attached to the PWM circuit has inductive elements.*

Example 19.17 DC-DC voltage conversion using a buck converter. The circuit developed in Example 19.16 is one step away from the circuit illustrated in Figure 19.16, called a **buck converter**, which inexpensively and efficiently solves the problem of DC-DC voltage step-down common in computers and electromechanical systems. Assuming a square wave at V_{PWM}, with current flowing from the PWM circuit when

$V_{\text{PWM}} = V_s$ and current flowing from ground (through the diode) when $V_{\text{PWM}} \approx 0$, and assuming (for the purpose of analysis) that the load is essentially resistive with some resistance R, V_b is determined as follows:

$$V_{\text{PWM}} - V_\text{b} = L\frac{dI_L}{dt}, \quad C\frac{dV_\text{b}}{dt} = I_C, \quad V_\text{b} = I_R R, \quad I_L = I_C + I_R \quad \Rightarrow \quad \frac{V_\text{b}(s)}{V_{\text{PWM}}(s)} = \frac{1/(LC)}{s^2 + s/(RC) + 1/(LC)}.$$

Thus, regardless of the precise value of R, if L and C are selected to be large enough that $\omega^2 \gg 1/LC$, where ω is the frequency of square wave output by the PWM, then the fundamental and higher harmonics comprising the square wave will all be damped by the second-order low-pass filter action of the buck converter, leaving only the average value of the PWM signal, which as derived previously is simply V_{in}, which is easily set by the user. Note also that, if a precise value is required for V_{in}, an appropriately-selected zener diode may be used instead of the lower half of the resistor in the first stage of the circuit illustrated in Figure 19.16.

Example 19.18 Brushed DC motor control. Brushed DC motors, which are both remarkably inexpensive and remarkably efficient for converting electrical power to mechanical (rotatary) power, do not operate effectively at low voltage due to stiction within the motor. They also have non-negligible inductance, because they contain coils of wires acting as electromagnets. The PWM strategy with flyback protection developed in Example 19.16 is thus especially useful for driving such motors at partial power.

A representative model of the voltage and torque balances, respectively, of a brushed DC motor are

$$V = RI + L\,dI/dt + K\omega, \tag{19.11a}$$
$$\tau = KI = J\,d\omega/dt + C\,\text{sgn}(\omega), \tag{19.11b}$$

where the last term in (19.11b) models dry friction (see Example 17.3), and where

- $V(t)$ is the **voltage** applied to the motor [averaged over each PWM cycle, and measured in volts],
- $I(t)$ is the **current** through the motor [measured in amps],
- $\omega(t)$ is the **rate of rotation** of the motor shaft [measured in rad/s],
- $\tau(t)$ is the **torque** applied by the motor to the mechanical load [measured in N·m],
- R is the motor **resistance** [measured in ohms],
- L is the motor **inductance** [measured in henries],
- K is the **torque constant** of the motor [measured in N·m/A = V/(rad/s)],
- J is the **rotational interia** of both the motor and its load [measured in N·m/(rad/s^2)], and
- C is the **dry friction coefficient** [measured in N·m; see (17.11c)].

Note that the torque constant K appears in both the computation of the torque generated by the motor, Ki, in (19.11b), as well as the **back emf** in the electric circuit, $K\omega$, in (19.11a).

In order to drive a brushed DC motor in both the forward and reverse directions, while still incorporating a PWM strategy as suggested above to efficiently drive the motor at partial power, an **H-bridge** circuit may be used. An example of such a circuit is given in Figure 19.17, which takes as input a PWM signal and two logic states, FWD and REV. In the example H-bridge circuit shown, all three of these logic states are electrically isolated[26] from the power electronics of the H-bridge via **optoïsolators**, which as indicated by their symbol in the schematic are simply light-emitting diodes (LEDs) packaged in close proximity with photodiodes. The **H-bridge** circuit shown in Figure 19.17 operates in four different modes based on the settings of the FWD and REV logic states:

[26] Implementing the H-bridge in this fashion protects the microcontroller used to generate the logic states from the voltage spikes that are sometimes produced by the power electronics hooked to a DC motor, which is an inductive load. The four flyback diodes protecting the transistors (Q_1 through Q_4) in Figure 19.17 go a long way towards minimizing such voltage spikes, but since these diodes take a finite amount of time to turn on, such voltage spikes still occur; **high-speed diodes**, which turn on quickly, are thus desired in such applications.

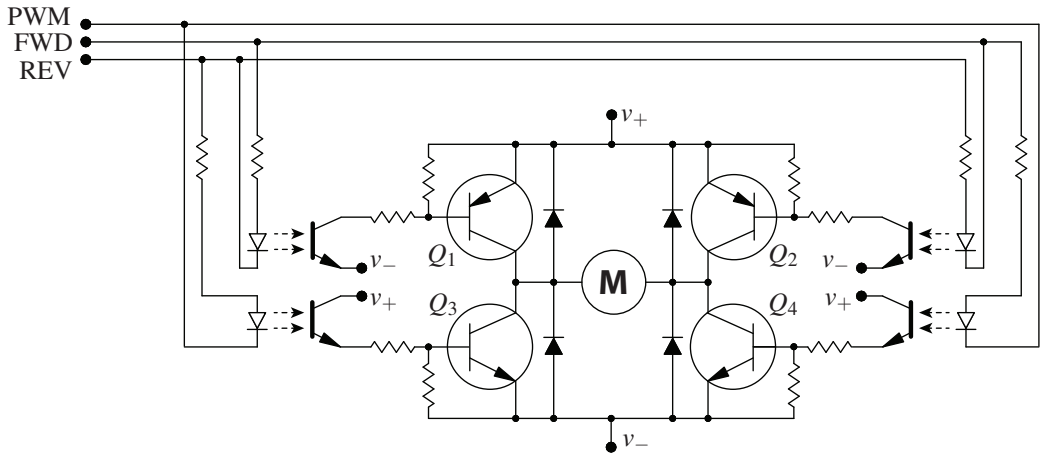


Figure 19.17: Implementation of an H-bridge for efficient bidirectional operation of a brushed DC motor via a PWM signal and two logic states, FWD and REV.

- **Driving forward** (FWD= 1, REV= 0). In this mode, Q_1 is on, Q_4 is on the same percentage of time that the PWM signal is low, and Q_2 and Q_3 are off. DC power thus provided from left to right across the motor at a duty cycle set by the PWM.
- **Driving reverse** (FWD= 0, REV= 1). In this mode, Q_2 is on, Q_3 is on the same percentage of time that the PWM signal is low, and Q_1 and Q_4 are off. DC power thus provided from right to left across the motor at a duty cycle set by the PWM.
- **Braking** (FWD= 1, REV= 1). In this mode, Q_3 and Q_4 are on the same percentage of time that the PWM signal is low, and Q_1 and Q_2 are off. DC power thus provided to the motor, at a duty cycle set by the PWM, that is equal and opposite to the back emf generated by the turning of the motor, resulting in a total of zero volts across the motor terminals, thus causing the motor to slow down.
- **Coasting** (FWD= 0, REV= 0). In this mode, Q_1, Q_2, Q_3 and Q_4 are all off, regardless of the PWM signal. No extra torque is generated by the motor.

In practice, integrated circuits implementing an entire H-bridge circuit are often convenient, such as the Toshiba TB6612FNG dual motor driver, which incorporates MOSFETs instead of BJTs to achieve high efficiency.

Exercises

Exercise 19.1 Combine the equations in (19.4) to compute V_o/I_o by hand, thus verifying (19.3). Then, modifying the Matlab code following (19.4) appropriately [making $\{V_o, R1, R2, R3, R4, R5\}$ symbolic variables], compute V_o/I_o symbolically. (Given the simplicity of the latter and the complexity of the former, the reader is encouraged to use symbolic manipulation to crank through such tedious algebraic computations at every opportunity!)

Exercise 19.2 Following an analogous derivation as that in Example 19.2, given one inductor L_3 of known inductance, quantify how a Wheatstone bridge may be used to measure precisely the inductance of an unknown inductor L_4.

Exercise 19.3 Following an analogous derivation as that in Example 19.3, compute the power provided or absorbed by the voltage and current sources of Figure 19.5 without making the assumption that $R_1 = I_L = 0$. Discuss.

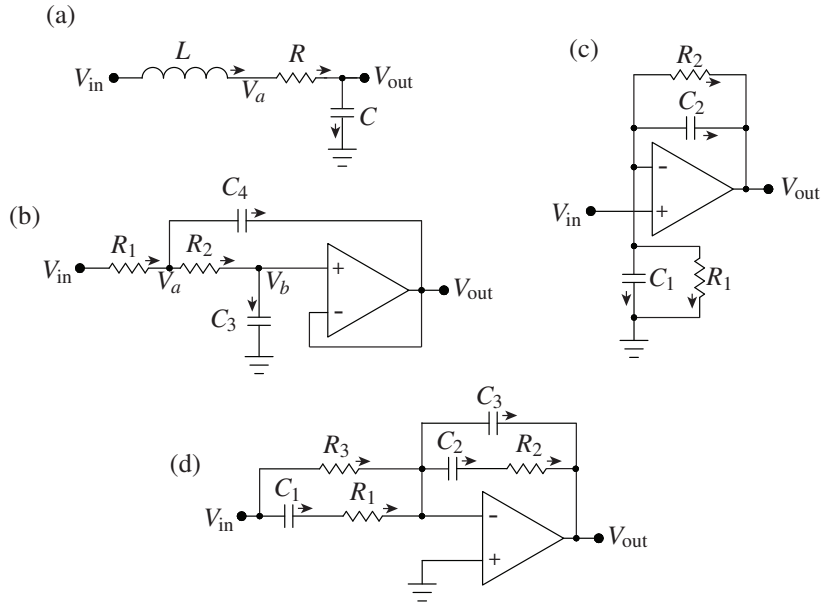


Figure 19.18: Some dynamic filters considered in the exercises. (a) A passive second-order low-pass filter. (b) An active second-order low-pass filter. (c) A general-purpose noninverting first-order filter. (d) A single op-amp implementation of a PID filter combined with two first-order low-pass filters.

Exercise 19.4 (a) Compute the transfer function $V_{out}(s)/V_{in}(s)$ of the passive circuit shown in Figure 19.18a, making the same two assumptions as in Example 19.4. Assuming $LC = 1$ and $RC = 0.5$, plot its Bode plot. What is the cutoff frequency and damping of this filter?
(b) Compute the transfer function $V_{out}(s)/V_{in}(s)$ of the active circuit shown in Figure 19.18b, again making the same two assumptions as in Example 19.4. Assuming $R_1 = 10$ kΩ and that the op amp is ideal with amplification $A \to \infty$, what values of $\{R_2, C_1, C_2\}$ result in the same frequency response as the passive filter considered in part (a)? Assuming all of the components are readily available (they are!), what advantages does a circuit of the type considered in part (b) have over the circuit considered in part (a)?
(c) Note that a fourth-order low-pass filter may be constructed simply by cascading together two second-order active low-pass filters of the type considered in part (b). Following the development in §17.5.2.1, design a fourth-order low-pass Butterworth filter and a fourth-order low-pass Bessel filter, both with $\omega_c = 100$ Hz. Specify the resistor and capacitor values used in each design.
(d) In the active circuit considered in part (b), replace the resistors with capacitors and the capacitors with resistors. Compute the corresponding transfer function and discuss.

Exercise 19.5 Example 19.12 developed a flexible general-purpose *inverting* first-order filter. Sketch the Bode plots of the nine filters that the circuit in Example 19.12 reduces to in the nine special cases enumerated. Then, develop a similarly flexible *noninverting* first-order filter, which is a bit more involved. Start by performing a careful analysis of the circuit in Figure 19.18c, and describe which of the nine cases itemized in the inverting case (Example 19.12) are realizable with this noninverting circuit. Identify precisely what limitations (if any) are present in each case. Reviewing the three special cases considered in Example 19.10, describe precisely how the limitations of this present circuit may be circumvented by reconnecting it appropriately.

Exercise 19.6 The circuit in Figure 19.18d (cf. Figure 19.12a) has a transfer function of

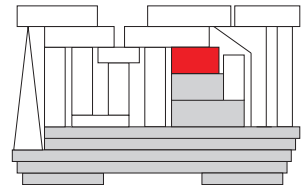
$$\frac{V_{out}(s)}{V_{in}(s)} = -K\frac{(s/z_1+1)(s/z_2+1)}{s}\cdot\frac{1}{(s/p_1+1)(s/p_2+1)},$$

and thus may be interpreted as an inverting PID filter combined with two first-order low-pass filters. Determine how each of the variables $\{K, z_1, z_2, p_1, p_2\}$ depend on $\{R_1, R_2, R_3, C_1, C_2, C_3\}$ (show your work). Then, assuming R_3 is set to some nominal value (say, 1 kΩ), solve for $\{R_1, R_2, C_1, C_2, C_3\}$ in terms of $\{K, z_1, z_2, p_1, p_2\}$.

References

Carter, B, & Mancini, R (2009) *Op Amps For Everyone*. Texas Instruments.

Chapter 20



Linear systems: state-space methods

Contents

20.1 State-space forms . . . 616
20.1.1 State transformations . . . 617
20.1.2 The solution of continuous-time linear systems in state-space form . . . 617
20.1.3 The solution of discrete-time linear systems in state-space form . . . 621
20.1.4 Subsystems . . . 625
20.1.5 MIMO transfer functions and the resolvent . . . 625
20.2 Characterizing solutions of nonorthogonal & defective systems . . . 628
20.2.1 Quantifying the maximum transient energy growth of a stable system . . . 631
20.2.2 Quantifying the transfer function of a stable system via system norms . . . 631
20.2.2.1 Computation of the transfer function 2-norm and ∞-norm of CT and DT systems† . . . 634
20.3 Canonical forms . . . 636
20.3.1 The four continuous-time canonical forms . . . 636
20.3.2 The six discrete-time canonical forms . . . 638
20.3.3 Markov parameters . . . 644
20.4 Feedback design via pole placement . . . 645
20.4.1 Controller design . . . 646
20.4.2 Observer design . . . 646
20.5 Controllability, observability, and related concepts . . . 648
20.5.1 Continuous-time controllability . . . 649
20.5.1.1 The continuous-time controllability matrix . . . 649
20.5.1.2 The continuous-time controllability gramian . . . 650
20.5.2 Continuous-time observability . . . 652
20.5.2.1 The continuous-time observability matrix . . . 652
20.5.2.2 The continuous-time observability gramian . . . 653
20.5.3 Discrete-time reachability and controllability . . . 654
20.5.3.1 A weaker condition: steering a discrete-time system to the origin . . . 656

20.5.4 Discrete-time observability and constructability . . . 657
20.5.4.1 A weaker condition: determining the final state only . . . 659
20.5.5 Output tracking in LTI systems . . . 660
20.6 Model reduction . . . 662
20.6.1 Removing uncontrollable and unobservable modes: minimal realizations . . . 662
20.6.2 Removing modes with poor controllability/observability: balanced truncation . . . 663
Exercises . . . 667

Before beginning this chapter, the reader is advised to read the introduction to §17, to put various related controls-oriented concepts in context, and to review specifically the transform-based techniques for the analysis of linear systems, as discussed in the remainder of §17, as well as the several linear-algebraic facts and decompositions laid out in §4, upon which the present chapter builds directly.

20.1 State-space forms

The essence of the subject of linear systems is the thorough characterization of continuous-time (**CT**) linear systems that may be written in the **continuous-time state-space form**

$$\left.\begin{aligned} \mathbf{x}'(t) &= A(t)\mathbf{x}(t) + B(t)\mathbf{u}(t) \\ \mathbf{y}(t) &= C(t)\mathbf{x}(t) + D(t)\mathbf{u}(t) \end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{c|c} A(t) & B(t) \\ \hline C(t) & D(t) \end{array}\right] \tag{20.1}$$

and discrete-time (**DT**) linear systems that may be written in the **discrete-time state-space form**

$$\left.\begin{aligned} \mathbf{x}_{k+1} &= F_k\mathbf{x}_k + G_k\mathbf{u}_k \\ \mathbf{y}_k &= H_k\mathbf{x}_k + D_k\mathbf{u}_k \end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{c|c} F_k & G_k \\ \hline H_k & D_k \end{array}\right], \tag{20.2}$$

denoting $\mathbf{x}_k \triangleq \mathbf{x}(t_k)$ where $t_k = hk$ for $k = 0,1,2,\ldots$ We will consider both cases in which $\{A,B,C,D\}$ in (20.1) and $\{F,G,H,D\}$ in (20.2) do *not* vary in time, referred to as **linear time-invariant** (**LTI**) systems, as well as cases in which $\{A(t),B(t),C(t),D(t)\}$ in (20.1) and $\{F_k,G_k,H_k,D_k\}$ in (20.2) do vary in time, referred to as **linear time-varying** (**LTV**) systems. Note the shorthand notations depicted above right (and implemented in `ShowSys.m` in the *NRC*), which are sometimes convenient to summarize compactly the four matrices defining a state-space form. The matrix A in the CT case, and the matrix F in the DT case, are often referred to as the **system matrices** of the corresponding state-space forms. The analyses of the CT and DT cases, when considered properly, are at each step analogous to one another. To highlight their close relationship, they are thus laid out in parallel in the sections that follow; to obtain a complete understanding, the reader is encouraged to master both cases as well as their interrelationship.

The direct feedthrough from the input $\mathbf{u}$ to the output $\mathbf{y}$, present when $D \neq 0$, causes difficulty in some of the analyses. We thus, at times, restrict our attention to the special case with $D = 0$ [which, as shown in (20.34), corresponds to a **strictly proper** transfer function]; note that such systems may always be formed by a simple change of variables, taking $\overline{\mathbf{y}} = \mathbf{y} - D\mathbf{u}$.

Algorithm 20.1: Perform a state transformation on a MIMO state-space model.

View Test

```
function [A,B,C]=SSTransform(A,B,C,R)
Ri=Inv(R); A=Ri*A*R; B=Ri*B; C=C*R;
end % function SSTransform
```

20.1.1 State transformations

The state-space realizations given in the CT case in (20.1) and in the DT case in (20.2) are not unique. Given one such realization, a different, equivalent state-space realization may be constructed via any nonsingular state transformation matrix R. Defining $\mathbf{x} = R\mathbf{x}_R$ for any nonsingular R and multiplying the first equation of (20.1) by R^{-1} allows us to perform a **state transformation** that reëxpresses (20.1) as

$$\left.\begin{aligned} \mathbf{x}_R'(t) &= A_R\mathbf{x}_R(t) + B_R\mathbf{u}(t) \\ \mathbf{y}(t) &= C_R\mathbf{x}_R(t) + D_R\mathbf{u}(t) \end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{c|c} A_R & B_R \\ \hline C_R & D_R \end{array}\right] = \left[\begin{array}{c|c} R^{-1}AR & R^{-1}B \\ \hline CR & D \end{array}\right]; \tag{20.3}$$

that is, taking $\mathbf{x}_R = R^{-1}\mathbf{x}$, with $A_R = R^{-1}AR$, $B_R = R^{-1}B$, $C_R = CR$, and $D_R = D$ (see Algorithm 20.1). An identical state transformation procedure may be applied to DT systems of the form (20.2).

Fact 20.1 *The eigenvalues of the system matrix are invariant under any state transformation* (20.3).

Proof: Since $|\lambda I - A_R| = |\lambda R^{-1}R - R^{-1}AR| = |R^{-1}|\,|\lambda I - A|\,|R| = |\lambda I - A|$, any eigenvalue of A_R is also an eigenvalue of A. □

20.1.2 The solution of continuous-time linear systems in state-space form

The first-order scalar SISO LTI case

Consider first the CT first-order scalar SISO LTI system given by

$$x'(t) = \lambda x(t) + bu(t). \tag{20.4}$$

[Note that, with $u = 0$, (20.4) is sometimes called the **continuous-time model problem**.] If $x(0)$ and $u(t)$ for $t \geq 0$ are specified, then the solution to this system is given by

$$x(t) = e^{\lambda t}x(0) + \int_0^t e^{\lambda(t-\tau)}bu(\tau)\,d\tau \quad \text{where} \quad e^{\lambda t} \triangleq 1 + \lambda t + \frac{(\lambda t)^2}{2!} + \frac{(\lambda t)^3}{3!} + \ldots, \tag{20.5}$$

as easily verified by substitution. Decomposing λ into its real and imaginary parts [that is, taking $\lambda = \lambda_R + \mathrm{i}\,\lambda_I$ with $\mathrm{i} = \sqrt{-1}$, where λ_R and λ_I denote the real and imaginary components of λ] and noting that $|e^{\mathrm{i}\,\lambda_I t}| = 1$, it follows that, if $u = 0$, then $x(t) = e^{\lambda t}x(0) = e^{\lambda_R t}e^{i\lambda_I t}x(0) = e^{\lambda_R t}[\cos(\lambda_I t) + \mathrm{i}\sin(\lambda_I t)]x(0)$, and thus

- $|x(t)|$ decays exponentially with t if $\lambda_R < 0$ (in which case the system is said to be **stable**),
- $|x(t)|$ is constant in t if $\lambda_R = 0$ (in which case the system is said to be **neutrally stable**), and
- $|x(t)|$ grows exponentially with t if $\lambda_R > 0$ (in which case the system is said to be **unstable**).

To recap, the region of stability of the exact solution to the CT model problem is the LHP of the complex plane λ, as denoted by the shaded region in Figure 20.1.

Vector LTI systems and the matrix exponential

Now consider the CT first-order vector LTI system given by

$$\mathbf{x}'(t) = A\mathbf{x}(t) + B\mathbf{u}(t). \tag{20.6}$$

The solution of this system is given by the natural generalization of (20.5),

$$\mathbf{x}(t) = e^{At}\mathbf{x}(0) + \int_0^t e^{A(t-\tau)}B\mathbf{u}(\tau)\,d\tau \quad \text{where} \quad e^{At} \triangleq I + At + \frac{(At)^2}{2!} + \frac{(At)^3}{3!} + \ldots, \tag{20.7}$$

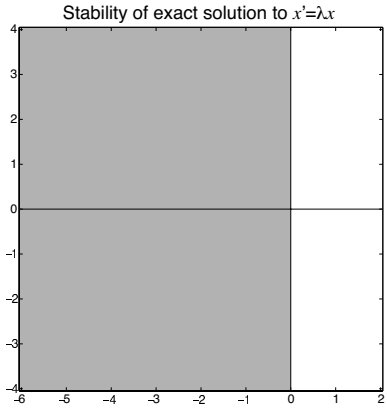


Figure 20.1: Stability of the exact solution to the model problem $x' = \lambda x$ in the complex plane λ.

as easily verified by substitution. The expression e^{At} is known as the **matrix exponential**. The triangle inequality (see §1.3) together with the submultiplicative property of matrix norms (see Fact 1.12) establishes that the series defining the matrix exponential converges:

$$\left\| \sum_{i=k}^{\infty} \frac{(At)^i}{i!} \right\|^2 \leq \sum_{i=k}^{\infty} \frac{\|(At)^i\|^2}{i!} \leq \sum_{i=k}^{\infty} \frac{\|(At)\|^i}{i!} \triangleq \varepsilon_k.$$

The expression on the right bounds the magnitude of the sum of all terms $i \geq k$ on the RHS of the expansion defining e^{At} in (20.7), and may be made arbitrarily small for sufficiently large k for any value of t.

Note that, by the Cayley-Hamilton theorem (Fact 4.15), A^p for $p \geq n$ may be expressed as a linear combination of lower powers of A. Thus, (20.7) may be written as a sum of a finite number of terms,

$$e^{At} \triangleq \alpha_0(t)I + \alpha_1(t)A + \alpha_2(t)A^2 + \alpha_3(t)A^3 + \ldots + \alpha_{n-1}(t)A^{n-1}; \tag{20.8}$$

the appropriate values of the coefficients $\alpha_i(t)$ in this expansion are derived in §20.1.5.

In the special case that the matrix A is diagonal, the matrix exponential defined above reduces to

$$\Lambda = \begin{pmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_n \end{pmatrix} \quad \Rightarrow \quad e^{\Lambda t} = I + \Lambda t + \frac{\Lambda^2 t^2}{2!} + \frac{\Lambda^3 t^3}{3!} + \ldots = \begin{pmatrix} e^{\lambda_1 t} & & 0 \\ & \ddots & \\ 0 & & e^{\lambda_n t} \end{pmatrix}. \tag{20.9}$$

In the special case that the matrix A is in Jordan form (see §4.4.6), the matrix exponential also reduces to a fairly simple form. For example,

$$J = \begin{pmatrix} \lambda_1 & 1 & & 0 \\ & \lambda_1 & 1 & \\ & & \lambda_1 & \\ 0 & & & \lambda_2 \end{pmatrix} \quad \Rightarrow \quad J^2 = \begin{pmatrix} \lambda_1^2 & 2\lambda_1 & 1 & 0 \\ & \lambda_1^2 & 2\lambda_1 & \\ & & \lambda_1^2 & \\ 0 & & & \lambda_2^2 \end{pmatrix}, \quad J^3 = \begin{pmatrix} \lambda_1^3 & 3\lambda_1^2 & 3\lambda_1 & 0 \\ & \lambda_1^3 & 3\lambda_1^2 & \\ & & \lambda_1^3 & \\ 0 & & & \lambda_2^3 \end{pmatrix}, \quad \ldots$$

$$\Rightarrow \quad e^{Jt} = I + Jt + \frac{J^2 t^2}{2!} + \frac{J^3 t^3}{3!} + \ldots = \begin{pmatrix} e^{\lambda_1 t} & te^{\lambda_1 t} & \frac{t^2}{2} e^{\lambda_1 t} & 0 \\ & e^{\lambda_1 t} & te^{\lambda_1 t} & \\ & & e^{\lambda_1 t} & \\ 0 & & & e^{\lambda_2 t} \end{pmatrix}. \tag{20.10}$$

The matrix exponential for other matrices in Jordan form follow analogously, noting that, if J_i is an $m \times m$ Jordan block with λ_i on the main diagonal and 1 on the first superdiagonal, then J_i^n is Toeplitz with 0 on the subdiagonals, λ_i^n on the main diagonal, and, noting (B.37), ${}_nC_k\,\lambda_i^{n-k}$ on the k'th superdiagonal for $k = 1,\dots,\max\{n, m-1\}$, with 0 on the other superdiagonals. The following useful results follow immediately:

Fact 20.2 (Semigroup property of the matrix exponential) $e^{A(t+\tau)} = e^{At}\,e^{A\tau} \quad \Rightarrow \quad e^{At}\,e^{-At} = I \quad \Rightarrow$ $(e^{At})^{-1} = e^{-At}$. *That is, e^{At} is invertible for any A.*

Fact 20.3 $\dfrac{d}{dt}(e^{A(t-\tau)}) = Ae^{A(t-\tau)} = e^{A(t-\tau)}A$. *Similarly,* $\dfrac{d}{d\tau}(e^{A(t-\tau)}) = -Ae^{A(t-\tau)} = -e^{A(t-\tau)}A$.

Modal coördinate form

If the system is LTI and the eigenvectors of A are linearly independent, then we may perform an eigen decomposition $A = S\Lambda S^{-1}$ (see §4.4.4), where Λ is the (diagonal) eigenvalue matrix, S has the corresponding eigenvectors as columns, and S^{-1} has the corresponding left eigenvectors of A as rows (see Fact 4.25). Performing the state transformation (20.3) with $R = S$ thus converts (20.1) to

$$\left.\begin{aligned}\mathbf{x}_m'(t) &= \Lambda\mathbf{x}_m(t) + B_m\mathbf{u}(t)\\ \mathbf{y}(t) &= C_m\mathbf{x}_m(t) + D\mathbf{u}(t)\end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{c|c} \Lambda & B_m \\ \hline C_m & D\end{array}\right] = \left[\begin{array}{c|c} S^{-1}AS & S^{-1}B \\ \hline CS & D\end{array}\right]. \tag{20.11}$$

In this representation, appropriately called **modal coördinate form**, the evolution of each **mode** $x_{m,i}$ of the uncontrolled system is seen to be *completely decoupled*. That is, taking $\mathbf{u} = 0$ for the purpose of illustration,

$$\left.\begin{aligned} x_{m,1}' &= \lambda_1 x_{m,1} \\ x_{m,2}' &= \lambda_2 x_{m,2} \\ &\vdots \end{aligned}\right. \quad \Rightarrow \quad \left.\begin{aligned} x_{m,1}(t) &= e^{\lambda_1 t}x_{m,1}(0) \\ x_{m,2}(t) &= e^{\lambda_2 t}x_{m,2}(0) \\ &\vdots \end{aligned}\right\} \quad \Rightarrow \quad \mathbf{x}_m(t) = e^{\Lambda t}\mathbf{x}_m(0).$$

In the original coördinates, the solution to (20.6) with $\mathbf{u} = 0$ may thus be written

$$\mathbf{x}(t) = S\mathbf{x}_m(t), \quad \text{where} \quad \mathbf{x}_m(t) = e^{\Lambda t}\mathbf{x}_m(0) \quad \text{and} \quad \mathbf{x}_m(0) = S^{-1}\mathbf{x}(0) \quad \Rightarrow \quad \mathbf{x}(t) = Se^{\Lambda t}S^{-1}\mathbf{x}(0). \tag{20.12}$$

This result may be confirmed by inserting the matrix decomposition $A = S\Lambda S^{-1}$ into the RHS of the definition of the matrix exponential in (20.7) and factoring out an S to the left and an S^{-1} to the right, resulting in

$$e^{At} = S\left[I + \Lambda t + \frac{\Lambda^2 t^2}{2!} + \frac{\Lambda^3 t^3}{3!} + \dots\right]S^{-1} = S[e^{\Lambda t}]S^{-1}. \tag{20.13}$$

Thus, the ODE (20.6) with $\mathbf{u} = 0$ is solved by $\mathbf{x}(t) = e^{At}\mathbf{x}(0) = S[e^{\Lambda t}]S^{-1}\mathbf{x}(0)$, consistent with (20.12).

If the system is LTI and the eigenvectors of A are not linearly independent, then an eigen decomposition is not available. In this case, the modal coördinate form is determined from the Jordan decomposition $A = MJM^{-1}$ (see §4.4.6); performing the state transformation (20.3) with $R = M$ converts (20.1) to

$$\left.\begin{aligned}\mathbf{x}_m'(t) &= J\mathbf{x}_m(t) + B_m\mathbf{u}(t)\\ \mathbf{y}(t) &= C_m\mathbf{x}_m(t) + D\mathbf{u}(t)\end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{c|c} J & B_m \\ \hline C_m & D\end{array}\right] = \left[\begin{array}{c|c} M^{-1}AM & M^{-1}B \\ \hline CM & D\end{array}\right]. \tag{20.14}$$

In this case, the evolution of the modes corresponding to each Jordan block are coupled. Taking $\mathbf{u} = 0$ and taking here the Jordan form J depicted in (20.10) for the purpose of illustration, we may write

$$\left.\begin{aligned} x_{m,1}' &= \lambda_1 x_{m,1} + x_{m,2} \\ x_{m,2}' &= \lambda_1 x_{m,2} + x_{m,3} \\ x_{m,3}' &= \lambda_1 x_{m,3} \\ x_{m,4}' &= \lambda_2 x_{m,4} \end{aligned}\right. \quad \Rightarrow \quad \left.\begin{aligned} x_{m,3}(t) &= e^{\lambda_1 t}x_{m,3}(0) \\ x_{m,2}(t) &= e^{\lambda_1 t}x_{m,2}(0) + te^{\lambda_1 t}x_{m,3}(0) \\ x_{m,1}(t) &= e^{\lambda_1 t}x_{m,1}(0) + te^{\lambda_1 t}x_{m,2}(0) + \tfrac{t^2}{2}e^{\lambda_1 t}x_{m,3}(0) \\ x_{m,4}(t) &= e^{\lambda_2 t}x_{m,2}(0) \end{aligned}\right\} \quad \Rightarrow \quad \mathbf{x}_m(t) = e^{Jt}\mathbf{x}_m(0).$$

Note that some modes [$x_{m,1}(t)$ and $x_{m,2}(t)$ in this example] are characterized by **algebraic growth followed by exponential decay**. In the original coördinates, the solution to (20.6) with $\mathbf{u} = 0$ may be written

$$\mathbf{x}(t) = M\mathbf{x}_m(t), \quad \text{where} \quad \mathbf{x}_m(t) = e^{Jt}\mathbf{x}_m(0) \quad \text{and} \quad \mathbf{x}_m(0) = M^{-1}\mathbf{x}(0) \quad \Rightarrow \quad \mathbf{x}(t) = Me^{Jt}M^{-1}\mathbf{x}(0). \tag{20.15}$$

This result may be confirmed by inserting the matrix decomposition $A = MJM^{-1}$ into the RHS of the matrix exponential definition in (20.7) and factoring out an M to the left and an M^{-1} to the right, resulting in

$$e^{At} = M\left[1 + Jt + \frac{J^2t^2}{2!} + \frac{J^3t^3}{3!} + \dots\right]M^{-1} = M[e^{Jt}]M^{-1}. \tag{20.16}$$

Thus, the ODE (20.6) with $\mathbf{u} = 0$ is solved by $\mathbf{x}(t) = e^{At}\mathbf{x}(0) = M[e^{Jt}]M^{-1}\mathbf{x}(0)$, consistent with (20.15).

Both of the above cases lead immediately to the following conclusion:

Fact 20.4 *The uncontrolled CT LTI system* $\mathbf{x}' = A\mathbf{x}$ *is stable if and only if all of the eigenvalues of the* **system matrix** *A are stable (that is, in the LHP), in which case A is said to be* **Hurwitz**.

Further characterization of the solution to CT LTI state-space systems is deferred to §20.2.

Computing the matrix exponential†

Unfortunately, the matrix exponential is, in general, difficult to compute accurately from (20.7) for large t. Note that (20.13) and (20.16) provide simple relations from which the matrix exponential may be determined from accurate eigen or Jordan decompositions; however, such decompositions are difficult to determine accurately. Defining $\Phi(t) = e^{At}$, we may instead write

$$\Phi(t) = I + At\Psi(t), \quad \text{where} \quad \Psi(t) = I + \frac{At}{2!} + \frac{(At)^2}{3!} + \dots + \frac{(At)^k}{(k+1)!} + O(t^{k+1}) \tag{20.17a}$$

$$= I + \frac{At}{2}\left(I + \frac{At}{3}\left(\dots + \frac{At}{k-1}\left(I + \frac{At}{k}\right)\dots\right)\right) + O(t^{k+1}). \tag{20.17b}$$

Computing (20.17b) with the $O(t^{k+1})$ terms truncated off (for sufficiently large k) is equivalent to computing (20.17a) truncated at the same order, but is better behaved numerically, as it involves fewer additions of elements of vastly different magnitudes (which can not be combined accurately using finite-precision arithmetic). Nonetheless, the calculation of e^{At} via truncation of (20.17b) is viable only for relatively small t. For larger t, many terms in the expansion are required for convergence; an effective work-around is to leverage the relation $e^{At} = e^{A(t/2)}\,e^{A(t/2)}$ [that is, $\Phi(t) = \Phi(t/2)\Phi(t/2)$], due to Fact 20.2, and thus

$$I + At\Psi(t) = [I + (At/2)\Psi(t/2)][I + (At/2)\Psi(t/2)] = I + At\Psi(t/2) + (At/2)^2\Psi^2(t/2)$$

$$\Rightarrow \quad At\Psi(t) = At\Psi(t/2) + (At/2)^2\Psi^2(t/2) \quad \Rightarrow \quad \Psi(t) = [I + (At/4)\Psi(t/2)]\Psi(t/2). \tag{20.18}$$

Thus, if t is too large to compute $\Psi(t)$ accurately via (20.17), one may instead calculate $\Psi(t/2)$, then determine $\Psi(t)$ according to (20.18). This idea can be cascaded, calculating $\Psi(t/2^m)$ for an appropriately large value of m using (20.17b), then extrapolating to compute $\Psi(t)$ via repeated application of (20.18). A simple and effective approach, implemented in Algorithm 20.2, is to select an integer $m \geq 0$ such that $\|At/2^m\|_{i1} \leq 1$ [that is, $m \geq c = \log_2 \|At\|_{i1}$], which facilitates convergence of (20.17b) using $k = 10$ to 15 terms.

LTV systems and the CT state transition matrix

Consider now the **linear time varying** (**LTV**) system

$$\mathbf{x}'(t) = A(t)\mathbf{x}(t) + B(t)\mathbf{u}(t).$$

Algorithm 20.2: Compute the matrix exponential using (20.17) and (20.18).

View
Test

```
function [Phi]=MatrixExponential(A,t)
c=log2(norm(A*t,1)); m=max(0,floor(c)+1); ti=t/(2^m); n=length(A); Psi=zeros(n); kmax=15;
Psi=(eye(n)+A*ti/kmax); for k=kmax-1:-1:2, Psi=eye(n)+(A*ti/k)*Psi; end
for j=1:m, Psi=(eye(n)+(A*(ti*2^j)/4)*Psi)*Psi; end, Phi=eye(n)+A*t*Psi;
end % function MatrixExponential
```

The solution of this system is given [cf. (20.7)] by

$$\mathbf{x}(t) = \Phi(t,0)\mathbf{x}(0) + \int_0^t \Phi(t,\tau)B(\tau)\mathbf{u}(\tau)\,d\tau \tag{20.19a}$$

where the **continuous-time state transition matrix** $\Phi(t,\tau)$ is given by the **Peano-Baker** formula

$$\begin{aligned}\Phi(t,\tau) &\triangleq I + \int_\tau^t A(t')dt' + \int_\tau^t A(t')\Big[\int_\tau^{t'} A(t'')dt''\Big]dt' + \int_\tau^t A(t')\Big[\int_\tau^{t'} A(t'')\Big(\int_\tau^{t''} A(t''')dt'''\Big)dt''\Big]dt' + \dots \\ &= I + O(t-\tau),\end{aligned} \tag{20.19b}$$

as easily verified by substitution, noting the identity $\frac{d}{dt}\int_\tau^t f(t')\,dt' = f(t)$. It is trivial to show that, in the case of constant A, the CT state transition matrix $\Phi(t,\tau)$ defined above reduces to the more familiar matrix exponential $e^{A(t-\tau)}$ defined in (20.7). Note also that the following useful results follow immediately:

Fact 20.5 $\dfrac{d}{dt}\Phi(t,\tau) = A(t)\Phi(t,\tau)$ *with* $\Phi(\tau,\tau) = I$.

Fact 20.6 (Semigroup property of the CT state-transition matrix) $\Phi(t,s) = \Phi(t,r)\,\Phi(r,s) = \Phi(r,s)\,\Phi(t,r)$ $\Rightarrow$ $\Phi(t,s)\,\Phi(s,t) = I$ $\Rightarrow$ $(\Phi(t,s))^{-1} = \Phi(s,t)$. *That is,* $\Phi(t,s)$ *is always invertible.*

Note that Fact 20.5 may in fact be used as an alternative to the Peano-Baker formula to *define* the CT state transition matrix mathematically. It also provides a numerically tractable differential equation from which $\Phi(t,\tau)$ may be calculated using the marching techniques described in §10.

Taking $\frac{d}{d\tau}[\Phi(\tau,t)\Phi(t,\tau) = I]$ and applying the above relations, it also follows that:

Fact 20.7 $\dfrac{d}{d\tau}\Phi(t,\tau) = -\Phi(t,\tau)A(\tau)$.

20.1.3 The solution of discrete-time linear systems in state-space form

The first-order scalar SISO LTI case

Consider the DT first-order scalar SISO LTI system given by

$$x_{k+1} = \sigma x_k + b u_k. \tag{20.20}$$

[Note that, with $u_k = 0$, (20.20) is sometimes called the **discrete-time model problem**.] If x_0 and u_k for $k = 0,1,2,\dots$ are specified, then the solution to this system is given by

$$x_k = \sigma^k x_0 + \sum_{k'=0}^{k-1} \sigma^{k-1-k'} b u_{k'},$$

as easily verified by substitution. It follows that, if $u_k = 0$, then

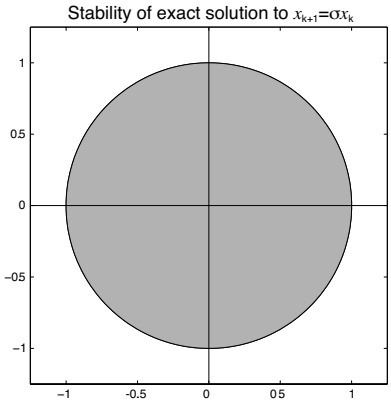


Figure 20.2: Stability of the exact solution to the model problem $x_{k+1} = \sigma x_k$ in the complex plane σ.

- $|x_k|$ decays exponentially with k if $|\sigma| < 1$ (in which case the system is said to be **stable**),
- $|x_k|$ is constant in k if $|\sigma| = 1$ (in which case the system is said to be **neutrally stable**), and
- $|x_k|$ grows exponentially with k if $|\sigma| > 1$ (in which case the system is said to be **unstable**).

To recap, the region of stability of the exact solution to the DT model problem is the interior of the unit circle in the complex plane σ, as denoted by the shaded region in Figure 20.2.

Vector LTI systems

Now consider the DT first-order vector LTI system given by

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + G\mathbf{u}_k. \tag{20.21}$$

The solution to this system is given by

$$\mathbf{x}_k = F^k\mathbf{x}_0 + \sum_{k'=0}^{k-1} F^{k-1-k'}G\mathbf{u}_{k'}, \tag{20.22}$$

as easily verified by substitution.

Exactly converting an LTI CT system to DT

Consider now an LTI CT system (20.1) in which the control input $\mathbf{u}(t)$ is held constant over each timestep of duration h. We may convert this CT problem exactly into the corresponding LTI DT form (20.2) by augmenting the CT matrix representation of the problem and computing the appropriate matrix exponential (20.7). Writing $\mathbf{x}(0) = \mathbf{x}_0$ and $\mathbf{u}(0) = \mathbf{u}_0$ and looking over the first timestep $0 \le t < h$,

$$\left.\begin{array}{l} \mathbf{x}'(t) = A\mathbf{x}(t) + B\mathbf{u}(t), \quad \mathbf{x}(0) = \mathbf{x}_0 \\ \mathbf{u}(t) = \mathbf{u}_0 \quad 0 \le t < h \end{array}\right\} \quad \Leftrightarrow \quad \begin{bmatrix} \mathbf{x}'(t) \\ \mathbf{u}'(t) \end{bmatrix} = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{u}(t) \end{bmatrix} \triangleq \mathscr{A} \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{u}(t) \end{bmatrix} \quad \Rightarrow$$

$$\begin{bmatrix} \mathbf{x}(h^-) \\ \mathbf{u}(h^-) \end{bmatrix} = e^{\mathscr{A}h} \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{u}_0 \end{bmatrix} \quad \text{where} \quad e^{\mathscr{A}h} = \begin{bmatrix} e^{Ah} & G \\ 0 & I \end{bmatrix}.$$

Writing similarly $\mathbf{x}(h) = \mathbf{x}_1$ and $\mathbf{u}(h) = \mathbf{u}_1$ and looking over the next timestep $h \le t < 2h$, etc., we may convert exactly into the DT form

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + G\mathbf{u}_k$$

where $F = e^{Ah}$ and $G = h(I + Ah/2! + (Ah)^2/3! + (Ah)^3/4! + \ldots)B$, and the latter is easily computed as the upper-right block of $e^{\mathscr{A}h}$, as shown above.

Modal coördinate form

If the system is LTI and the eigenvectors of F are linearly independent, then we may perform an eigen decomposition $F = S\Sigma S^{-1}$, where Σ is the (diagonal) eigenvalue matrix. Performing the state transformation (20.3) with $R = S$ thus converts (20.2) to

$$\left.\begin{aligned} \mathbf{x}_{m,k+1} &= \Sigma \mathbf{x}_{m,k} + G_m \mathbf{u}_k \\ \mathbf{y}_k &= H_m \mathbf{x}_{m,k} + D\mathbf{u}_k \end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{c|c} \Sigma & G_m \\ \hline H_m & D \end{array}\right] = \left[\begin{array}{c|c} S^{-1}FS & S^{-1}G \\ \hline HS & D \end{array}\right]. \tag{20.23}$$

In this representation, called **modal coördinate form**, the evolution of each **mode** of the uncontrolled system is seen to be completely decoupled. That is, taking $\mathbf{u} = 0$, we may write $\mathbf{x}_{m,k} = \Sigma^k \mathbf{x}_{m,0}$. In the original coördinates, the solution to (20.6) with $\mathbf{u} = 0$ may thus be written

$$\mathbf{x}_k = S\mathbf{x}_{m,k}, \quad \text{where} \quad \mathbf{x}_{m,k} = \Sigma^k \mathbf{x}_{m,0} \quad \text{and} \quad \mathbf{x}_{m,0} = S^{-1}\mathbf{x}_0 \quad \Rightarrow \quad \mathbf{x}_k = S\Sigma^k S^{-1}\mathbf{x}_0. \tag{20.24}$$

This result may be confirmed by inserting the matrix decomposition $F = S\Sigma S^{-1}$ into F^k in (20.22) with $\mathbf{u} = 0$, resulting in $\mathbf{x}_k = F^k\mathbf{x}_0 = S\Sigma^k S^{-1}\mathbf{x}_0$, consistent with (20.24).

If the system is LTI and eigenvectors of F are not linearly independent, then an eigen decomposition is not available. In this case, the modal coördinate form is determined from the Jordan decomposition $F = MJM^{-1}$, where J is in Jordan form; performing a state transformation with $R = M$ converts (20.2) to

$$\left.\begin{aligned} \mathbf{x}_{m,k+1} &= J\mathbf{x}_{m,k} + G_m \mathbf{u}_k \\ \mathbf{y}_k &= H_m \mathbf{x}_{m,k} + D\mathbf{u}_k \end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{c|c} J & G_m \\ \hline H_m & D \end{array}\right] = \left[\begin{array}{c|c} M^{-1}FM & M^{-1}G \\ \hline HM & D \end{array}\right]. \tag{20.25}$$

In this case, the evolution of the modes corresponding to each Jordan block are coupled. That is, taking $\mathbf{u} = 0$, we may write $\mathbf{x}_{m,k} = J^k \mathbf{x}_{m,0}$. In the original coördinates, the solution to (20.6) with $\mathbf{u} = 0$ is

$$\mathbf{x}_k = M\mathbf{x}_{m,k}, \quad \text{where} \quad \mathbf{x}_{m,k} = J^k \mathbf{x}_{m,0} \quad \text{and} \quad \mathbf{x}_{m,0} = M^{-1}\mathbf{x}_0 \quad \Rightarrow \quad \mathbf{x}_k = MJ^k M^{-1}\mathbf{x}_0. \tag{20.26}$$

This result may be confirmed by inserting the matrix decomposition $F = MJM^{-1}$ into F^k, resulting in $\mathbf{x}_k = F^k\mathbf{x}_0 = MJ^k M^{-1}\mathbf{x}_0$, consistent with (20.26).

Both of the above cases lead immediately to the following conclusion:

Fact 20.8 *The uncontrolled DT LTI system $\mathbf{x}_{k+1} = F\mathbf{x}_k$ is stable if and only if all of the eigenvalues of F are stable (that is, inside the unit circle).*

LTV systems and the discrete-time state transition matrix

In the case that the system of interest is a linear time varying (LTV) system, we write

$$\mathbf{x}_{k+1} = F_k\mathbf{x}_k + G_k\mathbf{u}_k.$$

The solution to this system is given by

$$\begin{aligned} \mathbf{x}_1 &= F_0\mathbf{x}_0 + G_0\mathbf{u}_0 \\ \mathbf{x}_2 &= F_1\mathbf{x}_1 + G_1\mathbf{u}_1 = F_1F_0\mathbf{x}_0 + F_1G_0\mathbf{u}_0 + G_1\mathbf{u}_1 \\ &\vdots \\ \mathbf{x}_n &= F_{n-1}\mathbf{x}_{n-1} + G_{n-1}\mathbf{u}_{n-1} = F_{n-1}\cdots F_1F_0\mathbf{x}_0 + \sum_{k=0}^{n-1} F_{n-1}\cdots F_{k+1}G_k\mathbf{u}_k = \Phi_{n,0}\mathbf{x}_0 + \sum_{k=0}^{n-1} \Phi_{n,k+1}G_k\mathbf{u}_k \end{aligned} \tag{20.27a}$$

where the **discrete-time state transition matrix** $\Phi_{n,m}$ is defined for $n > m$ such that

$$\Phi_{n,m} = F_{n-1}F_{n-2}\cdots F_m = \prod_{j=1}^{n-m} F_{n-j}. \tag{20.27b}$$

For consistency, we also define $\Phi_{m,m} = I$ and, when the inverses on the RHS exist, $\Phi_{m,n} = F_m^{-1}\cdots F_{n-2}^{-1}F_{n-1}^{-1}$. Thus, when the applicable F_i are each nonsingular, $\Phi_{n_3,n_1} = \Phi_{n_2,n_1}\Phi_{n_3,n_2} = \Phi_{n_3,n_2}\Phi_{n_2,n_1}$; in particular, $\Phi_{n,m}\Phi_{m,n} = I \;\Rightarrow\; (\Phi_{n,m})^{-1} = \Phi_{m,n}$ (cf. Fact 20.6). Note that, in general, the F_i are not necessarily invertible, so $\Phi_{n,m}$ in the DT case is not necessarily invertible either.

Reconciling the CT and DT LTV state transition matrices

For cases in which the DT state-space form (20.2) is set up to approximate a system written originally in CT state-space form (20.1), we now illustrate how the formulae for the DT state transition matrix given in (20.27b) and the CT state transition matrix given in (20.19b) may be reconciled. For sufficiently small Δt, we can approximate the CT system (20.6) with the **Explicit Euler** method (10.5) such that

$$\frac{\mathbf{x}_{k+1}-\mathbf{x}_k}{\Delta t} = A_k\mathbf{x}_k + B_k\mathbf{u}_k \quad\Rightarrow\quad \mathbf{x}_{k+1} = F_k\mathbf{x}_k + G_k\mathbf{u}_k \quad \text{with} \quad F_k = I + A_k\Delta t,\;\; G_k = B_k\Delta t,$$

where $A_k = A(t_k)$ and $B_k = B(t_k)$. [As discussed in §10, more accurate discretizations are certainly possible; however, the present discretization is adequate for the present illustration.] By (20.27b), we may write

$$\mathbf{x}_n = \Phi_{n,0}\mathbf{x}_0 + \sum_{k=0}^{n-1}\Phi_{n,k+1}B_k\mathbf{u}_k\Delta t \quad \text{where} \quad \Phi_{n,m} = \prod_{j=1}^{n-m}[I + A_{n-j}\Delta t].$$

We now consider the limit of the above expression as the discretization of the system is refined. Taking the limit as the number of steps n spanning the interval $[0,t]$ is increased, and the corresponding timestep $\Delta t = t/n$ is decreased (and, thus, the Explicit Euler approximation of the CT system becomes more accurate), the sum above converts to an integral, and the above expression may be written

$$\mathbf{x}(t) = \breve{\Phi}(t,0)\mathbf{x}_0 + \int_0^t \breve{\Phi}(t,\tau)B(\tau)\mathbf{u}(\tau)d\tau$$

where, denoting $m(n,\tau/t) = \text{round}(n\tau/t)$, $\breve{\Phi}(t,\tau)$ is defined by

$$\begin{aligned}
\breve{\Phi}(t,\tau) &= \lim_{n\to\infty}\left\{\prod_{j=1}^{n-m(n,\tau/t)}\left[I + A\Big(\frac{(n-j)t}{n}\Big)\frac{t}{n}\right]\right\} \\
&= \lim_{n\to\infty}\left\{I + \sum_{j=1}^{n-m(n,\tau/t)} A\Big(\frac{(n-j)t}{n}\Big)\frac{t}{n} + \sum_{j=1}^{n-m(n,\tau/t)} A\Big(\frac{(n-j)t}{n}\Big)\Big[\sum_{j'=j}^{n-m(n,\tau/t)} A\Big(\frac{(n-j')t}{n}\Big)\frac{t}{n}\Big]\frac{t}{n} + \ldots\right\} \\
&= \lim_{n\to\infty}\left\{I + \sum_{j=m(n,\tau/t)}^{n} A\Big(\frac{jt}{n}\Big)\frac{t}{n} + \sum_{j=m(n,\tau/t)}^{n} A\Big(\frac{jt}{n}\Big)\Big[\sum_{j'=m(n,\tau/t)}^{n-j} A\Big(\frac{j't}{n}\Big)\frac{t}{n}\Big]\frac{t}{n} + \ldots\right\},
\end{aligned}$$

thus showing that, in the limit that the grid is refined, the state transition matrix $\breve{\Phi}(t,\tau)$ for this DT form approaches a rectangular-rule approximation of the corresponding CT state transition matrix $\Phi(t,\tau)$ given in (20.19b).

In the case that the system is LTI and taking $\tau = 0$ for simplicity, noting (B.63) and (B.64), the above relation reduces to

$$\begin{aligned}\check{\Phi}(t,0) &= \lim_{n\to\infty}\left\{\prod_{j=1}^{n}\left[I + A\frac{t}{n}\right]\right\} = \lim_{n\to\infty}\left\{I + A\sum_{j=1}^{n}\frac{t}{n} + A^2\sum_{j=1}^{n}\left[\sum_{j'=j}^{n}\frac{t}{n}\right]\frac{t}{n} + \ldots\right\} \\ &= \lim_{n\to\infty}\left\{I + At\frac{n}{n} + (At)^2\sum_{j=1}^{n}\left[\frac{n-j}{n}\right]\frac{1}{n} + \ldots\right\} = I + At + \frac{(At)^2}{2} + \frac{(At)^3}{6} + \ldots,\end{aligned}$$

thus providing a different derivation of the Tayor-series expansion defining the matrix exponential.

20.1.4 Subsystems

A **subsystem** may easily be extracted from a CT or DT state-space form by performing an appropriate matrix decomposition of the system matrix, transforming the linear system using the procedure illustrated in (20.3), then extracting the equation describing the dynamics of certain modes of interest. For example, consider an ordered Schur decomposition $A = UTU^H$ and subsequent block partitioning of the transformation matrix U and upper triangular matrix T in this decomposition such that

$$U = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix}, \quad T = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix},$$

where the stable (LHP) eigenvalues appear on the main diagonal of T_{11} and the neutrally-stable (imaginary) and unstable (RHP) eigenvalues appear on the main diagonal of T_{22}. Performing the state transformation (20.3), taking $R = U$, transforms (20.1) to the form

$$\left.\begin{aligned}\begin{bmatrix}\mathbf{x}_s(t)\\ \mathbf{x}_u(t)\end{bmatrix}' &= \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22}\end{bmatrix}\begin{bmatrix}\mathbf{x}_s(t)\\ \mathbf{x}_u(t)\end{bmatrix} + \begin{bmatrix} B_s \\ B_u\end{bmatrix}\mathbf{u}(t) \\ \mathbf{y}(t) &= \begin{bmatrix} C_s & C_u\end{bmatrix}\begin{bmatrix}\mathbf{x}_s(t)\\ \mathbf{x}_u(t)\end{bmatrix} + D\mathbf{u}(t)\end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{cc|c} T_{11} & T_{12} & B_s \\ 0 & T_{22} & B_u \\ \hline C_s & C_u & D\end{array}\right] = \left[\begin{array}{c|c} U^HAU & U^HB \\ \hline CU & D\end{array}\right]. \tag{20.28}$$

We may then extract the following, which we will refer to as the **unstable subsystem** of the system (20.1):

$$\left.\begin{aligned}\mathbf{x}_u'(t) &= T_{22}\mathbf{x}_u(t) + B_u\mathbf{u}(t) \\ \mathbf{y}_u(t) &\triangleq C_u\mathbf{x}_u(t)\end{aligned}\right\} \quad \Leftrightarrow \quad \left[\begin{array}{c|c} T_{22} & B_u \\ \hline C_u & 0\end{array}\right]. \tag{20.29}$$

Note that the evolution of $\mathbf{x}_u(t)$ is decoupled from the evolution of $\mathbf{x}_s(t)$.

An analogous procedure may be applied to extract the unstable DT subsystem of (20.2), in this case grouping those eigenvalues with magnitude less than one on the main diagonal of T_{11} and those eigenvalues with magnitude greater than or equal to one on the main diagonal of T_{22}.

20.1.5 MIMO transfer functions and the resolvent

Returning to the discussion of transform-based methods as introduced in §17, we show now that we may also develop the **transfer function** $G(s)$ of a CT LTI system from its state-space representation, even in the MIMO case. Taking the Laplace transform of (20.1) [using the natural notation for vector functions of time such that, e.g., $\mathbf{X}(s)$ is the Laplace transform of $\mathbf{x}(t)$] gives

$$\left.\begin{aligned}s\mathbf{X}(s) &= A\mathbf{X}(s) + B\mathbf{U}(s) \\ \mathbf{Y}(s) &= C\mathbf{X}(s) + D\mathbf{U}(s)\end{aligned}\right\} \quad \Rightarrow \quad \mathbf{Y}(s) = G(s)\mathbf{U}(s) \quad \text{where} \quad G(s) = C(sI - A)^{-1}B + D. \tag{20.30}$$

In the SIMO case, we may write the above as $\mathbf{G}(s) = \mathbf{Y}(s)/U(s)$ where $\mathbf{G}(s)$ is a vector; in the MIMO case, we leave the expression in the above form, noting that the transfer function $G(s)$ in this case is a matrix.

Fact 20.9 *The transfer function of a system is invariant under state transformation* (20.3).

Proof: Noting Fact 1.7, we may write

$$G_R(s) = C_R(sI - A_R)^{-1}B_R + D_R = CR(sI - R^{-1}AR)^{-1}R^{-1}B + D = C(sI - A)^{-1}B + D = G(s). \qquad \square$$

The factor $(sI - A)^{-1}$ in (20.30) is called the **resolvent** matrix. To gain a better understanding of it, we may revisit the vector LTI system (20.6) and its solution built on the matrix exponential in (20.7). Applying (17.49a) to each component of the LHS of the Laplace transform of (20.6) in the case that $\mathbf{u} = 0$ gives

$$s\mathbf{X}(s) - \mathbf{x}(0) = A\mathbf{X}(s) \quad \Rightarrow \quad \mathbf{X}(s) = (sI - A)^{-1}\mathbf{x}(0), \tag{20.31}$$

whereas applying the fact (from Table 17.1a) that the Laplace transform of t^n is $n!/s^{n+1}$ to the Laplace transform of (20.7), in the case that $\mathbf{u} = 0$, gives

$$\mathbf{X}(s) = \Big(\frac{I}{s} + \frac{A}{s^2} + \frac{A^2}{s^3} + \ldots\Big)\mathbf{x}(0).$$

It thus follows that

$$(sI - A)^{-1} = \frac{1}{s}\Big(I + \frac{A}{s} + \frac{A^2}{s^2} + \ldots\Big). \tag{20.32}$$

[In the case in which $s = 1$ and $A = \varepsilon$, this reduces to the familiar expansion $(1 - \varepsilon)^{-1} = 1 + \varepsilon + \varepsilon^2 + \ldots$]

Comparing (20.7) and (20.31), it is seen that *the resolvent* $(sI - A)^{-1}$ *may be interpreted as the Laplace transform of the matrix exponential* e^{At}. This interpretation is useful, because it allows us rewrite the infinite series expansion for the Laplace transform of e^{At} as simply the inverse of a matrix. By Cramer's rule (Fact 4.3), we may reëxpress this inverse as $(sI - A)^{-1} = (sI - A)_{\text{cof}}/|sI - A|$, where the i, j'th element of the $n \times n$ matrix $(sI - A)_{\text{cof}}$ is $(-1)^{i+j}$ times the determinant of the matrix formed by removing the i'th column and the j'th row of the matrix $(sI - A)$. That is, each element of $(sI - A)_{\text{cof}}$ is an $(n-1)$'th-order polynomial in s, whereas $|sI - A| = s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0$ is the characteristic polynomial of A, which is an n'th order polynomial in s. We may thus write

$$(sI - A)^{-1} = \frac{S_{n-1}s^{n-1} + S_{n-2}s^{n-2} + \ldots + S_1 s + S_0}{s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0}. \tag{20.33}$$

Note that it follows directly from (20.30) and (20.33) that we may also write

$$G(s) = \frac{Ds^n + (CS_{n-1}B + a_{n-1}D)s^{n-1} + (CS_{n-2}B + a_{n-2}D)s^{n-2} + \ldots + (CS_1B + a_1D)s + (CS_0B + a_0D)}{s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0}, \tag{20.34}$$

thereby converting a state-space representation of a system into **transfer function form** (that is, into a rational function of s). Multiplying both sides of (20.33) by $(sI - A)$, it is easily verified that the values of the S_i given by the following **resolvent algorithm** satisfy the above equation:

$$\begin{aligned}
S_{n-1} &= I, & a_{n-1} &= -\text{trace}(S_{n-1}A),\\
S_{n-2} &= S_{n-1}A + a_{n-1}I, & a_{n-2} &= -\text{trace}(S_{n-2}A)/2,\\
S_{n-3} &= S_{n-2}A + a_{n-2}I, & a_{n-3} &= -\text{trace}(S_{n-3}A)/3,\\
&\vdots & &\vdots\\
S_1 &= S_2A + a_2I, & a_1 &= -\text{trace}(S_1A)/(n-1),\\
S_0 &= S_1A + a_1I, & a_0 &= -\text{trace}(S_0A)/n;
\end{aligned} \tag{20.35a}$$

Algorithm 20.3: Convert a MIMO state-space model to transfer function form using the resolvent algorithm.

View
Test

```
function [b,a]=SS2TF(A,B,C,D)
% Derive the transfer function form corresponding to a MIMO state-space form using the
% resolvent algorithm, for a system with ni inputs, no outputs, and n states. a is 1 x n.
% For SISO and SIMO systems, b is no x (n+1), consistent with Matlab's ss2tf.
% For MISO and MIMO systems, b is no x ni x (n+1). The a and b coefficients
% are enumerated in the opposite order here than in the textbook derivation.
n=size(A,1); ni=size(B,2); no=size(C,1); if nargin<4, D=zeros(no,ni); end
S(:,:,1)=zeros(n); a(1,1)=1; b(:,:,1)=D;
for i=2:n+1;
  S(:,:,i)=S(:,:,i-1)*A+a(i-1)*eye(n); a(1,i)=-trace(S(:,:,i)*A)/(i-1);    % (20.35a)
  b(:,:,i)=(C*S(:,:,i)*B+a(1,i)*D);                                        % (20.34)
end, if ni==1, b=reshape(b,no,n+1); end
end % function SS2TF
```

verification that the a_i given by the above algorithm are exactly the coefficients in the characteristic polynomial of A, as required, follows immediately from (4.68). Implementation of the above equations to convert a MIMO state-space model to transfer function form is given in Algorithm 20.3.

Substituting (20.35a) into (20.33), it follows that

$$(sI-A)^{-1} = \alpha_0(s)I + \alpha_1(s)A + \alpha_2(s)A^2 + \alpha_3(s)A^3 + \ldots + \alpha_{n-1}(s)A^{n-1}, \tag{20.35b}$$

where

$$\begin{aligned}
\alpha_{n-1}(s) &= 1/(s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0),\\
\alpha_{n-2}(s) &= (s + a_{n-1})/(s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0),\\
\alpha_{n-3}(s) &= (s^2 + a_{n-1}s + a_{n-2})/(s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0),\\
&\vdots\\
\alpha_0(s) &= (s^{n-1} + a_{n-1}s^{n-2} + \ldots + a_1)/(s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0).
\end{aligned} \tag{20.35c}$$

The inverse Laplace transform of each of the $\alpha_i(s)$ is easy to compute using partial fraction expansions and the methods of §17.3.2, thus determining the $\alpha_i(t)$ in the finite series expansion of e^{At} postulated in (20.8):

$$e^{At} \triangleq \alpha_0(t)I + \alpha_1(t)A + \alpha_2(t)A^2 + \alpha_3(t)A^3 + \ldots + \alpha_{n-1}(t)A^{n-1} \quad \text{for any } t \geq 0. \tag{20.36}$$

MIMO systems and the resolvent in discrete time

It is also straightforward to develop the transfer function $G(z)$ of a DT LTI system in state-space form, even in the MIMO case. Taking the Z transform of (20.2) [using the natural notation for vector functions of k that, e.g., $\mathbf{X}(z)$ is the Z transform of $\mathbf{x}_k$] gives

$$\left.\begin{aligned} z\mathbf{X}(z) &= F\mathbf{X}(z) + G\mathbf{U}(z)\\ \mathbf{Y}(z) &= H\mathbf{X}(z) + D\mathbf{U}(z) \end{aligned}\right\} \quad \Rightarrow \quad \mathbf{Y}(z) = G(z)\mathbf{U}(z) \quad \text{where} \quad G(z) = H(zI-F)^{-1}G + D. \tag{20.37}$$

In the SIMO case, we may write the above as $\mathbf{G}(z) = \mathbf{Y}(z)/U(z)$; in the MIMO case, we leave the expression in the above form, noting that the transfer function $G(z)$ in this case is a matrix. As stated for CT systems in Fact 20.9, the transfer function of DT systems is *invariant* under state transformation.

As in (20.32), the following expansion of the resolvent matrix $(zI-F)^{-1}$ is sometimes useful

$$(zI-F)^{-1} = \frac{1}{z}\left(I + \frac{F}{z} + \frac{F^2}{z^2} + \ldots\right), \tag{20.38}$$

where $(zI-F)^{-1}$ may be interpreted as the Z transform of F^k. Following the approach summarized in (20.33)-(20.35), with s replaced by z and the inverse Laplace transform replaced by the inverse Z transform, the resolvent algorithm may be used to express

$$F^k = \alpha_0^k I + \alpha_1^k F + \alpha_2^k F^2 + \alpha_3^k F^3 + \ldots + \alpha_{n-1}^k F^{n-1} \quad \text{for any } k \geq 0.$$

Note that, for $k < n$, $\alpha_i^k = \delta_{ik}$.

Reconciling the CT and DT transfer functions

Applying the Euler explicit approximation to approximate the CT system (20.1) in the DT form (20.2) (that is, defining $F = I + Ah$, $G = Bh$, and $H = C$), and using the approximate relation $z \approx 1 + sh$ given in (17.70) to connect the CT and DT variables, it follows that

$$H(zI-F)^{-1}G + D \approx C[(1+sh)I - (I+Ah)]^{-1}Bh + D = C(sI-A)^{-1}B + D,$$

thereby reconciling the equations for the CT and DT transfer functions.

20.2 Characterizing solutions of nonorthogonal & defective systems

Our study of CT and DT linear systems has focused thus far on the *eigenvalues* of the system matrix [equivalently, in §17, on the *poles* of the transfer function]. As seen in Facts 20.4 and 20.8, this is enough to determine stability. As seen in Examples 17.12 and 17.13, this is also enough to characterize the response of real second-order systems (which, by Fact 4.5, have orthogonal eigenvectors), as well as systems which are dominated by such second-order behavior. As demonstrated below, however, the locations of the eigenvalues/poles alone are *not* sufficient to characterize the system response in more general systems, due to energy amplification mechanisms related to *eigenvector nonorthogonality*, and thus further analysis is required.

The case of orthogonal eigenvectors

In continuous time, if the eigenvectors of A (that is, the columns of S) are orthogonal (for example, if A is symmetric or, in the complex case, Hermitian) and appropriately normalized, then $S^{-1} = S^H$, and, decomposing $\lambda_k = \lambda_{k,R} + \mathrm{i}\,\lambda_{k,I}$, it follows from (20.12) that the "energy" of the uncontrolled system [that is, the Euclidean norm of $\mathbf{x}(t)$] may be written

$$\|\mathbf{x}(t)\|^2 = \mathbf{x}^H(t)\mathbf{x}(t) = \mathbf{x}^H(0)Se^{(\Lambda^H+\Lambda)t}S^H\mathbf{x}(0), \quad \text{where} \quad e^{(\Lambda^H+\Lambda)t} = \begin{pmatrix} e^{2t\lambda_{1,R}} & & 0 \\ & \ddots & \\ 0 & & e^{2t\lambda_{n,R}} \end{pmatrix}.$$

Defining $\mathbf{z}(0) = S^H\mathbf{x}(0)$, it follows that

$$\|\mathbf{x}(t)\|^2 = e^{2t\lambda_{1,R}}|z_1(0)|^2 + e^{2t\lambda_{2,R}}|z_2(0)|^2 + \ldots + e^{2t\lambda_{n,R}}|z_n(0)|^2.$$

We thus conclude that, if all the eigenvalues of the A are stable (that is, $\lambda_{\kappa,R} < 0$ for all κ, as discussed in §20.1.2) and the corresponding eigenvectors are orthogonal, it follows that $\|\mathbf{x}(t)\|^2$ decreases monotonically in time in the uncontrolled system, as depicted in Figure 20.3a.

In discrete time, if the eigenvectors of F are orthogonal, then $S^{-1} = S^H$, and the energy of the system may be written

$$\|\mathbf{x}_k\|^2 = \mathbf{x}_k^H\mathbf{x}_k = \mathbf{x}_0^H S(\overline{\Sigma}^k\Sigma^k)S^H\mathbf{x}_0, \quad \text{where} \quad \overline{\Sigma}^k\Sigma^k = \begin{pmatrix} |\sigma_1|^{2k} & & 0 \\ & \ddots & \\ 0 & & |\sigma_n|^{2k} \end{pmatrix}.$$

Thus, defining $\mathbf{z}_0 = S^H \mathbf{x}_0$, it follows that

$$\|\mathbf{x}_k\|^2 = |\sigma_1|^{2k}|z_{1,0}|^2 + |\sigma_2|^{2k}|z_{2,0}|^2 + \ldots + |\sigma_n|^{2k}|z_{n,0}|^2.$$

We thus again conclude that, if all the eigenvalues of the F are stable (that is, $|\sigma_j| < 1$ for all j, as discussed in §20.1.3) and the corresponding eigenvectors are orthogonal, it follows that $\|\mathbf{x}_k\|^2$ decreases monotonically with k in the uncontrolled system.

The case of independent but nonorthogonal eigenvectors

It is important to note that, in the case that all the modes of the uncontrolled system matrix are stable and their eigenvectors linearly independent but *not* orthogonal, the 2-norm of the solution to the system does *not* necessarily decrease monotonically in time. We illustrate this effect here for CT systems, but the same effect is also seen in DT systems. In the CT case in which the eigenvectors $\mathbf{s}^\kappa$ are linearly independent but nonorthogonal, note that we may write

$$\mathbf{x}(t) = S\mathbf{z}(t) = z_1(t)\,\mathbf{s}^1 + z_2(t)\,\mathbf{s}^2 + \ldots + z_n(t)\,\mathbf{s}^n. \tag{20.39}$$

Though the individual $|z_\kappa(t)|$ decrease monotonically in time for all κ, the 2-norm of the linear combination of vectors on the RHS of (20.39) might grow substantially before it eventually decays.

This effect is well illustrated by the example already mentioned in (4.38): consider a 2×2 matrix A with eigenvalues λ_1 and $\lambda_1 + \varepsilon$ such that

$$A = \begin{pmatrix} \lambda_1 & 1 \\ 0 & \lambda_1 + \varepsilon \end{pmatrix}, \quad \Rightarrow \quad \mathbf{s}^1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{s}^2 = \begin{pmatrix} 1 \\ \varepsilon \end{pmatrix}, \tag{20.40}$$

where $\lambda_{1,R} < 0$ and $0 < |\varepsilon| \ll 1$. Note that the two eigenvectors are nearly parallel. Taking a "destructive" linear combination of these two eigenvectors by selecting $z_1(0) = 1$ and $z_2(0) = -1$, the initial 2-norm of $\mathbf{x}(0)$ is very small, $\|\mathbf{x}(0)\|^2 = \varepsilon^2$. As the two modes have different eigenvalues, the corresponding $z_\kappa(t)$ decay at different rates. Thus, the effectiveness of the "overlap" of the two components of $\mathbf{x}(t)$, which results from the nonorthogonality of the $\mathbf{s}^\kappa$ [see (20.39)], reduces with time. As a result, $\|\mathbf{x}\|^2$ might grow *substantially* before it eventually decays due to the stability of the eigenvalues, as depicted in Figures 20.3b and 20.3c as well as the cartoon in Figure 20.4. This type of response is ubiquitous in complex systems, and is referred to as **transient energy growth** or **peaking**.

The defective case

If the eigenvectors of the uncontrolled system matrix are stable but their eigenvectors are *not* linearly independent, a very similar effect is seen as discussed in the preceding section. Again, we illustrate this effect here for CT systems, but the same effect is also seen in DT systems. In the CT case in which the eigenvectors $\mathbf{s}^\kappa$ are not linearly independent, note that A may still be decomposed in Jordan form such that $A = MJM^{-1}$, where J is in Jordan form (close to a diagonal matrix, but with some 1's in the first superdiagonal), as noted in §4.4.6. Thus, by substitution of $\mathbf{x} = M\mathbf{z}$, into (20.6) and multiplication from the left by M^{-1}, it follows that $\mathbf{z}' = J\mathbf{z}$. In this representation, the coupling of the various modes of the system is readily apparent.

For example, in the case of the particular matrix J depicted in (20.10), we have

$$\begin{aligned} z_1' &= \lambda_1 z_1 + z_2 \\ z_2' &= \lambda_1 z_2 + z_3 \\ z_3' &= \lambda_1 z_3 \\ z_4' &= \lambda_2 z_4. \end{aligned} \quad \Rightarrow \quad \begin{aligned} z_1(t) &= e^{\lambda_1 t} z_1(0) + t e^{\lambda_1 t} z_2(0) + \frac{t^2}{2} e^{\lambda_1 t} z_3(0) \\ z_2(t) &= e^{\lambda_1 t} z_2(0) + t e^{\lambda_1 t} z_3(0) \\ z_3(t) &= e^{\lambda_1 t} z_3(0) \\ z_4(t) &= e^{\lambda_2 t} z_4(0). \end{aligned} \tag{20.41}$$

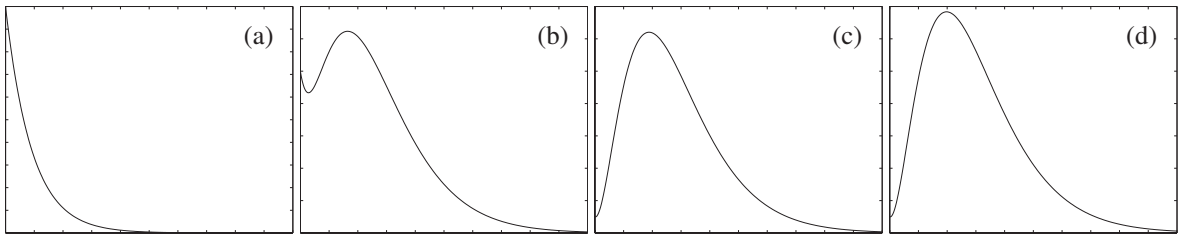


Figure 20.3: "Typical" system responses, $\|\mathbf{x}(t)\|$ versus t, as a function of eigenvector nonorthogonality. Parameters used: (a) $A = \begin{pmatrix} -0.1 & 0 \\ 0 & -0.11 \end{pmatrix}$, $\mathbf{x}(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$; (b) $A = \begin{pmatrix} -0.1 & 0.3 \\ 0 & -0.11 \end{pmatrix}$, $\mathbf{x}(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$; (c) $A = \begin{pmatrix} -0.1 & 1 \\ 0 & -0.11 \end{pmatrix}$, $\mathbf{x}(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$; (d) $A = \begin{pmatrix} -0.1 & 1 \\ 0 & -0.1 \end{pmatrix}$, $\mathbf{x}(0) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

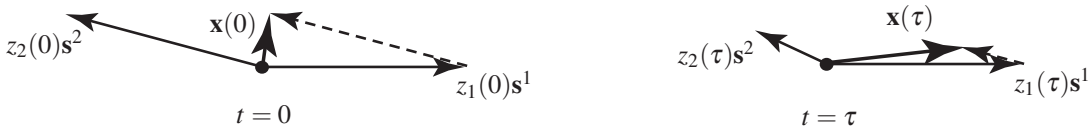


Figure 20.4: Graphical illustration of how the length of the resultant $\mathbf{x}(t) = z_1(t)\mathbf{s}^1 + z_2(t)\mathbf{s}^2$ can grow (from time $t = 0$ at left to time $t = \tau$ at right), before it eventually decays, even when the magnitude of its components, $|z_1(t)|$ and $|z_2(t)|$, decay exponentially for all time. [In short, the destructive interference of the two components at time $t = 0$ initially reduces in time if the two components $z_1(t)$ and $z_2(t)$ decay at different rates.] Note that the component directions $\mathbf{s}^1$ and $\mathbf{s}^2$ are not orthogonal; the closer $\mathbf{s}^1$ and $\mathbf{s}^2$ are to parallel, the worse this **transient energy growth** or **peaking** can be.

In matrix form, $\mathbf{z}(t) = e^{Jt}\mathbf{z}(0)$, where e^{Jt} is given in (20.10).

As another example, taking $\varepsilon \to 0$ in (20.40), the matrix A reduces directly to a Jordan form. In this case, (20.6) may be written

$$\begin{aligned} x_1' &= \lambda_1 x_1 + x_2, \\ x_2' &= \lambda_1 x_2. \end{aligned} \quad \Rightarrow \quad \begin{aligned} x_1(t) &= e^{\lambda_1 t} x_1(0) + t e^{\lambda_1 t} x_2(0) \\ x_2(t) &= e^{\lambda_1 t} x_2(0). \end{aligned} \tag{20.42}$$

In both of the above examples, the situation may be thought of as the limit of the nonorthogonal case as the linear independence of the eigenvectors is lost. If $\lambda_{1,R} < 0$ in either example, the response of the uncontrolled system is characterized by **algebraic growth followed by exponential decay**, as depicted in Figure 20.3d for the example given in (20.42). Note the similarity between Figures 20.3c and 20.3d; the response of the system does *not* suddenly change when ε reaches zero in (20.40) [though the peculiar Jordan decomposition of A *does* change suddenly when this limit is reached, as noted in §4.4.6].

The key point of this discussion is that, though a starting point for understanding low-dimensional linear systems and their control is simply getting the eigenvalues of the system sufficiently far into the LHP, this mindset is generally insufficient for adequate performance in high-dimensional systems. Nonorthogonality of the eigenvectors of A (or, in the limiting case, an A which is defective) leads to mechanisms for substantial energy amplification in stable systems which must also be accounted for and addressed.

20.2.1 Quantifying the maximum transient energy growth of a stable system

A quadratic measure or **cost** associated with the state of a system, $\|\mathbf{x}\|_Q^2 = \mathbf{x}^H Q\mathbf{x}$ for $Q > 0$, may be introduced and interpreted as a "generalized energy" measure of the system state. As illustrated in Figure 20.3, if a stable system matrix is either defective or characterized by nonorthogonal eigenvectors, then such a measure might grow substantially before it eventually decays at a rate corresponding to that of the least stable constituent eigenmode. We now quantify this effect.

For the CT LTI system $\mathbf{x}'(t) = A\mathbf{x}(t)$, the direction of the most amplified initial condition, $\mathbf{x}(0)$, and the corresponding maximum amplification factor $\theta_{\max}$ of the energy of this system $\|\mathbf{x}(t)\|_Q^2$ over any time interval of interest $t \in [0,\tau]$ follow as an immediate consequence the Rayleigh-Ritz Theorem (Fact 4.26) when the problem is transformed into **cost-decoupled** coördinates $\mathbf{z} = Q^{1/2}\mathbf{x}$, where $Q^{1/2}$ is the unique Hermitian positive-definite matrix such that $Q^{1/2}Q^{1/2} = Q$:

$$\begin{aligned}\max_{\mathbf{x}(0)\neq 0} \frac{\mathbf{x}^H(\tau)Q\mathbf{x}(\tau)}{\mathbf{x}^H(0)Q\mathbf{x}(0)} &= \max_{\mathbf{x}^H(0)Q\mathbf{x}(0)=1} \mathbf{x}^H(\tau)Q\mathbf{x}(\tau) = \max_{\mathbf{x}^H(0)Q\mathbf{x}(0)=1} \mathbf{x}^H(0)e^{A^H\tau}Qe^{A\tau}\mathbf{x}(0) \\ &= \max_{\mathbf{z}^H(0)\mathbf{z}(0)=1} \mathbf{z}^H(0)[Q^{-1/2}e^{A^H\tau}Qe^{A\tau}Q^{-1/2}]\mathbf{z}(0) = \theta_{\max}, \end{aligned} \tag{20.43a}$$

where $\theta_{\max}$ is the maximum eigenvalue of the Hermitian matrix $[Q^{-1/2}e^{A^H\tau}Qe^{A\tau}Q^{-1/2}]$.

For the DT LTI system $\mathbf{x}_{k+1} = F\mathbf{x}_k$, the maximum transient energy growth over m timesteps may be determined similarly:

$$\begin{aligned}\max_{\mathbf{x}_0\neq 0} \frac{\mathbf{x}_m^H Q\mathbf{x}_m}{\mathbf{x}_0^H Q\mathbf{x}_0} &= \max_{\mathbf{x}_0^H Q\mathbf{x}_0=1} \mathbf{x}_m^H Q\mathbf{x}_m = \max_{\mathbf{x}_0^H Q\mathbf{x}_0=1} \mathbf{x}_0^H (F^m)^H Q F^m \mathbf{x}_0 \\ &= \max_{\mathbf{z}_0^H\mathbf{z}(0)=1} \mathbf{z}_0^H[Q^{-1/2}(F^m)^H Q F^m Q^{-1/2}]\mathbf{z}_0 = \theta_{\max}, \end{aligned} \tag{20.43b}$$

where $\theta_{\max}$ is the maximum eigenvalue of the Hermitian matrix $[Q^{-1/2}(F^m)^H Q F^m Q^{-1/2}]$.

Implementation of the above formulae is given in Algorithm 20.4. In both cases, an optimization routine such as Brent's algorithm (see Algorithm 15.5) may be used to find the time interval τ, or the number of time steps m, over which the CT or DT system can lead to the greatest possible transient energy growth.

20.2.2 Quantifying the transfer function of a stable system via system norms

In §20.2, it was observed that nonorthogonality of the eigenvectors of a system matrix creates an important (and somewhat "hidden") mechanism for *significant energy amplification* in stable systems. This mechanism was quantified in §20.2.1 by calculating the *maximum transient energy growth* of an unforced system. In a system forced by external disturbances, precise characterizations of this "hidden" internal mechanism for energy amplification are possible from an input/output perspective. In both the CT and DT cases, we thus consider two **transfer function norms** (a.k.a. **system norms**) to quantify how an appropriately-defined **cost variable** output $\mathbf{z}$ [e.g., $\mathbf{z} = Q^{1/2}\mathbf{x}$] responds to a **disturbance input** $\mathbf{w}$ acting on the system.

To proceed, consider a CT LTI system in state-space form, initially at rest, with a disturbance input $\mathbf{w}(t)$ and a cost variable output $\mathbf{z}(t)$; the transfer function relating $\mathbf{w}(t)$ to $\mathbf{z}(t)$, denoted $G(s)$, is given by

$$\begin{aligned}\mathbf{x}'(t) &= A\mathbf{x}(t) + B\mathbf{w}(t) \\ \mathbf{z}(t) &= C\mathbf{x}(t) + D\mathbf{w}(t)\end{aligned} \quad \Rightarrow \quad \mathbf{Z}(s) = G(s)\mathbf{W}(s) \quad \text{with} \quad G(s) = C(sI-A)^{-1}B + D. \tag{20.44a}$$

Likewise, consider also a DT LTI system in state-space form, initially at rest, with a disturbance input $\mathbf{w}_k$ and a cost variable output $\mathbf{z}_k$; the **transfer function** relating $\mathbf{w}_k$ to $\mathbf{z}_k$, denoted $G(z)$, is given by

$$\begin{aligned}\mathbf{x}_{k+1} &= F\mathbf{x}_k + G\mathbf{w}_k \\ \mathbf{z}_k &= H\mathbf{x}_k + D\mathbf{w}_k\end{aligned} \quad \Rightarrow \quad \mathbf{Z}(z) = G(z)\mathbf{W}(z) \quad \text{with} \quad G(z) = H(zI-F)^{-1}G + D. \tag{20.44b}$$

The **CT transfer function 2-norm**, $\|G(s)\|_2$, is defined for a CT MIMO system (20.44a) such that

$$\|G(s)\|_2^2 \triangleq \frac{1}{2\pi}\int_{-\infty}^{\infty} \big\|G(\mathrm{i}\omega)\big\|_F^2\, d\omega = \frac{1}{2\pi}\int_{-\infty}^{\infty} \operatorname{trace}\big[G^H(\mathrm{i}\omega)G(\mathrm{i}\omega)\big]\, d\omega = \frac{1}{2\pi}\int_{-\infty}^{\infty} \sum_i \sigma_i^2\big[G(\mathrm{i}\omega)\big]\, d\omega, \quad (20.45)$$

where $\|\cdot\|_F$ denotes the Frobenius norm [§1.3.2] and σ_i denotes the i'th singular value [§4.4.7]. The transfer function 2-norm of a stable CT MIMO system has at least two natural interpretations:

- From a *deterministic* perspective, we take $\mathbf{w}(t)$ as a sequence of unit impulses in each component and denote $g(t)$ as the associated **impulse response matrix**[1] of the system (with corresponding transfer function $G(s)$; see Fact 17.6). Applying the form of Parseval's theorem given in (5.41c), noting the relation between the Fourier transform and the Laplace transform given in (17.4), applying causality, and applying the definition of the 2-norm of a CT signal (§1.3.3), it follows that

$$\|G(s)\|_2^2 = \frac{1}{2\pi}\operatorname{trace}\Big[\int_{-\infty}^{\infty} G^H(\mathrm{i}\omega)G(\mathrm{i}\omega)d\omega\Big] = \int_0^{\infty} \operatorname{trace}\Big[g^H(t)g(t)\Big]dt = \|g(t)\|_2^2. \quad (20.46a)$$

  Thus, *the square of the transfer function 2-norm is the* **total energy** *of the output* $\mathbf{z}(t)$ *of the CT system when the input* $\mathbf{w}(t)$ *contains a sequence of unit impulses in each component.*

- From a *stochastic* perspective, we take $\mathbf{w}(t)$ as a zero-mean **CT random process** (§6.3) with separable autocorrelation[2] $\mathscr{E}\{\mathbf{w}(t+\tau)\,\mathbf{w}^H(t)\} = \lim_{T\to\infty}\frac{1}{T}\int_0^T \mathbf{w}(t+\tau)\,\mathbf{w}^H(t)\,dt = R\,\delta^\sigma(\tau)$ and unit spectral density $R = I$, where $\delta^\sigma(\tau) = e^{-\tau^2/(2\sigma^2)}/(\sigma\sqrt{2\pi})$, and examine the mean energy of the output $\mathbf{z}(t)$ in the limit that $\sigma \to 0$ (that is, in the limit that the CT input $\mathbf{w}(t)$ is "white"). Writing $\mathbf{z}(t)$ as a convolution of the impulse response $g(t-\tau)$ and the input $\mathbf{w}(\tau)$, changing variables and rearranging the integrals appropriately, noting that $g(\tau)\to 0$ exponentially as $\tau\to\infty$, and applying (20.46a), it follows that

$$\begin{aligned}
\mathscr{E}\{\mathbf{z}^H(t)\mathbf{z}(t)\} &= \lim_{T\to\infty}\frac{1}{T}\int_0^T \mathbf{z}^H(t)\,\mathbf{z}(t)\,dt = \lim_{T\to\infty}\frac{1}{T}\int_0^T\int_0^t [g(t-\tau_1)\mathbf{w}(\tau_1)]^H\,d\tau_1\int_0^t g(t-\tau_2)\mathbf{w}(\tau_2)\,d\tau_2\,dt\\
&= \lim_{T\to\infty}\frac{1}{T}\int_0^T\int_0^t\int_0^t \mathbf{w}^H(t-\tau_1')\,g^H(\tau_1')\,g(\tau_2')\mathbf{w}(t-\tau_2')\,d\tau_1'\,d\tau_2'\,dt\\
&= \lim_{T\to\infty}\frac{1}{T}\int_0^T\int_0^\infty\int_0^\infty \operatorname{trace}\Big[g^H(\tau_1')\,g(\tau_2')\mathbf{w}(t-\tau_2')\mathbf{w}^H(t-\tau_1')\Big]\,d\tau_1'\,d\tau_2'\,dt \qquad (20.46b)\\
&= \int_0^\infty\int_0^\infty \operatorname{trace}\Big[g^H(\tau_1')\,g(\tau_2')\lim_{T\to\infty}\frac{1}{T}\int_0^T \mathbf{w}(t-\tau_2')\mathbf{w}^H(t-\tau_1')\,dt\Big]\,d\tau_1'\,d\tau_2'\\
&= \lim_{\sigma\to 0}\int_0^\infty\int_0^\infty \operatorname{trace}[g^H(\tau_1')\,g(\tau_2')\,R\,\delta^\sigma(\tau_2'-\tau_1')]d\tau_1'\,d\tau_2' = \int_0^\infty \operatorname{trace}[g^H(\tau)\,g(\tau)]\,d\tau = \|G(s)\|_2^2.
\end{aligned}$$

  Thus, *the square of the transfer function 2-norm is the* **expected mean energy** *of the output,* $\mathscr{E}\{\mathbf{z}^H(t)\mathbf{z}(t)\}$*, when the CT system is excited with a zero mean white random process* $\mathbf{w}(t)$ *with unit spectral density.*

Generalizing the definitions in §17.3.3.1 for the CT SISO case to CT MIMO systems via (20.34),

- a CT system is said to be **proper** if the norm of its transfer function is bounded as $\omega\to\infty$ [i.e., by (20.34), any CT system in state-space form], and
- a CT system is said to be **strictly proper** if its transfer function approaches zero as $\omega\to\infty$ [i.e., $D = 0$].

If a stable CT state-space system is strictly proper (that is, if $D = 0$), then its transfer function 2-norm is finite, and thus: (a) the output $\mathbf{z}(t)$ in the deterministic setting described above has finite total energy even though the input $\mathbf{w}(t)$ has infinite total energy, and (b) the output $\mathbf{z}(t)$ in the stochastic setting described above has finite mean energy even though the input $\mathbf{w}(t)$ has infinite mean energy.

[1] More precisely, $g_{ij}(t)$ is the (causal) response of the output $z_i(t)$ to a unit impulse on the input $w_j(t)$, taking the other inputs [$w_k(t)$ for $k \neq j$] as zero. This may be measured in a single experiment on a stable system with multiple inputs by first applying a unit impulse to $w_1(t)$, then waiting for the system response to settle back to essentially zero, then applying a unit impulse to $w_2(t)$, etc.

[2] We write $\mathscr{E}\{\mathbf{w}(t+\tau)\,\mathbf{w}^H(t)\}$ as $\lim_{T\to\infty}\frac{1}{T}\int_0^T \mathbf{w}(t+\tau)\,\mathbf{w}^H(t)\,dt$ for a stationary random process $\mathbf{w}(t)$ due to **ergodicity** (see §6).

Likewise, the **DT transfer function 2-norm**, $\|G(z)\|_2$, is defined for a DT system (20.44b) such that

$$\|G(z)\|_2^2 \triangleq \frac{h}{2\pi}\int_{-\pi/h}^{\pi/h}\|G(e^{\mathrm{i}\omega h})\|_F^2\,d\omega = \frac{1}{2\pi}\int_{-\pi}^{\pi}\|G(e^{\mathrm{i}\theta})\|_F^2\,d\theta = \frac{1}{2\pi}\int_{-\pi}^{\pi}\sum_i \sigma_i^2\big[G(e^{\mathrm{i}\theta})\big]\,d\theta. \tag{20.47}$$

If $G(z)$ is a consistent approximation[3] of $G(s)$, then $\|G(s)\|_2 = \lim_{h\to 0}\|G(z)\|_2/\sqrt{h}$. Thus, if $\|G(z)\|_2$ is nonzero as $h\to 0$, the corresponding $\|G(s)\|_2$ is infinite. The transfer function 2-norm of a stable DT MIMO system has, again, at least two natural interpretations:

- From a *deterministic* perspective, we take $\mathbf{w}_k$ as a sequence of unit impulses in each component and denote g_k as the associated impulse response matrix (a.k.a. the **Markov parameters** of the DT system; see §20.3.3). Applying the form of Parseval's theorem given in (5.71), noting the relation between the Fourier transform and the Z transform given in (17.5), applying causality, and applying the definition of the 2-norm of a DT signal (§1.3.3), it follows that

$$\|G(z)\|_2^2 = \frac{h}{2\pi}\mathrm{trace}\Big[\int_{-\pi/h}^{\pi/h} G^H(e^{\mathrm{i}\omega h})G(e^{\mathrm{i}\omega h})d\omega\Big] = \sum_{k=0}^{\infty}\mathrm{trace}\Big[g_k^H g_k\Big] = \|g_k\|_2^2. \tag{20.48a}$$

  Thus, *the square of the transfer function 2-norm is the* **total energy** *of the output* $\mathbf{z}_k$ *of the DT system when the input* $\mathbf{w}_k$ *contains a sequence of unit impulses in each component.*

- From a *stochastic* perspective, we take $\mathbf{w}_k$ as a zero-mean **DT random process** (§6.4) with separable autocorrelation $\mathscr{E}\{\mathbf{w}_{k+j}\mathbf{w}_k^H\} = \lim_{N\to\infty}\frac{1}{N}\sum_{k=0}^{N}\mathbf{w}_{k+j}\mathbf{w}_k^H = P\,\delta_{j0}$ (that is, a DT input $\mathbf{w}_k$ that is "white") and unit covariance $P = I$, and examine the mean energy of the output $\mathbf{z}_k$. Writing $\mathbf{z}_k$ as a convolution of the impulse response g_{k-i} and the input $\mathbf{w}_i$, changing variables and rearranging the sums appropriately, noting that $g_k\to 0$ exponentially as $k\to\infty$, and applying (20.48a), it follows that

$$\begin{aligned}
\mathscr{E}\{\mathbf{z}_k^H\mathbf{z}_k\} &= \lim_{N\to\infty}\frac{1}{N}\sum_{k=0}^{N}\mathbf{z}_k^H\mathbf{z}_k = \lim_{N\to\infty}\frac{1}{N}\sum_{k=0}^{N}\sum_{i=0}^{k}[g_{k-i}\mathbf{w}_i]^H\sum_{j=0}^{k}g_{k-j}\mathbf{w}_j = \lim_{N\to\infty}\frac{1}{N}\sum_{k=0}^{N}\sum_{i'=0}^{k}\sum_{j'=0}^{k}\mathbf{w}_{k-i'}^H g_{i'}^H g_{j'}\mathbf{w}_{k-j'}\\
&= \lim_{N\to\infty}\frac{1}{N}\sum_{k=0}^{N}\sum_{i'=0}^{\infty}\sum_{j'=0}^{\infty}\mathrm{trace}\Big[g_{i'}^H g_{j'}\mathbf{w}_{k-j'}\mathbf{w}_{k-i'}^H\Big] = \sum_{i'=0}^{\infty}\sum_{j'=0}^{\infty}\mathrm{trace}\Big[g_{i'}^H g_{j'}\lim_{N\to\infty}\frac{1}{N}\sum_{k=0}^{N}\mathbf{w}_{k-j'}\mathbf{w}_{k-i'}^H\Big]\\
&= \sum_{i'=0}^{\infty}\sum_{j'=0}^{\infty}\mathrm{trace}[g_{i'}^H g_{j'}P\,\delta_{j'-i',0}] = \sum_{k=0}^{\infty}\mathrm{trace}[g_k^H g_k] = \|G(z)\|_2^2.
\end{aligned} \tag{20.48b}$$

  Thus, *the square of the transfer function 2-norm is the* **expected mean energy** *of the output,* $\mathscr{E}\{\mathbf{z}_k^H\mathbf{z}_k\}$, *when the DT system is excited with a zero mean white random process* $\mathbf{w}_k$ *with unit covariance.*

As the integral in (20.47) is finite, a DT state-space system has finite transfer function 2-norm even if $D\neq 0$.

The **CT transfer function ∞-norm**, $\|G(s)\|_\infty$, is defined for a stable CT system (20.44a) such that

$$\|G(s)\|_\infty \triangleq \sup_{0\le\omega<\infty}\|G(\mathrm{i}\omega)\|_{i2} = \sup_{0\le\omega<\infty}\sigma_{\max}\Big[G(\mathrm{i}\omega)\Big], \tag{20.49a}$$

where $\|\cdot\|_{i2}$ denotes the induced 2-norm [§1.3.2] and $\sigma_{\max}[\cdot]$ denotes the maximum singular value [§4.4.7].

Likewise, **DT transfer function ∞-norm**, $\|G(z)\|_\infty$, is defined for a stable DT system (20.44b) such that

$$\|G(z)\|_\infty \triangleq \sup_{0\le\omega<\pi/h}\|G(e^{\mathrm{i}\omega h})\|_{i2} = \sup_{0\le\theta<\pi}\|G(e^{\mathrm{i}\theta})\|_{i2} = \sup_{0\le\theta<\pi}\sigma_{\max}\Big[G(e^{\mathrm{i}\theta})\Big]. \tag{20.49b}$$

The transfer function ∞-norm is finite for any type 0 (see §18.2.4.1), proper, CT or DT system.

[3] Note that $G(z)$ may be generated from $G(s)$ via, e.g., Tustin's rule (17.72).

The ∞-norm is the *peak* over all frequencies of the *largest singular value* of the transfer function G, whereas the 2-norm is the square root of the *integral* over all frequencies of the *sum of the squared singular values* of the transfer function G. Thus, though the former quantifies the *largest* singular value of the transfer function and the latter quantifies, in a way, *all* of the singular values of the transfer function, there is no clear relationship between the two (i.e., we can't bound one with a constant times the other).

The transfer function ∞-norm is a useful quantification of the "worst-case" amplification of disturbances by a stable system, and has at least two natural interpretations:

- In the *frequency* domain, if a (CT or DT) system is SISO, its transfer function ∞-norm is simply the maximum over all frequencies of the gain of the corresponding Bode plot. Recall that, in the MIMO case, a Bode plot may be drawn from any linear combination of inputs to any linear combination of outputs; the transfer function ∞-norm in the MIMO case thus quantifies the maximum over all frequencies of the gain of the corresponding Bode plot from the worst (i.e., "most disturbing") linear combination of inputs to the worst (i.e., "most sensitive") linear combination of outputs.
- In the *time* domain, it is straightforward to show that the infinity norm also quantifies the response of the stable CT or DT system to the "most disturbing" input $\mathbf{w}$ in the time domain; that is,

$$\|G(s)\|_\infty = \max_{\mathbf{w}(t)\neq 0} \frac{\|\mathbf{z}(t)\|_2}{\|\mathbf{w}(t)\|_2} = \max_{\|\mathbf{w}(t)\|_2=1} \|\mathbf{z}(t)\|_2 \qquad \text{and} \qquad \|G(z)\|_\infty = \max_{\mathbf{w}_k\neq 0} \frac{\|\mathbf{z}_k\|_2}{\|\mathbf{w}_k\|_2} = \max_{\|\mathbf{w}_k\|_2=1} \|\mathbf{z}_k\|_2 .$$

  These relations generalize the Rayleigh-Ritz Theorem (Fact 4.26) to time-varying signals, and their proof is similar: expanding the input as a linear combination of sinusoids[4] with unit 2-norm, the output with maximum 2-norm is given by a sinusoidal input at the "most disturbing" frequency with the "most disturbing" linear combination of inputs, as identified in the frequency-domain analysis above.

20.2.2.1 Computation of the transfer function 2-norm and ∞-norm of CT and DT systems†

Though the transfer function 2-norm and ∞-norm are defined unambiguously above, it is premature at this point in the text to derive from first principles how they may be calculated. For completeness, we include the relevant formulae here, with further explanations of these formulae deferred to the sections indicated.

The 2-norm of a stable, strictly proper ($D = 0$) CT system may be determined from the controllability or observability gramians via the following equivalent formulae (§20.5.1.2, §20.5.2.2):

$$\begin{aligned} 0 &= AP + PA^H + BB^H \quad &\Rightarrow \quad \|G(s)\|_2^2 &= \text{trace}(CPC^H), \\ 0 &= A^H Q + QA + C^H C \quad &\Rightarrow \quad \|G(s)\|_2^2 &= \text{trace}(B^H QB). \end{aligned}$$

Likewise, the 2-norm of a stable DT system may be found via the following formulae (§20.5.3, §20.5.4):

$$\begin{aligned} P &= FPF^H + GG^H \quad &\Rightarrow \quad \|G(z)\|_2^2 &= \text{trace}(HPH^H + DD^H), \\ Q &= F^H QF + H^H H \quad &\Rightarrow \quad \|G(z)\|_2^2 &= \text{trace}(G^H QG + D^H D). \end{aligned}$$

The ∞-norm of $G(s)$ is found via an iterative search; a simple algorithm for its computation (§**??**) is:

(1) Guess a value of $\gamma \geq \gamma_{\min} = \max(\|G(s=0)\|, \|G(s \to \mathrm{i}\infty)\|)$.

(2) Compute the eigenvalues of $Z = \begin{bmatrix} \tilde{A} & BR_1^{-1}B^H \\ -C^H R_2^{-1} C & -\tilde{A}^H \end{bmatrix}$ with $\begin{cases} R_1 = \gamma I - D^H D/\gamma, \\ R_2 = \gamma I - DD^H/\gamma, \\ \tilde{A} = A + BR_1^{-1} D^H C/\gamma. \end{cases}$

(3) $\|G(s)\|_\infty < \gamma$ if Z has no eigenvalues on the imaginary axis; modify γ accordingly, with $\gamma \geq \gamma_{\min}$, using the bisection algorithm of §3.1.2, and repeat from (2) until the bounds reach a desired tolerance.

[4] Via the infinite Fourier integral (17.1b) in the CT case, or the finite Fourier integral (17.1d) in the DT case.

Algorithm 20.4: Compute the maximum transient energy growth of a CT or DT system.

View
Test

```
function [thetamax,x0]=MaxEnergyGrowth(A,Q,tau,MODE)
% Compute the maximum possible growth of the quantity x^H Q x over the specified period.
if nargin==3, MODE='CT'; end, Qhi=Inv(sqrtm(Q));
if MODE=='CT', Phi=MatrixExponential(A,tau); else, Phi=A^tau; end
[lam,S]=Eig(Qhi*Phi'*Q*Phi*Qhi,'h');
thetamax=lam(end); x0=Qhi*S(:,end); x0=x0/sqrt(x0'*Q*x0);
end % function MaxEnergyGrowth
```

Algorithm 20.5: Compute the 2-norm or ∞-norm of a CT or DT system.

View
Test

```
function tfn=TFnorm(A,B,C,D,p,MODE)
% Compute the p norm of a transfer function for p='2' (default) or p='inf'
if nargin<6, MODE='CT'; end, if nargin<5, p='2'; end
[n,ni]=size(B); [no,n]=size(C); if nargin<4, D=zeros(no,ni); end
if p=='2', if (MODE=='CT' & norm(D)~=0), tfn=inf;
           else, PP=CtrbGrammian(A,B,MODE); tfn=sqrt(sum(diag(C*PP*C'+D'*D))); end
else, P.A=A; P.B=B; P.C=C; P.D=D; P.MODE=MODE; P.n=n; P.ni=ni; P.no=no;
  if MODE=='CT', P.gmin=max(norm(C*Inv(-A)*B+D),norm(D));
                 [x1,x2]=FindRootBracket(log10(P.gmin),log10(10*P.gmin),@CheckHamiltonian,P);
                 tfn=10^Bisection(x1,x2,@CheckHamiltonian,1e-6,0,P);
  else,          P.gmin=max(norm(C*Inv(eye(P.n)-A)*B+D),norm(C*Inv(-eye(P.n)-A)*B+D));
                 [x1,x2]=FindRootBracket(log10(P.gmin),log10(10*P.gmin),@CheckHamiltonian,P);
                 tfn=10^Bisection(x1,x2,@CheckHamiltonian,1e-6,0,P); end
end
end % function TFnorm
%%%
```

20.3 Canonical forms

20.3.1 The four continuous-time canonical forms

There are a variety of convenient ways to convert a proper transfer function of the form (17.55) into a (first-order, vector, SISO) continuous-time state-space form, as illustrated in (20.1). As shown below, four of the most convenient have natural interpretations in terms of simple block diagrams which facilitate their understanding. These **canonical forms** are easiest to understand by first developing them for systems for a given order; we thus develop each below for a CT SISO system with transfer function

$$G(s) = \frac{Y(s)}{U(s)} = \frac{b(s)}{a(s)}, \tag{20.50}$$

where $b(s)$ is a third-order polynomial in s and $a(s)$ is a third-order monic[6] polynomial in s. Noting the obvious patterns that develop, extrapolation to differential equations of any order n is straightforward. Note also that some of the resulting equations simplify when $b_n = 0$ (that is, when the system is strictly proper).

CT controller canonical form

Defining $w(t)$ such that $W(s) = U(s)/a(s)$, the following two relations are equivalent to (20.50)

$$b(s)W(s) = Y(s) \quad \Rightarrow \quad (b_3 s^3 + b_2 s^2 + b_1 s + b_0)W(s) = Y(s), \tag{20.51a}$$
$$a(s)W(s) = U(s) \quad \Rightarrow \quad (s^3 + a_2 s^2 + a_1 s + a_0)W(s) = U(s). \tag{20.51b}$$

Renaming $x_{3c}(t) = w(t)$ and defining the additional intermediate variables

$$x_{2c}(t) = x'_{3c}(t), \quad \Rightarrow \quad X_{2c}(s) = sX_{3c}(s), \tag{20.52a}$$
$$x_{1c}(t) = x'_{2c}(t), \quad \Rightarrow \quad X_{1c}(s) = sX_{2c}(s), \tag{20.52b}$$
$$v(t) = x'_{1c}(t), \quad \Rightarrow \quad V(s) = sX_{1c}(s), \tag{20.52c}$$

the relations (20.51a) and (20.51b) may be rewritten as

$$Y(s) = b_3 V(s) + b_2 X_{1c}(s) + b_1 X_{2c}(s) + b_0 X_{3c}(s), \tag{20.52d}$$
$$V(s) = -a_2 X_{1c}(s) - a_1 X_{2c}(s) - a_0 X_{3c}(s) + U(s). \tag{20.52e}$$

Together, the equations defined in (20.52) may easily be implemented in block diagram form, as illustrated in Figure 20.5a [note that equations (20.52a) through (20.52c) are implemented along the center row, whereas (20.52d) and (20.52e) are implemented at the summation junctions at the right and left respectively].

Defining $\mathbf{x}_c = \begin{pmatrix} x_{1c}(t) & x_{2c}(t) & x_{3c}(t) \end{pmatrix}^T$, writing (20.52e) [with $V(s)$ eliminated using (20.52c)] and (20.52b) and (20.52a) in matrix form, and writing (20.52d) [with $V(s)$ eliminated using (20.52e)] in matrix form, leads directly to the CT state-space form

$$\begin{aligned} \mathbf{x}'_c(t) &= A_c \mathbf{x}_c(t) + B_c u(t) \\ y(t) &= C_c \mathbf{x}_c(t) + D_c u(t) \end{aligned} \quad \text{with} \quad \left[\begin{array}{c|c} A_c & B_c \\ \hline C_c & D_c \end{array}\right] = \left[\begin{array}{ccc|c} -a_2 & -a_1 & -a_0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline b_2 - a_2 b_3 & b_1 - a_1 b_3 & b_0 - a_0 b_3 & b_3 \end{array}\right]. \tag{20.53}$$

Due to the simple structure of B_c and the corresponding structure of A_c in **top companion form**, which in turn makes pole-placement controller design particularly easy (see §20.4.1), this realization is called the **continuous-time controller canonical form**.

[6] A **monic polynomial** is a polynomial with a leading coefficient of one.

CT reachability canonical form

Now define the matrix R_1 as the lower triangular Toeplitz matrix with first column $\begin{pmatrix}1 & a_{n-1} & \dots & a_1\end{pmatrix}^T$,

$$R_1 = \begin{pmatrix} 1 & & & & 0 \\ a_{n-1} & 1 & & & \\ a_{n-2} & a_{n-1} & 1 & & \\ \vdots & \ddots & \ddots & \ddots & \\ a_1 & \dots & a_{n-2} & a_{n-1} & 1 \end{pmatrix}, \tag{20.54}$$

where in the present case $n = 3$. Defining $R = R_1^{-T}$ and $\mathbf{x}_{re} = R^{-1}\mathbf{x}_c$ and applying the state transformation (20.3) to the state-space realization (20.128) leads directly to the state-space form

$$\begin{aligned} \mathbf{x}'_{re}(t) &= A_{re}\mathbf{x}_{re}(t) + B_{re}u(t) \\ y(t) &= C_{re}\mathbf{x}_{re}(t) + D_{re}u(t) \end{aligned} \quad \text{with} \quad \left[\begin{array}{c|c} A_{re} & B_{re} \\ \hline C_{re} & D_{re} \end{array}\right] = \left[\begin{array}{ccc|c} 0 & 0 & -a_0 & 1 \\ 1 & 0 & -a_1 & 0 \\ 0 & 1 & -a_2 & 0 \\ \hline m_1 & m_2 & m_3 & m_0 \end{array}\right], \tag{20.55}$$

where the m_i coefficients [referred to as the first four **Markov parameters** of this CT SISO system] are

$$\begin{aligned} m_0 = b_3, \quad m_1 = b_2 - b_3 a_2, \quad m_2 = b_1 - b_2 a_2 + b_3(a_2^2 - a_1), \\ m_3 = b_0 - b_1 a_2 + b_2(a_2^2 - a_1) - b_3(a_2^3 - 2a_2 a_1 + a_0). \end{aligned} \tag{20.56}$$

The reader should easily be able to trace the implementation of (20.55) in Figure 20.5b.

Due to the simple structure of B_{re} and the structure of A_{re} in **right companion form**, which in turn makes the (equivalent) questions of

- CT reachability [that is, the question of whether or not it is possible to find a control input $\mathbf{u}(t)$ on $t \in [0,T]$ to take a CT system from any given initial state to any given final state], and
- CT controllability [that is, the question of whether or not it is possible to find a control input $\mathbf{u}(t)$ on $t \in [0,T]$ to take a CT system from any given initial state to a *zero* final state]

easy to address (see §20.5.1), this state-space realization may be called either the **continuous-time reachability canonical form** or the **continuous-time controllability canonical form**[7].

To illustrate the connection of this realization to CT reachability, note in particular that,

$$\text{if} \quad \mathbf{x}_{re}(\tau) = 0, \quad \text{then} \quad \mathbf{x}'_{re}(\tau) = \begin{pmatrix} u(\tau) \\ u'(\tau) \\ u''(\tau) \end{pmatrix}. \tag{20.57}$$

CT observer canonical form

It follows directly from (20.50) that

$$b(s)U(s) = a(s)Y(s) \quad \Rightarrow \quad (b_3 s^3 + b_2 s^2 + b_1 s + b_0)U(s) = (s^3 + a_2 s^2 + a_1 s + a_0)Y(s). \tag{20.58}$$

Defining the additional intermediate variables

$$x'_{3o}(t) = b_0 u(t) - a_0 y(t) \quad \Rightarrow \quad sX_{3o}(s) = b_0 U(s) - a_0 Y(s), \tag{20.59a}$$
$$x'_{2o}(t) = b_1 u(t) - a_1 y(t) + x_{3o}(t) \quad \Rightarrow \quad sX_{2o}(s) = b_1 U(s) - a_1 Y(s) + X_{3o}(s), \tag{20.59b}$$
$$x'_{1o}(t) = b_2 u(t) - a_2 y(t) + x_{2o}(t) \quad \Rightarrow \quad sX_{1o}(s) = b_2 U(s) - a_2 Y(s) + X_{2o}(s), \tag{20.59c}$$

[7] To avoid confusion when comparing with the discrete-time case, in which the classifications of reachability and controllability are *not* equivalent (and for which the canonical forms making these distinct classifications easy to perform are different), we recommend consistent use of the name **continuous-time reachability canonical form** for this realization.

and taking (20.58) plus (20.59a) plus s times (20.59b) plus s^2 times (20.59c), and dividing by s^3, gives

$$b_3 U(s) + X_{1o}(s) = Y(s). \tag{20.59d}$$

Together, the equations defined in (20.59) may easily be implemented in block diagram form, as illustrated in Figure 20.5c [note that equations (20.59a) through (20.59c) are implemented along the center row, whereas (20.59d) is implemented at the summation junction at the right].

Defining $\mathbf{x}_o = \begin{pmatrix} x_{1o}(t) & x_{2o}(t) & x_{3o}(t) \end{pmatrix}^T$, writing (20.59c) and (20.59b) and (20.59a) [with $Y(s)$ eliminated using (20.59d)] in matrix form, and writing (20.59d) in matrix form, leads to the CT state-space form

$$\begin{aligned} \mathbf{x}_o'(t) &= A_o \mathbf{x}_o(t) + B_o u(t) \\ y(t) &= C_o \mathbf{x}_o(t) + D_o u(t) \end{aligned} \quad \text{with} \quad \left[\begin{array}{c|c} A_o & B_o \\ \hline C_o & D_o \end{array}\right] = \left[\begin{array}{ccc|c} -a_2 & 1 & 0 & b_2 - a_2 b_3 \\ -a_1 & 0 & 1 & b_1 - a_1 b_3 \\ -a_0 & 0 & 0 & b_0 - a_0 b_3 \\ \hline 1 & 0 & 0 & b_3 \end{array}\right]. \tag{20.60}$$

As $A_o = A_c^T$, $B_o = C_c^T$, $C_o = B_c^T$, and $D_o = D_c^T$, the forms (20.128) and (20.60) are said to be **dual**.

Due to the simple structure of C_o and the corresponding structure of A_o in **left companion form**, which in turn makes pole-placement observer design easy (see §20.4.2), this realization is called the **continuous-time observer canonical form**.

CT observability canonical form

Defining $R = R_1$ [see (20.54)] and $\mathbf{x}_{ob} = R^{-1}\mathbf{x}_o$ and applying the state transformation (20.3) to the state-space realization (20.60) leads directly to the state-space form

$$\begin{aligned} \mathbf{x}_{ob}'(t) &= A_{ob} \mathbf{x}_{ob}(t) + B_{ob} u(t) \\ y(t) &= C_{ob} \mathbf{x}_{ob}(t) + D_{ob} u(t) \end{aligned} \quad \text{with} \quad \left[\begin{array}{c|c} A_{co} & B_{co} \\ \hline C_{co} & D_{co} \end{array}\right] = \left[\begin{array}{ccc|c} 0 & 1 & 0 & m_1 \\ 0 & 0 & 1 & m_2 \\ -a_0 & -a_1 & -a_2 & m_3 \\ \hline 1 & 0 & 0 & m_0 \end{array}\right], \tag{20.61}$$

where the Markov parameters $\{m_0, m_1, m_2, m_3\}$ are given in (20.56). The reader should easily recognize the implementation of (20.61) in Figure 20.5d.

As $A_{ob} = A_{co}^T$, $B_{ob} = C_{co}^T$, $C_{ob} = B_{oc}^T$, and $D_{ob} = D_{co}^T$, the forms (20.55) and (20.61) are said to be **dual**.

Due to the simple structure of C_{ob} and the structure of A_{ob} in **bottom companion form**, which in turn makes the question of CT observability [that is, the question of whether or not it is possible to reconstruct the initial state $\mathbf{x}_{ob}(0)$ and the final state $\mathbf{x}_{ob}(T)$ from the observations $y(t)$ on $t \in [0, T]$] particularly easy to address (see §20.5.2), this realization is called the CT **observability canonical form**.

To illustrate the connection of this realization to CT observability, note in particular that,

$$\text{if} \quad u(\tau) = 0, \quad \text{then} \quad \mathbf{x}_{ob}(\tau) = \begin{pmatrix} y(\tau) \\ y'(\tau) \\ y''(\tau) \end{pmatrix}. \tag{20.62}$$

20.3.2 The six discrete-time canonical forms

There are a variety of convenient ways to convert a high-order, scalar difference system of the form (17.64) into a (first-order, vector, SISO) DT state-space form, as defined by (20.2). We illustrate the development of six such **canonical forms** below for a system from an input u_k to an output y_k with a transfer function

$$G(z) = \frac{Y(z)}{U(z)} = \frac{b(z)}{a(z)}, \tag{20.63}$$

where $b(z)$ is a third-order polynomial in z and $a(z)$ is a third-order monic polynomial in z. Noting the obvious patterns that develop, extrapolation to difference systems of higher order is straightforward.

The derivation of four of the DT canonical forms below follow exactly as in the CT case, with the role of z replacing that of s in a manner analogous to that seen by comparing §17.3.3 to §17.4.3; these forms are thus summarized only briefly below. In addition, there are two extra canonical forms in the DT case related to the DT controllability and DT observability problems, which are presented in a bit more detail.

DT controller canonical form

Following an analogous derivation as that leading to (20.128), with the role of z replacing that of s, it follows that the DT system (20.63) may be written

$$\begin{aligned} \mathbf{x}_{c,k+1} &= F_c\mathbf{x}_{c,k} + G_c u_k \\ y_k &= H_c\mathbf{x}_{c,k} + D_c u_k \end{aligned} \quad \text{with} \quad \left[\begin{array}{c|c} F_c & G_c \\ \hline H_c & D_c \end{array}\right] = \left[\begin{array}{ccc|c} -a_2 & -a_1 & -a_0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline b_2 - a_2 b_3 & b_1 - a_1 b_3 & b_0 - a_0 b_3 & b_3 \end{array}\right]. \tag{20.64}$$

Due to the simple structure of G_c and the corresponding structure of F_c in top companion form, which in turn makes pole-placement control design particularly easy (see §20.4), this realization is called the **discrete-time controller canonical form**.

DT reachability canonical form

Following an analogous derivation as that leading to (20.55), it follows that (20.63) may also be written

$$\begin{aligned} \mathbf{x}_{re,k+1} &= F_{re}\mathbf{x}_{re,k} + G_{re} u_k \\ y_k &= H_{re}\mathbf{x}_{re,k} + D_{re} u_k \end{aligned} \quad \text{with} \quad \left[\begin{array}{c|c} F_{re} & G_{re} \\ \hline H_{re} & D_{re} \end{array}\right] = \left[\begin{array}{ccc|c} 0 & 0 & -a_0 & 1 \\ 1 & 0 & -a_1 & 0 \\ 0 & 1 & -a_2 & 0 \\ \hline m_1 & m_2 & m_3 & m_0 \end{array}\right], \tag{20.65}$$

where, again, the Markov parameters $\{m_0, m_1, m_2, m_3\}$ are given in (20.56).

Due to the simple structure of G_{re} and the structure of F_{re} in right companion form, which in turn makes the question of **discrete-time reachability** [that is, the question of whether or not it is possible to find a control input sequence $\mathbf{u}_k$ for $k = 0, \ldots, n-1$ that steers a DT system from any given initial state $\mathbf{x}_{re,0}$ to any given final state $\mathbf{x}_{re,n}$] particularly easy to address (see §20.5.3), this realization is called the **discrete-time reachability canonical form**.

To illustrate the connection of this realization to DT reachability, note in particular that,

$$\text{if} \quad \mathbf{x}_{re,0} = 0, \quad \text{then} \quad \mathbf{x}_{re,3} = \begin{pmatrix} u_2 \\ u_1 \\ u_0 \end{pmatrix}. \tag{20.66}$$

DT controllability canonical form

Now define the matrix R_2 as the Hankel matrix

$$R_2 = \begin{pmatrix} a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \\ a_{n-2} & a_{n-3} & \cdots & a_0 & \\ a_{n-3} & \cdots & a_0 & & \\ \vdots & \iddots & & & \\ a_0 & & & & 0 \end{pmatrix}, \tag{20.67}$$

where in the present case $n=3$. Defining $R=-R_2^{-1}$ and $\mathbf{x}_{co,k}=R^{-1}\mathbf{x}_{c,k}$ and applying the state transformation (20.3) to the state-space realization (20.64) leads directly to the DT state-space form

$$\begin{aligned}\mathbf{x}_{co,k+1}&=F_{co}\mathbf{x}_{co,k}+G_{co}u_k\\ y_k&=H_{co}\mathbf{x}_{co,k}+D_{co}u_k\end{aligned}\quad\text{with}\quad\left[\begin{array}{c|c}F_{co}&G_{co}\\\hline H_{co}&D_{co}\end{array}\right]=\left[\begin{array}{ccc|c}-a_2&1&0&-a_2\\-a_1&0&1&-a_1\\-a_0&0&0&-a_0\\\hline\gamma_1&\gamma_2&\gamma_3&m_0\end{array}\right],\tag{20.68a}$$

where $m_0=b_3$ and the γ_i coefficients work out to be

$$\gamma_1=(-b_0+b_3a_0)/a_0,\quad \gamma_2=(b_0a_1-b_1a_0)/a_0^2,\quad \gamma_3=(-a_0^2b_2+a_1a_0b_1+b_0a_2a_0-b_0a_1^2)/a_0^3.\tag{20.68b}$$

The reader should easily recognize the implementation of (20.68) in Figure 20.6a.

Due to the corresponding structures of G_{co} and F_{co} in left companion form, which in turn makes the question of **discrete-time controllability** [that is, the question of whether or not it is possible to find a control input sequence $\mathbf{u}_k$ for $k=0,\ldots,n-1$ that steers a DT system from any given initial state $\mathbf{x}_{co,0}$ to a *zero* final state $\mathbf{x}_{co,n}=0$] particularly easy to address (see §20.5.3.1), this realization is called the **discrete-time controllability canonical form**.

To illustrate the connection of this realization to DT controllability, note in particular that,

$$\text{if}\quad\left\{\begin{array}{l}u_0=-x_{1co,0}\\u_1=-x_{2co,0}\\u_2=-x_{3co,0}\end{array}\right\},\quad\text{then}\quad\mathbf{x}_{co,3}=0.\tag{20.69}$$

Note also that DT reachability implies DT controllability, but not visa-versa unless F is nonsingular. In CT, there is no distinction between the concepts of reachability and controllability, as e^{At} is nonsingular for any A (see Fact 20.2); however, when $|F|=0$, these distinctions are significant in DT.

DT observer canonical form

Following an analogous derivation as that leading to (20.60), it follows that (20.63) may also be written

$$\begin{aligned}\mathbf{x}_{o,k+1}&=F_o\mathbf{x}_{o,k}+G_ou_k\\ y_k&=H_o\mathbf{x}_{c,k}+D_ou_k\end{aligned}\quad\text{with}\quad\left[\begin{array}{c|c}F_o&G_o\\\hline H_o&D_o\end{array}\right]=\left[\begin{array}{ccc|c}-a_2&1&0&b_2-a_2b_3\\-a_1&0&1&b_1-a_1b_3\\-a_0&0&0&b_0-a_0b_3\\\hline 1&0&0&b_3\end{array}\right].\tag{20.70}$$

Due to the simple structure of G_o and the corresponding structure of F_o in left companion form, which in turn makes pole-placement observer design particularly easy (see §20.4), this realization is called the **discrete-time observer canonical form**.

DT observability canonical form

Following an analogous derivation as that leading to (20.61), it follows that (20.63) may also be written

$$\begin{aligned}\mathbf{x}_{ob,k+1}&=F_{ob}\mathbf{x}_{ob,k}+G_{ob}u_k\\ y_k&=H_{ob}\mathbf{x}_{ob,k}+D_{ob}u_k\end{aligned}\quad\text{with}\quad\left[\begin{array}{c|c}F_{ob}&G_{ob}\\\hline H_{ob}&D_{ob}\end{array}\right]=\left[\begin{array}{ccc|c}0&1&0&m_1\\0&0&1&m_2\\-a_0&-a_1&-a_2&m_3\\\hline 1&0&0&m_0\end{array}\right]\tag{20.71}$$

where, again, the Markov parameters $\{m_0,m_1,m_2,m_3\}$ are given in (20.56).

Algorithm 20.6: Converting SISO, SIMO, and MISO transfer functions into the state-space canonical forms.

View

```
function [A,B,C,D]=TF2SS(b,a,FORM)
% Convert a proper SISO, SIMO, or MISO CT or DT transfer function to one of the
% canonical state-space forms.  SIMO is handled with controller canonical form, MISO is
% handled with observer canonical form, and SISO is handled with the FORM specified.
b=b/a(1); a=a/a(1); p=size(a,2); n=p-1; if nargin<3, form='Controller'; end, s=size(b);
if size(s,1)==2                          % Standardize inputs for the various cases.
  if (s(1)>1), FORM='Controller'; end    % SIMO case.
  if size(s,2)==3
    if s(2)==1, b=reshape(b,s(1),s(3));  % Restructure b for SISO case if necessary.
    elseif s(1)==1, FORM='Observer'; end % MISO case.
end, end
m=size(b,2); b=[zeros(1,p-m) b]; switch FORM
  case 'Controller'
    A=[-a(2:p); eye(n-1,n)]; B=eye(n,1); C=b(:,2:p)-b(:,1)*a(2:p); D=b(:,1);
  case 'Reachability'
    A=[[zeros(1,n-1); eye(n-1)] -a(p:-1:2)'];
    m=TF2Markov(b,a); B=eye(n,1); C=m(2:p)'; D=m(1);
  case 'DTControllability'
    A=[-a(2:p)' eye(n,n-1)]; B=-a(2:p)'; D=b(1);
    R=Hankel(a(2:p),zeros(1,n)); R=-inv(R); C=( b(2:p)-a(2:p)*b(1) )*R;
  case 'Observer'
    A=[-a(2:p)' eye(n,n-1)]; B=b(2:p)'-a(2:p)'*b(1); C=eye(1,n); D=b(:,1);
  case 'Observability'
    A=[zeros(n-1,1) eye(n-1); -a(p:-1:2)];
    m=TF2Markov(b,a); B=m(2:p); C=eye(1,n); D=m(1);
  case 'DTConstructibility'
    A=[-a(2:p); eye(n-1,n)]; C=-a(2:p); D=b(1);
    R=Hankel(a(2:p),zeros(1,n)); R=-R; B=inv(R)*( b(2:p)'-a(2:p)'*b(1) );
end
end % function TF2SS
```

Due to the simple structure of H_{ob} and the structure of F_{ob} in bottom companion form, which in turn makes the question of **discrete-time observability** [that is, the question of whether or not it is possible to reconstruct the initial state $\mathbf{x}_{ob,0}$ and the final state $\mathbf{x}_{ob,n}$ from the observations y_0 through y_{n-1}] particularly easy to address (see §20.5.4), this realization is called the **discrete-time observability canonical form**.

To illustrate the connection of this realization to DT observability, note in particular that,

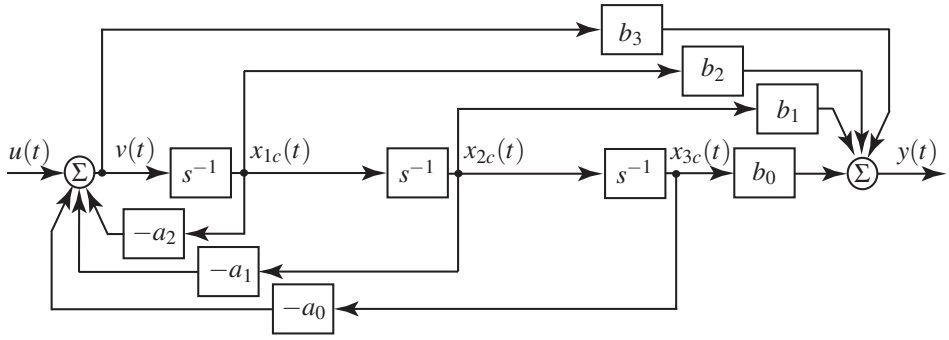
$$\text{if} \quad u=0, \quad \text{then} \quad \mathbf{x}_{ob,0}=\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}. \tag{20.72}$$

DT constructibility canonical form

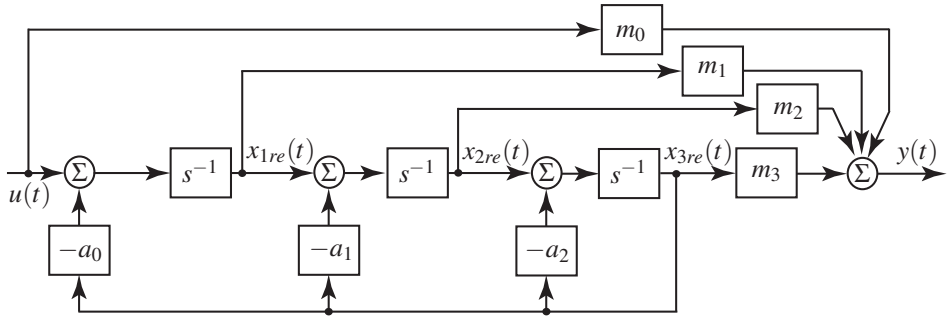
Defining $R=-R_2$ [see (20.67)] and applying the state transformation (20.3) to the state-space realization (20.70) leads directly to the state-space form

$$\begin{aligned} \mathbf{x}_{cs,k+1} &= F_{cs}\mathbf{x}_{cs,k}+G_{cs}u_k \\ y_k &= H_{cs}\mathbf{x}_{cs,k}+D_{cs}u_k \end{aligned} \quad \text{with} \quad \left[\begin{array}{c|c} F_{cs} & G_{cs} \\ \hline H_{cs} & D_{cs} \end{array}\right] = \left[\begin{array}{ccc|c} -a_2 & -a_1 & -a_0 & \gamma_1 \\ 1 & 0 & 0 & \gamma_2 \\ 0 & 1 & 0 & \gamma_3 \\ \hline -a_2 & -a_1 & -a_0 & m_0 \end{array}\right] \tag{20.73}$$

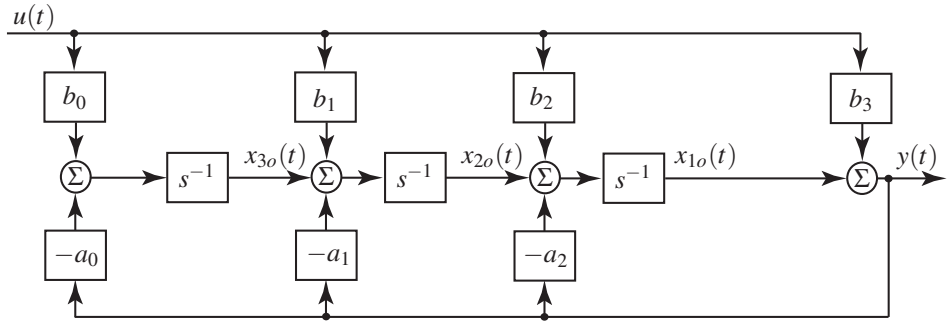
where, as before, $m_0=b_3$ and the γ_i coefficients are given in (20.68b). The reader should easily recognize the implementation of (20.68) in Figure 20.6b.



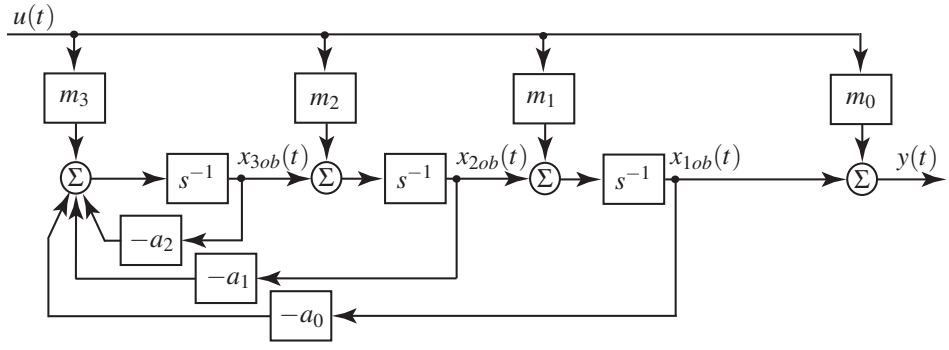
a) CT controller canonical form (20.128).



b) CT reachability canonical form (20.55).

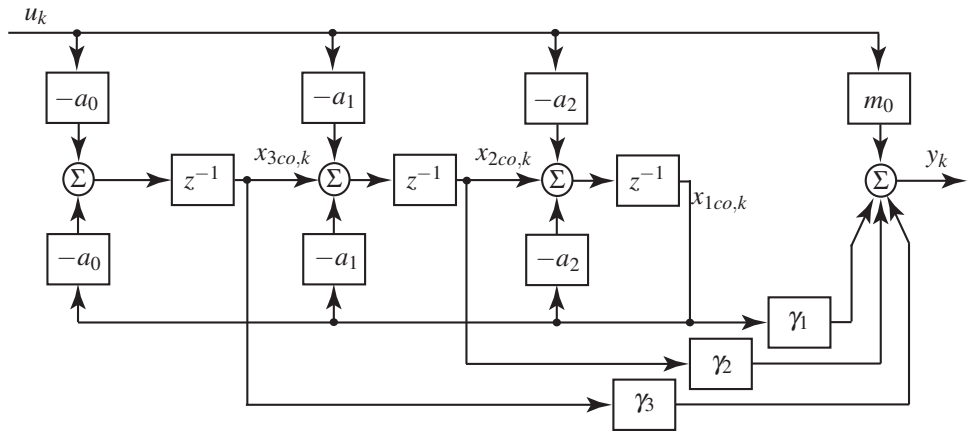


c) CT observer canonical form (20.60).

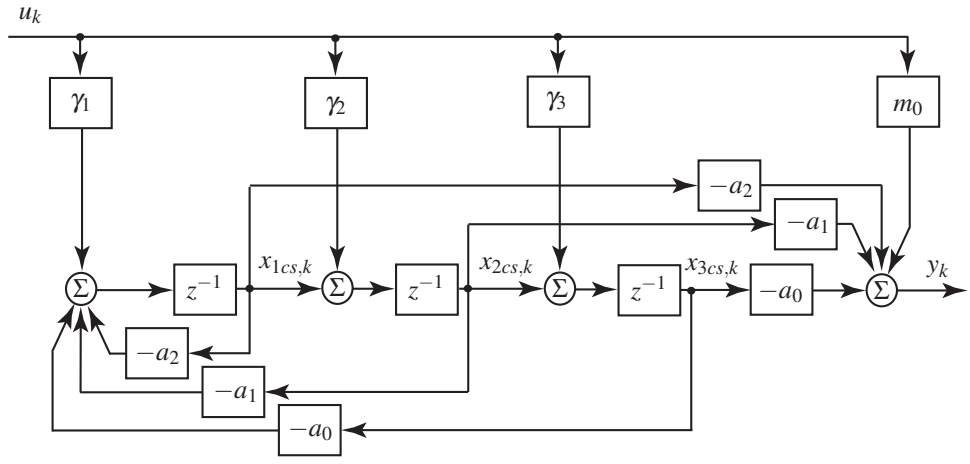


d) CT observability canonical form (20.61).

Figure 20.5: Block diagram representations of the four CT canonical forms. The four corresponding DT canonical forms are identical in structure, with the integrator (s^{-1}) blocks replaced by delay (z^{-1}) blocks.



a) DT controllability canonical form (20.68).



b) DT constructibility canonical form (20.73).

Figure 20.6: Block diagram representations of the two additional DT canonical forms.

As $F_{cs} = F_{co}^T$, $G_{cs} = H_{co}^T$, $H_{cs} = G_{co}^T$, and $D_{cs} = D_{co}^T$, the forms (20.68) and (20.73) are said to be **dual**.

Due to the corresponding structures of H_{cs} and F_{cs} in top companion form, which in turn makes the question of **discrete-time constructibility** [that is, the question of whether or not it is possible to reconstruct the *final* state $\mathbf{x}_{cs,n}$ from the observations y_0 through y_{n-1}] particularly easy to address (see §20.5.4.1), this realization is called the **discrete-time constructibility canonical form**.

To illustrate the connection of this realization to DT constructibility, note in particular that,

$$\text{if} \quad u = 0, \quad \text{then} \quad \mathbf{x}_{cs,3} = \begin{pmatrix} y_2 \\ y_1 \\ y_0 \end{pmatrix}. \tag{20.74}$$

Note also that DT observability implies DT constructibility, but not visa-versa unless F is nonsingular. In CT, there is no distinction between the concepts of observability and constructibility, as e^{At} is nonsingular for any A (see Fact 20.2); however, when $|F| = 0$, these distinctions are significant in DT.

A code which converts a proper CT or DT transfer function into any of the canonical state-space forms discussed above is provided in Algorithm 20.6.

20.3.3 Markov parameters

Discrete-time case

The **Markov parameters** of a DT MIMO system are defined as the coefficient matrices M_k of the Taylor series expansion of the transfer function of the system in the variable $1/z$ such that

$$G(z) = \sum_{k=0}^{\infty} \frac{M_k}{z^k}. \tag{20.75a}$$

If the transfer function is derived from a MIMO state-space representation of the form (20.2), such that $G(z) = H(zI - F)^{-1}G + D$ [see (20.37)], it follows from the series expansion (20.38) that

$$M_0 = D \quad \text{and} \quad M_k = HF^{k-1}G \ \text{ for } \ k \geq 1. \tag{20.75b}$$

This result is trivial to implement in code, as illustrated in Algorithm 20.7. Note that the Markov parameters of a DT system are *invariant* under any state transformation (20.3), as

$$H_R F_R^k G_R = (HR)(R^{-1}FR)^k(R^{-1}G) = HF^kG. \tag{20.76}$$

If the transfer function is derived from a SISO difference equation of the form (17.64), such that $G(z) = b(z)/a(z)$ [see (17.65)], the (scalar) coefficients m_k in the series expansion (20.75a) turn out to be rather involved functions of the coefficients a_i and b_i in the polynomials $a(z)$ and $b(z)$. [For the SISO $n = 3$ case (20.63), the first four (scalar) Markov parameters m_k are written out in (20.56).] These Markov parameters appear in a simple fashion in the DT reachability and observability canonical forms.

For SIMO systems, taking the input as a unit impulse [that is, $u_k = \delta_{k0}$ and thus $U(z) = 1$], it follows from (20.75a) and the definition of the Z transform (17.58a) that

$$\mathbf{Y}(z) = \mathbf{G}(z)U(z) = \mathbf{G}(z) = \sum_{k=0}^{\infty} \frac{\mathbf{m}_k}{z^k} = \sum_{k=0}^{\infty} \frac{\mathbf{y}_k}{z^k} \quad \Leftrightarrow \quad \mathbf{y}_k = \mathbf{m}_k;$$

that is, the k'th DT Markov parameter is simply the k'th value in the impulse response sequence. This result is also trivial to implement in code, as illustrated in Algorithm 20.8.

For MIMO systems, the k'th value in the response of the system to a unit impulse on the j'th input is given by the j'th column of the Markov parameter M_k in an identical fashion.

Continuous-time case

Analogous to the DT case, the **Markov parameters** of a CT MIMO system are defined as the coefficient matrices M_k of the Taylor series expansion of the transfer function of the system in the variable $1/s$ such that

$$G(s) = \sum_{k=0}^{\infty} \frac{M_k}{s^k}. \tag{20.77a}$$

If the transfer function is derived from a MIMO state-space representation of the form (20.1), such that $G(s) = C(sI - A)^{-1}B + D$ [see (20.30)], it follows from the series expansion (20.32) that

$$M_0 = D \quad \text{and} \quad M_k = CA^{k-1}B \ \text{ for } \ k \geq 1. \tag{20.77b}$$

This result is identical to that coded previously, in Algorithm 20.7. As in the DT case (20.76), the Markov parameters of a continuous-time system are *invariant* under any state transformation (20.3).

If the transfer function is derived from a SISO differential equation of the form (17.54), such that $G(s) = b(s)/a(s)$ [see (17.55)], the (scalar) coefficients m_k in the series expansion (20.77) turn out to be rather

Algorithm 20.7: Code to compute the n leading Markov parameters of a system in state-space form.

View

```
function [m]=SS2Markov(A,B,C,D,p)
% Compute the first p Markov parameters of a CT or DT MIMO system in SS form.
m(1,1)=D; for k=2:p, m(k,1)=C*A^(k-2)*B; end
end % function SS2Markov
```

Algorithm 20.8: Code to compute the n leading Markov parameters of a system in transfer-function form.

View

```
function [m]=TF2Markov(b,a)
% Compute the first n Markov parameters of a CT or DT SISO system in TF form.
n=length(a); m=length(b); b=[zeros(1,n-m) b]; u=[1; zeros(n-1,1)]; y=zeros(n,1);
for k=1:n, y(2:n)=y(1:n-1); y(1)=b*u-a(1,2:n)*y(2:n,1); u=[0; u(1:n-1)]; end; m=y(n:-1:1);
end % function TF2Markov
```

involved functions of the coefficients a_i and b_i in the polynomials $a(s)$ and $b(s)$. Note that, for SISO systems, they may be determined by calculating $R_1^{-1}B_o$ or $C_cR_1^{-T}$, as shown in the equations leading to (20.55) and (20.61), respectively. [For the SISO $n=3$ case (20.63), the first four (scalar) Markov parameters m_k are written out in (20.56).] These Markov parameters appear in a simple fashion in the CT reachability and observability canonical forms.

For SIMO systems, taking the input as a unit impulse [that is, taking $u(t)=\delta^\sigma(t)$ for small σ, and thus $U(s)\approx 1$], it follows from (20.77) that[8]

$$\mathbf{Y}(s)=\mathbf{G}(s)U(s)\approx\mathbf{G}(s)=\sum_{k=0}^{\infty}\frac{\mathbf{m}_k}{s^k}\quad\Leftrightarrow\quad\mathbf{y}(t)=\mathbf{m}_0\delta^\sigma(t)+\sum_{k=1}^{\infty}\mathbf{m}_k\frac{t^{k-1}}{(k-1)!},\quad\mathbf{m}_k=\left.\frac{d^{k-1}\mathbf{y}(t)}{dt^{k-1}}\right|_{t=0^+}\text{ for } k\geq 1;$$

note that, by the continuous-time initial value theorem (Fact 17.5), the impulse response of the system, $y(t)$, and the time derivatives of this impulse response, [$y'(t)$, $y''(t)$, etc.] evaluated at $t=0$ are given by the continuous-time Markov parameters. The algebraic relationship between the coefficients of the polynomials $a(s)$ and $b(s)$ and the continuous-time Markov parameters $\mathbf{m}_k$ are identical to the corresponding relationship in the DT case, and thus may again be determined via Algorithm 20.8.

For MIMO systems with $M_0=0$, the k'th time derivative of the response of the system to a unit impulse on the j'th input is given by the j'th column of the Markov parameter M_k in an identical fashion.

The CT and DT Markov parameters are reconciled in Exercise 20.8.

20.4 Feedback design via pole placement

We now introduce the ideas of controller and observer design with a simplistic approach based solely on placing the eigenvalues, or poles, of the **closed-loop** LTI system matrix[9] at specified, "favorable" (i.e., sufficiently stable) locations. As motivated in §20.2, some method of accounting for the orthogonality (or lack thereof) of the closed-loop system eigenvectors, as well as the magnitude of the control effort applied, is desired in feedback design, though such a capability is completely lacking in the pole placement method. Stated another way, the pole placement method can situate the closed-loop eigenvalues wherever you specify, but provides no guides to select *where* the several eigenvalues should go. Rest assured, the state-space control and estimation techniques presented in §§21-22 address these shortcomings. Regardless, pole placement is the natural starting point for introducing control and observer design for systems in state-space form, and is thus worth a brief introduction here. As the DT case is analogous to the CT case with the usual modifications (e.g., with the role of z replacing that of s, etc.), we discuss only the CT case here.

[8] Note that the output, the transfer function, and the Markov parameters in this case are vectors.

[9] That is, the matrix governing the dynamics of the LTI system after the feedback is incorporated.

20.4.1 Controller design

The **pole placement** approach to controller design is simply to select a **feedback gain matrix** K such that, in the controlled system

$$\mathbf{x}'(t) = A\mathbf{x}(t) + B\mathbf{u}(t), \quad \mathbf{u}(t) = K\mathbf{x}(t) \quad \Rightarrow \quad \mathbf{x}'(t) = (A + BK)\mathbf{x}(t),$$

the **closed-loop system matrix** $(A + BK)$ has its eigenvalues in prespecified stable locations (in the LHP).

In the SISO case, if the system is expressed in controller canonical form (20.128), solving this problem is particularly easy. For example, in the third-order SISO case, the transfer function

$$G(s) = \frac{b(s)}{s^3 + a_2 s^2 + a_1 s + a_0}$$

is equivalent to the controller canonical form

$$\mathbf{x}_c'(t) = A_c\mathbf{x}_c(t) + B_c u(t) \qquad \text{where} \qquad A_c = \begin{pmatrix} -a_2 & -a_1 & -a_0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad B_c = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

Recall that the eigenvalues of an **open-loop system** (i.e., with $u = 0$) are given by the roots of the characteristic polynomial of the system matrix. As A_c is in top companion form, this characteristic polynomial is

$$\lambda^3 + a_2\lambda^2 + a_1\lambda + a_0 = 0.$$

Now consider the feedback control problem in which we seek feedback

$$u = K\mathbf{x}_c = \begin{pmatrix} k_1 & k_2 & k_3 \end{pmatrix} \begin{pmatrix} x_{c,1} \\ x_{c,2} \\ x_{c,3} \end{pmatrix}$$

in order to place the closed-loop poles at the desired locations $\{\overline{\lambda}_1, \overline{\lambda}_2, \overline{\lambda}_3\}$; that is, in order that the closed-loop system matrix of the controlled system, $(A_c + B_c K)$, have a characteristic equation

$$(\lambda - \overline{\lambda}_1)(\lambda - \overline{\lambda}_2)(\lambda - \overline{\lambda}_3) = \lambda^3 - (\overline{\lambda}_1 + \overline{\lambda}_2 + \overline{\lambda}_3)\lambda^2 + (\overline{\lambda}_1\overline{\lambda}_2 + \overline{\lambda}_1\overline{\lambda}_3 + \overline{\lambda}_2\overline{\lambda}_3)\lambda - (\overline{\lambda}_1\overline{\lambda}_2\overline{\lambda}_3) = 0. \quad (20.78a)$$

Noting that $(A_c + B_c K)$ is itself in top companion form, its characteristic polynomial is given by

$$(A_c + B_c K) = \begin{pmatrix} k_1 - a_2 & k_2 - a_1 & k_3 - a_0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \Rightarrow \quad \lambda^3 - (k_1 - a_2)\lambda^2 - (k_2 - a_1)\lambda - (k_3 - a_0) = 0. \quad (20.78b)$$

Matching the coefficients of like powers of λ in the polynomials (20.78a) and (20.78b), simple expressions for $\{k_1, k_2, k_3\}$ follow immediately: $k_1 = a_2 + (\overline{\lambda}_1 + \overline{\lambda}_2 + \overline{\lambda}_3)$, $k_2 = a_1 - (\overline{\lambda}_1\overline{\lambda}_2 + \overline{\lambda}_1\overline{\lambda}_3 + \overline{\lambda}_2\overline{\lambda}_3)$, $k_3 = a_0 + (\overline{\lambda}_1\overline{\lambda}_2\overline{\lambda}_3)$.

20.4.2 Observer design

The feedback control problem considered in the previous section, often called the **state feedback control** problem, assumed that you can measure the state $\mathbf{x}(t)$ accurately. To account for the more typical situation in which all you can measure is some output $\mathbf{y}(t)$ of the system, we may build an **observer** (a.k.a. an **estimator**)

in order to estimate the state $\mathbf{x}(t)$ itself. The key idea of an observer is to model closely the dynamics of the actual state $\mathbf{x}$ with a state estimate $\hat{\mathbf{x}}$ (represented in the controlling electronics or computer) such that

$$\begin{aligned} \mathbf{x}'(t) &= A\mathbf{x}(t) + B\mathbf{u}(t), & \hat{\mathbf{x}}'(t) &= A\hat{\mathbf{x}}(t) + B\mathbf{u}(t) - L(\mathbf{y} - \hat{\mathbf{y}}), \\ \mathbf{y}(t) &= C\mathbf{x}(t) + D\mathbf{u}(t), & \hat{\mathbf{y}}(t) &= C\hat{\mathbf{x}}(t) + D\mathbf{u}(t). \end{aligned} \tag{20.79}$$

Note that, in general, we usually do *not* know precisely

- the initial state $\mathbf{x}(0)$,
- the system model $\{A, B, C, D\}$,
- the disturbances (e.g., wind gusts) banging around on the system, or
- the noise (e.g., random sensor noise) corrupting the measurements.

To account for these unknowns, we have subtracted the $L(\mathbf{y} - \hat{\mathbf{y}})$ correction term from the estimator in (20.79). Our goal is to select a **feedback gain matrix** L in this correction term to "nudge" consistently the state estimate $\hat{\mathbf{x}}(t)$ towards the actual state $\mathbf{x}(t)$ even in the presence of such unknowns. Stated another way, we seek to stabilize the dynamics of the **estimation error** $\tilde{\mathbf{x}} \triangleq \mathbf{x} - \hat{\mathbf{x}}$, which by (20.79) may be written

$$\tilde{\mathbf{x}}'(t) = (A + LC)\tilde{\mathbf{x}}(t). \tag{20.80}$$

Note the similarity of the pole placement controller problem, in which we seek the K such that $(A + BK)$ has eigenvalues in the desired locations, and the pole placement observer problem, in which we seek the L such that $(A + LC)$ has eigenvalues in the desired locations.

In the SISO case, if the system is expressed in observer canonical form (20.60), solving this problem is particularly easy. For example, in the third-order SISO case, the transfer function

$$G(s) = \frac{b(s)}{s^3 + a_2 s^2 + a_1 s + a_0}$$

is equivalent to the observer canonical form

$$\begin{aligned} \mathbf{x}_o'(t) &= A_o \mathbf{x}_o(t) + B_o u(t) \\ y(t) &= C_o \mathbf{x}_o(t) + D_o u(t) \end{aligned} \qquad \text{where} \qquad A_o = \begin{pmatrix} -a_2 & 1 & 0 \\ -a_1 & 0 & 1 \\ -a_0 & 0 & 0 \end{pmatrix}, \quad C_o = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}.$$

Recall that the eigenvalues of an **open-loop system** (i.e., with $u = 0$) are given by the roots of the characteristic polynomial of the system matrix. As A_o is in left companion form, this characteristic polynomial is

$$\lambda^3 + a_2\lambda^2 + a_1\lambda + a_0 = 0.$$

Now consider the problem of computing feedback into the estimator. We seek the gain matrix

$$L = \begin{pmatrix} l_1 \\ l_2 \\ l_3 \end{pmatrix}$$

in order to place the closed-loop poles of the estimation error system (20.80) at the locations $\{\overline{\lambda}_1, \overline{\lambda}_2, \overline{\lambda}_3\}$; i.e., that the closed-loop system matrix of the estimation error system, $(A_o + LC_o)$, have a characteristic equation

$$(\lambda - \overline{\lambda}_1)(\lambda - \overline{\lambda}_2)(\lambda - \overline{\lambda}_3) = \lambda^3 - (\overline{\lambda}_1 + \overline{\lambda}_2 + \overline{\lambda}_3)\lambda^2 + (\overline{\lambda}_1\overline{\lambda}_2 + \overline{\lambda}_1\overline{\lambda}_3 + \overline{\lambda}_2\overline{\lambda}_3)\lambda - (\overline{\lambda}_1\overline{\lambda}_2\overline{\lambda}_3) = 0. \tag{20.81a}$$

Noting that $(A_o + LC_o)$ is itself in left companion form, its characteristic polynomial is given by

$$(A_o + LC_o) = \begin{pmatrix} l_1 - a_2 & 1 & 0 \\ l_2 - a_1 & 0 & 1 \\ l_3 - a_0 & 0 & 0 \end{pmatrix} \quad \Rightarrow \quad \lambda^3 - (l_1 - a_2)\lambda^2 - (l_2 - a_1)\lambda - (l_3 - a_0) = 0. \tag{20.81b}$$

Matching the coefficients of like powers of λ in the polynomials (20.81a) and (20.81b), simple expressions for $\{l_1, l_2, l_3\}$ follow immediately: $l_1 = a_2 + (\overline{\lambda}_1 + \overline{\lambda}_2 + \overline{\lambda}_3)$, $l_2 = a_1 - (\overline{\lambda}_1\overline{\lambda}_2 + \overline{\lambda}_1\overline{\lambda}_3 + \overline{\lambda}_2\overline{\lambda}_3)$, $l_3 = a_0 + (\overline{\lambda}_1\overline{\lambda}_2\overline{\lambda}_3)$.

20.5 Controllability, observability, and related concepts

We now seek to quantify the control authority of the control inputs and the information content in the sensor outputs from a MIMO system via linear-algebraic arguments.

Continuous time. We begin with the following definitions:

- A *continuous-time* system is **controllable** if a control distribution $\mathbf{u}(t)$ on $t \in [0, T]$ may be always be found to steer the system from an *arbitrary initial state* $\mathbf{x}(0)$ to an *arbitrary final state* $\mathbf{x}(T)$.
- A *continuous-time* system is **observable** if the initial state $\mathbf{x}(0)$ and final state $\mathbf{x}(T)$ may be *uniquely determined from the measurements* $\mathbf{y}(t)$ on $t \in [0, T]$.

We will discuss continuous-time controllability and observability at length in §20.5.1 and §20.5.2 respectively. Both conditions are closely related, and are in practice very stringent requirements on a system. In fact, *we often do not need an unstable system to be either controllable or observable in order to command it to behave in approximately the desired manner*. Thus, two weaker conditions may be introduced, **stabilizability** and **detectability**, which are in fact the minimum requirements on a dynamic system in order to close a feedback loop around it with success.

- A *continuous-time* system is **stabilizable** if its unstable subsystem (see §20.1.4) is controllable (and thus feedback control can **stabilize** the complete system, eventually bringing it back to the origin exponentially fast as $t \to \infty$).
- A *continuous-time* system is **detectable** if its unstable subsystem is observable (and thus feedback into an estimator can stabilize the error of the estimate of the complete system, eventually bringing this error to zero exponentially fast as $t \to \infty$).

A mnemonic summarizing the relation between these four related properties *in continuous time* is thus:

controllability $\subset$ stabilizability :: observability $\subset$ detectability.

Discrete time. The situation in discrete time is slightly more subtle, as e^{At} is invertible for any A (see Fact 20.2), but F^k is not invertible for singular[10] F. Thus, in the discrete-time case, one must be more precise, and there are six possible characterizations. Three are related to the controllability question:

- A *discrete-time* system is **reachable**[11] if a control distribution $\mathbf{u}_k$ for $k = 0, \ldots, n-1$ may be always be found to steer the system from an arbitrary initial state $\mathbf{x}_0$ to an arbitrary final state $\mathbf{x}_n$.
- A *discrete-time* system is **controllable**[11] if a control distribution $\mathbf{u}_k$ for $k = 0, \ldots, n-1$ may be always be found to steer the system from an arbitrary initial state $\mathbf{x}_0$ to a *zero* final state $\mathbf{x}_n = 0$.
- A *discrete-time* system is **stabilizable** if its unstable subsystem (see §20.1.4) is controllable (and thus feedback control can stabilize the complete system, eventually bringing it back to the origin exponentially fast as $k \to \infty$).

Three are related to the observability question:

- A *discrete-time* system is **observable** if the initial state $\mathbf{x}_0$ and final state x_n may be determined uniquely from the measurements $\mathbf{y}_k$ of the system for $k = 0, \ldots, n-1$.
- A *discrete-time* system is **constructible** if the *final* state $\mathbf{x}_n$ may be determined uniquely from the measurements $\mathbf{y}_k$ of the system for $k = 0, \ldots, n-1$.

[10] Note that the following analysis does *not* anywhere assume that F is invertible.

[11] Discrete-time reachability is sometimes referred to as **controllability pointwise state from origin**, and discrete-time controllability is sometimes referred to as **controllability pointwise state to origin**.

- A *discrete-time* system is **detectable** if its unstable subsystem (see §20.1.4) is constructible (and thus feedback into an estimator can stabilize the error of the estimate of the complete system, eventually bringing this error to zero exponentially fast as $k \to \infty$).

We discuss discrete-time reachability/controllability and observability/constructibility at length in §20.5.3 and §20.5.4. Note that discrete-time reachability implies discrete-time controllability, but not visa versa. Similarly, discrete-time observability implies discrete-time constructibility, but not visa versa. This is because F might be singular, and thus there are some directions $\mathbf{x}_\mathsf{N}$ (precisely, those directions in the nullspace of F, as summarized in Figure 4.1) such that $F\mathbf{x}_\mathsf{N} = 0$. If one or more of these particular directions $\mathbf{x}_\mathsf{N}$ is uncontrollable, but all other directions are controllable, then though the system *can* be brought to a zero final state, it can *not* be brought to an arbitrary final state. Similarly, if one or more of these particular directions $\mathbf{x}_\mathsf{N}$ is unobservable, but all other directions are observable, then though we *can* correctly identify the final state from the measurements (which will have zero component in left nullspace of F), we can *not* correctly identify the initial state from the measurements (which will have an undeterminable component in nullspace of F). A mnemonic summarizing the relation between these six related properties *in discrete time* is thus:

reachability $\subset$ **controllability** $\subset$ **stabilizability :: observability** $\subset$ **constructibility** $\subset$ **detectability.**

20.5.1 Continuous-time controllability

20.5.1.1 The continuous-time controllability matrix

We first consider the question of **continuous-time controllability** [a.k.a. **continuous-time reachability**; that is, the question of whether or not a control distribution $\mathbf{u}(t)$ on $t \in [0,T]$ may be always be found to steer a continuous-time system from an arbitrary initial state $\mathbf{x}(0)$ to an arbitrary final state $\mathbf{x}(T)$], focusing initially on the LTI case. The solution to the forced continuous-time LTI system

$$\mathbf{x}'(t) = A\mathbf{x}(t) + B\mathbf{u}(t), \tag{20.82}$$

as shown in (20.7) and (20.36), may be written in the form

$$\begin{aligned}
\mathbf{x}(T) - e^{AT}\mathbf{x}(0) &= \int_0^T e^{A(T-t)}B\mathbf{u}(t)\,dt = \int_0^T \left\{ I + A(T-t) + \frac{[A(T-t)]^2}{2!} + \frac{[A(T-t)]^3}{3!} + \ldots \right\} B\mathbf{u}(t)\,dt \\
&= \int_0^T \left\{ \alpha_0(t)I + \alpha_1(t)A + \alpha_2(t)A^2 + \ldots + \alpha_{n-1}(t)A^{n-1} \right\} B\mathbf{u}(t)\,dt \\
&= \begin{bmatrix} B & AB & \ldots & A^{n-1}B \end{bmatrix} \begin{bmatrix} \int_0^T \alpha_0(t)\mathbf{u}(t)\,dt \\ \int_0^T \alpha_1(t)\mathbf{u}(t)\,dt \\ \vdots \\ \int_0^T \alpha_{n-1}(t)\mathbf{u}(t)\,dt \end{bmatrix}.
\end{aligned} \tag{20.83}$$

[Note that, as a consequence of the Cayley-Hamilton theorem (Fact 4.15), the infinite sum in the first line may be reduced to the finite sum in the second line via the resolvent algorithm (20.35).] It follows that:

Fact 20.10 *The continuous-time LTI system* (20.82) *is* **controllable** *iff the (fat or, if the system is SIMO, square)* **continuous-time controllability matrix**

$$\mathscr{C} \triangleq \begin{bmatrix} B & AB & \ldots & A^{n-1}B \end{bmatrix} \tag{20.84}$$

has full row rank; the system is called **uncontrollable** *if* $\mathrm{rank}(\mathscr{C}) < n$*, and* **null controllable** *if* $\mathrm{rank}(\mathscr{C}) = 0$.

Proof: If $\mathscr{C}$ does not have full row rank, then there are clearly some $\mathbf{x}(0)$ and $\mathbf{x}(T)$ for which (20.83) has zero solutions. On the other hand, if $\mathscr{C}$ has full row rank, then we may construct a control input $\mathbf{u}(t)$ on $t \in (0,T)$

that takes the system from the specified $\mathbf{x}(0)$ to the specified $\mathbf{x}(T)$ in any of a variety of ways. One such construction may be developed by defining $\mathbf{u}(t)$ via an orthogonal expansion[12]

$$\mathbf{u}(t) \triangleq \sum_{k=1}^{n} \hat{\mathbf{u}}_k^s \sin(k\pi t/T)$$

for n coefficients $\hat{\mathbf{u}}_k^s$ for $k = 1, \ldots n$ which we will determine, inserting this orthogonal expansion into (20.83), and multiplying from the left by $\mathscr{C}^+$, resulting in

$$\mathscr{C}^+[\mathbf{x}(T) - e^{AT}\mathbf{x}(0)] = \begin{bmatrix} \sum_{k=1}^n \int_0^T \alpha_0(t)\sin(k\pi t/T)\,dt\,\hat{\mathbf{u}}_k^s \\ \sum_{k=1}^n \int_0^T \alpha_1(t)\sin(k\pi t/T)\,dt\,\hat{\mathbf{u}}_k^s \\ \vdots \\ \sum_{k=1}^n \int_0^T \alpha_{n-1}(t)\sin(k\pi t/T)\,dt\,\hat{\mathbf{u}}_k^s \end{bmatrix} = M \begin{bmatrix} \hat{\mathbf{u}}_1^s \\ \hat{\mathbf{u}}_2^s \\ \vdots \\ \hat{\mathbf{u}}_n^s \end{bmatrix}$$

$$\Rightarrow \quad \begin{bmatrix} \hat{\mathbf{u}}_1^s \\ \hat{\mathbf{u}}_2^s \\ \vdots \\ \hat{\mathbf{u}}_n^s \end{bmatrix} = M^{-1}\mathscr{C}^+[\mathbf{x}(T) - e^{AT}\mathbf{x}(0)] \quad \text{with} \quad M = \frac{2}{T}\begin{bmatrix} \hat{\alpha}_{0,1}^s I & \hat{\alpha}_{0,2}^s I & \ldots & \hat{\alpha}_{0,n}^s I \\ \hat{\alpha}_{1,1}^s I & \hat{\alpha}_{1,2}^s I & \ldots & \hat{\alpha}_{1,n}^s I \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\alpha}_{n-1,1}^s I & \hat{\alpha}_{n-1,2}^s I & \ldots & \hat{\alpha}_{n-1,n}^s I \end{bmatrix}, \tag{20.85a}$$

where the $\hat{\alpha}_{i,k}^s$ are the coefficients in the sine expansion of $\alpha_i(t)$ on $t \in [0,T]$:

$$\alpha_i(t) = \sum_{k=1}^{\infty} \hat{\alpha}_{i,k}^s \sin(k\pi t/T) \quad \Rightarrow \quad \hat{\alpha}_{i,k}^s = \frac{2}{T}\int_0^T \alpha_i(t)\sin(k\pi t/T)dt. \tag{20.85b}$$

Note that the second equation above is obtained by multiplying the first equation by the test function $\sin(j\pi t/T)$, integrating over $[0,T]$, and applying the orthogonality of the sine functions [see (4.10b)]. □

There are a few remaining noteworthy issues. First, the question of controllability of an LTI system is not a function of T; if an LTI system is (not) controllable over $[0,T]$ for some $T > 0$, it is (not) controllable over $[0,T]$ for any $T > 0$. Second, if a SIMO continuous-time system is expressed in controllability canonical form (20.55), then $\mathscr{C} = I$. Finally, the rank of the continuous-time controllability matrix is *invariant* under any state transformation (20.3), as

$$\mathscr{C}_R \triangleq \begin{bmatrix} B_R & A_R B_R & \ldots & A_R^{n-1}B_R \end{bmatrix} = R^{-1}\begin{bmatrix} B & AB & \ldots & A^{n-1}B \end{bmatrix} \triangleq R^{-1}\mathscr{C}. \tag{20.86}$$

20.5.1.2 The continuous-time controllability gramian

We now expand our focus on the continuous-time controllability question to include the LTV case. A powerful method of analyzing the question of controllability of the continuous-time system

$$\mathbf{x}'(t) = A(t)\mathbf{x}(t) + B(t)\mathbf{u}(t), \tag{20.87}$$

is given by the following fact:

Fact 20.11 *The continuous-time LTV system* (20.87) *is* **controllable** *iff the* **continuous-time controllability gramian** $P(T)$ *is invertible, where* $P(T)$ *is defined in the LTV case, noting* (20.6) *and taking* $t' = T - t$, *as*

$$P(T) \triangleq \int_0^T \Phi(T,t)\,B(t)\,B^H(t)\,\Phi^H(T,t)\,dt = \Phi(T,0)\int_0^T \Phi(0,t)\,B(t)\,B^H(t)\,\Phi^H(0,t)\,dt\,\Phi^H(T,0), \tag{20.88a}$$

[12] Any set of orthogonal basis functions may be used in this expansion; we use sine functions here for convenience only.

which reduces in the LTI case, noting (20.2), to

$$P(T) = \int_0^T e^{A(T-t)}\, B B^H\, e^{A^H(T-t)}\, dt = \int_0^T e^{At'}\, B B^H\, e^{A^H t'}\, dt' = e^{AT} \int_0^T e^{-At}\, B B^H\, e^{-A^H t}\, dt\, e^{A^H T}. \tag{20.88b}$$

Proof: Note that, by construction, $P(T) \geq 0$. If $P(T)$ is invertible, then we may determine a control input $\mathbf{u}(t)$ on $t \in (0,T)$ that takes the LTV system from the specified $\mathbf{x}(0)$ to the specified $\mathbf{x}(T)$ as

$$\mathbf{u}(t) = B^H(t)\,\Phi^H(T,t) P^{-1}(T)[\mathbf{x}(T) - \Phi(T,0)\mathbf{x}(0)], \tag{20.89a}$$

which reduces in the LTI case to

$$\mathbf{u}(t) = B^H e^{A^H(T-t)} P^{-1}(T)[\mathbf{x}(T) - e^{AT}\mathbf{x}(0)], \tag{20.89b}$$

as easily verified by substitution into (20.19a) and (20.7) respectively [see Exercise 20.3]. On the other hand, if $P(T)$ is not invertible, then there is some direction $\mathbf{q} \neq 0$ such that $\mathbf{q}^H P(T)\mathbf{q} = 0$. Thus, in both the LTV and LTI cases,

$$\begin{aligned} \mathbf{q}^H \Big[\int_0^T \Phi(T,t)\, B(t)\, B^H(t)\, \Phi^H(T,t)\, dt \Big] \mathbf{q} &= \int_0^T \mathbf{q}^H \Phi(T,t) B(t) B^H(t) \Phi^H(T,t) \mathbf{q}\, dt \\ = \int_0^T \| B^H(t) \Phi^H(T,t) \mathbf{q} \|^2\, dt = 0 \qquad &\Rightarrow \qquad \mathbf{q}^H \Phi(T,t) B(t) = 0 \quad \text{for all } \; t \in (0,T). \end{aligned} \tag{20.90}$$

Now, *assuming* there exists a control input $\mathbf{u}(t)$ on $t \in (0,T)$ that takes the system from any specified initial state $\mathbf{x}(0)$ to any specified final state $\mathbf{x}(T)$, taking $\mathbf{q}^H$ times (20.19a) leads to

$$\mathbf{q}^H[\mathbf{x}(T) - \Phi(T,0)\mathbf{x}(0)] = \int_0^T \mathbf{q}^H \Phi(T,t) B(t) \mathbf{u}(t)\, dt = 0.$$

If $\mathbf{x}(0)$ and/or $\mathbf{x}(T)$ are taken as arbitrary, then this implies that $\mathbf{q} = 0$, thereby leading to a *contradiction*, thus implying that, if $P(T)$ is not invertible, then there is, in general, *not* always a control input $\mathbf{u}(t)$ on $t \in (0,T)$ that takes the system from any specified $\mathbf{x}(0)$ to any specified $\mathbf{x}(T)$. □

Noting in the LTV case the form of $P(T)$ on the right in (20.88a), and in the LTI case the form of $P(T)$ on the right in (20.88b), and applying the identity $\frac{d}{dT}\int_0^T f(t)\, dt = f(T)$ and Facts 20.5 (in the LTV case) and 20.3 (in the LTI case), it is easily verified by substitution that $P(t)$ satisfies the differential Lyapunov equation

$$\frac{dP(t)}{dt} = A(t)P(t) + P(t)A^H(t) + B(t)B^H(t) \quad \text{with} \quad P(0) = 0. \tag{20.91a}$$

In the LTI case, for large T, $P(T)$ approaches the solution of the continuous-time algebraic Lyapunov equation

$$0 = AP + PA^H + BB^H. \tag{20.91b}$$

Note also that the rank of the continuous-time controllability gramian is *invariant* under any state transformation (20.3), as [denoting by Φ_R the continuous-time state transition matrix of the transformed system]

$$\int_0^T \Phi_R(T,t)\, B_R(t)\, B_R^H(t)\, \Phi_R^H(T,t)\, dt = R^{-1} \Big[\int_0^T \Phi(T,t)\, B(t)\, B^H(t)\, \Phi^H(T,t)\, dt \Big] R^{-H}. \tag{20.92}$$

Now consider a stable, strictly proper ($D = 0$) CT LTI state-space form (20.1). The impulse response of this system may be written as simply $g(t) = C e^{At} B$ for $t > 0$. By (20.46a) and (20.88b),

$$\|G(s)\|_2^2 = \int_0^\infty \text{trace}\Big[g(t)\, g^H(t) \Big] dt = \text{trace}\Big[\int_0^\infty C e^{At} B B^H e^{A^H t} C^H\, dt \Big] = \text{trace}\big[C P C^H \big], \tag{20.93}$$

where P is the solution of (20.91b), as stated in §20.2.2.1.

20.5.2 Continuous-time observability

20.5.2.1 The continuous-time observability matrix

We now consider the question of **continuous-time observability** [that is, the question of whether or not one can reconstruct the initial state $\mathbf{x}(0)$ and the final state $\mathbf{x}(T)$ of a continuous-time system from the observations $\mathbf{y}(t)$ on $t \in [0,T]$], again focusing initially on the LTI case

$$\mathbf{x}'(t) = A\mathbf{x}(t), \quad \mathbf{y}(t) = C\mathbf{x}(t). \tag{20.94}$$

Leveraging (20.7) and (20.36), we may write

$$\begin{aligned}\mathbf{y}(t) = C\mathbf{x}(t) = Ce^{At}\mathbf{x}(0) &= C\Big\{I + At + \frac{[At]^2}{2!} + \frac{[At]^3}{3!} + \dots\Big\}\mathbf{x}(0) \\ &= C\{\alpha_0(t)I + \alpha_1(t)A + \dots + \alpha_{n-1}(t)A^{n-1}\}\mathbf{x}(0) \\ &= \begin{bmatrix}\alpha_0(t)I & \alpha_1(t)I & \dots & \alpha_{n-1}(t)I\end{bmatrix}\begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1}\end{bmatrix}\mathbf{x}(0).\end{aligned} \tag{20.95}$$

[Note that, as a consequence of the Cayley-Hamilton theorem (Fact 4.15), the infinite sum in the first line may be reduced to the finite sum in the second line via the resolvent algorithm (20.35).] It follows that:

Fact 20.12 *The continuous-time LTI system* (20.94) *is* **observable** *iff the (tall or, if the system is MISO, square)* **continuous-time observability matrix**

$$\mathscr{O} \triangleq \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1}\end{bmatrix} \tag{20.96}$$

has full column rank; the system is called **unobservable** *if* $\mathrm{rank}(\mathscr{O}) < n$*, and* **null observable** *if* $\mathrm{rank}(\mathscr{O}) = 0$.

Proof: If $\mathscr{O}$ does not have full column rank, then, by (20.95), then there are clearly some $\mathbf{x}(0) \neq 0$ which are not reflected in *any* of the measurements (that is, for which $\mathbf{y}(t) = 0$ on $t \in [0,T]$), and thus the component of $\mathbf{x}(0)$ in these directions can not be determined from the $\mathbf{y}(t)$. On the other hand, if $\mathscr{O}$ has full column rank, then we may construct the initial state $\mathbf{x}(0)$ from the measurements $\mathbf{y}(t)$ on $t \in (0,T)$ in any of a variety of ways. One such construction may be developed via an orthogonal expansion[13] of $\mathbf{y}(t)$:

$$\mathbf{y}(t) \triangleq \sum_{k=1}^{\infty} \hat{\mathbf{y}}_k^s \sin(k\pi t/T) \quad \Rightarrow \quad \hat{\mathbf{y}}_k^s = \frac{2}{T}\int_0^T \mathbf{y}(t)\sin(k\pi t/T)dt.$$

Substituting this expansion into (20.95), multiplying from the left by a vector of test functions $\sin(j\pi t/T)$, and integrating leads to

$$\int_0^T \begin{bmatrix}\sin(\pi t/T)I \\ \sin(2\pi t/T)I \\ \vdots \\ \sin(n\pi t/T)I\end{bmatrix}\left\{\sum_{k=1}^{\infty}\hat{\mathbf{y}}_k^s\sin(k\pi t/T) = \begin{bmatrix}\alpha_0(t)I & \alpha_1(t)I & \dots & \alpha_{n-1}(t)I\end{bmatrix}\begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1}\end{bmatrix}\mathbf{x}(0)\right\}dt$$

[13] Any set of orthogonal basis functions may be used in this expansion; as in the orthogonal expansions of the $\alpha_i(t)$ in (20.85b), we use sine functions here for convenience only.

$$\Rightarrow \quad \frac{T}{2}\begin{bmatrix}\hat{\mathbf{y}}_1^s\\ \hat{\mathbf{y}}_2^s\\ \vdots\\ \hat{\mathbf{y}}_n^s\end{bmatrix} = M^T\mathscr{O}\mathbf{x}(0) \quad \Rightarrow \quad \mathbf{x}(0) = \frac{T}{2}\mathscr{O}^+M^{-T}\begin{bmatrix}\hat{\mathbf{y}}_1^s\\ \hat{\mathbf{y}}_2^s\\ \vdots\\ \hat{\mathbf{y}}_n^s\end{bmatrix},$$

where M is defined as in (20.85). □

There are a few remaining noteworthy issues. First, the question of observability of an LTI system is not a function of T; if an LTI system is (not) observable over $[0,T]$ for some $T>0$, it is (not) observable over $[0,T]$ for any $T>0$. Second, if a MISO continuous-time system is expressed in observability canonical form (20.61), then $\mathscr{O}=I$. Finally, the rank of the continuous-time observability matrix is *invariant* under any state transformation (20.3), as

$$\mathscr{O}_R \triangleq \begin{bmatrix}C_R\\ C_RA_R\\ \vdots\\ C_RA_R^{n-1}\end{bmatrix} = \begin{bmatrix}C\\ CA\\ \vdots\\ CA^{n-1}\end{bmatrix}R \triangleq \mathscr{O}R. \tag{20.97}$$

20.5.2.2 The continuous-time observability gramian

We now expand our focus on the continuous-time observability question to include the LTV case. A powerful method of analyzing the question of observability of the continuous-time system

$$\mathbf{x}'(t) = A(t)\mathbf{x}(t), \quad \mathbf{y}(t) = C(t)\mathbf{x}(t), \tag{20.98}$$

is given by the following fact:

Fact 20.13 *The continuous-time LTV system* (20.98) *is* **observable** *iff the* **continuous-time observability gramian** $Q_T(0)$ *is invertible, where* $Q_T(0)$ *is defined in the LTV case (taking* $t'=T-t$*) as*

$$Q_T(0) \triangleq \int_0^T \Phi^H(t,0)C^H(t)C(t)\Phi(t,0)\,dt = \Phi^H(T,0)\int_0^T \Phi^H(0,t')C^H(t')C(t')\Phi(0,t')\,dt\Phi(T,0), \tag{20.99a}$$

which reduces in the LTI case to

$$Q_T(0) = \int_0^T e^{A^Ht}C^HCe^{At}\,dt = \int_0^T e^{A^H(T-t')}C^HCe^{A(T-t')}\,dt' = e^{A^HT}\int_0^T e^{-A^Ht'}C^HCe^{-At'}\,dt'\,e^{AT}. \tag{20.99b}$$

Proof: By construction, $Q_T(0)\geq 0$. If $Q_T(0)$ is invertible, then we may determine the initial state $\mathbf{x}(0)$ on the LTV system from the measurements $\mathbf{y}(t)$ on $t\in(0,T)$ by substituting $\mathbf{x}(t)=\Phi(t,0)\mathbf{x}(0)$ into $\mathbf{y}(t)=C(t)\mathbf{x}(t)$, multiplying the result from the left by $\Phi^H(t,0)C^H(t)$, and integrating from $t=0$ to T, resulting in

$$\mathbf{x}(0) = [Q_T(0)]^{-1}\int_0^T \Phi^H(t,0)C^H(t)\mathbf{y}(t)\,dt, \tag{20.100a}$$

which reduces in the LTI case to

$$\mathbf{x}(0) = [Q_T(0)]^{-1}\int_0^T e^{A^Ht}C^H\mathbf{y}(t)\,dt. \tag{20.100b}$$

On the other hand, if $Q_T(0)$ is not invertible, then there is some direction $\mathbf{x}(0) \neq 0$ such that $\mathbf{x}^H(0) Q_T(0) \mathbf{x}(0) = 0$. Thus, in both the LTV and LTI cases,

$$\mathbf{x}^H(0) \Big[\int_0^T \Phi^H(t,0) C^H(t) C(t) \Phi(t,0)\, dt \Big] \mathbf{x}(0) = \int_0^T \mathbf{x}^H(0)\, \Phi^H(t,0) C^H(t) C(t) \Phi(t,0) \mathbf{x}(0)\, dt$$
$$= \int_0^T \| C(t) \Phi(t,0) \mathbf{x}(0) \|^2 dt = 0 \qquad \Rightarrow \qquad \mathbf{y}(t) = C(t) \Phi(t,0) \mathbf{x}(0) = 0 \quad \text{for all } \; t \in [0,T];$$

that is, there are some $\mathbf{x}(0) \neq 0$ which are not reflected in *any* of the measurements, and thus the component of $\mathbf{x}(0)$ in these directions can not be determined from the $\mathbf{y}(t)$. □

Noting in the LTV case the form of $Q_T(0)$ in (20.99a), and in the LTI case the form of $Q_T(0)$ on the right in (20.99b), it may be verified that $Q_T(t)$ satisfies the differential Lyapunov equation

$$\frac{dQ_T(t)}{dt} = A^H(t) Q_T(t) + Q_T(t) A(t) + C^H(t) C(t) \quad \text{with} \quad Q_T(T) = 0. \tag{20.101}$$

In the LTI case, for large T, $Q_T(0)$ approaches the solution of the continuous-time algebraic Lyapunov equation

$$0 = A^H Q + Q A + C^H C. \tag{20.102}$$

Note also that the rank of the continuous-time observability gramian is *invariant* under any state transformation (20.3), as

$$\int_0^T \Phi_R^H(t,0) C_R^H(t) C_R(t) \Phi_R(t,0)\, dt = R^H \int_0^T \Phi^H(t,0) C^H(t) C(t) \Phi(t,0)\, dt\, R. \tag{20.103}$$

Finally, note that once $\mathbf{x}(0)$ is known, it follows that $\mathbf{x}(T) = e^{AT} \mathbf{x}(0)$ in the LTI case and $\mathbf{x}(T) = \Phi(T,0) \mathbf{x}(0)$ in the LTV case.

Again, consider a stable, strictly proper ($D = 0$) CT LTI state-space form (20.1), recalling that its impulse response may be written $g(t) = C e^{At} B$ for $t > 0$. By (20.46a) and (20.99b),

$$\| G(s) \|_2^2 = \int_0^\infty \text{trace} \Big[g^H(t)\, g(t) \Big] dt = \text{trace} \Big[\int_0^\infty B^H e^{A^H t} C^H C e^{At} B\, dt \Big] = \text{trace} \big[B^H Q B \big], \tag{20.104}$$

where Q is the solution of (20.102), as stated in §20.2.2.1.

20.5.3 Discrete-time reachability and controllability

The discrete-time reachability matrix

We now consider the question of **discrete-time reachability** [that is, the question of whether or not a control distribution $\mathbf{u}_k$ for $k = 0, \ldots, n-1$ may be always be found to steer a discrete-time system from an arbitrary initial state $\mathbf{x}_0$ to an arbitrary final state $\mathbf{x}_n$], focusing initially on the LTI case. The solution of the forced discrete-time LTI system

$$\mathbf{x}_{k+1} = F \mathbf{x}_k + G \mathbf{u}_k, \tag{20.105}$$

as given in (20.27), is

$$\mathbf{x}_n = F^n \mathbf{x}_0 + \underbrace{\begin{bmatrix} G & FG & \ldots & F^{n-1} G \end{bmatrix}}_{= \mathscr{R}} \begin{bmatrix} \mathbf{u}_{n-1} \\ \mathbf{u}_{n-2} \\ \vdots \\ \mathbf{u}_0 \end{bmatrix} \quad \Rightarrow \quad \mathscr{R} \begin{bmatrix} \mathbf{u}_{n-1} \\ \mathbf{u}_{n-2} \\ \vdots \\ \mathbf{u}_0 \end{bmatrix} = \mathbf{x}_n - F^n \mathbf{x}_0. \tag{20.106}$$

Thus,

Fact 20.14 *The discrete-time LTI system* (20.105) *is* **reachable** *iff the (fat or square)* **discrete-time reachability matrix**[14]

$$\mathscr{R} \triangleq \begin{bmatrix} G & FG & \dots & F^{n-1}G \end{bmatrix} \tag{20.107}$$

has full row rank; the system is called **unreachable** *if* $\mathrm{rank}(\mathscr{R}) < n$*, and* **null reachable** *if* $\mathrm{rank}(\mathscr{R}) = 0$.

Proof: If $\mathscr{R}$ does not have full row rank, then there are clearly some $\mathbf{x}_n$ for which (20.106) has zero solutions. On the other hand, if $\mathscr{R}$ has full row rank, then one control sequence that takes the system from the specified $\mathbf{x}_0$ to the specified $\mathbf{x}_n$ is

$$\begin{bmatrix} \mathbf{u}_{n-1} \\ \mathbf{u}_{n-2} \\ \vdots \\ \mathbf{u}_0 \end{bmatrix} = \mathscr{R}^{+}(\mathbf{x}_n - F^n \mathbf{x}_0). \qquad \square$$

There are a few remaining noteworthy issues. First, by the Cayley-Hamilton Theorem (Fact 4.15), if the matrix $\mathscr{R}$ does not have full row rank, than a larger matrix of the same form, quantifying the effect of $n+1$ (or more) control inputs on the evolution of the system, will not have full row rank either. Thus, the question of reachability of a discrete-time LTI system is not a function of k so long as $k \geq n$; if an LTI system is (not) reachable over $[0,n]$, it is (not) reachable over $[0,k]$ for any $k > n$. Second, if a SIMO discrete-time system is expressed in reachability canonical form (20.65), then $\mathscr{R} = I$. Finally, as in (20.86), the rank of the discrete-time reachability matrix is *invariant* under any state transformation.

The discrete-time reachability gramian

We now expand our focus on the discrete-time reachability question to include the LTV case. A powerful method of analyzing the question of reachability of the discrete-time LTV system

$$\mathbf{x}_{k+1} = F_k \mathbf{x}_k + G_k \mathbf{u}_k, \tag{20.108}$$

is given by the following fact:

Fact 20.15 *The discrete-time LTV system* (20.108) *is* **reachable** *iff the* **discrete-time reachability gramian** P_n *is invertible, where* P_n *is defined in the LTV case as*

$$P_n \triangleq \sum_{k=0}^{n-1} \Phi_{n,k+1} G_k G_k^H \Phi_{n,k+1}^H, \tag{20.109a}$$

which reduces in the LTI case to

$$P_n = \sum_{k=0}^{n-1} F^{n-1-k} G G^H (F^H)^{n-1-k} = \sum_{k'=0}^{n-1} F^{k'} G G^H (F^H)^{k'}. \tag{20.109b}$$

Proof: By construction, $P_n \geq 0$. If P_n is invertible, then a control input sequence $\mathbf{u}_k$ for $k = 0, 1, \dots, (n-1)$ that takes the LTV system from the specified $\mathbf{x}_0$ to the specified $\mathbf{x}_n$ is

$$\mathbf{u}_k = G_k^H \Phi_{n,k+1}^H P_n^{-1} [\mathbf{x}_n - \Phi_{n,0} \mathbf{x}_0], \tag{20.110a}$$

[14] Some texts refer to $\mathscr{R}$ defined in (20.106) and P_n defined in (20.109) as, respectively, the **discrete-time controllability matrix** and the **discrete-time controllability gramian**. However, as we show here, the tests of whether or not the matrix $\mathscr{R}$ is full rank and whether or not the matrix P_n is invertible are equivalent to the discrete-time reachability question; they are only sufficient (but not necessary) for discrete-time controllability (see §20.5.3.1). In the discrete-time case, we thus prefer names associated with "reachability" for $\mathscr{R}$ and P_n in the present text.

which reduces in the LTI case to

$$\mathbf{u}_k = G^H (F^H)^{n-k-1} P_n^{-1} [\mathbf{x}_n - F^n \mathbf{x}_0], \tag{20.110b}$$

as easily verified by substitution into (20.27) and (20.106) respectively [see Exercise 7.2]. On the other hand, if P_n is not invertible, then there is some direction $\mathbf{q} \neq 0$ such that $\mathbf{q}^H P_n \mathbf{q} = 0$. Thus, in both the LTV and LTI cases,

$$\mathbf{q}^H \Big[\sum_{k=0}^{n-1} \Phi_{n,k+1} G_k G_k^H \Phi_{n,k+1}^H \Big] \mathbf{q} = \sum_{k=0}^{n-1} \mathbf{q}^H \Phi_{n,k+1} G_k G_k^H \Phi_{n,k+1}^H \mathbf{q} \tag{20.111}$$

$$= \sum_{k=0}^{n-1} \| G_k^H \Phi_{n,k+1}^H \mathbf{q} \|^2 = 0 \qquad \Rightarrow \qquad \mathbf{q}^H \Phi_{n,k+1} G_k = 0 \quad \text{for} \quad k = 0, 1, \ldots, n-1. \tag{20.112}$$

Now, *assuming* there exists a control input sequence $\mathbf{u}_k$ for $k = 0, 1, \ldots, (n-1)$ that takes the system from the specified initial state $\mathbf{x}_0$ to any specified final state $\mathbf{x}_n$, taking $\mathbf{q}^H$ times (20.27) leads to

$$\mathbf{q}^H [\mathbf{x}_n - \Phi_{n,0} \mathbf{x}_0] = \sum_{k=0}^{n-1} \mathbf{q}^H \Phi_{n,k+1} G_k \mathbf{u}_k = 0.$$

If $\mathbf{x}_n$ is taken as arbitrary, then this implies that $\mathbf{q} = 0$, thereby leading to a *contradiction*, thus implying that, if P_n is not invertible, then there is, in general, *not* always a control input sequence $\mathbf{u}_k$ for $k = 0, 1, \ldots, (n-1)$ that takes the system from any specified $\mathbf{x}_0$ to any specified $\mathbf{x}_n$. □

Noting in the LTV case the form of P_n in (20.109a), and in the LTI case the form of P_n in (20.109b), it is easily verified by substitution that P_n satisfies the Lyapunov difference equation

$$P_{n+1} = F_n P_n F_n^H + G_n G_n^H \quad \text{with} \quad P_0 = 0. \tag{20.113}$$

In the LTI case, for large n, P_n approaches the solution of the discrete-time algebraic Lyapunov equation

$$P = F P F^H + G G^H. \tag{20.114}$$

Note also that, as in (20.92) the rank of the discrete-time reachability gramian is *invariant* under any state transformation (20.3).

Now consider a stable DT LTI state-space form (20.2). The impulse response of this system may be written, noting (B.69b) and (1.5), as simply $g_k = H F^k G h_{1k} + D \delta_{0k}$. By (20.48a) and (20.109b),

$$\| G(z) \|_2^2 = \sum_{k=0}^{\infty} \text{trace} \Big[g_k g_k^H \Big] = \text{trace} \Big[\sum_{k=0}^{\infty} (H F^k G h_{1k} + D \delta_{0k}) (D^H \delta_{0k} + G^H [F^H]^k H^H h_{1k}) \Big]$$

$$= \text{trace} \big[H P H^H + D D^H \big], \tag{20.115}$$

where P is the solution of (20.114), as stated in §20.2.2.1.

20.5.3.1 A weaker condition: steering a discrete-time system to the origin

We now focus specifically on the question of **discrete-time controllability** [that is, the question of whether or not a control distribution $\mathbf{u}_k$ for $k = 0, \ldots, n-1$ may be always be found to steer a discrete-time system from an arbitrary initial state $\mathbf{x}_0$ to a *zero* final state $\mathbf{x}_n = 0$], again focusing initially on the LTI case. Following the analysis presented above, we may conclude the following:

Fact 20.16 *The discrete-time LTI system* (20.105) *is* **controllable** *iff all columns of* F^n *are in the column space of the discrete-time reachability matrix* $\mathscr{R}$ *defined in Fact* 20.14.

Proof: If there are some columns of F^n that are not in the column space of $\mathscr{R}$, then there are clearly some $\mathbf{x}_0$ for which (20.106) has zero solutions. On the other hand, if all columns of F^n are in the column space of $\mathscr{R}$, then one control sequence that takes the system from any specified $\mathbf{x}_0$ to $\mathbf{x}_n = 0$ is given in the proof of Fact 20.14. □

Turning our attention to the more general LTV case, we may conclude the following:

Fact 20.17 *The discrete-time LTV system* (20.108) *is* **controllable** *iff all columns of* $\Phi_{n,0}$ *are in the column space of the discrete-time reachability gramian* P_n *defined in Fact* 20.15.

Proof: If all columns of $\Phi_{n,0}$ are in the column space of P_n, then a control input sequence $\mathbf{u}_k$ for $k = 0, 1, \ldots, (n-1)$ that takes the LTV system from any specified $\mathbf{x}_0$ to $\mathbf{x}_n = 0$ is

$$\mathbf{u}_k = -G_k^H \Phi_{n,k+1}^H P_n^+ \Phi_{n,0} \mathbf{x}_0, \tag{20.116a}$$

which reduces in the LTI case to

$$\mathbf{u}_k = -G^H (F^H)^{n-k-1} P_n^+ F^n \mathbf{x}_0. \tag{20.116b}$$

On the other hand, if not all columns of $\Phi_{n,0}$ are in the column space of P_n, and thus some vector $\mathbf{q} \neq 0$ in the column space of $\Phi_{n,0}$ is in the left nullspace of P_n, then we have both $\mathbf{q}^H \Phi_{n,0} \neq 0$ and, as shown in (20.112), $\mathbf{q}^H \Phi_{n,k+1} G_k = 0$ for $k = 0, 1, \ldots, n-1$. Now, *assuming* there exists a control input sequence $\mathbf{u}_k$ for $k = 0, 1, \ldots, (n-1)$ that takes the system from any specified initial state $\mathbf{x}_0$ to $\mathbf{x}_n = 0$, taking $\mathbf{q}^H$ times (20.27) leads to

$$-\mathbf{q}^H \Phi_{n,0} \mathbf{x}_0 = \sum_{k=0}^{n-1} \mathbf{q}^H \Phi_{n,k+1} G_k \mathbf{u}_k = 0.$$

If $\mathbf{x}_0$ is taken as arbitrary, then this implies that $\mathbf{q}^H \Phi_{n,0} = 0$, thereby leading to a *contradiction*, thus implying that, if not all columns of $\Phi_{n,0}$ are in the column space of P_n, then there is, in general, *not* always a control input sequence $\mathbf{u}_k$ for $k = 0, 1, \ldots, (n-1)$ that takes the system from any specified $\mathbf{x}_0$ to $\mathbf{x}_n = 0$. □

20.5.4 Discrete-time observability and constructability

The discrete-time observability matrix

We now consider the question of **discrete-time observability** [that is, the question of whether or not one can reconstruct both the initial state $\mathbf{x}_0$ and the final state $\mathbf{x}_n$ of a discrete-time system from the observations $\mathbf{y}_k$ for $k = 0, 1, \ldots, n-1$], again focusing initially on the LTI case

$$\mathbf{x}_{k+1} = F\mathbf{x}_k, \quad \mathbf{y}_k = H\mathbf{x}_k, \tag{20.117}$$

the solution of which may be written

$$\begin{bmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_{n-1} \end{bmatrix} = \begin{bmatrix} H \\ HF \\ \vdots \\ HF^{n-1} \end{bmatrix} \mathbf{x}_0. \tag{20.118}$$

It follows that:

Fact 20.18 *The discrete-time LTI system* (20.117) *is* **observable** *iff the (tall or square)* **discrete-time observability matrix**

$$\mathscr{O} \triangleq \begin{bmatrix} H \\ HF \\ \vdots \\ HF^{n-1} \end{bmatrix} \tag{20.119}$$

has full column rank; the system is called **unobservable** *if* $\text{rank}(\mathscr{O}) < n$*, and* **null observable** *if* $\text{rank}(\mathscr{O}) = 0$.

Proof: If $\mathscr{O}$ does not have full column rank, then there are clearly some $\mathbf{x}_0 \neq 0$ which are not reflected in *any* of the measurements (that is, for which $\mathbf{y}_k = 0$ for $k = 0, 1, \ldots, n-1$), and thus the component of $\mathbf{x}_0$ in these directions can not be determined from the $\mathbf{y}_k$. On the other hand, if $\mathscr{O}$ has full column rank, then we may determine $\mathbf{x}_0$ from the measurements $\mathbf{y}_k$ for $k = 0, 1, \ldots, n-1$ simply by multiplying (20.118) from the left by $\mathscr{O}^+$. □

There are a few remaining noteworthy issues. First, by the Cayley-Hamilton Theorem (Fact 4.15), if the matrix $\mathscr{O}$ does not have full column rank, than a larger matrix of the same form, quantifying the information in $n+1$ (or more) measurements of the system, will not have full column rank either. Thus, the question of observability of a discrete-time LTI system is not a function of k so long as $k \geq n$; if an LTI system is (not) observable over $[0, n]$, it is (not) observable over $[0, k]$ for any $k > n$. Second, if a MISO discrete-time system is expressed in observability canonical form (20.71), then $\mathscr{O} = I$. Finally, as in (20.97), the rank of the discrete-time observability matrix is *invariant* under any state transformation.

The discrete-time observability gramian

We now expand our focus on the discrete-time observability question to include the LTV case. A powerful method of analyzing the question of observability of the discrete-time LTV system with observations,

$$\mathbf{x}_{k+1} = F_k \mathbf{x}_k, \quad \mathbf{y}_k = H_k \mathbf{x}_k, \tag{20.120}$$

is given by the following fact:

Fact 20.19 *The discrete-time LTV system* (20.120) *is* **observable** *iff the* **discrete-time observability gramian** $Q_{n,0}$ *is invertible, where* $Q_{n,0}$ *is defined in the LTV case, as*

$$Q_{n,0} \triangleq \sum_{k=0}^{n-1} \Phi_{k,0}^H H_k^H H_k \Phi_{k,0}, \tag{20.121a}$$

which reduces in the LTI case to

$$Q_{n,0} = \sum_{k=0}^{n-1} (F^H)^{n-1-k} H^H H F^{n-1-k} = \sum_{k'=0}^{n-1} (F^H)^{k'} H^H H F^{k'}. \tag{20.121b}$$

Proof: By construction, $Q_{n,0} \geq 0$. If $Q_{n,0}$ is invertible, then we may determine the initial state $\mathbf{x}_0$ on the LTV system from the measurements $\mathbf{y}_k$ for $k = 0, 1, \ldots, n-1$ by substituting $\mathbf{x}_k = \Phi_{k,0} \mathbf{x}_0$ into $\mathbf{y}_k = H_k \mathbf{x}_k$, multiplying the result from the left by $\Phi_{k,0}^H H_k^H$, and summing from $k = 0$ to $n-1$, resulting in

$$\mathbf{x}_0 = Q_{n,0}^{-1} \sum_{k=0}^{n-1} \Phi_{k,0}^H H_k^H \mathbf{y}_k, \tag{20.122a}$$

which reduces in the LTI case to

$$\mathbf{x}_0 = Q_{n,0}^{-1} \sum_{k=0}^{n-1} (F^H)^k H^H \mathbf{y}_k. \tag{20.122b}$$

On the other hand, if $Q_{n,0}$ is not invertible, then there is some direction $\mathbf{x}_0 \neq 0$ such that $\mathbf{x}_0^H Q_{n,0} \mathbf{x}_0 = 0$. Thus, in both the LTV and LTI cases,

$$\begin{aligned} \mathbf{x}_0^H \Big[\sum_{k=0}^{n-1} \Phi_{k,0}^H H_k^H H_k \Phi_{k,0} \Big] \mathbf{x}_0 &= \sum_{k=0}^{n-1} \big[\mathbf{x}_0^H \Phi_{k,0}^H H_k^H H_k \Phi_{k,0} \mathbf{x}_0 \big] \\ = \sum_{k=0}^{n-1} \| H_k \Phi_{k,0} \mathbf{x}_0 \|^2 = 0 \qquad &\Rightarrow \qquad \mathbf{y}_k = H_k \Phi_{k,0} \mathbf{x}_0 = 0 \quad \text{for } k = 0, 1, \ldots, n-1; \end{aligned} \tag{20.123}$$

that is, there are some $\mathbf{x}_0 \neq 0$ which are not reflected in *any* of the measurements, and thus the component of $\mathbf{x}_0$ in these directions can not be determined from the $\mathbf{y}_k$. □

Noting in the LTV case the form of $Q_{n,0}$ in (20.121a), and in the LTI case the form of $Q_{n,0}$ in (20.121b), it is easily verified by substitution that $Q_{n,k}$ satisfies the Lyapunov difference equation

$$F_k^H Q_{n,k} F_k = Q_{n,k+1} - H_k^H H_k \quad \text{with} \quad Q_{n,n} = 0. \tag{20.124}$$

In the LTI case, for large n, $Q_{n,0}$ approaches the solution of the discrete-time algebraic Lyapunov equation

$$Q = F^H Q F + H^H H. \tag{20.125}$$

As in (20.103) the rank of the discrete-time observability gramian is *invariant* under any state transformation (20.3).

Again, consider a stable DT LTI state-space form (20.2), recalling that its impulse response may be written $g_k = H F^k G h_{1k} + D \delta_{0k}$. By (20.48a) and (20.121b),

$$\begin{aligned} \|G(z)\|_2^2 = \sum_{k=0}^{\infty} \operatorname{trace} \Big[g_k^H g_k \Big] &= \operatorname{trace} \Big[\sum_{k=0}^{\infty} (D^H \delta_{0k} + G^H [F^H]^k H^H h_{1k}) (H F^k G h_{1k} + D \delta_{0k}) \Big] \\ &= \operatorname{trace} \big[G^H Q G + D^H D \big], \end{aligned} \tag{20.126}$$

where Q is the solution of (20.125), as stated in §20.2.2.1.

Finally, note that once $\mathbf{x}_0$ is known, it follows that $\mathbf{x}_{n-1} = F^{n-1} \mathbf{x}_0$ in the LTI case and $\mathbf{x}_{n-1} = \Phi_{n-1,0} \mathbf{x}_0$ in the LTV case.

20.5.4.1 A weaker condition: determining the final state only

We now focus specifically on the question of **discrete-time constructability** [that is, the question of whether or not one can reconstruct the *final* state $\mathbf{x}_n$ of a discrete-time system from the observations $\mathbf{y}_k$ for $k = 0, 1, \ldots, n-1$], again focusing initially on the LTI case. Following the analysis presented above, we may conclude the following:

Fact 20.20 *The discrete-time LTI system* (20.117) *is* **constructable** *iff all vectors in the nullspace of the discrete-time observability matrix* $\mathscr{O}$ *defined in Fact* 20.18 *are in the nullspace of* F^n.

Proof: If some vector $\mathbf{x}_0 \neq 0$ in the nullspace of $\mathscr{O}$ is not in the nullspace of F^n, then for this initial state $\mathbf{x}_0$, it follows that $\mathbf{x}_n = F^n \mathbf{x}_0 \neq 0$, yet, by (20.118), $\mathbf{y}_k = 0$ for $k = 0, \ldots, n-1$. That is, the system evolution leading

to the nonzero final state $\mathbf{x}_n \neq 0$ is not reflected in *any* of the measurements, and thus this nonzero final state $\mathbf{x}_n$ can not be determined from the $\mathbf{y}_k$. On the other hand, if all vectors in the nullspace of $\mathscr{O}$ are in the nullspace of F^n, then we may determine $\mathbf{x}_n$ from the measurements $\mathbf{y}_k$ simply by multiplying (20.118) from the left by $F^n\mathscr{O}^+$. □

Turning our attention to the more general LTV case, we may conclude the following:

Fact 20.21 *The discrete-time LTV system* (20.120) *is* **constructable** *iff all vectors in the nullspace of the discrete-time observability gramian Q_n defined in Fact 20.19 are in the nullspace of $\Phi_{n,0}$.*

Proof: If all vectors in the nullspace of Q_n are in the nullspace of $\Phi_{n,0}$, then multiplying (20.122) by $\Phi_{n,0}^H$ and replacing Q^{-1} by Q^+ gives the unique answer for $\mathbf{x}_n$:

$$\mathbf{x}_n = \Phi_{n,0}\mathbf{x}_0 = \Phi_{n,0}Q_n^+ \sum_{k=0}^{n-1} \Phi_{k,0}^H H_k^H \mathbf{y}_k, \tag{20.127a}$$

which reduces in the LTI case to

$$\mathbf{x}_n = F^n\mathbf{x}_0 = F^n Q_n^+ \sum_{k=0}^{n-1} (F^H)^k H^H \mathbf{y}_k. \tag{20.127b}$$

On the other hand, if some vector $\mathbf{x}_0$ in the nullspace of Q_n is not in the nullspace of $\Phi_{n,0}$, then we have both $\mathbf{x}_n = \Phi_{n,0}\mathbf{x}_0 \neq 0$ and, as shown in (20.123), $\mathbf{y}_k = H_k\Phi_{k,0}\mathbf{x}_0 = 0$ for $k = 0, 1, \ldots, n-1$. That is, the system evolution leading to the nonzero final state $\mathbf{x}_n \neq 0$ is not reflected in *any* of the measurements, and thus this nonzero final state $\mathbf{x}_n$ can not be determined from the $\mathbf{y}_k$. □

The DT & CT controllability & observability matrices & gramians developed above are computed in the straightforward `CtrbMatrix.m`, `CtrbGrammian.m`, `ObsvMatrix.m`, and `ObsvGrammian.m` codes in the *NRC*.

20.5.5 Output tracking in LTI systems

The continuous-time controllability and discrete-time reachability questions discussed above address the problem of moving a system from any specified initial state to any specified final state, but do not address the path taken to move between these two states. In this section, we address the problem of following a specified path, not necessarily for the entire state, but for the specified outputs of interest. Such a problem is of interest in a variety of problems, such as moving the cutting head of milling machine or maneuvering a welder on a robotic arm into a hard-to-reach corner of a frame on an assembly line.

Continuous-time case

Starting from $\mathbf{y}(t) - C\mathbf{x}(t) = D\mathbf{u}(t)$, differentiating and substituting $\mathbf{x}'(t) = A\mathbf{x}(t) + B\mathbf{u}(t)$ to eliminate $\mathbf{x}'(t)$, and repeating m times (for any integer $m > 0$) leads to

$$\begin{bmatrix} \mathbf{y}(t) \\ \mathbf{y}'(t) \\ \mathbf{y}''(t) \\ \vdots \\ \mathbf{y}^{(m)}(t) \end{bmatrix} - \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^m \end{bmatrix} \mathbf{x}(t) = \begin{bmatrix} D & & & & 0 \\ CB & D & & & \\ CAB & CB & D & & \\ \vdots & \vdots & \ddots & \ddots & \\ CA^{m-1}B & CA^{m-2}B & \ldots & CB & D \end{bmatrix} \begin{bmatrix} \mathbf{u}(t) \\ \mathbf{u}'(t) \\ \mathbf{u}''(t) \\ \vdots \\ \mathbf{u}^{(m)}(t) \end{bmatrix}.$$

For any *prescribed* $\mathbf{y}(t)$, this equation may be solved for the unknown vector of inputs on the RHS iff the block lower-triangular Toeplitz matrix of Markov parameters on the RHS has full row rank. Note that, in

general, this equation is only solvable if $D \neq 0$ (more precisely, if D itself has full row rank). If $D = 0$, which is common, then solution(s) to the above equation may only exist if $\mathbf{y}(t) - C\mathbf{x}(t) = 0$ (that is, if the system "starts out at the right place"). This is a reasonable assumption for most tracking problems; if the system does not "start out at the right place" and $D = 0$, then an impulsive input $\mathbf{u}$ would be required to move the system the moment the control is turned on, which is nonphysical.

In practice, in order to track a *desired* output signal $\mathbf{y}_d(t)$ on the interval $t \in [0,T]$, assuming this desired output signal is well approximated by a smooth function $\mathbf{y}(t)$ with only m nonzero derivatives on this interval (for any m) such that

$$\mathbf{y}(t) = \mathbf{y}(0) + \mathbf{y}'(0)t + \mathbf{y}''(0)\frac{t^2}{2!} + \ldots + \mathbf{y}^{(m)}(0)\frac{t^m}{m!} \quad \text{on} \quad t \in [0,T],$$

we may differentiate this smoothed desired output $\mathbf{y}(t)$ m times, solve the above equation at $t = 0$, and use the values of $\{\mathbf{u}(0), \mathbf{u}'(0), \mathbf{u}''(0), \ldots, \mathbf{u}^{(m)}(0)\}$ so determined to construct a control $\mathbf{u}(t)$ that tracks $\mathbf{y}(t)$ using a truncated Taylor series as follows:

$$\mathbf{u}(t) = \mathbf{u}(0) + \mathbf{u}'(0)t + \mathbf{u}''(0)\frac{t^2}{2!} + \ldots + \mathbf{u}^{(m)}(0)\frac{t^m}{m!} \quad \text{on} \quad t \in [0,T].$$

Note that it is sufficient to approximate the desired output signal $\mathbf{y}_d(t)$ with a piecewise smooth function, solving the above problem on $t \in [0,T_1]$, then an analogous problem on $t \in [T_1,T_2]$, etc.

Discrete-time case

Starting from $\mathbf{y}_k - H\mathbf{x}_k = D\mathbf{u}_k$, evaluating for successive timesteps and substituting $\mathbf{x}_{k+1} = F\mathbf{x}_k + G\mathbf{u}_k$ to eliminate $\mathbf{x}_{k+1}$, and repeating m times (for any integer $m > 0$) leads to

$$\begin{bmatrix} \mathbf{y}_k \\ \mathbf{y}_{k+1} \\ \mathbf{y}_{k+2} \\ \vdots \\ \mathbf{y}_{k+m} \end{bmatrix} - \begin{bmatrix} H \\ HF \\ HF^2 \\ \vdots \\ HF^m \end{bmatrix} \mathbf{x}_k = \begin{bmatrix} D & & & & 0 \\ HG & D & & & \\ HFG & HG & D & & \\ \vdots & \vdots & \ddots & \ddots & \\ HF^{m-1}G & HF^{m-2}G & \ldots & HG & D \end{bmatrix} \begin{bmatrix} \mathbf{u}_k \\ \mathbf{u}_{k+1} \\ \mathbf{u}_{k+2} \\ \vdots \\ \mathbf{u}_{k+m} \end{bmatrix}.$$

Again, for any prescribed $\mathbf{y}_k$, this equation may be solved for the unknown vector on the RHS iff the block lower-triangular Toeplitz matrix of Markov parameters on the RHS has full row rank. In general, this equation is only solvable if $D \neq 0$; if $D = 0$, solution(s) to the above equation may only exist if $\mathbf{y}_k - H\mathbf{x}_k = 0$ (that is, if the system "starts out at the right place"). In order to track a desired output sequence $\{\mathbf{y}_k, \mathbf{y}_{k+1}, \ldots, \mathbf{y}_{k+m}\}$, we may solve the above equation to determine directly the necessary $\{\mathbf{u}_k, \mathbf{u}_{k+1}, \ldots, , \mathbf{u}_{k+m}\}$.

Alternatively, one may simply ignore $\mathbf{y}_k$ and instead seek a control sequence $\{\mathbf{u}_k, \mathbf{u}_{k+1}, \ldots, , \mathbf{u}_{k+m-1}\}$ with $\mathbf{u}_{k+m} = 0$ to track a desired output sequence $\{\mathbf{y}_{k+1}, \mathbf{y}_{k+2}, \ldots, \mathbf{y}_{k+m}\}$ by solving the following system:

$$\begin{bmatrix} \mathbf{y}_{k+1} \\ \mathbf{y}_{k+2} \\ \mathbf{y}_{k+3} \\ \vdots \\ \mathbf{y}_{k+m} \end{bmatrix} - \begin{bmatrix} HF \\ HF^2 \\ HF^3 \\ \vdots \\ HF^m \end{bmatrix} \mathbf{x}_k = \begin{bmatrix} HG & D & & & 0 \\ HFG & HG & D & & \\ HF^2G & HFG & HG & \ddots & \\ \vdots & \vdots & \ddots & \ddots & D \\ HF^{m-1}G & HF^{m-2}G & \ldots & HFG & HG \end{bmatrix} \begin{bmatrix} \mathbf{u}_k \\ \mathbf{u}_{k+1} \\ \mathbf{u}_{k+2} \\ \vdots \\ \mathbf{u}_{k+m-1} \end{bmatrix}.$$

This equation may be solved for the unknown vector on the RHS iff the block lower-Hessenberg Toeplitz matrix of Markov parameters on the RHS has full row rank.

20.6 Model reduction

If the available model of a system is too complex for computing feedback, it is useful to reduce the model complexity while still representing its inherent input-output relationship as faithfully as possible. There are a variety of approaches available to accomplish this; two of the essential methods are discussed below.

20.6.1 Removing uncontrollable and unobservable modes: minimal realizations

Consider now a strictly proper MIMO system in CT state-space form

$$\left.\begin{aligned}\overline{\mathbf{x}}'(t) &= \overline{A}(t)\overline{\mathbf{x}}(t) + \overline{B}(t)\mathbf{u}(t)\\ \mathbf{y}(t) &= \overline{C}(t)\overline{\mathbf{x}}(t)\end{aligned}\right\} \quad \Leftrightarrow \quad \overline{S} = \left[\begin{array}{c|c}\overline{A} & \overline{B}\\ \hline \overline{C} & 0\end{array}\right], \tag{20.128}$$

which may be uncontrollable, unobservable, or both. In this section, we develop state transformations, as described in §20.1.1, designed to isolate and eliminate the uncontrollable and unobservable modes of the system $\overline{S}$, which do not contribute to its input/output transfer function.

To begin, consider the transformation of the of the system $\overline{S}$ via a transformation matrix Q_c into the **controllability block staircase** form

$$S_{c\overline{c}} \triangleq \left[\begin{array}{c|c}Q_c^{-1}\overline{A}Q_c & Q_c^{-1}\overline{B}\\ \hline \overline{C}Q_c & 0\end{array}\right] = \left[\begin{array}{cc|c}A_c & A_{1,2} & B_c\\ 0 & A_{\overline{c}} & 0\\ \hline C_c & C_{\overline{c}} & 0\end{array}\right], \qquad \mathbf{x} = Q_c^{-1}\overline{\mathbf{x}} = \begin{bmatrix}\mathbf{x}_c\\ \mathbf{x}_{\overline{c}}\end{bmatrix}, \tag{20.129}$$

where the subsystem $S_c \triangleq \left[\begin{array}{c|c}A_c & B_c\\ \hline C_c & 0\end{array}\right]$ is controllable, and $S_{\overline{c}} \triangleq \left[\begin{array}{c|c}A_{\overline{c}} & 0\\ \hline C_{\overline{c}} & 0\end{array}\right]$ is null controllable. The matrix $Q_c = \begin{bmatrix}\underline{Q}_c & \overline{Q}_c\end{bmatrix}$ required to achieve this separation is easy to compute, since $\underline{Q}_c$ must form a basis for the column space of the controllability matrix of the original system, $\mathscr{C}(\overline{S})$, and $\overline{Q}_c$ forms a basis for its orthogonal complement [that is, for the left nullspace of $\mathscr{C}(\overline{S})$]. Thus, both $\underline{Q}_c$ and $\overline{Q}_c$ may be determined by applying Algorithm 4.6 to obtain a pivoted QR decomposition (see §4.4.2) of the controllability matrix $\mathscr{C}(\overline{S})$:

$$\begin{bmatrix}\underline{Q}_c & \overline{Q}_c\end{bmatrix}\begin{bmatrix}\underline{R}\\ \hline 0\end{bmatrix} = \mathscr{C}(\overline{S})\Pi,$$

noting that Π is the appropriate permutation matrix for this problem (recall that pivoting must be applied to a QR decomposition algorithm in order to identify the block partitioning shown above).

Likewise, consider the transformation of $\overline{S}$ via a transformation matrix Q_o into the **observability block staircase** form

$$S_{o\overline{o}} \triangleq \left[\begin{array}{c|c}Q_o^{-1}\overline{A}Q_o & Q_o^{-1}\overline{B}\\ \hline \overline{C}Q_o & 0\end{array}\right] = \left[\begin{array}{cc|c}A_o & 0 & B_o\\ A_{2,1} & A_{\overline{o}} & B_{\overline{o}}\\ \hline C_o & 0 & 0\end{array}\right], \qquad \mathbf{x} = Q_o^{-1}\overline{\mathbf{x}} = \begin{bmatrix}\mathbf{x}_o\\ \mathbf{x}_{\overline{o}}\end{bmatrix}, \tag{20.130}$$

where the subsystem $S_o \triangleq \left[\begin{array}{c|c}A_o & B_o\\ \hline C_o & 0\end{array}\right]$ is observable, and $S_{\overline{o}} \triangleq \left[\begin{array}{c|c}A_{\overline{o}} & B_{\overline{o}}\\ \hline 0 & 0\end{array}\right]$ is null observable. The matrix $Q_o = \begin{bmatrix}\underline{Q}_o & \overline{Q}_o\end{bmatrix}$ required to achieve this separation is also easy to compute, since $\underline{Q}_o$ must form a basis for the column space of observability matrix of the original system, $\mathscr{O}(\overline{S})$, and $\overline{Q}_o$ forms a basis for its orthogonal complement [that is, for the left nullspace of $\mathscr{O}(\overline{S})$]. Thus, both $\underline{Q}_o$ and $\overline{Q}_o$ may be determined by applying Algorithm 4.6 to obtain a pivoted QR decomposition of the observability matrix $\mathscr{O}(\overline{S})$:

$$\begin{bmatrix}\underline{Q}_o & \overline{Q}_o\end{bmatrix}\begin{bmatrix}\underline{R}\\ \hline 0\end{bmatrix} = \mathscr{O}(\overline{S})\Pi.$$

Finally, consider a transformation which first separates the controllable and uncontrollable modes, then further separates each of those sets of modes into those modes that are observable and those that are unobservable; if ordered carefully (see Algorithm 20.9), the resulting **block Kalman** form may be written

$$S \triangleq \left[\begin{array}{c|c} A & B \\ \hline C & 0 \end{array}\right] = \left[\begin{array}{c|c} Q^{-1}\overline{A}Q & Q^{-1}\overline{B} \\ \hline \overline{C}Q & 0 \end{array}\right] = \left[\begin{array}{cccc|c} A_{c,o} & 0 & A_{1,3} & 0 & B_{c,o} \\ A_{2,1} & A_{c,\overline{o}} & A_{2,3} & A_{2,4} & B_{c,\overline{o}} \\ 0 & 0 & A_{\overline{c},o} & 0 & 0 \\ 0 & 0 & A_{4,3} & A_{\overline{c},\overline{o}} & 0 \\ \hline C_{c,o} & 0 & C_{\overline{c},o} & 0 & 0 \end{array}\right], \quad \mathbf{x} = Q^{-1}\overline{\mathbf{x}} = \begin{bmatrix} \mathbf{x}_{c,o} \\ \mathbf{x}_{c,\overline{o}} \\ \mathbf{x}_{\overline{c},o} \\ \mathbf{x}_{\overline{c},\overline{o}} \end{bmatrix}, \tag{20.131}$$

where the subsystem $S_{c,o} \triangleq \left[\begin{array}{c|c} A_{c,o} & B_{c,o} \\ \hline C_{c,o} & 0 \end{array}\right]$ governing the evolution of the modes $\mathbf{x}_{c,o}$ is both controllable and observable, and the remaining modes are either null controllable, null observable, or both. Note that, by repeated application of Fact 4.12 (first on a 2×2 block partitioning of A with zero in the lower-left element, then on the 2×2 block partitioning of the diagonal elements of the result), it follows that the eigenvalues of A, and thus the eigenvalues of $\overline{A}$ itself, are given by the union of the eigenvalues of $A_{c,o}$, $A_{c,\overline{o}}$, $A_{\overline{c},o}$, and $A_{\overline{c},\overline{o}}$. Further, leveraging Facts 20.9 and 2.1, it is straightforward to verify that, in transfer function form,

$$G(s) = \overline{C}(sI - \overline{A})^{-1}\overline{B} = C(sI - A)^{-1}B = C_{c,o}(sI - A_{c,o})^{-1}B_{c,o}.$$

The $S_{c,o}$ subsystem is thus referred to as a **minimal** realization, as is reduces the order of the state-space form to the minimum possible while still preserving (exactly) the transfer function of the original system.

Algorithm 20.9 illustrates how to transform from a *general* (full) state-space form into the *controllability block staircase*, *observability block staircase*, *block Kalman*, and *minimal* forms discussed above, in addition to the several canonical forms discussed in §20.3, recalling that

- when mapping from a *general* state-space form into *reachability* canonical form, the new controllability matrix $\mathscr{C}_R$ (in CT), or the new reachability matrix $\mathscr{R}_R$ (in DT), is the identity matrix, and thus (20.86) reveals the required transformation matrix R;
- when mapping from *reachability* canonical form into *controller* canonical form, the required transformation matrix is $R = R_1^T$ [see (20.54)];
- when mapping from *DT controller* canonical form into *DT controllability* canonical form, the required transformation matrix is $R = -R_2^{-1}$ [see (20.67)];
- when mapping from a *general* state-space form into *observability* canonical form, the new observability matrix $\mathscr{O}_R$ is the identity matrix, and thus (20.97) reveals the required transformation matrix R;
- when mapping from *observability* canonical form into *observer* canonical form, the required transformation matrix is $R = R_1^{-1}$;
- when mapping from *DT observer* canonical form into *DT constructibility* canonical form, the required transformation matrix is $R = -R_2$.

20.6.2 Removing modes with poor controllability/observability: balanced truncation

The reduction to a minimal realization, as discussed in §20.6.1, is based on eliminating individual states from a state-space representation that are either null controllable, null observable, or both, as such states do not contribute to the input/output transfer function of the system. This idea may be extended to eliminate states in a state-space representation that have *small* (but nonzero) controllability and observability, as such states have *diminished* (though nonzero) effect on the input/output transfer function of the system.

To accomplish this, the system in question is first transformed into a **balanced realization** (Moore 1981) in which the transformed system has *equal* and *diagonal* controllability and observability grammians, with the non-negative elements on the main diagonal of both, called the **Hankel singular values** of the system,

listed in decreasing order. Once written in such a fashion, the trailing modes in this balanced realization (which are distinguished by both diminished observability *and* diminished controllability) may then simply be removed from the realization while having a diminished (though nonzero) impact on the overall input-output transfer function of the model; this process is referred to as **balanced truncation**.

To proceed, consider a CT[15], controllable[16], strictly-proper LTI state-space system

$$\begin{aligned} \mathbf{x}' &= A\mathbf{x} + B\mathbf{u} \\ \mathbf{y} &= C\mathbf{x} \end{aligned} \qquad \Leftrightarrow \qquad G = \left[\begin{array}{c|c} A & B \\ \hline C & 0 \end{array}\right]. \tag{20.132}$$

The controllability gramian $P \geq 0$ and observability gramian $Q \geq 0$, introduced in §20.5.1.2 and §20.5.2.2, are powerful tools that allow us to *quantify* the extent to which a particular state of such a system is controllable and observable. They are defined [see (20.88b) and (20.99b)] via the integrals

$$P(T) = \int_0^T e^{At}\, B B^H\, e^{A^H t}\, dt, \qquad Q_T(0) = \int_0^T e^{A^H t}\, C^H C\, e^{At}\, dt.$$

In the LTI case with stable A in the infinite-horizon ($T \to \infty$) limit, they are most easily computed, as shown in (20.91b) and (20.102), as the solutions of the continuous-time algebraic Lyapunov equations

$$0 = AP + PA^H + BB^H, \qquad 0 = A^H Q + QA + C^H C.$$

To achieve a balanced realization, as implemented in Algorithm 20.10, we will perform a state transform [see (20.3)] to (20.132) such that

$$\mathbf{x}_b = R^{-1}\mathbf{x}, \quad G_b = \left[\begin{array}{c|c} A_b & B_b \\ \hline C_b & 0 \end{array}\right] = \left[\begin{array}{c|c} R^{-1}AR & R^{-1}B \\ \hline CR & 0 \end{array}\right]. \tag{20.133}$$

Taking the Cholesky decompositions $P = G_P G_P^H$ and $Q = G_Q G_Q^H$, where the Cholesky factors G_P and G_Q are lower triangular (see §4.4.8), and defining $B = G_P^H G_Q$, we compute the singular value decomposition $B = U\Sigma V^H$ (see §4.4.7), where $\Sigma = \mathrm{diag}(\sigma_1, \cdots, \sigma_n)$ with $\sigma_1 \geq \cdots \geq \sigma_n \geq 0$; note that the σ_i are referred to as the **Hankel singular values** of the system. The **balanced realization** G_b is then given by taking $R = G_P U \Sigma^{-1/2}$ and performing the state transformation (20.133), from which it follows that

$$\begin{aligned} P_b &= R^{-1} P R^{-H} = \Sigma^{1/2} U^{-1} G_P^{-1} \left[G_P G_P^H\right] G_P^{-H} U^{-H} \Sigma^{1/2} = \Sigma, \\ Q_b &= R^H Q R = \Sigma^{-1/2} U^H G_P^H \left[G_Q G_Q^H\right] G_P U \Sigma^{-1/2} = \Sigma^{-1/2} U^H B B^H U \Sigma^{-1/2} = \Sigma^{-1/2} \Sigma V^H V \Sigma \Sigma^{-1/2} = \Sigma. \end{aligned}$$

We now endeavor to interpret $P(T)$ a bit more deeply. Recall from (20.89b) that

$$\mathbf{u}(t) = B^H e^{A^H(T-t)} P^{-1}(T)\, \overline{\mathbf{x}} \quad \text{for } t \in [0, T] \tag{20.134}$$

is one possible choice for a control input $\mathbf{u}(t)$ on $t \in [0,T]$ that takes the CT system (20.132) from a zero initial condition, $\mathbf{x}(0) = 0$, to a specified terminal condition $\mathbf{x}(T) = \overline{\mathbf{x}}$; that is, which satisfies

$$\mathbf{x}(T) = \overline{\mathbf{x}} \quad \text{where} \quad \mathbf{x}(t) = \int_0^t e^{A(t-\tau)} B\, \mathbf{u}(\tau)\, d\tau. \tag{20.135}$$

We now establish that the control input (20.134) is actually the control with *minimum* "input energy",

$$J_{\mathbf{u}} = \int_0^T \mathbf{u}(t)^H \mathbf{u}(t)\, dt, \tag{20.136}$$

[15] We illustrate the process of balanced truncation by examining the CT case; the DT case is entirely analogous.
[16] This condition is easily relaxed, using P^+ in place of P^{-1} in the analysis that follows.

that satisfies (20.135). To show this, we may simply perform a perturbation analysis: replacing $\mathbf{u}(t)$ with $\mathbf{u}(t)+\mathbf{u}'(t)$ and $J_{\mathbf{u}}$ with $J_{\mathbf{u}}+J'_{\mathbf{u}}$ in (20.136) and applying $\mathbf{x}'(T)=\int_0^T e^{A(T-\tau)}B\,\mathbf{u}'(\tau)\,d\tau=0$, it follows that

$$J_{\mathbf{u}}+J'_{\mathbf{u}}=\int_0^T[\mathbf{u}(t)+\mathbf{u}'(t)]^H[\mathbf{u}(t)+\mathbf{u}'(t)]\,dt \quad \Rightarrow$$
$$J'_{\mathbf{u}}=\overline{\mathbf{x}}^H P^{-1}(T)\underbrace{\int_0^T e^{A(T-t)}B\,\mathbf{u}'(t)\,dt}_{=0}+\underbrace{\int_0^T[\mathbf{u}'(t)]^H B^H e^{A^H(T-t)}\,dt}_{=0}P^{-1}(T)\,\overline{\mathbf{x}}+\int_0^T[\mathbf{u}'(t)]^H\mathbf{u}'(t)\,dt.$$

In the limit that $\|\mathbf{u}'(t)\|\to 0$, we may define the **gradient** $\mathbf{g}(t)$ and the **Hessian** $H(t)$ on $t\in[0,T]$ via

$$J'_{\mathbf{u}}=\int_0^T\mathbf{g}^H(t)\,\mathbf{u}'(t)\,dt+\int_0^T[\mathbf{u}'(t)]^H H(t)\,\mathbf{u}'(t)]\,dt; \tag{20.137}$$

it is thus observed that $\mathbf{g}(t)=0$ and $H(t)>0$ on $t\in[0,T]$ —in other words, that the solution (20.134) locally minimizes $J_{\mathbf{u}}$. Since $J_{\mathbf{u}}$ is quadratic in $\mathbf{u}$, this expression for $\mathbf{u}$, in fact, *globally* minimizes $J_{\mathbf{u}}$; note further that, combining (20.136) and (20.134), this minimum "input energy" is given by

$$J_{\mathbf{u}}=\int_0^T\overline{\mathbf{x}}^H[P(T)]^{-1}e^{A(T-t)}BB^He^{A^H(T-t)}[P(T)]^{-1}\overline{\mathbf{x}}\,dt=\overline{\mathbf{x}}^H[P(T)]^{-1}\overline{\mathbf{x}}. \tag{20.138}$$

We may similarly interpret $Q_T(0)$ a bit more deeply. Recall from (20.99b) that

$$Q_T(0)=\int_0^T e^{A^Ht}C^HCe^{At}\,dt.$$

Thus, assuming $\mathbf{u}(t)=0$, the "output energy" of $\mathbf{y}(t)$ over the interval $t\in[0,T]$ for a system initialized at $\mathbf{x}(0)=\overline{\mathbf{x}}$ may be written

$$J_{\mathbf{y}}=\int_0^T\mathbf{y}(t)^H\mathbf{y}(t)\,dt=\int_0^T\overline{\mathbf{x}}^He^{A^Ht}C^HCe^{At}\overline{\mathbf{x}}\,dt=\overline{\mathbf{x}}^HQ_T(0)\overline{\mathbf{x}}. \tag{20.139}$$

To summarize, the expression $\overline{\mathbf{x}}^H[P(T)]^{-1}\overline{\mathbf{x}}$ quantifies the minimum "input energy" necessary to steer a system from $\mathbf{x}(0)=0$ to $\mathbf{x}(T)=\overline{\mathbf{x}}$, whereas the expression $\overline{\mathbf{x}}^HQ_T(0)\overline{\mathbf{x}}$ quantifies the "output energy" over the interval $t\in[0,T]$ of a system initialized at $\mathbf{x}(0)=\overline{\mathbf{x}}$. States $\overline{\mathbf{x}}$ for which $\overline{\mathbf{x}}^H[P(T)]^{-1}\overline{\mathbf{x}}$ is large (that is, states for which $\overline{\mathbf{x}}^HP(T)\overline{\mathbf{x}}$ is small) are said to have **poor controllability**, whereas states $\overline{\mathbf{x}}$ for which $\overline{\mathbf{x}}^HQ_T(0)\overline{\mathbf{x}}$ is small are said to have **poor observability**; further, both properties may be considered for finite T or, as usually done when performing balanced truncation, in the "infinite horizon" limit $T\to\infty$.

As motivated above, the states corresponding to the diminished Hankel singular values have diminished impact on the input/output transfer function of the system, and may thus be eliminated without significantly corrupting the input-output transfer function of the system. Thus, let Σ_2 contain the negligible Hankel singular values $(\sigma_{r+1},\cdots,\sigma_n)$, and partition in the balanced realization G_b such that

$$G_b=\left[\begin{array}{cc|c}A_{11}&A_{12}&B_1\\A_{21}&A_{22}&B_2\\\hline C_1&C_2&0\end{array}\right],\qquad \Sigma=\left[\begin{array}{c|c}\Sigma_1&0\\\hline 0&\Sigma_2\end{array}\right]. \tag{20.140}$$

The desired reduced-order model G_r is then obtained simply by truncating those states associated with Σ_2:

$$G_r=\left[\begin{array}{c|c}A_{11}&B_1\\\hline C_1&0\end{array}\right]. \tag{20.141}$$

Algorithm 20.9: Convert a strictly-proper ($D = 0$) state-space form to any of a variety of canonical forms.

View
Test

```
function [A,B,C,r1,r2,r3,r4]=SS2CanonicalForm(A,B,C,FORM)
% Convert a general state-space model to one of a variety of canonical forms.
disp(' '), r1=0; r2=0; r3=0; r4=0; [n,ni]=size(B); [no,n]=size(C);
sisoforms={'Controller','Reachability', 'DTControllability', ...
           'Observer',  'Observability','DTConstructibility'};
if (ismember(FORM,sisoforms) & ni*no>1), disp('Error: invalid case.'), return, end, FORM
switch FORM          % Compute the transformation matrix
 case 'Reachability'
   Q=CtrbMatrix(A,B);
 case 'Controller'
   Q=CtrbMatrix(A,B); [A,B,C]=SSTransform(A,B,C,Q); a=-A(:,n)'; Q=R1(a,n)';
 case 'DTControllability'
   Q=CtrbMatrix(A,B); [A,B,C]=SSTransform(A,B,C,Q); a=-A(:,n)'; Q=-R1(a,n)'*Inv(R2(a,n));
 case 'Observability'
   Q=Inv(ObsvMatrix(A,C));
 case 'Observer'
   Q=Inv(ObsvMatrix(A,C)); [A,B,C]=SSTransform(A,B,C,Q); a=-A(n,:); Q=Inv(R1(a,n));
 case 'DTConstructibility'
   Q=Inv(ObsvMatrix(A,C)); [A,B,C]=SSTransform(A,B,C,Q); a=-A(n,:); Q=-R1(a,n)\R2(a,n);
 case 'ControllabilityBlockStaircase'
   [Q,R,pi,r1]=QRmgs(CtrbMatrix(A,B));  r2=n-r1;
 case 'ObservabilityBlockStaircase'
   [Q,R,pi,r1]=QRmgs(ObsvMatrix(A,C)'); r2=n-r1;
 case {'BlockKalman','Minimal'}
   % First, find orthogonal bases for the controllable/null-controllable subspaces,
   % and for the observable/null-observable subspaces.
   [Qcnc,R,pi,rc]=QRmgs(CtrbMatrix(A,B));   rnc=n-rc; Qc=Qcnc(:,1:rc); Qnc=Qcnc(:,rc+1:n);
   [Qono,R,pi,ro]=QRmgs(ObsvMatrix(A,C)'); rno=n-ro; Qo=Qono(:,1:ro); Qno=Qono(:,ro+1:n);
   % Find an orthogonal basis for the modes that are neither null-controllable nor
   % observable (that is, for the modes that are both controllable and null-observable).
   [Q,R,pi,r]=QRmgs([Qnc Qo]);              rcno =n-r;       Qcno =Q(:,r+1:n);
   % Find a basis for the remaining controllable modes, which are observable.
   [Q,R,pi,r]=QRmgs([Qcno Qc], rcno);       rco  =rc-rcno;   Qco  =Q(:,rcno+1:rc);
   % Find a basis for the remaining null-observable modes, which are null-controllable.
   [Q,R,pi,r]=QRmgs([Qcno Qno],rcno);       rncno=rno-rcno; Qncno=Q(:,rcno+1:rno);
   % Find a basis for the remaining modes, which are null-controllable and observable.
   [Q,R,pi,r]=QRmgs([Qco Qcno Qncno]);      rnco =n-r;       Qnco =Q(:,r+1:n);
   % Assemble these four bases into a transformation matrix Q.
   Q=[Qco Qcno Qnco Qncno]; r1=rco; r2=rcno; r3=rnco; r4=rncno;
 otherwise, disp('Error: invalid case.'), return
end
[A,B,C]=SSTransform(A,B,C,Q);  % Perform final transform of the system
if strcmp(FORM,'Minimal'), A=A(1:r1,1:r1); B=B(1:r1,:); C=C(:,1:r1); end
end % function SS2CanonicalForm
%%%
```

Exercises

Exercise 20.1 Dynamics of a three-story building, revisited Recall the three-story building modeled in Example 17.9.
(**a**) Defining $x_4 = dx_1/dt$, $x_5 = dx_2/dt$, $x_6 = dx_3/dt$, and taking the input $u = w$ with $v = 0$, write the system in (17.34) in state-space form.
(**b**) Repeat part (a), now taking the input $u = v$ with $w = 0$. Discuss.
(**c**) Taking $k = 10000$ kg/sec^2 and $c = 10$ kg/sec and using the `eig` command in Matlab, compute the eigenvalues of the system matrix A determined in part (a), and compare these eigenvalues to the poles of the corresponding transfer-function representation.
(**d**) Repeat part (c), now taking $c = 100$ kg/sec. Discuss.

Exercise 20.2 In general, $e^{A+B} \neq e^A e^B$. Under what conditions does equality hold in this expression?

Exercise 20.3 Verify the correctness of (20.89a)-(20.89b) by performing the substitutions mentioned in the text.

Exercise 20.4 In §20.1.5, we showed how to convert a continuous-time state-space system

$$\begin{aligned} \frac{d\mathbf{x}}{dt} &= A\mathbf{x} + B\mathbf{u} \\ \mathbf{y} &= C\mathbf{x} + D\mathbf{u} \end{aligned}$$

into the corresponding transfer function form $\mathbf{Y}(s) = G(s)\mathbf{U}(s)$, where

$$G(s) = \frac{Ds^n + (CS_{n-1}B + a_{n-1}D)s^{n-1} + (CS_{n-2}B + a_{n-2}D)s^{n-2} + \ldots + (CS_1B + a_1D)s + (CS_0B + a_0D)}{s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0},$$

and where the S_i and a_i could be computed easily using the resolvent algorithm [see (20.35) and Algorithm 20.3]. Describe how the eigenvalues of A are defined, and how the poles of $G(s)$ are defined. Based on the above result, describe why the eigenvalues of A and the poles of the corresponding $G(s)$ coincide.

Exercise 20.5 Following the general procedure described in the last paragraph of §20.6.1, noting Fact 1.8, attempt to convert, by hand, each of the following continuous-time state-space forms to *all four* of the continuous-time canonical forms, and discuss anything notable that comes up in each case:

(a) $A = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, B = \begin{pmatrix} 0 \\ 2 \end{pmatrix}, C = \begin{pmatrix} 4 & 0 \end{pmatrix}, D = 6.$ **(b)** $A = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, B = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, C = \begin{pmatrix} 4 & 0 \end{pmatrix}, D = 7.$

(c) $A = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, B = \begin{pmatrix} 0 \\ 2 \end{pmatrix}, C = \begin{pmatrix} 0 & 4 \end{pmatrix}, D = 8.$ **(d)** $A = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, B = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, C = \begin{pmatrix} 0 & 4 \end{pmatrix}, D = 9.$

Exercise 20.6 Consider a MIMO system in state-space form in which all of the eigenvalues λ_i of the system matrix A are distinct.

(a) What do we know about the eigenvectors $\mathbf{s}^i$ of A in this case?

(b) Is the matrix of eigenvalues, S, necessarily invertible in this case? [If the answer to this question is no, assume that S happens to be invertible for the A being considered in the rest of this problem.]

(c) Decompose $\mathbf{x}(t)$ in terms of its constituent eigenmodes, $\mathbf{x}(t) = \chi_1(t)\,\mathbf{s}^1 + \ldots + \chi_n(t)\,\mathbf{s}^n$. What is required of each eigenmode $\mathbf{s}^i$ for the corresponding component of $\mathbf{x}(t)$ to be be distinguishable in the measurements $\mathbf{y}(t)$?

(d) Stating question in part (c) another way, writing the system in modal coördinate form

$$\left\{\begin{array}{l}\mathbf{x}_m'(t) = \Lambda\mathbf{x}_m(t) + B_m\mathbf{u}(t)\\ \mathbf{y}(t) = C_m\mathbf{x}_m(t) + D\mathbf{u}(t)\end{array}\right\} \quad \text{with} \quad \left[\begin{array}{c|c}\Lambda & B_m\\ \hline C_m & D\end{array}\right] = \left[\begin{array}{c|c}S^{-1}AS & S^{-1}B\\ \hline CS & D\end{array}\right],$$

what property is required of C_m in order for the system to be observable?

(e) Finally, explain why the property identified in part (d) is required for observability in terms of $\mathscr{O}_m$ (that is, the observability matrix of the modal representation).

Exercise 20.7 Consider a MIMO system in state-space form in which some of the eigenvalues of the system matrix are repeated.

(a) What do we know about the eigenvectors $\mathbf{s}^i$ of A in this case?

(b) Is the matrix of eigenvalues, S, necessarily invertible in this case? [If the answer to this question is no, assume that S happens to be invertible for the A being considered in the rest of this problem.]

(c) What is required of each eigenmode $\mathbf{s}^i$ for the corresponding component of $\mathbf{x}(t)$ to be be distinguishable in the measurements $\mathbf{y}(t)$? Is this requirement any more restrictive than the requirement considered in part c of Exercise 20.6? If so, how? (think carefully here!)

(d) Stating question part (c) another way, writing the system in modal coördinate form, what property is required of C_m in order for the system to be observable? Again, is this requirement any more restrictive than the requirement considered in part d of Exercise 20.6? If so, how?

(e) Finally, explain why the property identified in (4d) is required for observability in terms of $\mathscr{O}_m$.

Exercise 20.8 Show that the series expansion of $G(z)$ in terms of the discrete-time Markov parameters, (20.75), is compatible with the series expansion of $G(s)$ in terms of the continuous-time Markov parameters, (20.77), in the case that the discrete-time state-space form considered is a consistent numerical approximation of the continous-time state-space form considered. Hint: follow similar substitutions as those performed in the last section of §20.1.5 to (20.75), regroup the infinite series into those terms that multiply CB, those terms that multiply CAB, etc., and then apply the identies given in (B.77) through (B.80) to reduce these sums into simple coefficients.

References

Antsakis, PJ, & Michel, AN (2005) *Linear Systems*. Birkhäuser.

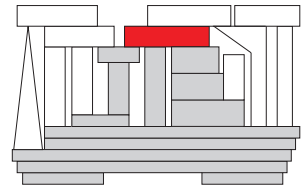
Chen, C-T (1998) *Linear System Theory & Design*. Oxford.

Kailath, T (1979) *Linear Systems*. Prentice-Hall.

Skelton, RE (1988) *Dynamic Systems Control, Linear Systems Analysis and Synthesis*. Wiley.

Skogestad, S, & Postlethwaite, I (2005) *Multivariable Feedback Control, Analysis and Design*. Wiley.

Chapter 21



State-space & model-predictive control design

Contents

21.1 Summary of the continuous-time (CT) case . . . 670
21.1.1 Control via adjoint-based iterative optimization ("model predictive control") . . . 671
21.1.2 Control via Riccati-based feedback ("optimal control") . . . 674
21.1.2.1 Interpretation of the matrix X in the optimal control setting . . . 675
21.1.3 Estimation via adjoint-based iterative optimization ("4Dvar" or "MHE") . . . 676
21.1.4 Estimation via Riccati-based feedback ("Kalman filtering") . . . 678
21.1.5 The separation principle: combining optimal control and Kalman filtering . . . 679
21.2 Summary of the discrete-time (DT) case . . . 683
21.2.1 Control via adjoint-based iterative optimization . . . 683
21.2.2 Control via Riccati-based feedback . . . 684
21.2.3 Estimation via adjoint-based iterative optimization . . . 685
21.2.4 Estimation via Riccati-based feedback . . . 686
21.2.5 The separation principle: putting it together . . . 688
21.2.6 Robust control . . . 688
21.3 Adjoint-based analysis of mixed CT/DT formulations . . . 689
21.3.1 Control of mixed CT/DT formulations . . . 689
21.3.2 Estimation of mixed CT/DT formulations . . . 689
21.4 MPDopt . . . 693
21.5 Feedback control of high-dimensional systems . . . 693
Exercises . . . 693

Leveraging the ODE simulation methods presented in §10, the numerical optimization methods discussed in §15, the transfer function approach to representation & control of dynamic systems in §17 & §18, and the state-space representations of dynamic systems §20, the stage is now set to:

a) optimize a distribution (in time) of nominal control inputs, $\bar{\mathbf{u}}(t)$, to drive a nonlinear ODE system along some nominal trajectory, $\bar{\mathbf{x}}(t)$, in order to satisfy a desired objective, while

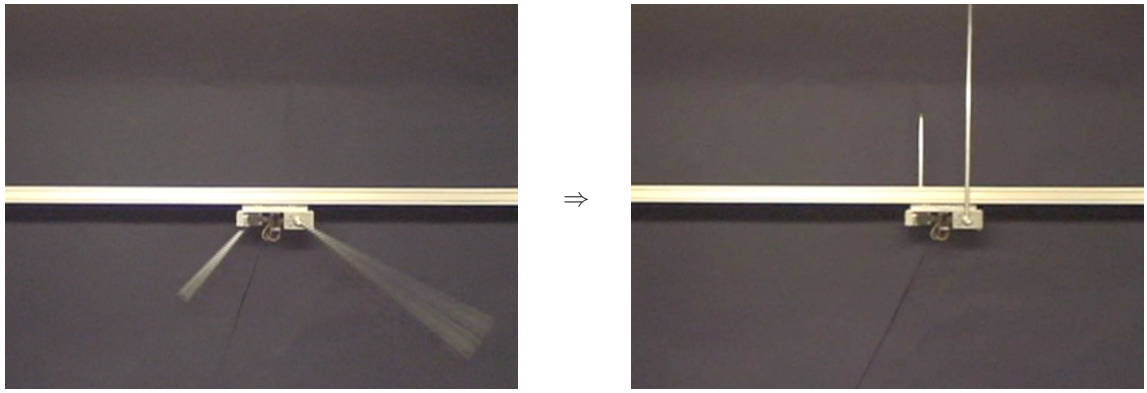


Figure 21.1: Swing up and stabilization of the **dual inverted pendulum**. [Construction by Robert Hughes, control solution by David Szeto (UCSD Coordinated Robotics Lab).]

b) coördinating small corrections $\mathbf{u}'(t)$ to these nominal control inputs $\bar{\mathbf{u}}(t)$ based on limited real-time measurements of the system $\mathbf{y}(t)$ in order to minimize the perturbations $\mathbf{x}'(t)$ from the optimized nominal trajectory $\bar{\mathbf{x}}(t)$ despite unknown state disturbances, measurement errors, and modeling errors.

This chapter explains the mathematical foundation for accomplishing these two tasks; ultimately, the control to be applied to the system will be the nominal plus the corrections, $\mathbf{u}(t) = \bar{\mathbf{u}}(t) + \mathbf{u}'(t)$. To focus the presentation, it is instructive to have a representative problem in mind. Thus, though the method presented has broad applicability and is easily generalized in a variety of ways, the **dual inverted pendulum** swing-up and stabilization problem illustrated in Figure 21.1 is useful to guide your thinking throughout the following discussion. In this problem, there are two pendulums of different lengths hanging from a cart; each pendulum rotates freely about the end attached to the cart (one pendulum hangs behind the track and the other hangs in front, so there is no interference between the two). The angles $\{\theta_1(t), \theta_2(t)\}$ of the pendulums are measured counterclockwise from upright, and the position $z(t)$ of the cart is measured from the center of the track. The goal is to bring the system to the unstable equilibrium state $\{z, \theta_1, \theta_2, \dot{z}, \dot{\theta}_1, \dot{\theta}_2\} = \{0,0,0,0,0,0\}$, and to keep it there, solely by pushing the cart to the left and right. This problem is studied in depth in Example 21.1.

21.1 Summary of the continuous-time (CT) case

An introduction to CT state-space control and estimation strategies, via both iterative adjoint-based optimization and direct Riccati-based feedback, is now given. The presentation is organized as follows:

- §21.1.1 & 21.1.2 consider the **control problem** (i.e., the determination of appropriate inputs to a system to achieve a desired objective assuming accurate knowledge of the system state), whereas
- §21.1.3 & 21.1.4 consider the **estimation problem** (i.e., the approximation of the system state based on recent, limited, noisy measurements of the actual system).

Denoting the current time as $t = 0$, the control problem considers possible evolutions of the system over a horizon of interest reaching into the *near future*, $[0, T]$, whereas the estimation problem considers possible fits of the system model to the history of available measurements over a horizon of interest reaching into the *recent past*, $[-T, 0]$. Together, solutions of the control and estimation problems facilitate the coördination of a limited number of actuators with a limited number of sensors to achieve a desired effect, as discussed in §21.1.5.

The *iterative approach* to these two problems (in §21.1.1 & 21.1.3) is applicable to both *nonlinear systems* and *nonquadratic cost functions*. Significantly, this approach only requires the computation of *vectors* (that

is, **state vectors** $\mathbf{x}(t)$ and corresponding **adjoint vectors** $\mathbf{r}(t)$, both of order n) evolving over the time horizon of interest, and thus extends readily to high-dimensional discretizations of unsteady PDE systems, even when $n \gtrsim 10^6$ is necessary to resolve adequately the dynamics of interest. In essence, for any smooth, differentiable system that one can afford to simulate computationally, one can also afford to simulate the adjoint field necessary to determine the gradient of a representative cost function in the space of the optimization variables, thereby enabling an (iterative) gradient-based optimization approach to the control and estimation problems.

The *direct approach* to these problems (in §21.1.2 & 21.1.4), on the other hand, is based on more strict assumptions: in particular, a *linearized governing equation* and a *quadratic cost function*. Subject to these assumptions, this approach proceeds directly (i.e., not iteratively) to the minimum of the cost function by setting the gradient equal to zero and solving the resulting **two-point boundary value problem** (**TPBVP**) for the state and adjoint fields. This TPBVP is made tractable by formulating a *matrix*, of dimension n^2, relating the state and adjoint fields in the optimal solution; the (quadratic) equations defining the matrices at the heart of this formulation for the control and estimation problems are called **Riccati equations**. An efficient technique to solve these matrix equations in the CT, LTI, infinite-horizon case is discussed in §4.5.2. Such matrix-based approaches do *not* extend readily to high-dimension discretizations of infinite-dimensional PDE systems, as they are prohibitively expensive for $n > O(10^3)$; however, as discussed in §21.5, there are a variety of techniques available to approximate either the formulation or the solution of the control and estimation problems in order to finesse oneself out of this dimensionality predicament.

Adjoint-based control optimization[1] (§21.1.1) is known as **model predictive control** (**MPC**). Riccati-based control feedback (§21.1.2) is known as $\mathscr{H}_2$ **state feedback**, **optimal control**, or **linear quadratic regulation** (**LQR**). Adjoint-based state estimation (§21.1.3) is known in the weather forecasting community as the **space-time variational** (**4Dvar**) method, and in the controls community as **moving-horizon estimation** (**MHE**). Riccati-based state estimation (§21.1.4) is commonly referred to as a **Luenberger observer** or, when interpreted from a stochastic point of view assuming Gaussian disturbances (see §22.1 and 22.2), as $\mathscr{H}_2$ **state estimation** or a **Kalman filter**. The combination of LQR and a Kalman filter, as discussed in §21.1.5, is known as **linear quadratic Gaussian** (**LQG**) control.

21.1.1 Control via adjoint-based iterative optimization ("model predictive control")

Assume the system of interest is governed by a CT **state equation** of the form

$$E(\mathbf{x},\mathbf{u})\frac{d\mathbf{x}}{dt} = N(\mathbf{x},\mathbf{u}) \quad \text{on } 0 < t < T, \tag{21.1a}$$

$$\mathbf{x} = \mathbf{x}_0 \quad \text{at } t = 0, \tag{21.1b}$$

where $t = 0$ is the present time and

- $\mathbf{x}(t)$ is the state vector, with $\mathbf{x}_0$ the (known) initial condition (at $t = 0$), and
- $\mathbf{u}(t)$ is the control input (e.g., some force on the system that we may prescribe).

The matrix $E(\mathbf{x},\mathbf{u})$, which may be singular, and the differentiable but possibly nonlinear function $N(\mathbf{x},\mathbf{u})$ are defined as necessary to represent any smooth ODE of interest, including both low-dimensional ODEs and high-dimensional discretizations of PDE systems. Noting that $\mathbf{x} = \mathbf{x}(\mathbf{u})$ by (21.1a)-(21.1b), we also define a **cost function** $J(\mathbf{u})$ which measures any trajectory of this system such that

$$J(\mathbf{u}) = \frac{1}{2}\int_0^T \left[|\mathbf{x}|_Q^2 + |\mathbf{u}|_R^2 \right] dt + \frac{1}{2}|E\mathbf{x}(T)|_{Q_T}^2. \tag{21.1c}$$

The norms are weighted such that, e.g., $|\mathbf{x}|_Q^2 \triangleq \mathbf{x}^H Q \mathbf{x}$, with $Q \geq 0$, $R > 0$, and $Q_T \geq 0$. The cost function (specifically, the choice of Q, R, and Q_T) represents mathematically what we would like the control $\mathbf{u}$ to

[1] The **adjoint** at the heart of this powerful and generalizable optimization approach is also known as a **costate** or **dual** state or, when derived via a slightly different but ultimately equivalent formulation, as a **Lagrange multiplier**.

accomplish in this system[2], and must be tuned with care. *In short, the problem at hand is to minimize* $J(\mathbf{u})$ *with respect to the control distribution* $\mathbf{u}(t)$ *in* (21.1c)*, where* $\mathbf{x}(t)$ *follows from* $\mathbf{u}(t)$ *via* (21.1a)-(21.1b).

We now consider what happens when we simply perturb the inputs to our original system (21.1) a small amount. Small perturbations $\mathbf{u}'(t)$ to the control distribution $\mathbf{u}(t)$ cause small perturbations $\mathbf{x}'(t)$ to the state $\mathbf{x}(t)$. Such perturbations are governed by the **first perturbation equation**, a.k.a. **tangent linear equation**,

$$\mathscr{L}\mathbf{x}' = B\mathbf{u}' \quad \Leftrightarrow \quad E\frac{d\mathbf{x}'}{dt} = A\mathbf{x}' + B\mathbf{u}' \qquad \text{on } \ 0 < t < T, \tag{21.2a}$$

$$\mathbf{x}' = 0 \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{at } \ t = 0, \tag{21.2b}$$

where the operator $\mathscr{L} = (E\frac{d}{dt} - A)$ and matrices $E(\mathbf{x},\mathbf{u})$, $A(\mathbf{x},\mathbf{u})$ and $B(\mathbf{x},\mathbf{u})$ are obtained via **linearization**[3] of (21.1a) about the trajectory $\mathbf{x}(\mathbf{u})$, which itself is given by marching (21.1a)-(21.1b) from $t = 0$ to $t = T$. The concomitant small perturbation to the cost function J in (21.1c) is given by

$$J' = \int_0^T (\mathbf{x}^H Q\mathbf{x}' + \mathbf{u}^H R\mathbf{u}')\, dt + \mathbf{x}^H(T)E^H Q_T E\mathbf{x}'(T). \tag{21.2c}$$

Note that (21.2a) implicitly represents a linear relationship between $\mathbf{x}'$ and $\mathbf{u}'$. Knowing this, the task before us is simply to reëxpress J' in (21.2c) in such a way as to make the resulting linear relationship between J' and $\mathbf{u}'$ explicitly evident, at which point the gradient $\mathscr{D}J/\mathscr{D}\mathbf{u}$ may readily be extracted. To this end, define the weighted inner product $\langle\langle \mathbf{a},\mathbf{b}\rangle\rangle \triangleq \int_0^T \mathbf{a}^H\mathbf{b}\, dt$ and express the following useful **adjoint identity**

$$\langle\langle \mathbf{r}, \mathscr{L}\mathbf{x}'\rangle\rangle = \langle\langle \mathscr{L}^*\mathbf{r}, \mathbf{x}'\rangle\rangle + \mathbf{b}. \tag{21.3}$$

Using integration by parts, it follows that $\mathscr{L}^*\mathbf{r} = -(E^H\frac{d}{dt} + A^H)\mathbf{r}$ and $\mathbf{b} = \left[\mathbf{r}^H E\mathbf{x}'\right]_{t=0}^{t=T}$. We now *define* the relevant **adjoint equation** by

$$\mathscr{L}^*\mathbf{r} = Q\mathbf{x} \quad \Leftrightarrow \quad -E^H\frac{d\mathbf{r}}{dt} = A^H\mathbf{r} + Q\mathbf{x} \qquad \text{on } \ 0 < t < T, \tag{21.4a}$$

$$\mathbf{r} = Q_T E\mathbf{x} \qquad\qquad\qquad\qquad\quad \text{at } \ t = T. \tag{21.4b}$$

The **adjoint field** $\mathbf{r}$ defined by this equation is easy to compute via a *backward* march, from $t = T$ back to $t = 0$. Both A^H and the forcing term $Q\mathbf{x}$ in (21.4a) are functions of $\mathbf{x}(t)$, which itself must be determined from a *forward* march of (21.1), from $t = 0$ to $t = T$; thus, $\mathbf{x}(t)$ must be *saved* during this forward march over the interval $t \in [0,T]$ in order to calculate (21.4) via a backward march from $t = T$ back to $t = 0$. The need for storing $\mathbf{x}(t)$ on $[0,T]$ during this forward march in order to construct the adjoint on the backward march can present a significant storage problem. This problem may be averted with a **checkpointing** algorithm which saves $\mathbf{x}(t)$ only occasionally on the forward march, then recomputes $\mathbf{x}(t)$ as necessary from these "checkpoints" during the backward march for $\mathbf{r}(t)$. Noting (21.2) and (21.4), it follows from (21.3) that

$$\int_0^T \mathbf{r}^H B\mathbf{u}'\, dt = \int_0^T \mathbf{x}^H Q\mathbf{x}'\, dt + \mathbf{x}^H(T)E^H Q_T E\mathbf{x}'(T) \quad \Rightarrow \quad J' = \int_0^T \left[B^H\mathbf{r} + R\mathbf{u}\right]^H \mathbf{u}'\, dt \triangleq \left\langle\!\!\left\langle \frac{\mathscr{D}J}{\mathscr{D}\mathbf{u}}, \mathbf{u}'\right\rangle\!\!\right\rangle.$$

As $\mathbf{u}'$ is arbitrary, the desired **gradient** $\mathbf{g}(t)$ on $t \in [0,T]$ may be identified as

$$\mathbf{g} = \frac{\mathscr{D}J}{\mathscr{D}\mathbf{u}} = B^H\mathbf{r} + R\mathbf{u}, \tag{21.5}$$

[2] Physically, we can say that the control objective is to minimize some measure of the state [as quantified by the first and third terms of J in (21.1c)] without using too much control effort to do it [as quantified by the second term of J]. Nonquadratic forms for J are also possible, as illustrated in Example 21.1. Note also that the terminal cost [the last term of (21.1c)] enables, in effect, the penalization of the dynamics likely to come *after* the finite optimization horizon $t \in [0,T]$; including such a term in the optimization problem significantly improves its long-time behavior when applied in the **receding-horizon model predictive control** framework.

[3] That is, to obtain (21.2a), substitute $\mathbf{x}+\mathbf{x}'$ for $\mathbf{x}$ and $\mathbf{u}+\mathbf{u}'$ for $\mathbf{u}$ in (21.1a), multiply out (using Taylor series where necessary), and retain all terms that are linear in the perturbation quantities; (21.2c) is obtained similarly from (21.1c), also substituting $J+J'$ for J.

and is readily determined from the adjoint field $\mathbf{r}(t)$ defined by (21.4). This gradient may be used to update $\mathbf{u}(t)$ at each iteration k via any of a number of optimization strategies, including the steepest descent and nonquadratic conjugate gradient methods (see §16.2, a review of which is advised before proceeding).

In short, optimization of the nominal control input $\mathbf{u}(t)$ on $t \in [0,T]$ to the system defined in (21.1a)-(21.1b) in order to minimize the cost function defined in (21.1c) proceeds as follows:

0. Define $k = 0$, and guess an initial value for the control input, denoted $\mathbf{u}_0(t)$, on $t \in [0,T]$. Note that this initial guess doesn't need to be particularly good.
1. Using the control $\mathbf{u}_k(t)$ on $t \in [0,T]$, march the state equation (21.1a)-(21.1b) *forward* in time, from $t = 0$ to $t = T$, using an appropriate technique from §10 (e.g., RK4); denote the resulting trajectory of the state $\mathbf{x}_k(t)$ on $t \in [0,T]$, and save this state trajectory (or checkpoints thereof) during this march.
2. Using the control $\mathbf{u}_k(t)$ and the state trajectory $\mathbf{x}_k(t)$ on $t \in [0,T]$, march the adjoint equation (21.4a)-(21.4b) *backward* in time, from $t = T$ back to $t = 0$, again using an appropriate technique from §10; denote the resulting trajectory of the adjoint $\mathbf{r}_k(t)$ on $t \in [0,T]$.
3. Using the control $\mathbf{u}_k(t)$, the state trajectory $\mathbf{x}_k(t)$, and the adjoint trajectory $\mathbf{r}_k(t)$ on $t \in [0,T]$, compute the gradient $\mathbf{g}_k(t) = \mathscr{D}J/\mathscr{D}\mathbf{u}$ according to (21.5). (To reduce storage, steps 2 and 3 may be combined.)
4. Using the gradient $\mathbf{g}_k(t)$ [and, if $k > 0$, the update direction used at the previous iteration, $\mathbf{p}_{k-1}(t)$], update the control $\mathbf{u}_{k+1}(t) = \mathbf{u}_k(t) + \alpha \mathbf{p}_k(t)$ on $t \in [0,T]$ in an appropriate direction $\mathbf{p}_k(t)$ using a gradient-based optimization method from §16.2, such as the nonquadratic conjugate gradient method.
5. Update $k \leftarrow k+1$, and repeat from step 1 until convergence. Denote the converged value of $\mathbf{u}_k(t)$ that results from this optimization process as $\bar{\mathbf{u}}(t)$, the corresponding optimized trajectory as $\bar{\mathbf{x}}(t)$, and the corresponding optimized cost as $\bar{J}(t)$.

In step 4 of the above iteration, for given $\mathbf{u}_k(t)$ and $\mathbf{p}_k(t)$ on $t \in [0,T]$, one needs to determine a parameter of descent α to perform a *line minimization*: that is, to minimize $J(\mathbf{u}_k + \alpha \mathbf{p}_k)$ with respect to α. By solving the perturbation equation (21.2) for $\mathbf{x}'(t)$ in the direction $\mathbf{u}'(t) = \mathbf{p}_k(t)$ linearized about the trajectory $\mathbf{u}(t) = \mathbf{u}_k(t)$ and $\mathbf{x}(t) = \mathbf{x}_k(t)$ on $t \in [0,T]$, and denoting the resulting value of the state perturbation as $\mathbf{x}'_k(t)$, it is straightforward to get an estimate of the most suitable value for α at this iteration in the case that $J(\mathbf{u})$ is nearly quadratic in $\mathbf{u}(t)$. Fixing $\mathbf{u}_k(t)$ and $\mathbf{p}_k(t)$, performing a Taylor series expansion for $J(\mathbf{u}_k + \alpha \mathbf{p}_k)$ about $\alpha = 0$, truncating at second order, and setting the derivative with respect to α equal to zero gives

$$\begin{aligned} J(\mathbf{u}_k + \alpha \mathbf{p}_k) &= \mathbf{u}_k + \alpha \cdot \left.\frac{dJ(\mathbf{u}_k + \alpha \mathbf{p}_k)}{d\alpha}\right|_{\alpha=0} + \frac{\alpha^2}{2} \cdot \left.\frac{d^2 J(\mathbf{u}_k + \alpha \mathbf{p}_k)}{d\alpha^2}\right|_{\alpha=0} + \ldots, \\ \frac{dJ(\mathbf{u}_k + \alpha \mathbf{p}_k)}{d\alpha} = 0 \quad &\Rightarrow \quad \alpha \approx -\left.\frac{dJ(\mathbf{u}_k + \alpha \mathbf{p}_k)}{d\alpha}\right|_{\alpha=0} \Bigg/ \left.\frac{d^2 J(\mathbf{u}_k + \alpha \mathbf{p}_k)}{d\alpha^2}\right|_{\alpha=0}, \end{aligned} \tag{21.6a}$$

where the derivatives shown are simple functions of $\{\mathbf{u}_k, \mathbf{x}_k, \mathbf{p}_k, \mathbf{x}'_k\}$, as readily determined from (21.1c):

$$\left.\frac{dJ(\mathbf{u}_k + \alpha \mathbf{p}_k)}{d\alpha}\right|_{\alpha=0} = \int_0^T (\mathbf{x}_k^H Q \mathbf{x}'_k + \mathbf{u}_k^H R \mathbf{p}_k)\, dt + \mathbf{x}_k^H(T) E^H Q_T E \mathbf{x}'_k(T), \tag{21.6b}$$

$$\left.\frac{d^2 J(\mathbf{u}_k + \alpha \mathbf{p}_k)}{d\alpha^2}\right|_{\alpha=0} \approx \frac{1}{2}\int_0^T [(\mathbf{x}'_k)^H Q \mathbf{x}'_k + (\mathbf{p}_k)^H R \mathbf{p}_k]\, dt + \frac{1}{2}[\mathbf{x}'_k(T)]^H E^H Q_T E \mathbf{x}'_k(T). \tag{21.6c}$$

The value of α given by (21.6) minimizes $J(\mathbf{u}_k + \alpha \mathbf{p}_k)$ if J is essentially quadratic in $\mathbf{u}$. If it is not (e.g., if the relationship $\mathbf{x}(\mathbf{u})$ implied by (21.1a) is significantly nonlinear in the region of interest[4]), this value of α will not minimize $J(\mathbf{u}_k + \alpha \mathbf{p}_k)$ with respect to α, and may in fact lead to an unstable algorithm if used at each iteration of the optimization algorithm. However, (21.6) is still useful to *initialize* the guess for α at each iteration; Brent's method (§15.1.3) may then be used to optimize α based on this initial guess.

[4] In this case, a **second perturbation equation** [cf. the first perturbation equation in (21.2)] needs to be solved and accounted for to compute $d^2J/d\alpha^2$ accurately [cf. (21.6c)]; however, as the importance of this second perturbation increases, the validity of (21.6a) itself reduces; the calculation of the second perturbation is thus, in our experience, not usually worth the computational effort.

The expression in (21.5) allows us to *interpret the adjoint field* $\mathbf{r}$ *simply as the gradient of the cost* J *with respect to the control* $\mathbf{u}$ *in the case that* $B = I$ *and* $R = 0$; if $B \neq I$, the gradient is extracted from $\mathbf{r}$ by shaping it through B^H, and if $R \neq 0$, the gradient is augmented with a discount term $R\mathbf{u}$.

21.1.2 Control via Riccati-based feedback ("optimal control")

Once a desired **nominal trajectory** $\{\bar{\mathbf{u}}(t), \bar{\mathbf{x}}(t)\}$ on $t \in [0,T]$ is identified, as done in §21.1.1, we focus next on *minimizing the perturbations* to this desired nominal trajectory by augmenting the nominal control $\bar{\mathbf{u}}(t)$ on $t \in [0,T]$ with appropriate **control corrections** $\mathbf{u}'(t)$ on $t \in [0,T]$. The analysis leading to the computation of $\mathbf{u}'(t)$ is a very slight modification of the analysis in the previous section. [Note that most problems can be framed with $E = I$, which simplifies the following equations substantially.] We consider the perturbation equation (21.2a)-(21.2b) linearized about the desired nominal trajectory $\{\bar{\mathbf{u}}(t), \bar{\mathbf{x}}(t)\}$. We also consider a quadratic **perturbation cost** $J(\mathbf{u}')$, measuring the perturbations alone, such that [cf. (21.1c)]

$$J(\mathbf{u}') = \frac{1}{2}\int_0^T \left[|\mathbf{x}'|_Q^2 + |\mathbf{u}'|_R^2 \right] dt + \frac{1}{2}|E\mathbf{x}'(T)|_{Q_T}^2. \tag{21.7}$$

The derivation of the expression for the gradient of this perturbation cost follows as before, from a **perturbation adjoint** field driven by $\mathbf{x}'$ instead of $\mathbf{x}$ [cf. (21.4)]

$$\mathscr{L}^*\mathbf{r}' = Q\mathbf{x}' \quad \Leftrightarrow \quad -E^H\frac{d\mathbf{r}'}{dt} = A^H\mathbf{r}' + Q\mathbf{x}' \qquad \text{on } 0 < t < T, \tag{21.8a}$$

$$\mathbf{r}' = Q_T E\mathbf{x}' \qquad \text{at } t = T, \tag{21.8b}$$

from which it is seen that the control corrections $\mathbf{u}'(t)$ which minimizes the perturbation cost $J(\mathbf{u}')$ in (21.7) is given by [cf. (21.5)]

$$\frac{\mathscr{D}J}{\mathscr{D}\mathbf{u}'} = B^H\mathbf{r}' + R\mathbf{u}' = 0 \quad \Rightarrow \quad \mathbf{u}' = -R^{-1}B^H\mathbf{r}'. \tag{21.9}$$

Combining the perturbation equation (21.2) and perturbation adjoint equation (21.8) into a single matrix form, while applying the optimal value of the control corrections $\mathbf{u}'$ from (21.9), gives

$$\begin{bmatrix} E & 0 \\ 0 & E^H \end{bmatrix} \frac{d}{dt}\begin{bmatrix} \mathbf{x}' \\ \mathbf{r}' \end{bmatrix} = \begin{bmatrix} A & -BR^{-1}B^H \\ -Q & -A^H \end{bmatrix}\begin{bmatrix} \mathbf{x}' \\ \mathbf{r}' \end{bmatrix} \quad \text{where} \quad \begin{cases} \mathbf{x}' = 0 & \text{at } t = 0, \\ \mathbf{r}' = Q_T E\mathbf{x}' & \text{at } t = T. \end{cases} \tag{21.10}$$

This ODE, with both initial and terminal conditions, is called a **two-point boundary value problem** (**TPBVP**). Its general solution may be found via the **sweep method** (Bryson & Ho 1969): assuming there exists a relation between the perturbation $\mathbf{x}'(t)$ and the perturbation adjoint $\mathbf{r}'(t)$ via a matrix $X = X(t)$ such that

$$\mathbf{r}' = XE\mathbf{x}', \tag{21.11}$$

inserting this assumed form of the solution (a.k.a. **solution ansatz**) into the combined matrix form (21.10) to eliminate $\mathbf{r}'$, combining rows to eliminate $(E\,d\mathbf{x}'/dt)$, factoring out $\mathbf{x}'$ to the right, and noting that this equation holds for all $\mathbf{x}'$, it follows that X obeys the **differential Riccati equation** (DRE)

$$-E^H\frac{dX}{dt}E = A^HXE + E^HXA - E^HXBR^{-1}B^HXE + Q \quad \text{where} \quad X(T) = Q_T, \tag{21.12a}$$

where the condition at $X(T)$ follows from (21.10) and (21.11). Solutions $X = X(t)$ of this matrix equation are Hermetian ($X^H = X$), and may easily be determined via marching procedures similar to those used to march ODEs (see §10). By the characterization of the optimal point, we may now write the control corrections $\mathbf{u}'$ as

$$\mathbf{u}' = K\mathbf{x}' \quad \text{where} \quad K = -R^{-1}B^HX. \tag{21.12b}$$

To recap, this value of K minimizes the perturbation cost

$$J(\mathbf{u}') = \frac{1}{2}\int_0^T \left[|\mathbf{x}'|_Q^2 + |\mathbf{u}'|_R^2 \right] dt + \frac{1}{2}|E\mathbf{x}'(T)|_{Q_T}^2 \quad \text{where} \quad \frac{d\mathbf{x}'}{dt} = A\mathbf{x}' + B\mathbf{u}'. \tag{21.12c}$$

The resulting matrix $K(t)$ is referred to as the optimal control feedback gain matrix, and is a function of the solution $X(t)$ to (21.12a). This equation may be solved for linear time-varying (LTV) or linear time-invariant (LTI) systems based solely on knowledge of A and B in the system model and Q, R, and Q_T in the cost function; that is, the gain matrix K may be computed **offline**. Note in particular that, for LTI systems in limit that $T \to \infty$ (that is, for the the **infinite-horizon** control problem), the matrix X in (21.12a) may be marched to steady state; this steady state solution for X satisfies the **CT algebraic Riccati equation** (**CARE**)

$$0 = A^H XE + E^H XA - E^H XBR^{-1}B^H XE + Q. \tag{21.13}$$

Efficient algorithms to solve this quadratic matrix equation directly are discussed in §4.5.2.

21.1.2.1 Interpretation of the matrix X in the optimal control setting

Recall from (20.138) that the solution of the appropriately-defined Lyapunov equation, $P(T)$, may be interpreted such that $\overline{\mathbf{x}}^H [P(T)]^{-1} \overline{\mathbf{x}}$ quantifies the minimum "input energy" necessary to steer a system from $\mathbf{x}(0) = 0$ to $\mathbf{x}(T) = \overline{\mathbf{x}}$. We now seek a similarly intuitive interpretation of the matrix $X(t)$ used in the analysis of the optimal control setting above.

Recall first the correspondence between the solution of the DLE (20.91a), marched from $t = 0$ to $t = T$, and the integral form (20.92). Renaming the variables in these two equations appropriately [replacing A with $(A+BK)^H$ and BB^H with $(Q+K^HRK)$ for a given value of K], a similar correspondence may be established between the following DLE and integral forms:

$$\frac{dX}{dt} = (A+BK)^H X + X(A+BK) + (Q+K^HRK) \quad \text{with} \quad X(0) = 0, \tag{21.14a}$$

$$\Leftrightarrow \qquad X(T) = \int_0^T e^{(A+BK)^H t}\,(Q+K^HRK)\,e^{(A+BK)t}\,dt; \tag{21.14b}$$

note that, in the limit that $T \to \infty$, X satisfies the CALE

$$0 = (A+BK)^H X + X(A+BK) + (Q+K^HRK). \tag{21.14c}$$

Now take $E = I$ and consider the evolution equation, initial state, cost function, and feedback relation

$$\frac{d\mathbf{x}}{dt} = A\mathbf{x} + B\mathbf{u}, \quad \mathbf{x}(0) = \mathbf{x}_0, \quad J = \int_0^T [\mathbf{x}^H Q\mathbf{x} + \mathbf{u}^H R\mathbf{u}]\,dt, \quad \mathbf{u} = K\mathbf{x}. \tag{21.15}$$

For a given value of K [see (21.12b)] and $\mathbf{x}_0$, we have $\mathbf{x}(t) = e^{(A+BK)t}\mathbf{x}_0$. Thus, by (21.14b), we have

$$J = \int_0^T \mathbf{x}^H\,(Q+K^HRK)\,\mathbf{x}\,dt = \int_0^T \mathbf{x}_0^H e^{(A+BK)^H t}\,(Q+K^HRK)\,e^{(A+BK)t}\mathbf{x}_0\,dt = \mathbf{x}_0^H X(T)\,\mathbf{x}_0, \tag{21.16}$$

where, noting (21.14), X satisfies the DLE (21.14a) or, in the $T \to \infty$ limit, the CALE (21.14c). Now, since

$$(A+BK)^H X + X(A+BK) + (Q+K^HRK) = A^H X + XA - XBR^{-1}B^H X + Q + (XBR^{-1}+K^H)R(R^{-1}B^H X + K),$$

taking $K = -R^{-1}B^H X$ in these expressions [see (21.12b)] sets the last term on the RHS above to zero, thus equating the DLE in (21.14a) to the DRE in (21.12a), and the CALE in (21.14b) to the CARE in (21.13).

The relation $J = \mathbf{x}_0^H X(T)\mathbf{x}_0$ derived above [see (21.16)], where $X(T)$ is the solution to the DRE in (21.12a) or, in the $T \to \infty$ limit, the CARE in (21.13), allows us to *interpret the expression* $\mathbf{x}_0^H X(T)\mathbf{x}_0$ *simply as the* **cost to go** *in the case that the optimal controller* $K = -R^{-1}B^H X$ *is to be applied to the system.*

21.1.3 Estimation via adjoint-based iterative optimization ("4Dvar" or "MHE")

The derivation presented here is analogous to that presented in §21.1.1. We first write the **state equation** modeling the system of interest in ODE form:

$$E(\mathbf{x},\mathbf{v},\mathbf{w})\frac{d\mathbf{x}}{dt} = N(\mathbf{x},\mathbf{v},\mathbf{w}) \quad \text{on } -T<t<0, \tag{21.17a}$$

$$\mathbf{x} = \mathbf{u} \quad \text{at } t=-T, \tag{21.17b}$$

where $t=0$ is the present time and $\mathbf{x}(t)$ is the state vector, and the quantities to be optimized are:

- $\mathbf{u}$, representing the unknown initial condition of the model (at $t=-T$),
- $\mathbf{v}$, representing the unknown constant parameters of the model, and
- $\mathbf{w}(t)$, representing the unknown external inputs which we would like to determine.

We next write a **cost function** which measures the misfit of the available measurements $\mathbf{y}(t)$ with the corresponding quantity in the computational model, $C\mathbf{x}(t)$, and additionally penalizes the deviation of the initial condition $\mathbf{u}$ from any *a priori* estimate[5] of the initial condition $\overline{\mathbf{u}}$, the deviation of the parameters $\mathbf{v}$ from any *a priori* estimate of the parameters $\overline{\mathbf{v}}$, and the magnitude of the disturbance terms $\mathbf{w}(t)$:

$$J = \frac{1}{2}\int_{-T}^{0} |C\mathbf{x}-\mathbf{y}|^2_{Q_\mathbf{y}}\,dt + \frac{1}{2}|\mathbf{u}-\overline{\mathbf{u}}|^2_{Q_\mathbf{u}} + \frac{1}{2}|\mathbf{v}-\overline{\mathbf{v}}|^2_{Q_\mathbf{v}} + \frac{1}{2}\int_{-T}^{0} |\mathbf{w}|^2_{Q_\mathbf{w}}\,dt. \tag{21.17c}$$

The norms are weighted with positive semi-definite matrices such that, e.g., $|\mathbf{y}|^2_{Q_\mathbf{y}} \triangleq \mathbf{y}^H Q_\mathbf{y}\mathbf{y}$ with $Q_\mathbf{y} \geq 0$. *In short, the problem at hand is to minimize J with respect to $\{\mathbf{u},\mathbf{v},\mathbf{w}(t)\}$ subject to* (21.17).

Small perturbations $\{\mathbf{u}',\mathbf{v}',\mathbf{w}'(t)\}$ to $\{\mathbf{u},\mathbf{v},\mathbf{w}(t)\}$ cause small perturbations $\mathbf{x}'$ to the state $\mathbf{x}$. Such perturbations are governed by the **tangent linear equation**

$$\mathscr{L}\mathbf{x}' = B_\mathbf{v}\mathbf{v}' + B_\mathbf{w}\mathbf{w}' \;\Leftrightarrow\; E\frac{d\mathbf{x}'}{dt} = A\mathbf{x}' + B_\mathbf{v}\mathbf{v}' + B_\mathbf{w}\mathbf{w}' \quad \text{on } -T<t<0, \tag{21.18a}$$

$$\mathbf{x}' = \mathbf{u}' \quad \text{at } t=-T, \tag{21.18b}$$

where the operator $\mathscr{L} = (E\frac{d}{dt} - A)$ and the matrices E, A, $B_\mathbf{v}$, and $B_\mathbf{w}$ are obtained via the linearization of (21.17a) about the trajectory $\mathbf{x}(\mathbf{u},\mathbf{v},\mathbf{w})$. The concomitant small perturbation to the cost function J is given by

$$J' = \int_{-T}^{0} (C\mathbf{x}-\mathbf{y})^H Q_\mathbf{y} C\mathbf{x}'\,dt + (\mathbf{u}-\overline{\mathbf{u}})^H Q_\mathbf{u}\mathbf{u}' + (\mathbf{v}-\overline{\mathbf{v}})^H Q_\mathbf{v}\mathbf{v}' + \int_{-T}^{0} \mathbf{w}^H Q_\mathbf{w}\mathbf{w}'\,dt. \tag{21.19}$$

Again, the task before us is to reëxpress J' in such a way as to make the resulting linear relationship between J' and $\{\mathbf{u}',\mathbf{v}',\mathbf{w}'(t)\}$ explicitly evident, at which point the necessary gradients may readily be defined. To this end, we define the inner product $\langle\langle \mathbf{a},\mathbf{b}\rangle\rangle_R \triangleq \int_0^T \mathbf{a}^H R\mathbf{b}\,dt$ for some $R>0$ and express the **adjoint identity**

$$\langle\langle \mathbf{r}, \mathscr{L}\mathbf{x}'\rangle\rangle_R = \langle\langle \mathscr{L}^*\mathbf{r}, \mathbf{x}'\rangle\rangle_R + \mathbf{b}. \tag{21.20}$$

Using integration by parts, it follows that $\mathscr{L}^*\mathbf{r} = -(R^{-1}E^H R\frac{d}{dt} + R^{-1}A^H R)\,\mathbf{r}$ and $\mathbf{b} = \left[\mathbf{r}^H RE\mathbf{x}'\right]_{t=-T}^{t=0}$. Based on this adjoint operator, we now define an **adjoint equation** of the form

$$\mathscr{L}^*\mathbf{r} = R^{-1}C^H Q_\mathbf{y}(C\mathbf{x}-\mathbf{y}) \;\Leftrightarrow\; -E^H R\frac{d\mathbf{r}}{dt} = A^H R\mathbf{r} + C^H Q_\mathbf{y}(C\mathbf{x}-\mathbf{y}) \quad \text{on } -T<t<0, \tag{21.21a}$$

$$\mathbf{r} = 0 \quad \text{at } t=0. \tag{21.21b}$$

[5] In the 4Dvar setting, such an estimate $\overline{\mathbf{u}}$ for $\mathbf{x}(-T)$ is obtained from the previously-computed forecast, and the corresponding term in the cost function is called the "background" term. The effect of this term on the time evolution of the forecast is significant and sometimes detrimental, as it constrains the update to $\mathbf{u}$ to be small when, in some circumstances, a large update might be warranted.

Again, the difficulty involved with numerically solving the ODE given by (21.21) via a backward march from $t = 0$ to $t = -T$ is essentially the same as the difficulty involved with solving the original ODE (21.17).

Finally, combining (21.18) and (21.21) into the identity (21.20) and substituting into (21.19) gives

$$\begin{aligned} J' &= \left[E^H R\mathbf{r}(-T) + R(\mathbf{u}-\overline{\mathbf{u}})\right]^H \mathbf{u}' + \left[\int_{-T}^{0} B_{\mathbf{v}}^H R\mathbf{r}\,dt + Q_{\mathbf{v}}(\mathbf{v}-\overline{\mathbf{v}})\right]^H \mathbf{v}' + \int_{-T}^{0} \left[B_{\mathbf{w}}^H R\mathbf{r} + Q_{\mathbf{w}}\mathbf{w}\right]^H \mathbf{w}'\,dt \\ &\triangleq \left\langle \frac{\mathscr{D}J}{\mathscr{D}\mathbf{u}}, \mathbf{u}' \right\rangle_{S_{\mathbf{u}}} + \left\langle \frac{\mathscr{D}J}{\mathscr{D}\mathbf{v}}, \mathbf{v}' \right\rangle_{S_{\mathbf{v}}} + \left\langle\!\!\left\langle \frac{\mathscr{D}J}{\mathscr{D}\mathbf{w}}, \mathbf{w}' \right\rangle\!\!\right\rangle_{S_{\mathbf{w}}}, \end{aligned}$$

for some $S_{\mathbf{u}} > 0$, $S_{\mathbf{v}} > 0$, and $S_{\mathbf{w}} > 0$ where $\langle \mathbf{a}, \mathbf{b} \rangle_S \triangleq \mathbf{a}^H S\mathbf{b}$ and $\langle\langle \mathbf{a}, \mathbf{b} \rangle\rangle_S \triangleq \int_{-T}^{0} \mathbf{a}^H S\mathbf{b}\,dt$, and thus

$$\begin{aligned} &\frac{\mathscr{D}J}{\mathscr{D}\mathbf{u}} = S_{\mathbf{u}}^{-1}\left[E^H R\mathbf{r}(-T) + Q_{\mathbf{u}}(\mathbf{u}-\overline{\mathbf{u}})\right], \quad \frac{\mathscr{D}J}{\mathscr{D}\mathbf{v}} = S_{\mathbf{v}}^{-1}\left[\int_{-T}^{0} B_{\mathbf{v}}^H R\mathbf{r}\,dt + Q_{\mathbf{v}}(\mathbf{v}-\overline{\mathbf{v}})\right], \quad \text{and} \\ &\frac{\mathscr{D}J}{\mathscr{D}\mathbf{w}(t)} = S_{\mathbf{w}}^{-1}\left[B_{\mathbf{w}}^H R\mathbf{r}(t) + Q_{\mathbf{w}}\mathbf{w}(t)\right], \qquad \text{for } t \in [-T, 0]. \end{aligned} \tag{21.22}$$

We have thus identified the **gradient** of the cost function with respect to the optimization variables $\{\mathbf{u}, \mathbf{v}, \mathbf{w}(t)\}$ as a function of the adjoint field $\mathbf{r}$ defined in (21.21) which, for any trajectory $\mathbf{x}(\mathbf{u}, \mathbf{v}, \mathbf{w})$ of our original system (21.17), may easily be computed.

The parallels between the adjoint formulation for the estimation problem discussed above and the control problem discussed in §21.1.1 should be clear. There is substantial flexibility in the framing of such optimization problems; the derivation given above is given in a slightly more general form than the derivation shown in §21.1.1 in order to illustrate this flexibility. Once the equation governing the system is specified, this flexibility comes in exactly three forms:

- **targeting the cost function** via selection of the Q_i matrices,
- **regularizing the adjoint operator** via selection of the R matrix, and
- **preconditioning the gradient** via selection of the S_i matrices.

In the ODE setting, the simplest approach is to select the identity matrix for some if not all of these weighting matrices. However, especially when considering a high-dimensional ODE approximation of a PDE problem, this is not always the best choice. By incorporating finite-dimensional discretizations of Sobolev inner products (with derivatives and/or antiderivatives in space and/or time) in place of L^2 inner products, different spatial and/or temporal scales of the problem may be emphasized or de-emphasized in the statement of the cost function, in the dynamics of the corresponding adjoint field, and in the extraction of the gradient. Such alternative inner products (in the infinite-dimensional setting) or weighting matrices (in the finite-dimensional setting) can have a substantially beneficial effect in the resulting optimization process when applied to multiscale problems. In fact, the flexibility of the inner products implied by the Q_i, R, and S_i operators parameterizes exactly *all* of the available options to target/regularize/precondition the *entire* optimization framework (Protas, Bewley, & Hagen 2004). Note in particular that changing the R and S_i matrices change the gradients calculated during the optimization process, but does not change the minimum point(s) of the cost function considered; that is, different gradients ultimately lead to the same minimizer(s), but emphasize different scales of the problem at different iterations of the optimization. Note also that, in high-dimensional problems, we never actually achieve "complete convergence", so these differences are significant. For clarity of presentation, other sections of this chapter take $R = S_i = I$ everywhere, just to simplify in the form of the equations presented, recognizing that other choices can (and, in some cases, should) be preferred.

Another technique that has been introduced to accelerate the adjoint-based estimation technique described above is **multiple shooting**. With this technique, the horizon of interest is split into two or more subintervals. The initial conditions (and, in some implementations, the time-varying model error term) for each subinterval are first initialized and optimized independently, then these several independent solutions are gradually adjusted so that the trajectories coincide at the matching points between the subintervals.

21.1.4 Estimation via Riccati-based feedback ("Kalman filtering")

We now simplify and convert a Riccati-based estimation problem into an equivalent control problem of the form already solved (in §21.1.2). Consider the following linear equations for the state $\mathbf{x}$, the state estimate $\hat{\mathbf{x}}$, and the state estimation error $\tilde{\mathbf{x}} = \mathbf{x} - \hat{\mathbf{x}}$:

$$d\mathbf{x}/dt = A\mathbf{x} + B\mathbf{u}, \qquad \mathbf{y} = C\mathbf{x}, \tag{21.23}$$

$$d\hat{\mathbf{x}}/dt = A\hat{\mathbf{x}} + B\mathbf{u} - L(\mathbf{y} - \hat{\mathbf{y}}), \qquad \hat{\mathbf{y}} = C\hat{\mathbf{x}}, \tag{21.24}$$

$$d\tilde{\mathbf{x}}/dt = A\tilde{\mathbf{x}} + L\tilde{\mathbf{y}}, \qquad \tilde{\mathbf{y}} = C\tilde{\mathbf{x}}. \tag{21.25}$$

The **output injection** term $L(\mathbf{y} - \hat{\mathbf{y}})$ applied to the equation for the state estimate $\hat{\mathbf{x}}$ is to be designed to "nudge" this equation appropriately based on the available measurements $\mathbf{y}$ of the actual system. If this term is doing its job correctly, $\hat{\mathbf{x}}$ is driven towards $\mathbf{x}$ (that is, $\tilde{\mathbf{x}}$ is driven towards zero) even if the initial condition on $\mathbf{x}$ is unknown and (21.23) is only an approximate model of reality. For convenience and without loss of generality, we now return our focus to the time interval $[0, T]$ rather than the interval $[-T, 0]$.

We thus set out to minimize some measure of the state estimation error $\tilde{\mathbf{x}}$ by appropriate selection of L. To this end, taking $\tilde{\mathbf{x}}^H$ times (21.25), we obtain

$$\tilde{\mathbf{x}}^H \left[\frac{d\tilde{\mathbf{x}}}{dt} = A\tilde{\mathbf{x}} + LC\tilde{\mathbf{x}}\right] = \left[\frac{d\tilde{\mathbf{x}}}{dt} = A^H\tilde{\mathbf{x}} + C^H L^H \tilde{\mathbf{x}}\right]^H \tilde{\mathbf{x}} = \frac{1}{2}\frac{d\tilde{\mathbf{x}}^H\tilde{\mathbf{x}}}{dt}. \tag{21.26}$$

Motivated by the second relation in brackets above, consider a new system

$$d\mathbf{z}/dt = A^H\mathbf{z} + C^H\tilde{\mathbf{u}} \quad \text{where} \quad \tilde{\mathbf{u}} = L^H\mathbf{z} \quad \text{and} \quad \mathbf{z}(0) = \tilde{\mathbf{x}}(0). \tag{21.27}$$

Though the *dynamics* of $\tilde{\mathbf{x}}(t)$ and $\mathbf{z}(t)$ are different, the evolution of their *energy* is the same, by (21.26). That is, $\tilde{\mathbf{x}}^H\tilde{\mathbf{x}} = \mathbf{z}^H\mathbf{z}$ for all t even though, in general, $\mathbf{z}(t) \neq \tilde{\mathbf{x}}(t)$. We will thus, for convenience, design L to minimize a cost function related to this auxiliary variable $\mathbf{z}$, defined here such that, taking $Q_1 = I$,

$$\tilde{J} = \frac{1}{2}\int_0^T [\mathbf{z}^H Q_1 \mathbf{z} + \tilde{\mathbf{u}}^H Q_2 \tilde{\mathbf{u}}]\, dt + \frac{1}{2}\mathbf{z}^H(0) P_0 \mathbf{z}(0), \tag{21.28a}$$

where, renaming $\tilde{A} = A^H$, $\tilde{B} = C^H$, and $\tilde{K} = L^H$, (21.27) may be written as

$$d\mathbf{z}/dt = \tilde{A}\mathbf{z} + \tilde{B}\tilde{\mathbf{u}} \quad \text{where} \quad \tilde{\mathbf{u}} = \tilde{K}\mathbf{z}. \tag{21.28b}$$

Finding the feedback gain matrix $\tilde{K}$ in (21.28b) that minimizes the cost function $\tilde{J}$ in (21.28a) is *exactly* the same problem that is solved in (21.12), just with different variables. Thus, the optimal gain matrix L which minimizes a linear combination of the energy of the state estimation error, $\tilde{\mathbf{x}}^H\tilde{\mathbf{x}}$, and some measure of the estimator feedback gain L is again determined from the solution P of a Riccati equation which, making the appropriate substitutions into the solution presented in (21.12), is given by

$$\frac{dP}{dt} = AP + PA^H - PC^H Q_2^{-1} CP + Q_1, \quad P(0) = P_0, \quad L = -PC^H Q_2^{-1}. \tag{21.29}$$

The compact derivation presented above gets quickly to the Riccati equation for an optimal estimator, but as the result of a somewhat contrived optimization problem. A more intuitive formulation is to replace the state equation (21.23) with

$$d\mathbf{x}/dt = A\mathbf{x} + B\mathbf{u} + \mathbf{w}_1, \qquad \mathbf{y} = C\mathbf{x} + \mathbf{w}_2, \tag{21.30}$$

where $\mathbf{w}_1$ (the "state disturbance") and $\mathbf{w}_2$ (the "measurement noise") are assumed to be uncorrelated, zero mean, white Gaussian processes with modeled covariance $\mathscr{E}\{\mathbf{w}_1\mathbf{w}_1^H\} = Q_1$ and $\mathscr{E}\{\mathbf{w}_2\mathbf{w}_2^H\} = Q_2$ respectively. As shown in §22.1, going through the necessary steps to minimize the expected energy of the estimation error, $\mathscr{E}\{\tilde{\mathbf{x}}^H\tilde{\mathbf{x}}\} = \text{trace}(P)$ where $P = \mathscr{E}\{\tilde{\mathbf{x}}\tilde{\mathbf{x}}^H\}$, we again arrive at an estimator of the form given in (21.24) with the feedback gain matrix L as given by (21.29), but now with a much clearer physical understanding of P_0, Q_1, and Q_2.

21.1.5 The separation principle: combining optimal control and Kalman filtering

In §21.1.2, a convenient feedback relationship was derived for determining optimal control inputs based on full state information in the **LQ** setting (that is, for a **linear** state equation and a **quadratic** cost function). In §21.1.4, a convenient feedback relationship was derived for determining an optimal estimate of the full state based on the available system measurements, again in the LQ setting; in §22.1, this feedback relationship is given a logical theoretical foundation assuming **Gaussian** disturbances. It might seem like a good idea, then, to combine the results of §21.1.2 and §21.1.4: that is, in the practical setting in which control inputs must be determined based on available system measurements, to develop an estimate of the state $\hat{\mathbf{x}}$ based on the results of §21.1.4, then to apply control $\mathbf{u} = K\hat{\mathbf{x}}$ based on this state estimate and the results of §21.1.2. This is in fact a generally good idea, and the reason why is called the **separation principle**. Collecting the equations presented previously, taking $E = I$, and adding a reference control input $\mathbf{r}$, we have

Plant :	$d\mathbf{x}/dt = A\mathbf{x} + B\mathbf{u} + \mathbf{w}_1,$	$\mathbf{y} = C\mathbf{x} + \mathbf{w}_2,$
Estimator :	$d\hat{\mathbf{x}}/dt = A\hat{\mathbf{x}} + B\mathbf{u} - L(\mathbf{y} - \hat{\mathbf{y}}),$	$\hat{\mathbf{y}} = C\hat{\mathbf{x}},$
Controller :	$\mathbf{u} = K\hat{\mathbf{x}} + \mathbf{r},$	

where K is determined as in (21.12) and L is determined as in (21.29). This very practical state-space feedback control framework is commonly called **linear quadratic gaussian** (**LQG**) control. In block matrix form (noting that $\tilde{\mathbf{x}} = \mathbf{x} - \hat{\mathbf{x}}$), this composite system may be written

$$\frac{d}{dt}\begin{bmatrix}\mathbf{x}\\ \tilde{\mathbf{x}}\end{bmatrix} = \begin{bmatrix}A+BK & -BK\\ 0 & A+LC\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \tilde{\mathbf{x}}\end{bmatrix} + \begin{bmatrix}B\\ 0\end{bmatrix}\mathbf{r} + \begin{bmatrix}I & 0\\ I & L\end{bmatrix}\begin{bmatrix}\mathbf{w}_1\\ \mathbf{w}_2\end{bmatrix} \tag{21.31a}$$

$$\mathbf{y} = \begin{bmatrix}C & 0\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \tilde{\mathbf{x}}\end{bmatrix} + \begin{bmatrix}0 & I\end{bmatrix}\begin{bmatrix}\mathbf{w}_1\\ \mathbf{w}_2\end{bmatrix}. \tag{21.31b}$$

Since this system matrix is block triangular, its eigenvalues are given by the union of the eigenvalues of $A+BK$ and those of $A+LC$; thus, selecting K and L to stabilize the control and estimation problems separately effectively stabilizes the composite system. Further, assuming that $\mathbf{w}_1 = \mathbf{w}_2 = 0$ and the initial condition on all variables are zero, taking the Laplace transform[6] of the composite system (21.31) gives

$$\mathbf{Y}(s) = \begin{bmatrix}C & 0\end{bmatrix}\begin{bmatrix}sI-(A+BK) & BK\\ 0 & sI-(A+LC)\end{bmatrix}^{-1}\begin{bmatrix}B\\ 0\end{bmatrix}\mathbf{R}(s) = C[sI-(A+BK)]^{-1}B\mathbf{R}(s).$$

That is, the transfer function from $\mathbf{r}$ to $\mathbf{y}$ is unaffected by the estimator. Note that, in the SIMO case, this transfer function may be written in the form

$$\frac{Y(s)}{R(s)} = C[sI-(A+BK)]^{-1}B.$$

As a matter of practice, the estimator feedback L is often designed (by adjusting the relative magnitude of Q_1 and Q_2) according to a convenient rule of thumb such that the slowest eigenvalues of $A+LC$ are a factor of 2 to 5 faster than the slowest eigenvalues of $A+BK$.

[6] In effect, simply replacing d/dt by the Laplace variable s and replacing the time-domain signals $\{\mathbf{y},\mathbf{r},\mathbf{x},\tilde{\mathbf{x}}\}$ with their Laplace transforms $\{\mathbf{Y}(s),\mathbf{R}(s),\mathbf{X}(s),\tilde{\mathbf{X}}(s)\}$, where $\mathbf{F}(s) = \int_{0^-}^{\infty}\mathbf{f}(t)e^{-st}dt$; see §17.3 to review.

Example 21.1 Swing up and stabilization of the dual inverted pendulum

To illustrate the power of the

The nonlinear equations of motion of a single pendulum are given in §17.5 [see (17.16)]. Denoting $\{m_1, I_1, \ell_1\}$ and $\{m_2, I_2, \ell_2\}$ the mass, moment of inertia, and distance from the center of mass to the point where it is attached to the cart of pendulums 1 and 2 respectively, these equations may easily be extended to the dual pendulum case depicted in Figure 21.1 as follows:

$$\begin{aligned}
&(m_c+m_1+m_2)\frac{d^2x}{dt^2} - m_1\ell_1\cos\theta_1\frac{d^2\theta_1}{dt^2} - m_2\ell_2\cos\theta_2\frac{d^2\theta_2}{dt^2} + m_1\ell_1\sin\theta_1\left(\frac{d\theta_1}{dt}\right)^2 + m_2\ell_2\sin\theta_2\left(\frac{d\theta_2}{dt}\right)^2 = u,\\
&-m_1\ell_1\cos\theta_1\frac{d^2x}{dt^2} + (I_1+m_1\ell_1^2)\frac{d^2\theta_1}{dt^2} - m_1 g\ell_1\sin\theta_1 = 0,\\
&-m_2\ell_2\cos\theta_2\frac{d^2x}{dt^2} + (I_2+m_2\ell_2^2)\frac{d^2\theta_2}{dt^2} - m_2 g\ell_2\sin\theta_2 = 0.
\end{aligned} \tag{21.32}$$

Linearization of this system is performed by taking $x = \overline{x} + x'$, $\theta_1 = \overline{\theta}_1 + \theta_1'$, $\theta_2 = \overline{\theta}_2 + \theta_2'$, and $u = \overline{u} + u'$ in (21.32), expanding with Taylor series, multiplying out, applying the fact that the nominal condition $\{\overline{x}, \overline{\theta}_1, \overline{\theta}_2, \overline{u}\}$ is itself also a solution of (21.32), and keeping only those terms which are linear in the perturbation (primed) quantities, as terms that are quadratic or higher in the perturbations are negligible if the perturbations are sufficiently small. Taking $\{\overline{x}=0, \overline{\theta}_1=0, \overline{\theta}_2=0, \overline{u}=0\}$, the linearized equations of motion of the dual inverted pendulum are

$$\begin{aligned}
&(m_c+m_1+m_2)\frac{d^2x'}{dt^2} - m_1\ell_1\frac{d^2\theta_1'}{dt^2} - m_2\ell_2\frac{d^2\theta_2'}{dt^2} = u',\\
&-m_1\ell_1\frac{d^2x'}{dt^2} + (I_1+m_1\ell_1^2)\frac{d^2\theta_1'}{dt^2} - m_1 g\ell_1\theta_1' = 0,\\
&-m_2\ell_2\frac{d^2x'}{dt^2} + (I_2+m_2\ell_2^2)\frac{d^2\theta_2'}{dt^2} - m_2 g\ell_2\theta_2' = 0.
\end{aligned} \tag{21.33}$$

On the other hand, considering an unsteady nominal trajectory $\{\overline{x}(t), \overline{\theta}_1(t), \overline{\theta}_2(t), \overline{u}(t)\}$ gives the tangent linear equations for the dual inverted pendulum:

$$\begin{aligned}
&(m_c+m_1+m_2)\frac{d^2x'}{dt^2} - m_1\ell_1\cos(\overline{\theta}_1)\frac{d^2\theta_1'}{dt^2} - m_2\ell_2\cos(\overline{\theta}_2)\frac{d^2\theta_2'}{dt^2}\\
&\quad + m_1\ell_1\left[\frac{d^2\overline{\theta}_1}{dt^2}\sin(\overline{\theta}_1)\,\theta_1' + \left(\frac{d\overline{\theta}_1}{dt}\right)^2(\cos\overline{\theta}_1)\theta_1' + 2\frac{d\overline{\theta}_1}{dt}\sin(\overline{\theta}_1)\frac{d\theta_1'}{dt}\right]\\
&\quad + m_2\ell_2\left[\frac{d^2\overline{\theta}_2}{dt^2}\sin(\overline{\theta}_2)\,\theta_2' + \left(\frac{d\overline{\theta}_2}{dt}\right)^2(\cos\overline{\theta}_2)\theta_2' + 2\frac{d\overline{\theta}_2}{dt}\sin(\overline{\theta}_2)\frac{d\theta_2'}{dt}\right] = u',\\
&-m_1\ell_1\cos(\overline{\theta}_1)\frac{d^2x'}{dt^2} + (I_1+m_1\ell_1^2)\frac{d^2\theta_1'}{dt^2} - m_1\ell_1\left[g(\cos\overline{\theta}_1)\,\theta_1' - \frac{d^2\overline{x}}{dt^2}\sin(\overline{\theta}_1)\,\theta_1'\right] = 0,\\
&-m_2\ell_2\cos(\overline{\theta}_2)\frac{d^2x'}{dt^2} + (I_2+m_2\ell_2^2)\frac{d^2\theta_2'}{dt^2} - m_2\ell_2\left[g(\cos\overline{\theta}_2)\,\theta_2' - \frac{d^2\overline{x}}{dt^2}\sin(\overline{\theta}_2)\,\theta_2'\right] = 0.
\end{aligned} \tag{21.34}$$

Step 1: Optimizing $\overline{\mathbf{u}}(t)$ on $t \in [0,T]$. We first optimize the nominal control input $\bar{\mathbf{u}}(t)$ on $t \in [0,T]$ to swing up both pendulums. The nonlinear equation of motion (21.32) may be written in the first-order form (21.1a)-(21.1b) by defining

$$\mathbf{x} = \begin{pmatrix} x \\ \theta_1 \\ \theta_2 \\ dx/dt \\ d\theta_1/dt \\ d\theta_2/dt \end{pmatrix}, \quad E(\theta_1,\theta_2) = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & m_c+m_1+m_2 & -m_1\ell_1\cos\theta_1 & -m_2\ell_2\cos\theta_2 \\ & & & -m_1\ell_1\cos\theta_1 & I_1+m_1\ell_1^2 & \\ & & & -m_2\ell_2\cos\theta_2 & & I_2+m_2\ell_2^2 \end{pmatrix},$$

$$N(\mathbf{x},\mathbf{f},\mathbf{u}) = \begin{pmatrix} dx/dt \\ d\theta_1/dt \\ d\theta_2/dt \\ -m_1\ell_1\sin\theta_1\,(d\theta_1/dt)^2 - m_2\ell_2\sin\theta_2\,(d\theta_2/dt)^2 + u \\ m_1 g\ell_1\sin\theta_1 \\ m_2 g\ell_2\sin\theta_2 \end{pmatrix}$$

The initial condition on the system is $\mathbf{x}^T(0) = \begin{pmatrix} 0 & \pi & \pi & 0 & 0 & 0 \end{pmatrix}$; the desired (unstable) terminal condition is $\mathbf{x}^T(T) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$. We target the dynamics of interest on the horizon $(0,T)$ by defining the cost function (21.1c), at least initially, such that $Q = I$ and $R = \alpha^2 I$ and $Q_T = \beta^2 I$. We will take $\alpha = O(1)$ and β as relatively large in order to emphasize bringing the system close to the desired state at the end of the swingup (that is, at $t = T$). We can adjust these weighting matrices in the cost function after optimizing the nominal trajectory of the system if there are any features of this initial optimized trajectory that are found to be undesirable. The tangent linear equation (21.34) may be written in the form (21.2) by taking $E = E(\overline{\theta}_1, \overline{\theta}_2)$, using the (symmetric) formula for $E(\theta_1,\theta_2)$ given above, and defining

$$\mathbf{x}' = \begin{pmatrix} x' \\ \theta_1' \\ \theta_2' \\ dx'/dt \\ d\theta_1'/dt \\ d\theta_2'/dt \end{pmatrix}, \quad A = \begin{pmatrix} & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \\ -a_{42} & -a_{43} & & -a_{45} & -a_{46} & \\ a_{52} & & & & & \\ & a_{63} & & & & \end{pmatrix}, \quad B = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix},$$

where

$$a_{42} = m_1\ell_1\left[\frac{d^2\overline{\theta}_1}{dt^2}\sin\overline{\theta}_1 + \left(\frac{d\overline{\theta}_1}{dt}\right)^2\cos\overline{\theta}_1\right], \quad a_{45} = 2m_1\ell_1\frac{d\overline{\theta}_1}{dt}\sin\overline{\theta}_1, \quad a_{52} = m_1\ell_1\left[g\cos\overline{\theta}_1 - \frac{d^2\overline{x}}{dt^2}\sin\overline{\theta}_1\right],$$
$$a_{43} = m_2\ell_2\left[\frac{d^2\overline{\theta}_2}{dt^2}\sin\overline{\theta}_2 + \left(\frac{d\overline{\theta}_2}{dt}\right)^2\cos\overline{\theta}_2\right], \quad a_{46} = 2m_2\ell_2\frac{d\overline{\theta}_2}{dt}\sin\overline{\theta}_2, \quad a_{63} = m_2\ell_2\left[g\cos\overline{\theta}_2 - \frac{d^2\overline{x}}{dt^2}\sin\overline{\theta}_2\right].$$

With the state vector $\mathbf{x}$, the nonlinear equation of motion in first-order form, the matrices $\{Q, Q_\mathbf{u}, Q_T\}$ defining the cost function, and the matrices $\{E, A, B\}$ defining the tangent linear model so identified, the powerful machinery of (21.1.1) may be used to optimize the control input $\bar{\mathbf{u}}(t)$ on $t \in [0,T]$ to swing up both pendulums.

Step 2: Computing $K(t)$ on $t \in [0,T]$.

Step 3: Computing $L(t)$ on $t \in [0,T]$.

Step 4: Computing K for $t \in [T,\infty)$.

Step 5: Computing L for $t \in [T,\infty)$.

Algorithm 21.1: Adjoint-based optimization of the dual pendulum swing-up.

View
Test

```
function [u_k,x_k]=Example_21_1(T,u_k)
s.h=0.01; s.N=T/s.h; s.mc=10; t=[0:s.N]*s.h;                    % STEP 0: initialize simulation
s.m1=0.2; s.L1=1;     s.ell1=s.L1/2; s.I1=s.m1*s.ell1^2/12; % system, & derived parameters
s.m2=0.1; s.L2=0.5;   s.ell2=s.L2/2; s.I2=s.m2*s.ell2^2/12; alpha=0.1;
s.B=[0; 0; 0; 1; 0; 0]; s.Q=diag([0 0 0 0 0 0]); s.R=0; s.QT=diag([5 40 10 .1 60 10]);
if nargin<2, u_k=zeros(s.N+1,1); end, s.x0=[0; pi; pi; 0; 0; 0]; x_k(1:6,1)=s.x0; res=0;
for k=0:100, k
  u=u_k(1); x=s.x0; J=0.25*s.h*(x'*s.Q*x+u'*s.R*u); c=.5; % STEP 1: march/save state
  for n=1:s.N, u=u_k(n);                                    % (from t=0 -> T), compute cost
    f1=RHS(x,u,s); f2=RHS(x+s.h*f1/2,u,s); f3=RHS(x+s.h*f2/2,u,s); f4=RHS(x+s.h*f3,u,s);
    x=x+s.h*(f1/6+(f2+f3)/3+f4/6); x_k(1:6,n+1)=x; u=u_k(n+1);
    x_k(7:9,n)=f1(4:6); if n==s.N, c=.25; end, J=J+c*s.h*(x'*s.Q*x+u'*s.R*u);
  end, f1=RHS(x,u,s); x_k(7:9,s.N+1)=f1(4:6); E=Compute_E(x,s); J=J+0.5*(x'*E'*s.QT*E*x);
  r=s.QT*E*x; g(s.N+1,1)=s.B'*r+s.R*u_k(s.N+1);          % STEPS 2 & 3: march adjoint
  for n=s.N:-1:1, xh=(x_k(:,n+1)+x_k(:,n))/2;            % (from t=T -> 0), compute gradient
    f1=RHSa(r,x_k(:,n+1),s); f2=RHSa(r-s.h*f1/2,xh,s); f3=RHSa(r-s.h*f2/2,xh,s);
    f4=RHSa(r-s.h*f3,x_k(:,n),s); r=r-s.h*(f1/6+(f2+f3)/3+f4/6); g(n,1)=s.B'*r+s.R*u_k(n);
  end, res1=res; res=g'*g;                                % STEPS 4 & 5: update u and repeat
  if (mod(k,4)==0|alpha<1e-4), p_k=-g; else, p_k=-g+p_k*res/res1; end % conjugate gradient
  figure(1); clf; subplot(2,1,1); plot(t,x_k(1,:),'r-',t,x_k(2,:),'b-',t,x_k(3,:),'g-');
                  subplot(2,1,2); plot(t,u_k,'r--');
  [AA,AB,AC,JA,JB,JC]=Bracket(@Compute_J_Ex21_1,0,alpha,J,u_k,p_k,s);  % find triplet
  [alpha,J]=Brent(@Compute_J_Ex21_1,AA,AB,AC,JA,JB,JC,1e-5,u_k,p_k,s)  % refine triplet
  u_k=u_k+alpha*p_k; pause(0.01); if abs(alpha)<1e-12, break, end       % update u_k
end
s.mc=1; for n=1:s.N+1       % Compute u_k corresponding to different s.mc to give same x_k
  E=Compute_E(x_k(1:6,n),s); N=Compute_N(x_k(1:6,n),0,s); u_k(n,1)=s.B'*(E*x_k(4:9,n)-N);
end, end % function Example_21_1
%%%
```

21.2 Summary of the discrete-time (DT) case

An introduction to DT state-space control and estimation strategies, via both iterative adjoint-based optimization and direct Riccati-based feedback, is now presented, following closely the corresponding CT derivations presented in §21.1 at each step, albeit with the discussion abbreviated to minimize repetition. Again, the presentation is divided into four parts: §21.2.1 and 21.2.2 consider the control problem, whereas §21.2.3 and 21.2.4 consider the estimation problem. The iterative approach to these two problems is considered in §21.2.1 and 21.2.3, whereas the direct approach to these two problems is considered in §21.2.2 and 21.2.4. Together, solutions of the control and estimation problems enable the coördination of a limited number of actuators with a limited number of sensors in order to achieve a desired effect (§21.2.5). An efficient technique to calculate the matrix equation at the heart of these problems in the DT, infinite-horizon, linear time invariant (LTI) case is discussed in §4.5.4.

21.2.1 Control via adjoint-based iterative optimization

We now assume the system of interest is governed by a DT **state equation** of the form

$$E\mathbf{x}_{k+1} = N(\mathbf{x}_k, \mathbf{f}_k, \mathbf{u}_k) \quad \text{on } 0 \le k < K, \tag{21.35a}$$

$$\mathbf{x}_0 = \text{specified} \tag{21.35b}$$

where $k = 0$ is the present time step and

- $\mathbf{x}_k$ is the state vector with $\mathbf{x}_0$ the (known) initial conditions,
- $\mathbf{f}_k$ is the (known) applied external force (e.g., gravity), and
- $\mathbf{u}_k$ is the "control" (e.g., some force on the system that we may prescribe).

We also define a **cost function** $\mathscr{J}$ which measures any trajectory of this system such that

$$\mathscr{J} = \frac{1}{2}\sum_{k=0}^{K}\Big[|\mathbf{x}_k|_Q^2 + |\mathbf{u}_k|_R^2\Big]. \tag{21.35c}$$

The norms are weighted such that, e.g., $|\mathbf{x}|_Q^2 \triangleq \mathbf{x}^H Q\mathbf{x}$, with $Q \ge 0$ and $Q_\mathbf{u} > 0$. The cost function (specifically, the selection of Q and $Q_\mathbf{u}$) represents mathematically what we would like the controls $\mathbf{u}_k$ to accomplish in this system. *In short, the problem at hand is to minimize $\mathscr{J}$ with respect to the control distribution $\mathbf{u}_k$ subject to* (21.35).

Small perturbations $\mathbf{u}'$ to the control $\mathbf{u}$ cause small perturbations $\mathbf{x}'$ to the state $\mathbf{x}$. Such perturbations are governed by the **perturbation equation**

$$(\mathscr{L}\mathbf{x}')_{k+1} = G\mathbf{u}'_k \;\Leftrightarrow\; E\mathbf{x}'_{k+1} = F\mathbf{x}'_k + G\mathbf{u}'_k \qquad \text{on } 0 \le k < K, \tag{21.36a}$$

$$\mathbf{x}'_0 = 0, \tag{21.36b}$$

where the operation $(\mathscr{L}\mathbf{x}')_{k+1} = E\mathbf{x}'_{k+1} - F\mathbf{x}'_k$ and matrices F and G are obtained via the **linearization** of (21.35a) about the trajectory $\mathbf{x}(\mathbf{u})$. The concomitant small perturbation to the cost function $\mathscr{J}$ is given by

$$\mathscr{J}' = \sum_{k=0}^{K}(\mathbf{x}_k^H Q\mathbf{x}'_k + \mathbf{u}_k^H Q_\mathbf{u}\mathbf{u}'_k). \tag{21.36c}$$

Note that (21.36a) implicitly represents a linear relationship between $\mathbf{x}'$ and $\mathbf{u}'$. The task before us is to reëxpress $\mathscr{J}'$ in such a way as to make the resulting linear relationship between $\mathscr{J}'$ and $\mathbf{u}'$ explicitly evident, at which point the gradient $\mathscr{D}\mathscr{J}/\mathscr{D}\mathbf{u}$ may readily be defined. To this end, define the inner product $\langle\langle \mathbf{a}, \mathbf{b}\rangle\rangle \triangleq \sum_{k=1}^{K}\mathbf{a}_k^H\mathbf{b}_k$ and express the following **adjoint identity**

$$\langle\langle \mathbf{r}, \mathscr{L}\mathbf{x}'\rangle\rangle = \langle\langle \mathscr{L}^*\mathbf{r}, \mathbf{x}'\rangle\rangle + \mathbf{b}. \tag{21.37}$$

By rearranging the sums, it follows that $(\mathscr{L}^*\mathbf{r})_k = E^H r_k - F^H \mathbf{r}_{k+1}$ and $\mathbf{b} = \left[\mathbf{r}_{k+1}^H F \mathbf{x}_k'\right]_{k=0}^{k=K}$. We now *define* the relevant **adjoint equation** by

$$(\mathscr{L}^*\mathbf{r})_k = Q\mathbf{x}_k \quad \Leftrightarrow \quad E^H \mathbf{r}_k = F^H \mathbf{r}_{k+1} + Q\mathbf{x}_k \qquad \text{on } \ 0 < k \le K, \tag{21.38a}$$

$$\mathbf{r}_{K+1} = 0 \tag{21.38b}$$

The **adjoint field** $\mathbf{r}$ so defined is easy to compute via a *backward* march from $k = K$ back to $k = 0$. Both A^H and the forcing term $Q\mathbf{x}_k$ in (21.38a) are functions of $\mathbf{x}_k$, which itself must be determined from a *forward* march of (21.35) from $k = 0$ to $k = K$; thus $\mathbf{x}_k$ must be saved on this forward march in order to calculate (21.38) via a backward march from $k = K$ back to $k = 0$. Noting (21.36) and (21.38), it follows from (21.37) that

$$\sum_{k=1}^{K} \mathbf{r}_k^H G \mathbf{u}_{k-1}' = \sum_{k=1}^{K} \mathbf{x}_k^H Q \mathbf{x}_k' \quad \Rightarrow \quad \mathscr{J}' = \sum_{k=0}^{K} \left[G^H \mathbf{r}_{k+1} + Q_{\mathbf{u}} \mathbf{u}_k \right]^H \mathbf{u}_k' \triangleq \left\langle\!\!\left\langle \frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{u}}, \mathbf{u}' \right\rangle\!\!\right\rangle_{S_{\mathbf{u}}}.$$

As $\mathbf{u}'$ is arbitrary, the desired **gradient** is thus given by

$$\left(\frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{u}}\right)_k = S_{\mathbf{u}}^{-1} \left[G^H \mathbf{r}_{k+1} + Q_{\mathbf{u}} \mathbf{u}_k \right], \tag{21.39}$$

and is readily determined from the adjoint field $\mathbf{r}$ defined by (21.38). This gradient may be used to update $\mathbf{u}$ at each iteration k via any of a number of standard optimization strategies, including steepest descent, preconditioned nonquadratic conjugate gradient, and limited-memory BFGS (see §**??**).

21.2.2 Control via Riccati-based feedback

By (21.39), the control $\mathbf{u}_k$ which minimizes $\mathscr{J}$ is given by

$$\frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{u}} = 0 \quad \Rightarrow \quad \mathbf{u}_k = -Q_{\mathbf{u}}^{-1} G^H \mathbf{r}_{k+1}.$$

We now consider the problem that arises when we start with a DT governing equation (21.35a)-(21.35b) for the state variable $\mathbf{x}_k$ that is already in the linearized form of a perturbation equation [as in (21.36a)-(21.36b)]. In other words, we perturb the (already linear) system about the control distribution $\mathbf{u} = 0$ and the trajectory $\mathbf{x}(\mathbf{u}) = 0$, and thus the perturbed system is $\mathbf{u} = \mathbf{u}'$ and $\mathbf{x} = \mathbf{x}'$. Combining the perturbation and adjoint equations (21.36) and (21.38) into a single matrix form, assuming for simplicity that F is invertible and $E = I$ and thus (21.38) may be written in the simplified form

$$\mathbf{r}_{k+1} = F^{-H} \mathbf{r}_k - F^{-H} Q \mathbf{x}_k,$$

and applying the optimal value of the control $\mathbf{u}$ as noted above gives:

$$\begin{bmatrix} \mathbf{x}' \\ \mathbf{r} \end{bmatrix}_{k+1} = \begin{bmatrix} F + GR^{-1}G^H F^{-H} Q & -GQ_{\mathbf{u}}^{-1} G^H F^{-H} \\ -F^{-H} Q & F^{-H} \end{bmatrix} \begin{bmatrix} \mathbf{x}' \\ \mathbf{r} \end{bmatrix}_k \quad \text{where} \quad \begin{cases} \mathbf{x}_0' = 0 \\ \mathbf{r}_{K+1} = 0 \end{cases} \tag{21.40}$$

This difference equation, with both initial and terminal conditions, is referred to as a **two-point boundary value problem**. Its general solution may again be found via a **sweep method**: assuming there exists a relation between the perturbation vector $\mathbf{x}_k'$ and the adjoint vector $\mathbf{r}_k$ via a matrix X_k such that

$$\mathbf{r}_k = X_k \mathbf{x}_k', \tag{21.41}$$

inserting this assumed form of the solution into the combined matrix form (21.40) to eliminate $\mathbf{r}_k$ and performing the manipulations discussed in §4.5.4 leads to the form [see (4.57c)]

$$X_k = F^H X_{k+1} F - F^H X_{k+1} G (R + G^H X_{k+1} G)^{-1} G^H X_{k+1} F + Q \quad \text{where} \quad X_{K+1} = 0, \tag{21.42a}$$

where the condition on X_{K+1} follows from (21.40) and (21.41). Solutions X_k of this matrix equation satisfy $X^H = X$, and may easily be determined simply by marching from $k = K$ back to $k = 0$. By the characterization of the optimal point, we may now write the control $\mathbf{u}$ as

$$\mathbf{u}_k = K_k \mathbf{x}_k \quad \text{where} \quad K_k = -Q_{\mathbf{u}}^{-1} G^H F^{-H} (X_k - Q). \tag{21.42b}$$

To recap, this value of K minimizes

$$\mathscr{J} = \frac{1}{2} \sum_{k=0}^{K} \left[\mathbf{x}_k^H Q \mathbf{x}_k + \mathbf{u}_k^H R \mathbf{u}_k \right] \quad \text{where} \quad \mathbf{x}_{k+1} = F\mathbf{x}_k + G\mathbf{u}_k. \tag{21.42c}$$

The matrix K_k is referred to as the optimal control feedback gain matrix, and is a function of the solution X_k to (21.42a). This equation may be solved for linear time-varying (LTV) or linear time-invariant (LTI) systems based solely on knowledge of F and G in the system model and Q and $Q_{\mathbf{u}}$ in the cost function. Alternatively, if we take the limit that $K \to \infty$ (that is, if we consider the **infinite-horizon** control problem) and the system is LTI, the matrix X_k in (21.42a) may be marched to steady state. This steady state solution for X satisfies the **DT algebraic Riccati equation** (**DARE**)

$$X = F^H X F - F^H X G (R + G^H X G)^{-1} G X F + Q \tag{21.43}$$

Efficient algorithms to solve this quadratic matrix equation are discussed in §4.5.4.

21.2.3 Estimation via adjoint-based iterative optimization

The derivation presented here is analogous to that presented in §21.2.1. We first write the **state equation** modeling the system of interest in ODE form:

$$E\mathbf{x}_{k+1} = N(\mathbf{x}_k, \mathbf{f}_k, \mathbf{v}_k, \mathbf{w}_k) \quad \text{on } -K \le k < 0, \tag{21.44a}$$

$$\mathbf{x}_{-K} = \mathbf{u}, \tag{21.44b}$$

where $k = 0$ is the present time and

- $\mathbf{x}_k$ is the state vector,
- $\mathbf{f}_k$ models the known external forcing,

and the quantities to be optimized are:

- $\mathbf{u}$, representing the unknown initial conditions of the model (at $k = -K$),
- $\mathbf{v}$, representing the unknown constant parameters of the model, and
- $\mathbf{w}_k$, representing the unknown external inputs which we would like to determine.

We next write a **cost function** which measures the misfit of the available measurements $\mathbf{y}_k$ with the corresponding quantity in the computational model, $C\mathbf{x}_k$, and additionally penalizes the deviation of the initial condition $\mathbf{u}$ from any *a priori* estimate of the initial conditions $\overline{\mathbf{u}}$, the deviation of the parameters $\mathbf{v}$ from any *a priori* estimate of the parameters $\overline{\mathbf{v}}$, and the magnitude of the disturbance terms $\mathbf{w}_k$:

$$\mathscr{J} = \frac{1}{2} \sum_{k=-K}^{0} \left| C\mathbf{x}_k - \mathbf{y}_k \right|_{Q_{\mathbf{y}}}^2 + \frac{1}{2} \left| \mathbf{u} - \overline{\mathbf{u}} \right|_{Q_{\mathbf{u}}}^2 + \frac{1}{2} \left| \mathbf{v} - \overline{\mathbf{v}} \right|_{Q_{\mathbf{v}}}^2 + \frac{1}{2} \sum_{k=-K}^{0} \left| \mathbf{w}_k \right|_{Q_{\mathbf{w}}}^2. \tag{21.44c}$$

The norms are weighted with positive semi-definite matrices such that, e.g., $|\mathbf{y}|_{Q_{\mathbf{y}}}^2 \triangleq \mathbf{y}^H Q_{\mathbf{y}} \mathbf{y}$ with $Q_{\mathbf{y}} \ge 0$. *In short, the problem at hand is to minimize* $\mathscr{J}$ *with respect to* $\{\mathbf{u}, \mathbf{v}, \mathbf{w}_k\}$ *subject to* (21.44).

Small perturbations $\{\mathbf{u}', \mathbf{v}', \mathbf{w}_k'\}$ to $\{\mathbf{u}, \mathbf{v}, \mathbf{w}_k\}$ cause small perturbations $\mathbf{x}'$ to the state $\mathbf{x}$. Such perturbations are governed by the **perturbation equation**

$$(\mathscr{L}\mathbf{x}')_{k+1} = G_{\mathbf{v}}\mathbf{v}_k' + G_{\mathbf{w}}\mathbf{w}_k' \quad \Leftrightarrow \quad E\mathbf{x}_{k+1}' = F\mathbf{x}' + G_{\mathbf{v}}\mathbf{v}' + G_{\mathbf{w}}\mathbf{w}' \qquad \text{on} \quad -K \le k < 0, \tag{21.45a}$$

$$\mathbf{x}_{-K}' = \mathbf{u}', \tag{21.45b}$$

where the operation $(\mathscr{L}x')_{k+1} = E\mathbf{x}_{k+1}' - F\mathbf{x}_k'$ and the matrices F, $G_{\mathbf{v}}$, and $G_{\mathbf{w}}$ are obtained via the linearization of (21.44a) about the trajectory $\mathbf{x}(\mathbf{u})$. The concomitant small perturbation to the cost function $\mathscr{J}$ is given by

$$\mathscr{J}' = \sum_{k=-K}^{0} (C\mathbf{x}_k - \mathbf{y}_k)^H Q_{\mathbf{y}} C\mathbf{x}_k' + (\mathbf{u} - \overline{\mathbf{u}})^H Q_{\mathbf{u}}\mathbf{u}' + (\mathbf{v} - \overline{\mathbf{v}})^H Q_{\mathbf{v}}\mathbf{v}' + \sum_{k=-K}^{0} \mathbf{w}_k^H Q_{\mathbf{w}}\mathbf{w}_k' \, dt. \tag{21.46}$$

Again, the task before us is to reëxpress $\mathscr{J}'$ in such a way as to make the resulting linear relationship between $\mathscr{J}'$ and $\{\mathbf{u}', \mathbf{v}', \mathbf{w}_k'\}$ explicitly evident, at which point the necessary gradients may readily be defined. To this end, we define the inner product $\langle\langle \mathbf{a}, \mathbf{b} \rangle\rangle \triangleq \sum_{k=-K+1}^{0} \mathbf{a}_k^H \mathbf{b}_k$ and express the **adjoint identity**

$$\langle\langle \mathbf{r}, \mathscr{L}\mathbf{x}' \rangle\rangle = \langle\langle \mathscr{L}^*\mathbf{r}, \mathbf{x}' \rangle\rangle + \mathbf{b}. \tag{21.47}$$

By rearranging the sums, it follows that $(\mathscr{L}^*\mathbf{r})_k = E^H r_k - F^H \mathbf{r}_{k+1}$ and $\mathbf{b} = \left[\mathbf{r}_{k+1}^H F\mathbf{x}_k'\right]_{k=-K}^{k=0}$. Based on this adjoint operator, we now define an **adjoint equation** of the form

$$(\mathscr{L}^*\mathbf{r})_k = C^H Q_{\mathbf{y}}(C\mathbf{x}_k - \mathbf{y}_k) \quad \Leftrightarrow \quad -E^H\mathbf{r}_k = A^H\mathbf{r}_{k+1} + C^H Q_{\mathbf{y}}(C\mathbf{x}_k - \mathbf{y}_k) \qquad \text{on} \quad -K < k < 0, \tag{21.48a}$$

$$\mathbf{r}_0 = 0. \tag{21.48b}$$

The difficulty involved with numerically solving the ODE given by (21.48) via a backward march from $t = 0$ to $t = -T$ is essentially the same as the difficulty involved with solving the original ODE (21.44). Finally, combining (21.45) and (21.48) into the identity (21.47) and substituting into (21.46), it follows that

$$\begin{aligned}
\mathscr{J}' &= \left[E^H\mathbf{r}_{-K} + R(\mathbf{u} - \overline{\mathbf{u}})\right]^H \mathbf{u}' + \left[\sum_{k=-K}^{0} B_{\mathbf{v}}^H\mathbf{r}_k + Q_{\mathbf{v}}(\mathbf{v} - \overline{\mathbf{v}})\right]^H \mathbf{v}' + \sum_{k=-K}^{0} \left[B_{\mathbf{w}}^H\mathbf{r} + Q_{\mathbf{w}}\mathbf{w}\right]^H \mathbf{w}' \, dt \\
&\triangleq \left\langle \frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{u}}, \mathbf{u}' \right\rangle_{S_{\mathbf{u}}} + \left\langle \frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{v}}, \mathbf{v}' \right\rangle_{S_{\mathbf{v}}} + \left\langle\!\!\left\langle \frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{w}}, \mathbf{w}' \right\rangle\!\!\right\rangle_{S_{\mathbf{w}}},
\end{aligned}$$

for some $S_{\mathbf{u}} > 0$, $S_{\mathbf{v}} > 0$, and $S_{\mathbf{w}} > 0$ where $\langle \mathbf{a}, \mathbf{b} \rangle_S \triangleq \mathbf{a}^H S\mathbf{b}$ and $\langle\langle \mathbf{a}, \mathbf{b} \rangle\rangle_S \triangleq \int_{-T}^{0} \mathbf{a}^H S\mathbf{b} \, dt$, and thus

$$\begin{aligned}
&\frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{u}} = S_{\mathbf{u}}^{-1}\left[E^H\mathbf{r}_{-K} + Q_{\mathbf{u}}(\mathbf{u} - \overline{\mathbf{u}})\right], \quad \frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{v}} = S_{\mathbf{v}}^{-1}\left[\sum_{k=-K}^{0} B_{\mathbf{v}}^H\mathbf{r}_k + Q_{\mathbf{v}}(\mathbf{v} - \overline{\mathbf{v}})\right], \quad \text{and} \\
&\frac{\mathscr{D}\mathscr{J}}{\mathscr{D}\mathbf{w}_k} = S_{\mathbf{w}}^{-1}\left[B_{\mathbf{w}}^H\mathbf{r}_k + Q_{\mathbf{w}}\mathbf{w}_k\right], \qquad \text{for } k \in [-K, 0].
\end{aligned} \tag{21.49}$$

We have thus defined the **gradient** of the cost function with respect to the optimization variables $\{\mathbf{u}, \mathbf{v}, \mathbf{w}_k\}$ as a function of the adjoint field $\mathbf{r}_k$ defined in (21.21), which, for any trajectory $\mathbf{x}_k$ of our original system (21.44), may easily be computed.

21.2.4 Estimation via Riccati-based feedback

We now convert the Riccati-based estimation problem into an equivalent control problem of the form already solved (in §21.2.2). Consider the linear equations for the state $\mathbf{x}$, the state estimate $\hat{\mathbf{x}}$, and the state estimation

error $\tilde{\mathbf{x}} = \mathbf{x} - \hat{\mathbf{x}}$:

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + G\mathbf{u}_k, \qquad \mathbf{y}_k = H\mathbf{x}_k, \tag{21.50}$$

$$\hat{\mathbf{x}}_{k+1} = F\hat{\mathbf{x}}_k + G\mathbf{u}_k - L(\mathbf{y}_k - \hat{\mathbf{y}}_k), \qquad \hat{\mathbf{y}}_k = H\hat{\mathbf{x}}_k, \tag{21.51}$$

$$\tilde{\mathbf{x}}_{k+1} = F\tilde{\mathbf{x}}_k + L\tilde{\mathbf{y}}_k, \qquad \tilde{\mathbf{y}}_k = H\tilde{\mathbf{x}}_k. \tag{21.52}$$

The **output injection** term $L(\mathbf{y}_k - \hat{\mathbf{y}}_k)$ applied to the equation for the state estimate $\hat{\mathbf{x}}_k$ is to be designed to "nudge" this equation appropriately based on the available measurements $\mathbf{y}_k$ of the actual system. If this term is doing its job correctly, $\hat{\mathbf{x}}_k$ is driven towards $\mathbf{x}_k$ (that is, $\tilde{\mathbf{x}}_k$ is driven towards zero) even if the initial conditions on $\mathbf{x}_k$ are unknown and (21.50) is only an approximate model of reality.

The section still under construction!!!!! need to convert to discrete time.

For convenience and without loss of generality, we now return our focus to the time interval $[0,T]$ rather than the interval $[-T,0]$.

We thus set out to minimize some measure of the state estimation error $\tilde{\mathbf{x}}$ by appropriate selection of L. To this end, taking $\tilde{\mathbf{x}}^H$ times (21.25), we obtain

$$\tilde{\mathbf{x}}^H \left[\frac{d\tilde{\mathbf{x}}}{dt} = A\tilde{\mathbf{x}} + LC\tilde{\mathbf{x}}\right] = \left[\frac{d\tilde{\mathbf{x}}}{dt} = A^H\tilde{\mathbf{x}} + C^H L^H \tilde{\mathbf{x}}\right]^H \tilde{\mathbf{x}} = \frac{1}{2}\frac{d\tilde{\mathbf{x}}^H\tilde{\mathbf{x}}}{dt}. \tag{21.53}$$

Motivated by the second relation in brackets above, consider a new system

$$d\mathbf{z}/dt = A^H\mathbf{z} + C^H\tilde{\mathbf{u}} \quad \text{where} \quad \tilde{\mathbf{u}} = L^H\mathbf{z} \quad \text{and} \quad \mathbf{z}(0) = \tilde{\mathbf{x}}(0). \tag{21.54}$$

Though the *dynamics* of $\tilde{\mathbf{x}}(t)$ and $\mathbf{z}(t)$ are different, the evolution of their *energy* is the same, by (21.26). That is, $\tilde{\mathbf{x}}^H\tilde{\mathbf{x}} = \mathbf{z}^H\mathbf{z}$ even though, in general, $\mathbf{z}(t) \neq \tilde{\mathbf{x}}(t)$. We will thus, for convenience, design L to minimize a cost function related to this auxiliary variable $\mathbf{z}$, defined here such that, taking $Q_1 = I$,

$$\tilde{J} = \frac{1}{2}\int_0^T [\mathbf{z}^H Q_1 \mathbf{z} + \tilde{\mathbf{u}}^H Q_2 \tilde{\mathbf{u}}]\, dt + \frac{1}{2}\mathbf{z}^H(-T)P_0\mathbf{z}(-T), \tag{21.55a}$$

where, renaming $\tilde{A} = A^H$, $\tilde{B} = C^H$, and $\tilde{K} = L^H$, (21.27) may be written as

$$d\mathbf{z}/dt = \tilde{A}\mathbf{z} + \tilde{B}\tilde{\mathbf{u}} \quad \text{where} \quad \tilde{\mathbf{u}} = \tilde{K}\mathbf{z}. \tag{21.55b}$$

Finding the feedback gain matrix $\tilde{K}$ in (21.28b) that minimizes the cost function $\tilde{J}$ in (21.28a) is *exactly* the same problem that is solved in (21.12), just with different variables. Thus, the optimal gain matrix L which minimizes a linear combination of the energy of the state estimation error, $\tilde{\mathbf{x}}^H\tilde{\mathbf{x}}$, and some measure of the estimator feedback gain L is again determined from the solution P of a Riccati equation which, making the appropriate substitutions into the solution presented in (21.12), is given by

$$\frac{dP}{dt} = AP + PA^H - PC^H Q_2^{-1} CP + Q_1, \quad P(0) = P_0, \quad L = -PC^H Q_2^{-1}. \tag{21.56}$$

The compact derivation presented above gets quickly to the Riccati equation for an optimal estimator, but as the result of a somewhat contrived optimization problem. A more intuitive formulation is to replace the state equation (21.23) with

$$d\mathbf{x}/dt = A\mathbf{x} + B\mathbf{u} + \mathbf{w}_1, \qquad \mathbf{y} = C\mathbf{x} + \mathbf{w}_2, \tag{21.57}$$

where $\mathbf{w}_1$ (the "state disturbance") and $\mathbf{w}_2$ (the "measurement noise") are assumed to be uncorrelated, zero mean, white Gaussian processes with modeled covariance $\mathscr{E}\{\mathbf{w}_1\mathbf{w}_1^H\} = Q_1$ and $\mathscr{E}\{\mathbf{w}_2\mathbf{w}_2^H\} = Q_2$ respectively. As shown in §22.1, going through the necessary steps to minimize the expected energy of the estimation error, $\mathscr{E}\{\tilde{\mathbf{x}}^H\tilde{\mathbf{x}}\} = \text{trace}(P)$ where $P = \mathscr{E}\{\tilde{\mathbf{x}}\tilde{\mathbf{x}}^H\}$, we again arrive at an estimator of the form given in (21.24) with the feedback gain matrix L as given by (21.29).

21.2.5 The separation principle: putting it together

In §21.2.2, a convenient feedback relationship was derived for determining optimal control inputs based on full state information. In §21.2.4, a convenient feedback relationship was derived for determining an optimal estimate of the full state based on the available system measurements. It might seem like a good idea, then, to combine the results of these two sections: that is, in the practical case in which control inputs must be determined based on available system measurements, to develop an estimate of the state $\hat{\mathbf{x}}$ based on the results of §21.2.4, then to apply control $\mathbf{u} = K\hat{\mathbf{x}}$ based on this state estimate and the results of §21.2.2. This is in fact a good idea, and the reason why is called the **separation principle**. Collecting the equations presented previously and adding a reference control input $\mathbf{r}$, we have

Plant :	$\mathbf{x}_{k+1} = F\mathbf{x}_k + G\mathbf{u}_k + \mathbf{w}_{1,k},$	$\mathbf{y}_k = H\mathbf{x}_k + \mathbf{w}_{2,k},$
Estimator :	$\hat{\mathbf{x}}_{k+1} = F\hat{\mathbf{x}}_k + G\mathbf{u}_k - L(\mathbf{y}_k - \hat{\mathbf{y}}_k),$	$\hat{\mathbf{y}}_k = H\hat{\mathbf{x}}_k,$
Controller :	$\mathbf{u}_k = K\hat{\mathbf{x}}_k + \mathbf{r}_k,$	

where K is determined as in (21.42) and L is determined as in (21.56). In block matrix form (noting that $\tilde{\mathbf{x}}_k = \mathbf{x}_k - \hat{\mathbf{x}}_k$), this composite system may be written

$$\begin{bmatrix}\mathbf{x}\\ \tilde{\mathbf{x}}\end{bmatrix}_{k+1} = \begin{bmatrix}F+GK & -GK\\ 0 & F+LH\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \tilde{\mathbf{x}}\end{bmatrix}_k + \begin{bmatrix}G\\ 0\end{bmatrix}\mathbf{r}_k + \begin{bmatrix}I & 0\\ I & L\end{bmatrix}\begin{bmatrix}\mathbf{w}_1\\ \mathbf{w}_2\end{bmatrix}_k \tag{21.58a}$$

$$\mathbf{y}_k = \begin{bmatrix}H & 0\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ \tilde{\mathbf{x}}\end{bmatrix}_k + \begin{bmatrix}0 & I\end{bmatrix}\begin{bmatrix}\mathbf{w}_1\\ \mathbf{w}_2\end{bmatrix}_k. \tag{21.58b}$$

Since this system matrix is block triangular, its eigenvalues are given by the union of the eigenvalues of $F + GK$ and those of $F + LH$; thus, selecting K and L to stabilize the control and estimation problems separately effectively stabilizes the composite system. Further, assuming that $\mathbf{w}_1 = \mathbf{w}_2 = 0$ and the initial condition on all variables are zero, taking the Z transform of the composite system (21.58) gives

$$\mathbf{Y}(z) = \begin{bmatrix}H & 0\end{bmatrix}\begin{bmatrix}zI-(F+GK) & GK\\ 0 & zI-(F+LH)\end{bmatrix}^{-1}\begin{bmatrix}G\\ 0\end{bmatrix}\mathbf{R}(z) = H[zI-(F+GK)]^{-1}G\mathbf{R}(z).$$

That is, the transfer function from $\mathbf{r}_k$ to $\mathbf{y}_k$ is unaffected by the estimator. Note that, in the SISO case, this transfer function may be written in the form

$$\frac{Y(z)}{R(z)} = H[zI-(G+HK)]^{-1}G.$$

21.2.6 Robust control

This section still under construction.

21.3 Adjoint-based analysis of mixed CT/DT formulations

21.3.1 Control of mixed CT/DT formulations

21.3.2 Estimation of mixed CT/DT formulations

For analysis, let the variational window be defined as $t \in [-T, 0]$. Additionally, let there be $K+1$ measurements in this interval, with measurement indices given by the set

$$M = \{k \mid t_k \in [-T,0]\} \;\Rightarrow\; M = \{-K, \cdots, -1, 0\}. \tag{21.59}$$

Without loss of generality, it will be assumed that there are measurements at both edges of the window (i.e. $t_{-K} = -T$ and $t_0 = 0$). Then, the cost function $\mathscr{J}(\mathbf{u})$ that 4DVar attempts to minimize (with respect to $\mathbf{u}$) is defined as follows:

$$\begin{aligned}\mathscr{J}(\mathbf{u}) = &\frac{1}{2}(\mathbf{u} - \overline{\mathbf{x}}_{-K|-K})^H \mathscr{P}^{-1}_{-K|-K}(\mathbf{u} - \overline{\mathbf{x}}_{-K|-K}) + \\ &\frac{1}{2}\sum_{k=-K}^{0}\left(\mathbf{y}_k - H\tilde{\mathbf{x}}_k\right)^H \mathscr{R}^{-1}\left(\mathbf{y}_k - H\tilde{\mathbf{x}}_k\right),\end{aligned} \tag{21.60}$$

where the "optimization variable" $\mathbf{u}$ is the initial condition on the refined state estimate $\tilde{\mathbf{x}}$ on the interval $[-T,0]$; that is,

$$\frac{d\tilde{\mathbf{x}}(t)}{dt} = f(\tilde{\mathbf{x}}(t), 0), \tag{21.61a}$$

$$\tilde{\mathbf{x}}_{-K} = \mathbf{u}. \tag{21.61b}$$

The first term in the cost function (21.60), known as the "background" term, summarizes the fit of $\mathbf{u}$ with the current probability distribution before the optimization (i.e., the effect of all past measurement updates). Like with the KF, $\overline{\mathbf{x}}_{-K|-K}$ is the estimate at time t_{-K} not including any of the new measurements in the window, and the covariance $\mathscr{P}^{-1}_{-K|-K}$ quantifies the second moment of the uncertainty in that estimate. Assuming an a priori Gaussian probability distribution of this uncertainty, the background mean and covariance exactly describe this distribution. The second term in the cost function (21.60) summarizes the misfit between the estimated system trajectory and the observations within the variational window. Thus, the solution $\mathbf{u}$ to this optimization problem is the estimate that best "fits" the observations over the variational window while also accounting for the existing information from observations prior to the variational window.

In practice, a 4DVar iteration is usually initialized with the background mean, $\mathbf{u} = \overline{\mathbf{x}}_{-K|-K}$. Given this initial guess for $\mathbf{u}$, the trajectory $\tilde{\mathbf{x}}(t)$ may be found using the full nonlinear equations for the system (21.61). To find the gradient of the cost function (21.60), consider a small perturbation of the optimization variable, $\mathbf{u} \leftarrow \mathbf{u} + \mathbf{u}'$, and the resulting perturbed trajectory, $\tilde{\mathbf{x}}(t) \leftarrow \tilde{\mathbf{x}}(t) + \tilde{\mathbf{x}}'(t)$, and perturbed cost function, $\mathscr{J}(\mathbf{u}) \leftarrow \mathscr{J}(\mathbf{u}) + \mathscr{J}'(\mathbf{u}')$. The local gradient of (21.60), $\bigtriangledown\mathscr{J}(\mathbf{u})$, is defined here as the sensitivity of the perturbed cost function $\mathscr{J}'(\mathbf{u}')$ to the perturbed optimization variable $\mathbf{u}'$:

$$\mathscr{J}'(\mathbf{u}') = \left[\bigtriangledown\mathscr{J}(\mathbf{u})\right]^H \mathbf{u}'. \tag{21.62}$$

The following derivation illustrates how to write $\mathscr{J}'(\mathbf{u}')$ in this simple form, leveraging the definition of an appropriate adjoint field.

The full derivation of the gradient $\bigtriangledown\mathscr{J}(\mathbf{u})$ is included here due to the unusual setting considered (that is, of a CT system with DT measurements). Perturbing the nonlinear model equations (**??**) and linearizing about

$\tilde{\mathbf{x}}(t)$ gives:

$$\frac{d\tilde{\mathbf{x}}'(t)}{dt} = A\,\tilde{\mathbf{x}}'(t) \quad \text{with} \quad \tilde{\mathbf{x}}'_{-K} = \mathbf{u}' \tag{21.63}$$

$$\Rightarrow \quad \mathscr{L}\,\tilde{\mathbf{x}}' = 0 \quad \text{where} \quad \mathscr{L} = \frac{d}{dt} - A. \tag{21.64}$$

Similarly, the perturbed cost function is:

$$\mathscr{J}'(\mathbf{u}') = (\mathbf{u} - \overline{\mathbf{x}}_{-K|-K})^H\, \mathscr{P}^{-1}_{-K|-K}\, \mathbf{u}' - \sum_{k=-K}^{0} (\mathbf{y}_k - H\,\tilde{\mathbf{x}}_k)^H\, \mathscr{R}^{-1}\, H\,\tilde{\mathbf{x}}'_k. \tag{21.65}$$

The perturbed cost function (21.65) is not quite in the form necessary to extract the gradient, as illustrated in (21.62). However, there is an implicitly defined linear relationship between $\mathbf{u}'$ and $\tilde{\mathbf{x}}'(t)$ on $t \in [-T, 0]$ given by (21.63). To re-express this relationship, a set of K adjoint functions $\mathbf{r}^{(k)}(t)$ are defined over the measurement intervals such that, for all $k \in [1, K]$, the adjoint function $\mathbf{r}^{(k)}(t)$ is defined on the closed interval $t \in \left[t_{-k}, t_{1-k}\right]$. These adjoint functions will be used to identify the gradient. To this end, a suitable duality pairing is defined here as:

$$\langle\, \mathbf{r}^{(k)}, \tilde{\mathbf{x}}' \,\rangle = \int_{t_{-k}}^{t_{1-k}} (\mathbf{r}^{(k)})^H\, \tilde{\mathbf{x}}'\, dt. \tag{21.66}$$

Then, the necessary adjoint identity is given by

$$\langle\, \mathbf{r}^{(k)}, \mathscr{L}\,\tilde{\mathbf{x}}' \,\rangle = \langle\, \mathscr{L}^*\, \mathbf{r}^{(k)}, \tilde{\mathbf{x}}' \,\rangle + b^{(k)}. \tag{21.67a}$$

Using the definition of the operator $\mathscr{L}$ given by (21.64) and the appropriate integration by parts, it is easily shown that

$$\mathscr{L}^*\, \mathbf{r}^{(k)} = -\frac{d\mathbf{r}^{(k)}(t)}{dt} - A^H\, \mathbf{r}^{(k)}(t), \tag{21.67b}$$

$$b^{(k)} = (\mathbf{r}^{(k)}_{1-k})^H\, \tilde{\mathbf{x}}'_{1-k} - (\mathbf{r}^{(k)}_{-k})^H\, \tilde{\mathbf{x}}'_{-k}. \tag{21.67c}$$

Returning to the perturbed cost function, (21.65) can be rewritten as:

$$\begin{aligned} \mathscr{J}'(\mathbf{u}') = {} & (\mathbf{u} - \overline{\mathbf{x}}_{-K|-K})^H\, \mathscr{P}^{-1}_{-K|-K}\, \mathbf{u}' - \mathscr{J}'_1 \\ & - \sum_{k=-K}^{-1} (\mathbf{y}_k - H\,\tilde{\mathbf{x}}_k)^H\, \mathscr{R}^{-1}\, H\,\tilde{\mathbf{x}}'_k, \end{aligned} \tag{21.68a}$$

$$\mathscr{J}'_1 = \left[H^H\, \mathscr{R}^{-1}\, (\mathbf{y}_0 - H\,\tilde{\mathbf{x}}_0)\right]^H\, \tilde{\mathbf{x}}'_0. \tag{21.68b}$$

Looking at the adjoint defined over the last interval, $\mathbf{r}^{(1)}(t)$, the following criteria is enforced:

$$\mathscr{L}^*\, \mathbf{r}^{(1)} = 0 \quad \Rightarrow \quad \langle\, \mathscr{L}^*\, \mathbf{r}^{(1)}, \tilde{\mathbf{x}}' \,\rangle = 0, \tag{21.69a}$$

$$\mathbf{r}^{(1)}_0 = H^H\, \mathscr{R}^{-1}\, (\mathbf{y}_0 - H\,\tilde{\mathbf{x}}_0). \tag{21.69b}$$

Substituting (21.64) and (21.69a) into (21.67a) for $k = 1$ gives:

$$\begin{aligned} & b^{(1)} = 0 \\ & \Rightarrow \quad (\mathbf{r}^{(1)}_0)^H\, \tilde{\mathbf{x}}'_0 - (\mathbf{r}^{(1)}_{-1})^H\, \tilde{\mathbf{x}}'_{-1} \;=\; 0, \\ & \Rightarrow \quad \left[H^H\, \mathscr{R}^{-1}\, (\mathbf{y}_0 - H\,\tilde{\mathbf{x}}_0)\right]^H\, \tilde{\mathbf{x}}'_0 \;=\; (\mathbf{r}^{(1)}_{-1})^H\, \tilde{\mathbf{x}}'_{-1}, \end{aligned} \tag{21.70}$$

which allows us to re-express $\mathscr{J}_1'$ in (21.68b) as

$$\mathscr{J}_1' = (\mathbf{r}_{-1}^{(1)})^H \, \tilde{\mathbf{x}}_{-1}'. \tag{21.71}$$

Note that (21.69a) and (21.69b) give the full evolution equation and starting condition for the adjoint $\mathbf{r}^{(1)}$ defined on the interval $t \in [t_{-1}, t_0]$. Hence, a backward march over this interval will lead to the term $\mathbf{r}_{-1}^{(1)}$ contained in (21.71).

The perturbed cost function (21.68a) can now be rewritten such that

$$\begin{aligned} \mathscr{J}'(\mathbf{u}') =& (\mathbf{u} - \overline{\mathbf{x}}_{-K|-K})^H \, \mathscr{P}_{-K|-K}^{-1} \, \mathbf{u}' - \mathscr{J}_2' \\ & - \sum_{k=-K}^{-2} (\mathbf{y}_k - H\tilde{\mathbf{x}}_k)^H \mathscr{R}^{-1} H \tilde{\mathbf{x}}_k', \end{aligned} \tag{21.72a}$$

$$\Rightarrow \quad \mathscr{J}_2' = \left[H^H \mathscr{R}^{-1} (\mathbf{y}_{-1} - H\tilde{\mathbf{x}}_{-1}) + \mathbf{r}_{-1}^{(1)} \right]^H \tilde{\mathbf{x}}_{-1}'. \tag{21.72b}$$

Enforcing the following conditions [cf. (21.69)] for the adjoint on this interval, $\mathbf{r}^{(2)}(t)$,

$$\mathscr{L}^* \mathbf{r}^{(2)} = 0, \tag{21.73a}$$

$$\mathbf{r}_{-1}^{(2)} = H^H \mathscr{R}^{-1} (\mathbf{y}_{-1} - H\tilde{\mathbf{x}}_{-1}) + \mathbf{r}_{-1}^{(1)}, \tag{21.73b}$$

it can be shown via a derivation similar to (21.70) that

$$\mathscr{J}_2' = (\mathbf{r}_{-2}^{(2)})^H \, \tilde{\mathbf{x}}_{-2}', \tag{21.74}$$

which is of identical form as (21.71). Thus, it follows that each of the adjoints can be defined in such a way as to collapse the sum in the perturbed cost function (21.65) as above, until the last adjoint equation $\mathbf{r}^{(K)}$ reduces the perturbed cost function to the following:

$$\begin{aligned} \mathscr{J}'(\mathbf{u}') =& (\mathbf{u} - \overline{\mathbf{x}}_{-K|-K})^H \, \mathscr{P}_{-K|-K}^{-1} \, \mathbf{u}' - (\mathbf{r}_{-K}^{(K)})^H \, \tilde{\mathbf{x}}_{-K}' \\ & - (\mathbf{y}_{-K} - H\tilde{\mathbf{x}}_{-K})^H \mathscr{R}^{-1} H \tilde{\mathbf{x}}_{-K}', \end{aligned} \tag{21.75}$$

with the adjoints over the K intervals being defined as:

$$\begin{aligned} \frac{d\mathbf{r}^{(1)}(t)}{dt} &= -A^H \mathbf{r}^{(1)}(t), & \mathbf{r}_0^{(1)} &= 0 & &+ H^H \mathscr{R}^{-1} (\mathbf{y}_0 - H\tilde{\mathbf{x}}_0), \\ \frac{d\mathbf{r}^{(2)}(t)}{dt} &= -A^H \mathbf{r}^{(2)}(t), & \mathbf{r}_{-1}^{(2)} &= \mathbf{r}_{-1}^{(1)} & &+ H^H \mathscr{R}^{-1} (\mathbf{y}_{-1} - H\tilde{\mathbf{x}}_{-1}), \\ &\vdots & &\vdots \\ \frac{d\mathbf{r}^{(K)}(t)}{dt} &= -A^H \mathbf{r}^{(K)}(t), & \mathbf{r}_{1-K}^{(K)} &= \mathbf{r}_{1-K}^{(K-1)} & &+ H^H \mathscr{R}^{-1} (\mathbf{y}_{1-K} - H\tilde{\mathbf{x}}_{1-K}). \end{aligned} \tag{21.76}$$

Upon further examination, the system of adjoints (21.76) all have the same form. Each adjoint variable $\mathbf{r}^{(k+1)}$ is endowed with a starting condition that is the final condition of the adjoint march $\mathbf{r}^{(k)}$ plus a correction due to the discrete measurement $\mathbf{y}_{-k}$ at the measurement time t_{-k}. Thus, the total adjoint march can be thought of as one CT march of a single adjoint variable $\mathbf{r}(t)$ backward over the window $[t_{-K}, t_0]$, with discrete "jumps" in $\mathbf{r}$ at each measurement time t_k. That is, (21.76) can be rewritten as:

$$\frac{d\mathbf{r}(t)}{dt} = -A^H \mathbf{r}(t), \tag{21.77a}$$

which is marched backward over the entire interval $t \in [t_{-K}, t_0]$ with $\mathbf{r}_0 = 0$. At the measurement times (t_k for $k \in M$) the adjoint is updated such that

$$\mathbf{r}_k \leftarrow \mathbf{r}_k + H^H \mathscr{R}^{-1} (\mathbf{y}_k - H \tilde{\mathbf{x}}_k). \tag{21.77b}$$

Note that this update is performed right at the beginning of the march, at t_0, and also right at the end of the march, at t_{-K}, as well at all the measurement times in between. Then, this definition of the adjoint can be substituted into (21.75) to give:

$$\mathscr{J}'(\mathbf{u}') = (\mathbf{u} - \overline{\mathbf{x}}_{-K|-K})^H \mathscr{P}^{-1}_{-K|-K} \mathbf{u}' - \mathbf{r}^H_{-K} \tilde{\mathbf{x}}'_{-K}, \tag{21.78}$$

$$\Rightarrow \quad \mathscr{J}'(\mathbf{u}') = \left[\mathscr{P}^{-1}_{-K|-K} (\mathbf{u} - \overline{\mathbf{x}}_{-K|-K}) - \mathbf{r}_{-K} \right]^H \mathbf{u}', \tag{21.79}$$

where (21.79) is found by noting that $\tilde{\mathbf{x}}'_{-K} = \mathbf{u}'$. Then finally, from (21.62) and (21.79), the gradient sought may be written as:

$$\bigtriangledown \mathscr{J}(\mathbf{u}) = \mathscr{P}^{-1}_{-K|-K} (\mathbf{u} - \overline{\mathbf{x}}_{-K|-K}) - \mathbf{r}_{-K}. \tag{21.80}$$

The resulting gradient[7] can then be used iteratively to update the current estimate via a suitable minimization algorithm (steepest descent, conjugate gradient, limited-memory BFGS, etc.).

Being vector based [see (21.77), (21.80)] makes 4DVar well suited for multiscale problems, and as a result is currently used extensively by the weather forecasting community. However, it has several key disadvantages. Most significantly, upon convergence, the algorithm provides an updated mean estimate $\overline{\mathbf{x}}_{-K|0}$, but provides no clear formula for computing the updated estimate uncertainty covariance or its inverse, $\mathscr{P}^{-1}_{-K|0}$. That is, the statistical distribution of the estimate probability is not contained in the output of a traditional 4DVar algorithm. It can be shown that, upon full convergence for a linear system, the resulting analysis covariance $\mathscr{P}_{-K|0}$ is simply the Hessian of the original cost function (21.60). However, this is merely an analytical curiosity; computing the analysis covariance in this fashion requires as much matrix algebra as would be required to propagate a sequential filter through the entire variational window, defeating the purpose of the vector-based method.

Additionally, as posed above, the width of the variational window is fixed in the traditional 4DVar formulation. Thus, the cost function and associated n-dimensional minimization surface are also constant throughout the iterations. For nonlinear systems, especially chaotic systems, this makes traditional 4DVar extremely sensitive to initial conditions. Because of the nature of these systems, the optimization surfaces are highly irregular and fraught with local minima. The gradient-based algorithms associated with 4DVar are only guaranteed to converge to local minima. Thus, if the initial background estimate is located in the region of attraction of one of these local minima, the solution of the 4DVar algorithm will tend to converge to a suboptimal estimate.

Lastly, the computation time required for full convergence of the fixed-horizon 4DVar algorithm in complex systems is usually non-negligible when compared with the characteristic time scales of the system. As iterations of 4DVar over a fixed horizon proceed, one is effectively solving more and more accurately a problem which, as time bears on, one cares less and less about. When the 4DVar algorithm finally converges, the estimate so determined is for a time that has already slipped far into the past, and is of reduced relevance for producing an accurate forecast.

[7] Omitted in this gradient derivation is the substantial flexibility in the choice of the gradient definition (21.62) and the duality pairing (21.66). There is freedom in the choice of these inner products (e.g. by incorporating derivative and/or integral operators as well as weighting factors) that can serve to better precondition the optimization problem at hand without affecting its minimum points.

21.4 MPDopt

Model Predictive Dynamic Optimization (MPDopt)

This section still under construction.

21.5 Feedback control of high-dimensional systems

Exercises

Exercise 21.1 Uprighting and stabilization of a two-wheeled balancing rover.

In this project, you will design and simulate an optimal LQG controller for balancing a rover on its two wheels. The equations of motion of this system are:

g = 9.81 m/s/s mp = .4 kg Jp = .015 kg-m2 L = .0095 m mw = .2 kg [combined weight of both drive wheels] Jw = 3e-4 kg-m2 [combined moment of inertia of both drive wheels] r = .0635 meters k = 1.8 N-m/amp [torque constant of each drive motor*gear-reduction*(number of motors)] R = 1.2 ohms [armature resistance of each drive motor] l1 = .0127 m l2 = .152 m v1 = 1.5e-3 (m/s/s)2 [Accelerometer Noise Variance] v2 = 1e-3 (rad)2 [Encoder Noise Variance]

Problems:

1) Linearize the equations of motion, (1) and (2), about , in order to locally approximate them in state-space form: , using the state vector , where u represents the voltage input into both drive motors. Substitute for values of mass, inertia..etc., and solve for .

2) Use the Matlab care.m function to design an optimal linear quadratic regulator (LQR), of the form , using this state space system, and weighting matrices, Q and R of your choice. Justify your choice of Q, R in terms of their effect on the controller performance.

3) Test the performance of this controller by simulating the nonlinear equations of motion, (1) and (2), starting from a nonzero initial angle, using RK4 (or your method of choice). Use in the motor torque equation (assume that we have knowledge of the entire state, , for now). The attached m-file, plotrover.m, can be used to animate the simulation results. Please provide time-plots of u, theta, and phi.

4) Let the scalar, w, represent zero-mean, white noise with a Gaussian distribution. Modify equations (1) and (2) to include plant disturbances, w, on the system, where w may be pre-multiplied by some scalar in each equation, in order to adjust the level of exposure of the horizontal and angular dynamics to this noise. Now repeat/modify part (1) for your new equations, in order to express them in the form: . Please justify your choice of the weighting on w in terms of how it models the expected plant disturbances (wind, modeling errors.. etc.)

5) In reality, we cannot directly measure each individual component of , namely we are limited to measurements provided by two accelerometers and an encoder. The output of these three sensors is: , where v1 and v2 represent zero-mean, white noise on the accelerometer, and encoder measurements, respectively (see constants on page 2 for values of v1 and v2). As in part 1), linearize these outputs about , in order to locally approximate them as . Use the Matlab care.m command in order to design a Kalman filter for this system.

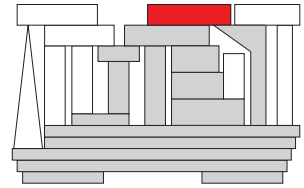
6) Augment the simulation from part 3) in order to march your actual system, (equations (1) and (2) modified with weighted noise, w, from part (4)), alongside your estimated system, , where (note that Bw and Dw indirectly enter the estimator dynamics via their influence on optimizing L). Run the closed-loop simulation using your estimated state to drive the controller, i.e. plug into the motor torque equation (LQG control). Plot your estimated state in order to verify that it converges to the actual state.

7) Modify the up-righting equations of motion, (original equations of motion (1) and (2)), in order to model the rover when the boom is touching the ground (horizontal mode), and write down the conditions under which the dynamics switch between these two modes.

8) Describe how to use the adjoint, along with the two sets of dynamics (horizontal and up-righting) in order to optimize a control trajectory, u(t) from t = 0 to t = T, to up-right the rover starting from horizontal mode, to a vertical orientation ( ) with minimum final forward velocity (i.e., minimum).

References

Green & Limebeer, (1995) *Linear Robust Control*
Zhou Doyle & Glover
Skogestad, S, & Postlethwaite, I (1996) *Multivariable Feedback Control, Analysis & Design*. Wiley.
Skelton, RE, Iwasaki, T, & Grigoriadis, K (1997) *A Unified Algebraic Approach to Linear Control Design*. Taylor & Francis.
Protas, B, Bewley, T, & Hagen, G (2004) A computational framework for the regularization of adjoint analysis in multiscale PDE systems. *J. Comp. Phys.* **195**, 49-89.



Chapter 22

State estimation & adaptive observation

Contents

22.1 The continuous-time Kalman-Bucy Filter (KBF) . . . 695
22.1.1 Deriving the equation governing the evolution of $P(t)$ . . . 697
22.1.2 Minimizing J by appropriate selection of the $L(t)$ . . . 698
22.2 The discrete-time Kalman Filter (KF) . . . 699
22.2.1 Deriving the equation governing the evolution of P_k . . . 701
22.2.2 Minimizing J by appropriate selection of the L_k . . . 701
22.2.3 Reconciling the continuous-time and discrete-time forms . . . 703
22.3 Extending the Kalman Filter to nonlinear systems . . . 704
22.3.1 The Extended Kalman Filter (EKF) . . . 704
22.3.2 The Ensemble Kalman Filter (EnKF) . . . 704
22.3.3 The Particle Filter (PF) . . . 705
22.4 Adaptive observation: a case study . . . 706
22.A HEnS . . . 711
22.B EnVO . . . 711

22.1 The continuous-time Kalman-Bucy Filter (KBF)

We now consider a continuous-time system described by[1]

$$\frac{d\mathbf{x}}{dt} = A\mathbf{x} + B\mathbf{u} + \mathbf{w}, \tag{22.1a}$$

$$\mathbf{y} = C\mathbf{x} + \mathbf{v}, \tag{22.1b}$$

[1] Note that, in the LTV case, A, B, and C are functions of t, whereas in the LTI case they are not. The present analysis may be applied to either case; for simplicity of notation, we will suppress the (t) dependence of $A(t)$, $B(t)$, and $C(t)$ in the equations that follow.

where $\mathbf{x}(t)$ is the state, $\mathbf{u}(t)$ is the control, $\mathbf{w}(t)$ is the state disturbance, $\mathbf{y}(t)$ is the measurement, and $\mathbf{v}(t)$ is the measurement noise. We model the state disturbance $\mathbf{w}(t)$ and measurement noise $\mathbf{v}(t)$ as random and unknown processes (functions of time), and the initial condition $\mathbf{x}_0$ as a random unknown quantity as well. Note that we can usually model with reasonably fidelity the relevant *statistics* of these random quantities (specifically, their mean and covariance, defined in §6). The KBF reconciles this knowledge of the relevant statistics of the random quantities $\mathbf{w}(t)$, $\mathbf{v}(t)$, and $\mathbf{x}_0$ driving this continuous-time dynamic system with the recent measurements $\mathbf{y}(t)$ that are taken of the dynamic system in order to determine a best estimate $\hat{\mathbf{x}}(t)$ of the state of the system $\mathbf{x}(t)$, as presented below. As the derivation of the KBF is based on a precise definition of the statistics of random variables and continuous-time random processes, a careful review of §6 is advised before proceeding.

We now focus on the estimation of the system described by (22.1). We assume that the initial condition $\mathbf{x}_0$ is an unknown random variable with some expected value $\overline{\mathbf{x}}_0$ and covariance P_0 which we can model. We will also assume that the state disturbance $\mathbf{w}(t)$ and the measurement noise $\mathbf{v}(t)$ are **zero-mean, nearly-white continuous-time random processes** (see §6 for definition of these terms) such that[2]

$$R_{\mathbf{w}}(\tau;t) = \mathscr{E}\{\mathbf{w}(t+\tau)\,\mathbf{w}^H(t)\} = Q\delta^{\sigma}(\tau) \tag{22.2a}$$

$$R_{\mathbf{v}}(\tau;t) = \mathscr{E}\{\mathbf{v}(t+\tau)\,\mathbf{v}^H(t)\} = R\delta^{\sigma}(\tau) \tag{22.2b}$$

for some small but finite σ, with spectral densities $Q \geq 0$ and $R > 0$ which we can model. We also assume that the random variables $\mathbf{x}_0$, $\mathbf{w}(t)$, and $\mathbf{v}(t)$ are **uncorrelated**.

We now propose an **estimator** of the form

$$\left.\begin{aligned} \frac{d\hat{\mathbf{x}}}{dt} &= A\hat{\mathbf{x}} + B\mathbf{u} - L(\mathbf{y}-\hat{\mathbf{y}}) \\ \hat{\mathbf{y}} &= C\hat{\mathbf{x}} \end{aligned}\right\} \quad \Rightarrow \quad \frac{d\hat{\mathbf{x}}}{dt} = (A+LC)\hat{\mathbf{x}} + B\mathbf{u} - L\mathbf{y} \tag{22.3}$$

to model the dynamics of the actual system (22.1). Note that neither $\mathbf{w}(t)$ nor $\mathbf{v}(t)$ appear in this model, as these quantities are unknown. Also, as the initial conditions of $\mathbf{x}$ are unknown, we will initialize $\hat{\mathbf{x}}$ simply as the expected value of $\mathbf{x}(0)$, that is, $\hat{\mathbf{x}}_0 = \overline{\mathbf{x}}_0$. To correct for all of these errors in the model, we have subtracted an **output injection** term $L(\mathbf{y}-\hat{\mathbf{y}})$ from the estimator equation for $\hat{\mathbf{x}}$. The matrix $L = L(t)$ in this term will be designed to minimize some measure of the **estimation error** $\tilde{\mathbf{x}}(t)$, defined by

$$\tilde{\mathbf{x}}(t) = \mathbf{x}(t) - \hat{\mathbf{x}}(t).$$

As $\mathbf{x}_0$ is a random vector and $\mathbf{w}(t)$ and $\mathbf{v}(t)$ are continuous-time random processes, the state $\mathbf{x}(t)$, the estimate $\hat{\mathbf{x}}(t)$, and the estimation error $\tilde{\mathbf{x}}(t)$ are continuous-time random processes as well. The evolution equation for the estimation error is easily found by subtracting (22.3) from (22.1), which leads to

$$\frac{d\tilde{\mathbf{x}}}{dt} = (A+LC)\tilde{\mathbf{x}} + \mathbf{w} + L\mathbf{v}, \tag{22.4}$$

with initial conditions $\tilde{\mathbf{x}}_0 = \mathbf{x}_0 - \overline{\mathbf{x}}_0$. Applying the above assumptions on $\mathbf{x}_0$, $\mathbf{w}(t)$, and $\mathbf{v}(t)$, it follows immediately that both $\mathscr{E}\{\tilde{\mathbf{x}}_0\} = 0$ and, indeed, that $\mathscr{E}\{\tilde{\mathbf{x}}(t)\} = 0$ for all t. Indeed, even if the expected value of $\mathbf{x}(0)$ is not accurately known, as long as the matrix L resulting from the present derivation is such that $(A+LC)$ is stable, then it follows from the expected value of (22.4) that $\mathscr{E}\{\tilde{\mathbf{x}}(t)\} \to 0$ as $t \to \infty$. We thus say that the estimate $\hat{\mathbf{x}}(t)$ defined by (22.3) is an **unbiased estimate**. Since $\tilde{\mathbf{x}}$ is a random process, however, knowing that $\mathscr{E}\{\tilde{\mathbf{x}}(t)\} \to 0$ only tells us that the components of $\tilde{\mathbf{x}}(t)$, roughly speaking, are as often positive as they

[2] In the LTV formulation of the KBF, Q and R may be functions of t, whereas in the LTI formulation they may not. The following analysis may be applied to either formulation; for simplicity of notation, we will suppress the (t) dependence of $Q(t)$ and $R(t)$ in the equations that follow.

are negative; it is not sufficient to establish that $\mathscr{E}\{\|\tilde{\mathbf{x}}(t)\|^2\}$ is by any means small. Thus, to establish this stronger condition, consider the covariance of $\tilde{\mathbf{x}}$ which, for simplicity of notation, we will denote

$$P(t) = \mathscr{E}\{\tilde{\mathbf{x}}(t)\,\tilde{\mathbf{x}}^H(t)\} \geq 0, \tag{22.5}$$

and define an appropriate metric J which measures $P(t)$ over an interval $t \in (0,T)$ such that

$$J = \int_0^T \operatorname{trace}(P)\,dt = \int_0^T \mathscr{E}\left\{\|\tilde{\mathbf{x}}(t)\|^2\right\} dt \geq 0.$$

Note that J is the integral over the time interval of interest of the expected value of the energy of the estimation error. In the derivation that follows, we will first develop an evolution equation for $P(t)$, then select $L(t)$ to minimize the metric J, thereby establishing that the estimator (22.3) with this optimized value of $L(t)$ provides a **best linear unbiased estimate** (**BLUE**).

22.1.1 Deriving the equation governing the evolution of $P(t)$

The solution of the equation for the estimation error (22.4) may be written in terms of the state transition matrix $\Phi(\tau,t)$ for continuous-time LTV systems, as introduced in §**??**:

$$\tilde{\mathbf{x}}(t) = \Phi(0,t)\mathbf{x}_0 + \int_0^t \Phi(\tau,t)\left[\mathbf{w}(\tau) + L(\tau)\mathbf{v}(\tau)\right] d\tau. \tag{22.6}$$

Differentiating (22.5) with respect to time, we may write

$$\frac{dP}{dt} = \mathscr{E}\left\{\frac{d\tilde{\mathbf{x}}(t)}{dt}\tilde{\mathbf{x}}^H(t)\right\} + \mathscr{E}\left\{\tilde{\mathbf{x}}(t)\frac{d\tilde{\mathbf{x}}^H(t)}{dt}\right\}. \tag{22.7}$$

The second term on the RHS is just the conjugate transpose of the first, so we focus our attention on the first. Applying (22.4) and then (22.6), the fact that $\mathbf{x}_0$, $\mathbf{w}(t)$, and $\mathbf{v}(t)$ are uncorrelated, the autocorrelations given in (22.2), the definition of δ^σ in (6.6), the fact that $\Phi(\tau,t) = I + O(t-\tau)$, and assuming σ is small as compared to the magnitude of the eigenvalues of A leads to

$$\begin{aligned}
\mathscr{E}\left\{\frac{d\tilde{\mathbf{x}}(t)}{dt}\tilde{\mathbf{x}}^H(t)\right\} &= \mathscr{E}\{(A+LC)\tilde{\mathbf{x}}(t) + \mathbf{w}(t) + L\mathbf{v}(t)]\tilde{\mathbf{x}}^H(t)\} \\
&= (A+LC)P(t) + \mathscr{E}\left\{\left[\mathbf{w}(t) + L(t)\mathbf{v}(t)\right]\left[\Phi(0,t)\mathbf{x}_0 + \int_0^t \Phi(\tau,t)\left[\mathbf{w}(\tau) + L(\tau)\mathbf{v}(\tau)\right] d\tau\right]^H\right\} \\
&= (A+LC)P(t) + \int_0^t \left[\mathscr{E}\{\mathbf{w}(t)\mathbf{w}^H(\tau)\} + L(t)\mathscr{E}\{\mathbf{v}(t)\mathbf{v}^H(\tau)\}L^H(\tau)\right]\Phi^H(\tau,t)\,d\tau \\
&= (A+LC)P(t) + \int_0^t \left[Q(\tau)\delta^\sigma(t-\tau) + L(t)R(\tau)\delta^\sigma(t-\tau)L^H(\tau)\right]\Phi^H(\tau,t)\,d\tau \\
&\approx (A+LC)P(t) + \frac{1}{2}\left[Q(t) + L(t)R(t)L^H(t)\right].
\end{aligned}$$

Noting that P, Q, and R are Hermitian, it thus follows from (22.7) that, for small σ,

$$\frac{dP}{dt} \approx (A+LC)P + P(A+LC)^H + LRL^H + Q.$$

Note that this relation approaches an equality as σ is made small, so the following analysis will assume equality in this expression. Note also that $P(0) = \mathscr{E}\{\tilde{\mathbf{x}}_0\tilde{\mathbf{x}}_0^H\} = \mathscr{E}\{(\mathbf{x}_0 - \overline{\mathbf{x}}_0)(\mathbf{x}_0 - \overline{\mathbf{x}}_0)^H\} = P_0$.

22.1.2 Minimizing J by appropriate selection of the $L(t)$

The remainder of the analysis now follows closely the derivation in §21.1.1-21.1.2, albeit with a matrix state equation (for the covariance) rather than a vector state equation. It will therefore be written in an analogous style, so that the parallels may easily be identified.

Summarizing the results derived above, the system of interest is now the equation for evolution of the covariance matrix $P(t)$:

$$\frac{dP}{dt} = (A+LC)P + P(A+LC)^H + LRL^H + Q \quad \text{on } 0<t<T, \tag{22.8a}$$

$$P = P_0 \quad \text{at } t=0, \tag{22.8b}$$

where $\{A, C, P_0, Q, R\}$ are specified, $Q \geq 0$, $R > 0$, and the function $L(t)$ is yet to be determined. For a given $L(t)$, the evolution of $P(t)$ is governed by the (linear) **continuous-time Lyapunov equation** given above. The **cost function** J which measures the trajectory $P(t)$ of this system is defined such that

$$J = \int_0^T \text{trace}(P)\, dt \geq 0. \tag{22.8c}$$

The problem at hand is to minimize J with respect to the function $L(t)$ subject to (22.8).

We now consider what happens when we perturb the inputs to our original system (22.8) a small amount. Small perturbations $L'(t)$ to $L(t)$ cause small perturbations $P'(t)$ to $P(t)$. Such perturbations are governed by the **perturbation equation**

$$\underbrace{\frac{dP'}{dt} - (A+LC)P' - P'(A+LC)^H}_{\mathscr{L}(P')} = \underbrace{L'CP + PC^H(L')^H + L'RL^H + LR(L')^H}_{\mathscr{B}(L')} \quad \text{on } 0<t<T, \tag{22.9a}$$

$$P' = 0 \quad \text{at } t=0, \tag{22.9b}$$

where the linear operations $\mathscr{L}(P')$ and $\mathscr{B}(L')$ are obtained via the linearization[3] of (22.8a) about the trajectory $P(L)$. The concomitant small perturbation to the (linear) cost function J is given by

$$J' = \int_0^T \text{trace}(P')\, dt. \tag{22.10}$$

Note that (22.9a) implicitly represents a linear relationship between P' and L'. Knowing this, the task before us is to reëxpress J' in such a way as to make the resulting linear relationship between J' and L' explicitly evident. To this end, define the inner product $\langle\langle A, B\rangle\rangle \triangleq \int_0^T \Re[\text{trace}(A^H B)]\, dt$ and express the **adjoint identity**

$$\langle\langle S, \mathscr{L}(P')\rangle\rangle = \langle\langle \mathscr{L}^*(S), P'\rangle\rangle + \mathbf{b}. \tag{22.11}$$

It follows after integration by parts and application of Fact 4.43 that $\mathscr{L}^*(S) = -dS/dt - (A+LC)^H S - S(A+LC)$ and $\mathbf{b} = \Re[\text{trace}(S^H P')]_{t=0}^{t=T}$. We now define the relevant **adjoint equation** by

$$\mathscr{L}^* S = I \;\Leftrightarrow\; -\frac{dS}{dt} = (A+LC)^H S + S(A+LC) + I \quad \text{on } 0<t<T, \tag{22.12a}$$

$$S = 0 \quad \text{at } t=T. \tag{22.12b}$$

[3] That is, substitute $P+P'$ for P and $L+L'$ for L in (22.8a), multiply out, and retain all terms that are linear in the perturbation quantities.

It follows from (22.12a) that S is positive definite (and, therefore, nonsingular) for $0 \leq t < T$. [This is, in fact, all we need to know about S; in this case, it turns out, we will not even have to compute it.] Substituting (22.12) and (22.9) into (22.11) and using the result to simplify (22.10), it follows that

$$\begin{aligned} J' &= \int_0^T \Re\Big[\mathrm{trace}\Big(S^H\big[L'CP + PC^H(L')^H + L'RL^H + LR(L')^H\big]\Big)\Big]\,dt \\ &= \int_0^T \Re\Big[\mathrm{trace}\Big(\big[2(CP+RL^H)S\big]\,L'\Big)\Big]\,dt \triangleq \Big\langle\!\Big\langle \frac{\mathscr{D}J}{\mathscr{D}L}, L' \Big\rangle\!\Big\rangle \quad \Rightarrow \quad \frac{\mathscr{D}J}{\mathscr{D}L} = 2S(CP+RL^H)^H \end{aligned}$$

Noting that S is nonsingular, the feedback gain L which minimizes J is thus given by

$$\frac{\mathscr{D}J}{\mathscr{D}L} = 2S(CP+RL^H)^H = 0 \quad \Rightarrow \quad L = -PC^HR^{-1}.$$

Substituting this expression for L into (22.8a), it is seen that the evolution of P is governed by a (quadratic) **continuous-time Riccati equation**, from which the L that minimizes J may readily be determined, such that

$$\boxed{\frac{dP}{dt} = AP + PA^H - PC^HR^{-1}CP + Q, \quad P(0) = P_0, \quad L = -PC^HR^{-1}.} \tag{22.13}$$

Note that this is precisely the same form as determined in (21.29). However, in this setting, we now have a much more intuitive justification / explanation for the tunable matrices $\{P_0, Q, R\}$ defining the optimal filter: P_0 parameterizes the covariance of the initial condition, $Q \geq 0$ parameterizes the covariance of the state disturbance $\mathbf{w}(t)$, and R parameterizes the covariance of the measurement noise $\mathbf{v}(t)$.

To interpret (22.13), note that if the state disturbance $\mathbf{w}(t)$ is negligible (that is, if $\mathrm{trace}(Q) \approx 0$) and the measurements worthless (that is, $\mathrm{trace}(R^{-1}) \approx 0$, which implies that all eigenvalues of R are large, which, in turn, implies lots of measurement noise $\mathbf{v}(t)$ in all measurements), then $P(t)$ simply propogates according to $dP/dt = AP + PA^H$. There are two additional effects that modify this evolution:

- Significant state disturbances $\mathbf{w}$ (with $\mathrm{trace}(Q) > 0$) *add* a positive semidefinite term to the RHS of (22.13), thereby increasing $d[\mathrm{trace}(P)]/dt$ (that is, increasing the uncertainty of the estimate).
- Accurate measurements $\mathbf{y}$ (with noise $\mathbf{v}$ characterized by $\mathrm{trace}(R^{-1}) > 0$) *subtract* a positive semidefinite term from the RHS of (22.13), thereby decreasing $d[\mathrm{trace}(P)]/dt$ (that is, decreasing the uncertainty of the estimate).

22.2 The discrete-time Kalman Filter (KF)

We now consider a discrete-time system described by[4]

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + G\mathbf{u}_k + \mathbf{w}_k, \tag{22.14a}$$

$$\mathbf{y}_k = H\mathbf{x}_k + \mathbf{v}_k. \tag{22.14b}$$

We will now develop the KF for this discrete-time system in an analogous fashion as the KBF in §22.1, where the corresponding continuous-time formulae were derived. Note that, as the derivation of the KF is based on a precise definition of the statistics of random variables and discrete-time random processes, a careful review of §6 is advised before proceeding. Note also that the following presentation is abbreviated in places to minimize repetition; the reader is thus advised to read the corresponding continuous-time derivation in §22.1 first. Finally, note that, to simplify the derivation, we will assume that F is nonsingular.

[4] Note that, in the LTV case, F, G, and H are functions of k, whereas in the LTI case they are not. The present analysis may be applied to either case; for simplicity of notation, we will suppress the k subscripts on F_k, G_k, and H_k in the equations that follow.

We now focus on the estimation of the system described by (22.14). We assume that the initial condition $\mathbf{x}_0$ is an unknown random variable with some expected value $\overline{\mathbf{x}}_0$ and covariance P_0 which we can model. We also assume that the state disturbance $\mathbf{w}_k$ and the measurement noise $\mathbf{v}_k$ are **zero-mean, white, discrete-time random processes** such that[5]

$$R_{\mathbf{w}}(j;k) = \mathscr{E}\{\mathbf{w}_{k+j}\mathbf{w}_k^H\} = Q\delta_{j0} \tag{22.15a}$$

$$R_{\mathbf{v}}(j;k) = \mathscr{E}\{\mathbf{v}_{k+j}\mathbf{v}_k^H\} = R\delta_{j0} \tag{22.15b}$$

with covariances $Q \geq 0$ and $R > 0$ which we can model. We will also assume that the random variables $\mathbf{x}_0$, $\mathbf{w}_k$, and $\mathbf{v}_k$ are **uncorrelated**.

To proceed without ambiguity in the discrete-time case, it is useful to define the notation $\hat{\mathbf{x}}_{k/j}$ (pronounced "$\mathbf{x}$ hat k given j") as the "best" estimate of $\mathbf{x}_k$ (at time t_k) accounting for the measurements up to and including $\mathbf{y}_j$ (at time t_j). In particular, we will make extensive use of $\hat{\mathbf{x}}_{k/k-1}$ (and $\hat{\mathbf{x}}_{k+1/k}$, etc., referred to as **prediction estimates**) and $\hat{\mathbf{x}}_{k/k}$ (and $\hat{\mathbf{x}}_{k+1/k+1}$, etc., referred to as **current estimates**) in the derivation that follows[6].

We now propose a two-step prediction/current **estimator** of the form

$$\left.\begin{aligned} \hat{\mathbf{x}}_{k+1/k} &= F\hat{\mathbf{x}}_{k/k} + G\mathbf{u}_k \\ \hat{\mathbf{y}}_{k+1} &= H\hat{\mathbf{x}}_{k+1/k} \\ \hat{\mathbf{x}}_{k+1/k+1} &= \hat{\mathbf{x}}_{k+1/k} - L_{k+1}(\mathbf{y}_{k+1} - \hat{\mathbf{y}}_{k+1}) \end{aligned}\right\} \quad \Rightarrow \quad \hat{\mathbf{x}}_{k+1/k} = (F + FL_kH)\hat{\mathbf{x}}_{k/k-1} + G\mathbf{u}_k - FL_k\mathbf{y}_k \tag{22.16}$$

to model the dynamics of the actual system (22.14). Note that neither $\mathbf{w}_k$ nor $\mathbf{v}_k$ appear in this model, as these quantities are unknown. Also, as the initial conditions on $\mathbf{x}_k$ are unknown, we will initialize the prediction estimate $\hat{\mathbf{x}}_{k/k-1}$ simply as the expected value of $\mathbf{x}_0$, that is, $\hat{\mathbf{x}}_{0/-1} = \overline{\mathbf{x}}_0$. In an analogous manner as the continuous-time case in (22.3), in order to correct for all of these errors in the model, we subtract an **output injection** term $L_{k+1}(\mathbf{y}_{k+1} - \hat{\mathbf{y}}_{k+1})$ from the prediction estimate $\hat{\mathbf{x}}_{k+1/k}$ in order to generate the current estimate $\hat{\mathbf{x}}_{k+1/k+1}$ at each step k. Note that the three equations at left above may be combined into the single equation shown at right. The matrix L_k will be designed to minimize some measure of the **prediction estimation error** $\tilde{\mathbf{x}}_{k+1/k}$, defined by

$$\tilde{\mathbf{x}}_{k+1/k} = \mathbf{x}_{k+1} - \hat{\mathbf{x}}_{k+1/k}.$$

As $\mathbf{x}_0$ is a random vector and $\mathbf{w}_k$ and $\mathbf{v}_k$ are discrete-time random processes, the state $\mathbf{x}_k$, the prediction estimate $\hat{\mathbf{x}}_{k/k-1}$, the current estimate $\hat{\mathbf{x}}_{k/k}$, and the prediction estimation error $\tilde{\mathbf{x}}_{k/k-1}$ are discrete-time random processes as well. The evolution equation for the prediction estimation error is easily found by subtracting (22.16) from (22.14), which leads to

$$\tilde{\mathbf{x}}_{k+1/k} = (F + FL_kH)\tilde{\mathbf{x}}_{k/k-1} + \mathbf{w}_k + FL_k\mathbf{v}_k, \tag{22.17}$$

with initial conditions $\tilde{\mathbf{x}}_{0/-1} = \mathbf{x}_0 - \overline{\mathbf{x}}_0$. Applying the above assumptions on $\mathbf{x}_0$, $\mathbf{w}_k$ and $\mathbf{v}_k$, it follows immediately that both $\mathscr{E}\{\tilde{\mathbf{x}}_{0/-1}\} = 0$ and, indeed, that $\mathscr{E}\{\tilde{\mathbf{x}}_{k+1/k}\} = 0$ for all k; we thus say that the estimate $\hat{\mathbf{x}}_{k+1/k}$ defined by (22.16) is an **unbiased estimate**. To establish that $\|\tilde{\mathbf{x}}_{k+1/k}\|$ is also small, consider the covariance of $\tilde{\mathbf{x}}_{k+1/k}$ which, for simplicity of notation, we will denote

$$P_k \triangleq \mathscr{E}\{\tilde{\mathbf{x}}_{k/k-1}\tilde{\mathbf{x}}_{k/k-1}^H\} \geq 0, \tag{22.18}$$

[5] In the LTV formulation of the KF, Q and R may be functions of k, whereas in the LTI formulation they may not. The following analysis may be applied to either formulation; for simplicity of notation, we will suppress the k subscripts on Q_k and R_k in the equations that follow.

[6] From the perspective of computing feedback, the prediction estimate $\hat{\mathbf{x}}_{k/k-1}$ is typically the most useful, as it is available prior to time t_k (and is thus available to compute feedback of the form $\mathbf{u}_k = K\hat{\mathbf{x}}_{k/k-1}$, to be applied at time t_k), whereas the current estimate $\hat{\mathbf{x}}_{k/k}$ is not. (Note that the current estimate $\hat{\mathbf{x}}_{k/k}$ is a function of the measurement $\mathbf{y}_k$, which one must wait until time t_k to take before the computation of $\hat{\mathbf{x}}_{k/k}$ may be completed.) On the other hand, if the computation speed of the computer being used is very fast as compared with the sample speed of the data acquisition hardware being used, then the current estimate $\hat{\mathbf{x}}_{k/k}$ may be used anyway to compute feedback of the form $\mathbf{u}_k = K\hat{\mathbf{x}}_{k/k}$, noting that there must be a small delay between taking the measurement $\mathbf{y}_k$ and applying the control $\mathbf{u}_k$.

and define an appropriate metric J which measures P_k over an interval $k \in [0,K]$ such that

$$J = \sum_{k=0}^{K} \text{trace}(P_k) = \sum_{k=0}^{K} \mathscr{E}\{\|\tilde{\mathbf{x}}_k\|^2\} \geq 0.$$

Note that J is the sum over the interval of interest of the expected value of the energy of the estimation error. In the derivation that follows, we will first develop an evolution equation for P_k, then select L_k to minimize the metric J, thereby establishing that the estimator (22.16) with this optimized value of L_k provides a **best linear unbiased estimate** (**BLUE**).

22.2.1 Deriving the equation governing the evolution of P_k

By (22.18), (22.17), the fact that $\mathbf{x}_0$, $\mathbf{w}_k$, and $\mathbf{v}_k$ are uncorrelated, and the autocorrelations given in (22.15), it follows that $R_{\mathbf{xw}}(0;k) = 0$ and $R_{\mathbf{xv}}(0;k) = 0$, and thus

$$\begin{aligned} P_{k+1} &= \mathscr{E}\{\tilde{\mathbf{x}}_{k+1/k}\tilde{\mathbf{x}}_{k+1/k}^H\} = \mathscr{E}\{[(F+FL_kH)\tilde{\mathbf{x}}_{k/k-1} + \mathbf{w}_k + FL_k\mathbf{v}_k]\,[(F+FL_kH)\tilde{\mathbf{x}}_{k/k-1} + \mathbf{w}_k + FL_k\mathbf{v}_k]^H\} \\ &= (F+FL_kH)P_k(F+FL_kH)^H + Q + FL_kRL_k^HF^H. \end{aligned}$$

Note also that $\mathscr{E}\{\tilde{\mathbf{x}}_{0/-1}\tilde{\mathbf{x}}_{0/-1}^H\} = \mathscr{E}\{(\mathbf{x}_0 - \overline{\mathbf{x}}_0)(\mathbf{x}_0 - \overline{\mathbf{x}}_0)^H\} = P_0$.

22.2.2 Minimizing J by appropriate selection of the L_k

The remainder of the analysis now follows closely the derivation in §21.2.1-21.2.2, albeit with a matrix state equation rather than a vector state equation. It will therefore be written in an analogous style, so that the parallels may easily be identified.

Summarizing the results derived above, the system of interest is now the equation for evolution of the covariance matrix P_k:

$$P_{k+1} = (F+FL_kH)P_k(F+FL_kH)^H + Q + FL_kRL_k^HF^H \quad \text{on } 0 < k < K, \tag{22.19a}$$

where $\{A, C, P_0, Q, R\}$ are specified, $Q \geq 0$, $R > 0$, and the L_k are yet to be determined. For given L_k, the evolution of P_k is governed by the **discrete-time Lyapunov equation** given above. The **cost function** J which measures the trajectory P_k of this system is defined such that

$$J = \sum_{k=0}^{K} \text{trace}(P_k) \geq 0. \tag{22.19b}$$

The problem at hand is to minimize J with respect to the L_k subject to (22.19).

We now consider what happens when we perturb the inputs to our original system (22.19) a small amount. Small perturbations L_k' to the L_k cause small perturbations P_k' to the P_k. Such perturbations are governed by the **perturbation equation**

$$\underbrace{P_{k+1}' - (F+FL_kH)P_k'(F+FL_kH)^H}_{\mathscr{L}(P')_k} = \underbrace{\begin{aligned} &FL_k'HP_k(F+FL_kH)^H + (F+FL_kH)P_kH^H(L_k')^HF^H \\ &+ FL_k'RL_k^HF^H + FL_kR(L_k')^HF^H \end{aligned}}_{\mathscr{B}(L')_k} \tag{22.20}$$

on $0 \leq k < K$ with $P_0' = 0$, where the linear operations $\mathscr{L}(P')_k$ and $\mathscr{B}(L')_k$ are obtained via the linearization of (22.19a) about the trajectory $P(L)$. The concomitant small perturbation to the cost function J is given by

$$J' = \sum_{k=0}^{K} \text{trace}(P_k'). \tag{22.21}$$

Note that (22.20) implicitly represents a linear relationship between the P_k' and the L_k'. Knowing this, the task before us is to reëxpress J' in such a way as to make the resulting linear relationship between J' and the L_k' explicitly evident. To this end, define the inner product $\langle\langle A,B\rangle\rangle \triangleq \sum_{k=0}^{K} \Re[\text{trace}(A_k^H B_k)]$ and express the **adjoint identity**

$$\langle\langle S, \mathscr{L}(P')\rangle\rangle = \langle\langle \mathscr{L}^*(S), P'\rangle\rangle + \mathbf{b}. \tag{22.22}$$

It follows after integration by parts and application of Fact 4.43 that $\mathscr{L}^*(S) = S_{k-1} - (F+FL_kH)^H S_k (F+FL_kH)$ and $\mathbf{b} = \Re[\text{trace}(S_k^H P_{k+1}')]_{k=-1}^{k=K}$. We now define the relevant **adjoint equation** by

$$\mathscr{L}^* S = I \;\Leftrightarrow\; S_{k-1} = (F+FL_kH)^H S_k (F+FL_kH) + I \qquad \text{on } 0 < k \le K, \tag{22.23a}$$

$$S = 0 \qquad \text{at } k = K. \tag{22.23b}$$

It follows directly from (22.23a) that S_k is positive definite (and, therefore, nonsingular) for $0 \le k < K$. [This is, in fact, all we need to know about S_k; in this case, it turns out, we will not even have to compute it.] Substituting (22.23) and (22.20) into (22.22) and using the result to simplify (22.21), it follows that

$$\begin{aligned} J' &= \sum_{k=0}^{K} \Re\Big[\text{trace}\Big(S_k^H \big[FL_k' HP_k(F+FL_kH)^H + (F+FL_kH)P_kH^H(L_k')^H F^H + FL_k' RL_k^H F^H + FL_kR(L_k')^H F^H\big]\Big)\Big] \\ &= \sum_{k=0}^{K} \Re\Big[\text{trace}\Big(2\big[F^H S_k(F+FL_kH)P_kH^H + F^H S_k F L_k R\big]^H L_k'\Big)\Big] \triangleq \Big\langle\Big\langle \frac{\mathscr{D}J}{\mathscr{D}L}, L'\Big\rangle\Big\rangle \\ &\Rightarrow \quad \frac{\mathscr{D}J}{\mathscr{D}L} = 2F^H S_k F\big[P_kH^H + L_k(R+HP_kH^H)\big] \end{aligned}$$

Noting that F and S are nonsingular, the L_k which minimize J are thus given by

$$\frac{\mathscr{D}J}{\mathscr{D}L} = 2F^H S_k F\big[P_kH^H + L_k(R+HP_kH^H)\big] = 0 \quad \Rightarrow \quad L_k = -P_kH^H(R+HP_kH^H)^{-1}.$$

Substituting this expression for L_k into (22.19a) and simplifying, it is seen that the evolution of P_k is governed by a **discrete-time Riccati equation**, from which the L_k that minimizes J may readily be determined, such that

$$\boxed{P_{k+1} = FP_kF^H - FP_kH^H(R+HP_kH^H)^{-1}HP_kF^H + Q, \quad P(0) = P_0, \quad L_k = -P_kH^H(HP_kH^H+R)^{-1}.} \tag{22.24}$$

Note that this is precisely the same form as determined in (21.56). However, in this setting, we now have a much more intuitive justification / explanation for the tunable matrices $\{P_0, Q, R\}$ defining the optimal filter: P_0 parameterizes the covariance of the initial condition, $Q \ge 0$ parameterizes the covariance of the state disturbance $\mathbf{w}_k$, and R parameterizes the covariance of the measurement noise $\mathbf{v}_k$.

To interpret (22.24), note first that, if P is nonsingular, then by the Matrix Inversion Lemma (Fact 1.9), (22.24) may be written

$$P_{k+1} = FP_kF^H - FP_kH^H[R^{-1} - R^{-1}H(P_k^{-1} + H^HR^{-1}H)^{-1}H^HR^{-1}]HP_kF^H + Q.$$

If the state disturbance $\mathbf{w}$ is negligible (that is, if $\text{trace}(Q) \approx 0$) and the measurements worthless (that is, $\text{trace}(R^{-1}) \approx 0$, which implies that all eigenvalues of R are large, which, in turn, implies lots of measurement noise $\mathbf{v}$ in all measurements), then P_k simply propogates according to $P_{k+1} = FP_kF^H$. There are two additional effects that modify this evolution:

- Significant state disturbances $\mathbf{w}_k$ (with $\text{trace}(Q) > 0$) *add* a positive semidefinite term to the RHS of this equation, thereby increasing $\text{trace}(P_{k+1})$.
- Accurate measurements $\mathbf{y}_k$ (with noise $\mathbf{v}_k$ characterized by $\text{trace}(R^{-1}) > 0$) *subtract* a positive semidefinite [by the form in (22.24)] term from the RHS of this equation, thereby decreasing $\text{trace}(P_{k+1})$.

22.2.3 Reconciling the continuous-time and discrete-time forms

Performing the substitutions

$$F_k \to I + \Delta t\, A(t_k), \quad G_k \to \Delta t\, B(t_k), \quad H_k \to C(t_k), \quad Q_k \to \Delta t\, Q(t_k), \quad R_k \to R(t_k)/\Delta t,$$

into the discrete-time system (22.14) and making Δt small provides an explicit Euler approximation of the continuous-time system (22.1). Performing the same substitutions into the discrete-time Riccati equation (22.24), we obtain

$$\frac{P_{k+1} - P_k}{\Delta t} = AP_k + P_k A^H - P_k C^H R^{-1} C P_k + Q + O(\Delta t);$$

as Δt is made small, the discrete-time Riccati equation (22.24) provides an explicit Euler approximation of the corresponding continuous-time Riccati equation (22.13), thus demonstrating that the (continuous-time) KBF and the (discrete-time) KF formulations are consistent.

22.3 Extending the Kalman Filter to nonlinear systems

22.3.1 The Extended Kalman Filter (EKF)

22.3.2 The Ensemble Kalman Filter (EnKF)

The Ensemble Kalman Filter (EnKF) is a sequential data assimilation method useful for nonlinear multiscale systems with substantial uncertainties. In practice, it has been shown repeatedly to provide significantly improved state estimates in systems for which the traditional EKF breaks down. Unlike in the KF and EKF, the statistics of the estimation error in the EnKF are not propagated via a covariance matrix, but rather are implicitly approximated via the appropriate nonlinear propagation of several perturbed trajectories ("ensemble members") centered about the ensemble mean, as illustrated in Figure **??**. The collection of these ensemble members (itself called the "ensemble"), propagates the statistics of the estimation error exactly in the limit of an infinite number of ensemble members. Realistic approximations arise when the number of ensemble members, N, is (necessarily) finite. Even with a finite ensemble, the propagation of the statistics is still consistent with the nonlinear nature of the model. Conversely, the EKF propagates only the lowest-order components of the second-moment statistics about some assumed trajectory of the system. This difference is a primary strength of the EnKF.

In practice, the ensemble members $\hat{\mathbf{x}}^j$ in the EnKF are initialized with some known statistics about an initial mean estimate $\overline{\mathbf{x}}$. The ensemble members are propagated forward in time using the fully nonlinear model equation (**??**), incorporating random forcing $\mathbf{w}^j(t)$ with statistics consistent with those of the actual state disturbances $\mathbf{w}(t)$ [see (**??**)]:

$$\frac{d\hat{\mathbf{x}}^j(t)}{dt} = f(\hat{\mathbf{x}}^j(t), \mathbf{w}^j(t)). \tag{22.25}$$

At the time t_k (for integer k), an observation $\mathbf{y}_k$ is taken [see (**??**)]. Each ensemble member is updated using this observation, incorporating random forcing $\mathbf{v}_k^j$ with statistics consistent with those of the actual measurement noise, $\mathbf{v}_k$ [see (**??**)]:

$$\mathbf{d}_k^j = \mathbf{y}_k + \mathbf{v}_k^j. \tag{22.26}$$

Given this perturbed observation $\mathbf{d}_k^j$, each ensemble member is updated in a manner consistent with the KF and EKF:

$$\hat{\mathbf{x}}_{k|k}^j = \hat{\mathbf{x}}_{k|k-1}^j + \mathscr{P}_{k|k-1}^e H^H (H \mathscr{P}_{k|k-1}^e H^H + \mathscr{R})^{-1} (\mathbf{d}_k^j - H \hat{\mathbf{x}}_{k|k-1}^j), \tag{22.27}$$

where H is the linearization of the output operator $h(\cdot)$ in (**??**). Unlike the EKF, in which the entire covariance matrix $\mathscr{P}$ is propagated using the appropriate Riccati equation, the EnKF estimate covariance $\mathscr{P}^e$ is computed "on the fly" using the second moment of the ensembles from the ensemble mean:

$$\begin{aligned} \mathscr{P}^e &= \frac{(\delta\widehat{X})\,(\delta\widehat{X})^H}{N-1}, \quad \text{where} \quad \delta\widehat{X} = \begin{bmatrix} \delta\hat{\mathbf{x}}^1 & \delta\hat{\mathbf{x}}^2 & \cdots & \delta\hat{\mathbf{x}}^N \end{bmatrix}, \\ \delta\hat{\mathbf{x}}^j &= \hat{\mathbf{x}}^j - \overline{\mathbf{x}}, \quad \text{and} \quad \overline{\mathbf{x}} = \frac{1}{N}\sum_j \hat{\mathbf{x}}^j, \end{aligned} \tag{22.28}$$

where N is the number of ensemble members, and the time subscripts have been dropped for notational clarity[7].

Thus, like the KF and EKF, the EnKF is propagated with a forecast step (22.25) and an update step (22.27). The ensemble members $\hat{\mathbf{x}}^j(t)$ are propagated forward in time using the system equations [with state

[7] Note also that the factor $N-1$ (instead of N) is used in (22.28) to obtain an unbiased estimate of the covariance matrix [see ???].

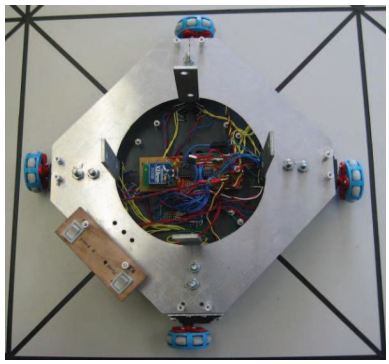
disturbances $\mathbf{w}^j(t)$] until a new measurement $\mathbf{y}_k$ is obtained, then each ensemble member $\hat{\mathbf{x}}^j(t_k) = \hat{\mathbf{x}}_k^j$ is updated to include this new information [with measurement noise $\mathbf{v}_k^j$]. The covariance matrix is not propagated explicitly, as its evolution is implicitly represented by the evolution of the ensemble itself.

It is convenient to think of the various estimates during such a data assimilation procedure in terms of the set of measurements that have been included to obtain that estimate. Just as it is possible to propagate the ensemble members forward in time accounting for new measurements, ensemble members can also be propagated *backward* in time, either retaining the effect of each measurements or subtracting this information back off. In the case of a linear system, the former approach is equivalent to the Kalman smoother, while the later approach simply retraces the forward march of the Kalman filter backward in time. In order to make this distinction clear, the notation $\widehat{X}_{j|k}$ will represent the estimate ensemble at time t_j given measurements up to and including time t_k. Similarly, $\overline{\mathbf{x}}_{j|k}$ will represent the corresponding ensemble mean; that is, the average of the ensemble and the "highest-likelihood" estimate of the system.

While the EnKF significantly outperforms the more traditional EKF for chaotic systems, further approximations need to be made for multiscale systems such as atmospheric models. When assimilating data for 3D PDEs, the discretized state dimension n is many orders of magnitude larger than the number of ensemble members N that is computationally feasible (i.e., $N \ll n$). The consequences of this are twofold. First, the ensemble covariance matrix $\mathscr{P}^e$ is guaranteed to be singular, which can lead to difficulty when trying to solve linear systems constructed with this matrix. Second, this singularity combined with an insufficient statistical sample size produces directions in phase space in which no information is gained through the assimilation. This leads to spurious correlations in the covariance that would cause improper updates across the domain of the system. This problem can be significantly diminished via the ad hoc method of "covariance localization" mentioned previously, which artificially suppresses these spurious correlations using a distance-dependent damping function.

22.3.3 The Particle Filter (PF)

The Particle Filter (PF) propagates a set of "particles" representing a very large number of potential trajectories of the system in a very similar manner as the EnKF propogates its ensemble. In the PF method, however, each particle has an associated "weighting factor" that is used to compute a biased mean and its corresponding higher moment statistics. Unlike the EnKF, at the measurement times, the particle filter uses the new observations to update the weighting factor of each particle, without actually updating the particle's position in phase space. As a result, in the limit of an infinite number of particles, this update strategy can be shown to be optimal, even for nonlinear systems with non-Gaussian uncertainties. Unfortunately, compared to the EnKF, the PF method requires excessive of computational resources even in fairly systems due to the relatively large number of particles required for adequate performance. Further, particle re-population strategies which "prune" particles with low weights from the set, and then initialize new particles near the current best estimate, are computationally intensive.



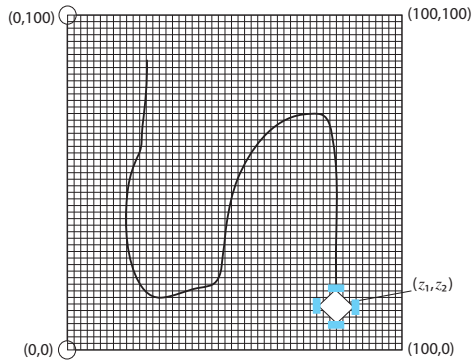


Figure 22.1: (left) A simple robot

22.4 Adaptive observation: a case study

With the foundation for both model predictive control and Kalman filtering now set, we now turn our attention to a combined problem, called the **adaptive observation** problem, in which control inputs are designed to move one or more mobile sensors in an optimized fashion in order to improve the resulting estimate of the state of the system over an area which the mobile sensors are deployed to inspect. To focus our attention, we consider first a single, extremely simple autonomous robot deployed to inspect, as thoroughly as possible, a square 100m $\times$ 100m field. The continuous-time dynamics of the state of this simple robot, denoted $\mathbf{q}$, are given by

$$\begin{aligned} z_1'' &= -z_1' + u_1 \\ z_2'' &= -z_2' + u_2 \end{aligned} \qquad \text{where} \quad \mathbf{q} = \begin{pmatrix} z_1 \\ z_1' \\ z_2 \\ z_2' \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix}, \quad \mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix},$$

where $\{z_1, z_2\}$ is the position of the robot on the field and $\{u_1, u_2\}$ are the control inputs. We assume we know the position and orientation of the robot at any given time well, using an appropriate localization system.

We assume a timestep of $h = 1$s, and we also assume that the control inputs $\{u_1, u_2\}$ are held constant over each timestep. We may thus convert the continuous-time dynamics of the plant into discrete-time state-space form (with no disturbances):

$$\mathbf{q}_{k+1} = F\mathbf{q}_k + G\mathbf{u}_k, \qquad \mathbf{q}_0 \text{ specified.} \tag{22.29}$$

[Do not bother to do this derivation again here, as you already did it for the midterm.]

We now imagine a 101×101 grid of points covering the field, and define our "state variable" $\mathbf{f}$ as the density of some contaminant (that has unfortunately been distributed over an unknown portion of the field) at each of these $101 \times 101 = 10,201$ gridpoints on the field. *Our goal is to plan a trajectory to move the robot around the field in an optimal manner over a finite period of time (that is, over $n = 300$ seconds) in order to measure the contamination distribution over the entire field with a specialized imaging system on the robot which takes a 360-degree-view photo of the field, at an appropriate wavelength, every second.* We assume that the actual state of the field, $\mathbf{f}$, (that is, the actual density of the contaminant at each of the gridpoints) is not evolving appreciably in time, and thus our estimate of the state of the field, $\hat{\mathbf{f}}$, is modelled with null dynamics:

$$\begin{aligned} \mathbf{f}_{k+1} &= \mathbf{f}_k, & \hat{\mathbf{f}}_{k+1} &= \hat{\mathbf{f}}_k - L_{k+1}(\mathbf{y}_{k+1} - \hat{\mathbf{y}}_{k+1}), \\ \mathbf{y}_{k+1} &= H\mathbf{f}_{k+1} + \mathbf{v}_k, & \hat{\mathbf{y}}_{k+1} &= H\hat{\mathbf{f}}_{k+1}. \end{aligned}$$

At each second, we assume that our camera can essentially "see" each of the 101×101 grid points on the

field, so $H = I$. The "measurement noise" in this video imaging system is modelled as a discrete-time zero-mean white Gaussian process with diagonal covariance matrix, where each component on the diagonal (that is, the variance of each measurement) is taken as being proportional to the distance squared between location of the robot and the location of the corresponding measurement point:

$$R_k = \mathscr{E}\{\mathbf{v}_k \mathbf{v}_k^H\} = \frac{1}{2}\text{diag}[(\mathbf{x}_1 - q_{1k}) \bullet (\mathbf{x}_1 - q_{1k}) + (\mathbf{x}_2 - q_{3k}) \bullet (\mathbf{x}_2 - q_{3k})], \tag{22.30}$$

$$\mathbf{x}_1 = \begin{pmatrix} 0 & \ldots & 0 & 1 & \ldots & 1 & \ldots\ldots & 100 & \ldots & 100 \end{pmatrix}^T, \tag{22.31}$$

$$\mathbf{x}_2 = \begin{pmatrix} 0 & \ldots & 100 & 0 & \ldots & 100 & \ldots\ldots & 0 & \ldots & 100 \end{pmatrix}^T, \tag{22.32}$$

where $\mathbf{a} \bullet \mathbf{b}$ denotes the elementwise product of $\mathbf{a}$ and $\mathbf{b}$. In other words, $\mathbf{v}_k$ is modelled as a random vector function of the timestep k such that the probability density function of $\mathbf{v}_k$ is given by the **Gaussian**

$$p_{\mathbf{v}_k}(\mathbf{v}') = \frac{1}{(2\pi)^{n/2}|R_k|^{1/2}} \exp\Big[-\frac{1}{2}(\mathbf{v}' - \overline{\mathbf{v}})^H R_k^{-1} (\mathbf{v}' - \overline{\mathbf{v}})\Big],$$

and thus the expected value, or mean, of $\mathbf{v}_k$ is $\mathscr{E}\{\mathbf{v}_k\} = \overline{\mathbf{v}}$, and the covariance of $\mathbf{v}_k$ is $\mathscr{E}\{\mathbf{v}_k \mathbf{v}_k^T\} = R_k$. We model $\mathbf{v}_k$ as being **zero mean** (that is, $\overline{\mathbf{v}} = 0$), as having **diagonal covariance** R_k (which means that the value of any component of $\mathbf{v}_k$ is decorrelated from that of any other component of $\mathbf{v}_k$), and **white** (which means that the autocorrelation of $\mathbf{v}_k$ is given by $\mathscr{E}\{\mathbf{v}_{k+j} \mathbf{v}_k^T\} = R_k \delta_{j0}$; in other words, this means that the value of the random vector $\mathbf{v}_k$ at any timestep t_k is decorrelated with the value of the random vector at any other timestep).

In other words, we can only image very poorly those points that are far away from the robot. Every time we make an observation, we update our estimate of the state using the standard discrete-time Kalman filter gains which, noting that, for the evolution of $\mathbf{f}_k$, the system matrix and observation matrix are I and the covariance of the state disturbances are 0, we may write as

$$L_k = -P_k(P_k + R_k)^{-1} \quad \text{with} \quad P_{k+1} = P_k - P_k(R_k + P_k)^{-1} P_k, \quad P_0 \text{ specified.} \tag{22.33}$$

Our problem is to design a sequence of control inputs $\mathbf{u}_k$, for $k \in [0, n-1]$, which moves the robot around the field in a manner that minimizes a measure of our uncertainty of the state of the field at timestep $n+1$ while not using too much control energy to move the robot, that is, to minimize a (scalar) cost function J with respect to $\mathbf{u}_k$ for $k \in [0, n-1]$ where

$$J = \text{trace}(P_{n+1}) + \frac{1}{2}\sum_{k=0}^{n-1} \mathbf{u}_k^T \mathbf{u}_k, \quad \text{where} \quad P_k = \mathscr{E}\{\tilde{\mathbf{f}}_k \tilde{\mathbf{f}}_k^T\} \quad \text{and} \quad \tilde{\mathbf{f}}_k = \mathbf{f}_{k+1} - \hat{\mathbf{f}}_{k+1}. \tag{22.34}$$

Our adaptive observation problem is now well framed as a model predictive control problem. Our task is to minimize J with respect to $\mathbf{u}_k$ for $k \in [0, n-1]$. Note in the 4 numbered equations presented thus far that we identify 4 cascading relationships: $\mathbf{u}_k$ affects the evolution of $\mathbf{q}_k$, $\mathbf{q}_k$ affects R_k, R_k affects the evolution of P_k, and P_{n+1} affects J (note that two of these relations are evolution equations, and two of these relations are algebraic; the two evolution equations will require adjoint analyses, as discussed below).

To proceed, we need to develop an expression for the gradient of the cost with respect to the control, which we can readily identify if we can write the perturbation problem in the form:

$$J' = \sum_{k=0}^{n-1} \left(\frac{DJ}{D\mathbf{u}}\right)^T \mathbf{u}_k.$$

Q.2: To accomplish this, **you first need to do a perturbation analysis, replacing $\mathbf{u}_k$ with $\mathbf{u}_k + \mathbf{u}_k'$, $\mathbf{q}_k$ with $\mathbf{q}_k + \mathbf{q}_k'$, R_k with $R_k + R_k'$, P_k with $P_k + P_k'$, and J with $J + J'$ in the four numbered relationships presented above, and subtracting out the nominal equations to obtain 4** *linear* **relationships relating**

the perturbation quantities. The first, second, and fourth of these relationships are easy to derive. To write the third in a linear fashion, $\mathscr{L}(P')_k = \mathscr{B}(R')_k$, you'll need the identity:

$$(\Phi^{-1})' = -\Phi^{-1}\Phi'\Phi^{-1}.$$

Q.2: To recap, the four coupled equations of interest in this optimization problem are:

$$\mathbf{q}_{k+1} = F\mathbf{q}_k + G\mathbf{u}_k, \quad \mathbf{q}_0 \text{ specified}, \tag{22.35a}$$

$$R_k = \frac{1}{2}\text{diag}[(q_{1k} - \mathbf{x}_1) \bullet (q_{1k} - \mathbf{x}_1) + (q_{3k} - \mathbf{x}_2) \bullet (q_{3k} - \mathbf{x}_2)], \tag{22.35b}$$

$$P_{k+1} = P_k - P_k(R_k + P_k)^{-1}P_k, \quad P_0 \text{ specified}, \tag{22.35c}$$

$$J = \text{trace}(P_{n+1}) + \frac{1}{2}\sum_{k=0}^{n-1} \mathbf{u}_k^T \mathbf{u}_k; \tag{22.35d}$$

that is, $\mathbf{u}_k$ affects the evolution of $\mathbf{q}_k$, $\mathbf{q}_k$ affects R_k, R_k affects the evolution of P_k, and P_{n+1} affects J. Note that the P_k so defined is symmetric. Linearizing to obtain the equation for the perturbations, we obtain immediately

$$\mathbf{q}'_{k+1} = F\mathbf{q}'_k + G\mathbf{u}'_k, \quad \mathbf{q}'_0 = 0, \tag{22.36a}$$

$$R'_k = \text{diag}[(q_{1k} - \mathbf{x}_1)q'_{1k} + (q_{3k} - \mathbf{x}_2)q'_{3k}], \tag{22.36b}$$

$$P'_{k+1} = P'_k - P'_k(R_k + P_k)^{-1}P_k - P_k(R_k + P_k)^{-1}P'_k + P_k(R_k + P_k)^{-1}(R'_k + P'_k)(R_k + P_k)^{-1}P_k, \quad P'_0 = 0, \tag{22.36c}$$

$$J' = \text{trace}(P'_{n+1}) + \sum_{k=0}^{n-1} \mathbf{u}_k^T \mathbf{u}'_k. \tag{22.36d}$$

Note that the P'_k so defined is symmetric.

Q.3: By doing a perturbation analysis of the problem $\Psi\Phi = I$ (that is, expanding $(\Psi + \Psi')(\Phi + \Phi') = I$ and simplifying), prove this identity.

Q.3: $\Psi\Phi = I$, $(\Psi + \Psi')(\Phi + \Phi') = I \;\Rightarrow\; \Psi\Phi' + \Psi'\Phi = 0 \;\Rightarrow\; \Psi' = -\Psi\Phi'\Phi^{-1}$. As $\Psi = \Phi^{-1}$, $(\Phi^{-1})' = -\Phi^{-1}\Phi'\Phi^{-1}$.

Q.4: Then, defining an inner product $\langle\langle A, B\rangle\rangle \triangleq \sum_{k=0}^{n} \text{trace}(A_k^T B_k)$, express an adjoint identity

$$\langle\langle S, \mathscr{L}(P')\rangle\rangle = \langle\langle \mathscr{L}^*(S), P'\rangle\rangle + b. \tag{22.37}$$

and derive the adjoint operator $\mathscr{L}^*(S)$ for this problem. Now, define an adjoint field using this adjoint operator such that the linear dependence of J' on P'_{n+1} can be rewritten as a linear depencence of J' on R'_k, for $k \in [0, n-1]$.

Q.4: First, identify the linear operators relating P'_k and R'_k in (22.36c):

$$\underbrace{P'_{k+1} - P'_k + P'_k(R_k + P_k)^{-1}P_k + P_k(R_k + P_k)^{-1}P'_k - P_k(R_k + P_k)^{-1}P'_k(R_k + P_k)^{-1}P_k}_{\mathscr{L}(P')_k} = \underbrace{P_k(R_k + P_k)^{-1}R'_k(R_k + P_k)^{-1}P_k}_{\mathscr{B}(R')_k} \tag{22.38}$$

with $P'_0 = 0$. We now frame an adjoint identity based on a relevant inner product:

$$\langle\langle S, \mathscr{L}(P')\rangle\rangle = \langle\langle \mathscr{L}^*(S), P'\rangle\rangle + b \quad \text{where} \quad \langle\langle A, B\rangle\rangle \triangleq \sum_{k=0}^{n} \text{trace}(A_k^T B_k). \tag{22.39}$$

Doing the necessary rearrangement of sums [and applying the identity $\text{trace}(ABC) = \text{trace}(BCA)$ as necessary] to recast the form on the LHS of (22.39) into the form on the RHS, we can readily identify the required expressions for $\mathscr{L}^*(S)_k$ and b:

$$\mathscr{L}^*(S)_k = S_{k-1} - S_k + S_k P_k (R_k + P_k)^{-1} + (R_k + P_k)^{-1} P_k S_k - (R_k + P_k)^{-1} P_k S_k P_k (R_k + P_k)^{-1}, \tag{22.40a}$$

$$b = \text{trace}(S_n P'_{n+1}) - \text{trace}(S_{-1} P'_0). \tag{22.40b}$$

Leveraging these expression, we now define a relevant adjoint equation

$$\mathscr{L}^*(S)_k = 0, \quad S_n = I; \tag{22.41}$$

noting that S so defined is symmetric, and substituting (22.41) and (22.38) into (22.39) gives

$$\sum_{k=0}^{n} \text{trace}[S_k \mathscr{B}(R')_k] = \text{trace}(P'_{n+1}),$$

which allows us to rewrite the cost function perturbation (22.36d) in the form:

$$J' = \sum_{k=0}^{n} \text{trace}[S_k P_k (R_k + P_k)^{-1} R'_k (R_k + P_k)^{-1} P_k] + \sum_{k=0}^{n-1} \mathbf{u}_k^T \mathbf{u}'_k. \tag{22.42}$$

Q.5: Once you have the above accomplished, rewriting the linear dependence of J' on R'_k, for $k \in [0, n-1]$, as a linear dependence of J' on $\mathbf{q}'$, for $k \in [0, n-1]$, is just algebraic. Perform this substitution.

Q.5: Inserting (22.36b) into (22.42) gives

$$\begin{aligned} J' &= \sum_{k=0}^{n} \text{trace}\{\underbrace{(R_k + P_k)^{-1} P_k S_k P_k (R_k + P_k)^{-1}}_{A_k} [\underbrace{\text{diag}(q_{1k} - \mathbf{x}_1)}_{B_k} q'_{1k} + \underbrace{\text{diag}(q_{3k} - \mathbf{x}_2)}_{C_k} q'_{3k}]\} + \sum_{k=0}^{n-1} \mathbf{u}_k^T \mathbf{u}'_k \\ &= \sum_{k=0}^{n} [\text{trace}\{A_k B_k\} q'_{1k} + \text{trace}\{A_k C_k\} q'_{3k}] + \sum_{k=0}^{n-1} \mathbf{u}_k^T \mathbf{u}'_k \end{aligned} \tag{22.43}$$

Q.6: Finally, rewrite the linear dependence of J' on $\mathbf{q}'_k$, for $k \in [0, n-1]$, as a linear dependence of J' on $\mathbf{u}'_k$, for $k \in [0, n-1]$, via the appropriate definition of another adjoint field $\mathbf{r}_k$.

Q.6: First, identify the linear operators relating P'_k and R'_k in (22.36c):

$$\underbrace{\mathbf{q}'_{k+1} - F \mathbf{q}'_k}_{\mathscr{L}(\mathbf{q}')_k} = G \mathbf{u}'_k \tag{22.44}$$

with $\mathbf{q}'_0 = 0$. We now frame an adjoint identity based on a relevant inner product:

$$\langle\langle \mathbf{r}, \mathscr{L}(\mathbf{q}')_k \rangle\rangle = \langle\langle \mathscr{L}^*(\mathbf{r}), \mathbf{q}' \rangle\rangle + b \quad \text{where} \quad \langle\langle \mathbf{a}, \mathbf{b} \rangle\rangle \triangleq \sum_{k=0}^{n} \mathbf{a}_k^T \mathbf{b}_k. \tag{22.45}$$

Doing the necessary rearrangement of sums to recast the form on the LHS of (22.45) into the form on the RHS, we can readily identify the required expressions for $\mathscr{L}^*(\mathbf{r})_k$ and b:

$$\mathscr{L}^*(\mathbf{r})_k = \mathbf{r}_{k-1} - F^T \mathbf{r}_k, \tag{22.46a}$$

$$b = \mathbf{r}_n^T \mathbf{q}'_{n+1} - \mathbf{r}_{-1}^T \mathbf{q}'_0. \tag{22.46b}$$

Leveraging these expression, we now define a relevant adjoint equation

$$\mathscr{L}^*(\mathbf{r})_k = \begin{pmatrix} \text{trace}\{A_k B_k\} \\ 0 \\ \text{trace}\{A_k C_k\} \\ 0 \end{pmatrix}, \quad \mathbf{r}_n = 0; \tag{22.47}$$

substituting (22.47) and (22.44) into (22.45) gives

$$\sum_{k=0}^{n} \mathbf{r}_k^T G \mathbf{u}_k' = \sum_{k=0}^{n} [\text{trace}\{A_k B_k\} q_{1k}' + \text{trace}\{A_k C_k\} q_{3k}'],$$

which allows us to rewrite the cost function perturbation (22.43) in the form:

$$J' = \sum_{k=0}^{n-1} \underbrace{[G^T \mathbf{r}_k + \mathbf{u}_k]^T}_{(DJ/D\mathbf{u})_k} \mathbf{u}_k'. \tag{22.48}$$

Q.7: Even if you didn't work out all of the adjoint equations described above precisely to every last detail, the flow of information forward with the states and backward with the adjoints should now be clear. Describe the resulting complete process of determining the gradient $\frac{DJ}{D\mathbf{u}}$ as precisely as you can. Once you have a workable set of equations to calculate this gradient, how can you optimize the sequence of control inputs which you seek? Discuss in detail.

Q.7: Putting it all together:

(A) Guess $\mathbf{u}_k$ for $k \in [0, n-1]$.

(B) March (22.35a) to compute $\mathbf{q}_k$ for $k = 1$ to n:

$$\mathbf{q}_{k+1} = F\mathbf{q}_k + G\mathbf{u}_k, \quad \mathbf{q}_0 \text{ specified.} \tag{22.49a}$$

(C) March (22.35c), with (22.35b), to compute P_k for $k = 1$ to $n+1$:

$$P_{k+1} = P_k - P_k \left\{ \frac{1}{2}\text{diag}[(q_{1k} - \mathbf{x}_1) \bullet (q_{1k} - \mathbf{x}_1) + (q_{3k} - \mathbf{x}_2) \bullet (q_{3k} - \mathbf{x}_2)] + P_k \right\}^{-1} P_k, \quad P_0 \text{ specified.} \tag{22.49b}$$

The variables $\mathbf{q}_k$ and P_k need to be saved on these forward marches for later use; however, note that steps (B) and (C) may be performed simultaneously to streamline memory references.

(D) March (22.41) to compute S_k for $k = n$ back to 0:

$$S_{k-1} = S_k - S_k P_k (R_k + P_k)^{-1} - (R_k + P_k)^{-1} P_k S_k + (R_k + P_k)^{-1} P_k S_k P_k (R_k + P_k)^{-1}, \quad S_n = I. \tag{22.49c}$$

(E) March (22.47) to compute $\mathbf{r}_k$ for $k = n$ back to 0:

$$\mathbf{r}_{k-1} = F^T \mathbf{r}_k + \begin{pmatrix} \text{trace}\{(R_k + P_k)^{-1} P_k S_k P_k (R_k + P_k)^{-1} \text{diag}(q_{1k} - \mathbf{x}_1)\} \\ 0 \\ \text{trace}\{(R_k + P_k)^{-1} P_k S_k P_k (R_k + P_k)^{-1} \text{diag}(q_{3k} - \mathbf{x}_2)\} \\ 0 \end{pmatrix}, \quad \mathbf{r}_n = 0. \tag{22.49d}$$

(F) Identify the gradient via (22.48) for $k \in [0, n-1]$:

$$\left(\frac{DJ}{D\mathbf{u}}\right)_k = G^T \mathbf{r}_k + \mathbf{u}_k. \tag{22.49e}$$

Note that steps (D), (E), and (F) may be performed simultaneously, thereby eliminating the need to store the S_k and $\mathbf{r}_k$ for later use.

(G) Update $\mathbf{u}_k$ for $k \in [0, n-1]$ using your favorite gradient-based algorithm: e.g., for sufficiently small α,

$$\mathbf{u}_k \leftarrow \mathbf{u}_k - \alpha \left(\frac{DJ}{D\mathbf{u}} \right)_k . \tag{22.49f}$$

[In fact, this gradient-based algorithm converges very slowly; a much better method which is just as simple to apply, called the Conjugate Gradient method, is presented in §16 of *Numerical Renaissance*...]

(H) Repeat from step (A) until convergence.

22.A HEnS

22.B EnVO

Exercises

References

Anderson, BDO, & Moore, JB (1979) *Optimal Filtering*. Dover.

Supplement

A Programming: a brief introduction **3**

B Assorted mathematical foundations **15**

Numerical Renaissance is written in a manner which attempts to go way back to a fairly basic level and work up several related concepts essentially from first principles, in a common language, while minimizing repetition. The text is written assuming only a prior exposure to

- the arithmetic of complex variables,
- the essentials of computer programming (`for` loops, `if` statements, `function` calls, and floating-point operations on scalars, vectors, and matrices), and
- the basic framework of undergraduate-level linear algebra (matrix/vector multiplication and solution of the problem $A\mathbf{x} = \mathbf{b}$).

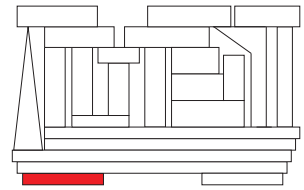
The first two of these prerequisite areas are reviewed and extended in these two appendices:

- §A introduces Matlab/Octave programming syntax, highlighting some of its more subtle features, and
- §B discusses a variety important mathematical concepts used throughout the text.

The third prerequisite area, undergraduate-level linear algebra, is reviewed and extended in §1 and §2.

Recall also that the numerical codes presented throughout the text, together with test scripts which clearly demonstrate how they work, are maintained online at `http://numerical-renaissance.com/`; *before embarking upon serious study of this text, the ready is advised to download and install the Numerical Renaissance Codebase on his/her computer.*

While you are at the webpage for the text, it is advised that the reader also download the electronic version of the text; since it is freely searchable, the electronic version is handy for reference, cross-reference, and quickly reviewing various concepts. (Note that we dispense entirely with an index for the text, as the electronic version is easy to search.) On the other hand, the printed version of the text is generally the most comfortable and effective to use for focused study (e.g., classwork and thesis work); your purchase of the printed version of the text goes directly to support further refinements of the codebase and, ultimately, a new-and-improved second edition of the text.



Appendix A

Programming: a brief introduction

Matlab (a **portmanteau word**[1] formed from **mat**rix **lab**oratory) is a powerful high-level programming language marketed by The MathWorks, and is available at http://www.mathworks.com. Though expensive[2], Matlab has become (for small-scale problems[3]) a de facto industry standard for, among other things, linear algebra, data analysis & visualization, control design, system identification, and optimization.

GNU Octave is a powerful, free, user-developed alternative to Matlab that is mostly compatible with Matlab syntax, and is available at http://www.octave.org. A distinct advantage of designing all codes you write in Matlab syntax to run in both Matlab & Octave is that both you and others are assured free access to a legal copy of Octave for any computer platform that you might someday use.

All of the numerical algorithms presented in Matlab syntax in the present text are designed to run in both Matlab & Octave (with a little work, yours may be developed in a similar manner). Thus, you have the choice when following this text to support either high-end, industry-standard commercial software or free, user-developed open-source software, per your individual preference. Please contact the author if you encounter coding errors running any of the algorithms presented in this text in recent versions of either language.

Many numerical problems encountered in science, engineering, and other disciplines are indeed fairly small, and may thus be addressed well with Matlab & Octave. Both languages are also quite useful as intuitive programming languages in which one can experiment, on small-scale systems, with maximally efficient numerical algorithms designed for large-scale systems in an interactive, user-friendly environment in which plotting the simulation results is especially simple. It is thus highly recommended that those who develop and use numerical algorithms today acquire and use (when appropriate) either Matlab or Octave. However, large-scale systems which require intensive computations are much more efficiently solved in compiler-based low-level languages, such as Fortran and C, in which the programmer has much more precise control over both the memory usage and the parallelization of the numerical algorithm. Conversion of numerical algorithms from Matlab/Octave syntax to Fortran or C syntax is straightforward (see §11).

Note that, if you have a fast connection to the internet and access to a unix-based **server** with Matlab already installed, it is possible to run Matlab remotely on the server from your own computer (referred to in this setting as the **client**), using a standard communication protocol known as **The X Window System**

[1] A portmanteau word is formed out of parts of other words, which is common in the naming of computer hardware and software. For example, **Fortran** is a portmanteau word formed from **for**mula **tran**slation, **codec** from **co**der/**dec**oder, **voxel** from **vo**lumetric pi**xel**, etc. Such words are often formed informally as new technology is developed, then become established through usage.

[2] Note that the student edition of Matlab is available at many university bookshops at a significant discount.

[3] A useful definition of a **small-scale numerical problem** is one that takes a significantly longer time to code than it does to run.

(a.k.a. **XWindows**) to transmit the graphical output of Matlab from the server to the client. If you choose this option, it is recommended that you start Matlab on the server in a unix window with the command

```
matlab -nojvm -nosplash
```

in order to run Matlab directly in the unix window without its communication-intensive **graphical user interface** (**GUI**). Users of the unix shells[4] **csh** or **tcsh** can set this to the default action when the command `matlab` is typed by putting the following command in their `~/.cshrc` file on the server:

```
alias matlab 'matlab -nojvm -nosplash'
```

Users of the **sh**, **ksh**, or **bash** can set this by putting the following in `~/.shrc`, `~/.kshrc`, or `~/.bashrc` :

```
alias matlab='matlab -nojvm -nosplash'
```

Note that, to get XWindows to successfully port the graphics from a server with an IP number 987.654.32.10 to a client with an IP number 123.456.78.90 (assuming the csh or tcsh is being run on both), you might need to run the following command on the client

```
xhost +987.654.32.10
```

and the following command on the server (before starting Matlab)

```
setenv DISPLAY 123.456.78.90:0.0
```

(or `:1.0`, depending on the configuration of the client).

Fundamentals of both Matlab and Octave

Installing either Matlab or Octave is straightforward, following the instructions distributed with each package. Once you get Matlab or Octave running, take it for a test drive, and you will most likely find that no manual or classroom instruction on either language is necessary. Most of the basic constructs available in these languages (primarily `for` loops, `if` statements, and `function` calls) can be understood easily by examining the sample codes available in this text. Also, the (lowercase[5]) built-in command names in Matlab/Octave are all intuitive (`sin` for computing the sine, `eig` for computing eigenvalues/eigenvectors, etc.), and extensive online help for all commands is available in both Matlab and Octave by typing `help` <*command name*>, with even more information on both languages available on the web. These help pages also point you to several related commands, which can be used to learn what you need to know about any given aspect of either language very quickly. To help get you off to a fast start, we now introduce some of the fundamental constructs used by both Matlab and Octave, then explain some of their more subtle features.

To begin, Matlab or Octave can function as an ordinary calculator. At the prompt, try typing

```
>> 1+1
```

Matlab or Octave should reassure you that the universe is still in good order. To enter a matrix, type

```
>> A = [1 2 3; 4 5 6; 7 8 0]
```

Matlab/Octave responds with

```
A =
     1     2     3
     4     5     6
     7     8     0
```

By default, Matlab/Octave operates in an **echo mode**; that is, the elements of a matrix or vector will be printed on the screen as it is created. This behavior quickly become tedious—to suppress it, type a semicolon after the command; this also allows several commands to be included on a single line:

[4] In order to customize your unix environment, you first need to know what **shell** (the protocol defining the set of commands available at the unix command line) you are running in (type `echo $SHELL` to find out).

[5] The names of the codes developed in this text are mixed case, to distinguish them from the (lowercase) built-in commands.

```
>> A = [1 2 3; 4 5 6; 7 8 0];  x = 5;
```

To put multiple commands on a single line without suppressing echo mode, separate the commands by commas. Matrix elements are separated by spaces or commas, and a semicolon indicates the end of a row. Three periods in a row means that the present command is continued on the following line, as in:

```
>> A = [1 2 3;    ...
        4 5 6;    ...
        7 8 0];
```

When working with long expressions, the legibility of your code can often be improved significantly by aligning entries in a natural fashion, as illustrated above. Elements of a matrix can also be arithmetic expressions, such as `3*pi`, etc.

Matlab/Octave syntax has control flow statements, such as `for` loops, similar to most other programming languages. Note that each `for` must be matched by an `end`. To illustrate, the commands

```
>> for j=1:10; a(j)  =j^2; end;   b=[0:2:10];
```

build row vectors (type `a,b` to see the result), whereas the commands

```
>> for j=1:10; c(j,1)=j^2; end;   d=[0:2:10]';
```

build column vectors. In most cases, you want the latter, not the former. *The most common mistake made in Matlab/Octave syntax is to build a row vector when you intend to build a column vector*, as they are often not interchangeable; thus, pay especially close attention to this issue if your code is misbehaving.

An `if` statement may be used as follows:

```
>> n = 7;
>> if n > 0, sgn = 1, elseif n < 0, sgn = -1, else sgn = 0, end
```

The format of a `while` statement is similar to that of `for`, but exits at the control of a logical condition:

```
>> m = 0;
>> while m < 7, m = m+2; end, m
```

A column vector `y` can be premultiplied by a matrix `A` and the result stored in a column vector `z` with

```
>> z = A*y
```

Multiplication of a vector `y` by a scalar may be accomplished with

```
>> z = 3.0*y
```

The transpose, $B = A^T$, and the conjugate transpose, $C = A^H$, are obtained as follows

```
>> B = A.', C=A'
```

The inverse of a square matrix (if it exists) may be obtained by typing

```
>> D = inv(A)
```

This command is rewritten in §2 of the present text (see Algorithm 2.3); as mentioned in §2 (indeed, as identified as early as §1.2.7), you should not ever compute a matrix inverse (which is expensive to calculate) in a production code, though it is sometimes convenient to compute a matrix inverse in a test code.

A 5×5 identity matrix may be constructed with

```
>> E = eye(5)
```

Tridiagonal matrices may be constructed by the following command and variations thereof:

```
>> 1*diag(ones(m-1,1),-1) - 2*diag(ones(m,1),0) + 1*diag(ones(m-1,1),1)
```

See also Algorithm 1.1. There are two "matrix division" symbols in Matlab/Octave, `\` and `/` — if `A` is a nonsingular square matrix, then `A\B` and `B/A` correspond formally to left and right multiplication of `B` (which must be of the appropriate size) by the inverse of `A`, that is `inv(A)*B` and `B*inv(A)`, but the result is obtained directly (via Gaussian elimination with complete pivoting, as discussed, and rewritten from scratch, in §2, but leaving the matrix `A` in tact) without the computation of the inverse (which is a very expensive computation). Thus, to solve a system `A*x=b` for the unknown vector `x`, one may type

```
>> A=[1 2 3; 4 5 6; 7 8 0];  b=[5 8 -7]';  x=A\b
```

which results in

```
x =
   -1
    0
    2
```

To check this result, just type

```
>> A*x
```

which verifies that

```
ans =
    5
    8
   -7
```

Starting with the innermost group(s) of operations nested in parentheses and working outward, the usual precedence rules are observed by Matlab/Octave. First, all the exponentials are calculated. Then, all the multiplications and divisions are calculated. Finally, all the additions and subtractions are calculated. In each of these three catagories, the calculation proceeds from left to right through the expression. Thus

```
>> 5/5*3
ans = 3
```

and

```
>> 5/(5*3)
ans = 0.3333
```

If in doubt, use parentheses to ensure the order of operations is as you intend. Note that the matrix sizes must be correct for the requested operations to be defined or an error will result.

Suppose we have two vectors **x** and **y** and wish to perform the componentwise operations:

$$z_\alpha = x_\alpha * y_\alpha \quad \text{for } \alpha = 1, n.$$

The Matlab/Octave command to execute this is

```
>> x=[1:5]';  y=[6:10]';  z=x.*y
```

Note that `z=x*y` gives an error, since this implies matrix multiplication and is undefined in this case. (A row vector times a column vector, however, is a well defined operation, so `z = x'*y` is successful. Try it!) The period distinguishes matrix operations (`*` `^` and `/`) from component-wise operations (`.*` `.^` and `./`).

Typing `whos` lists all the variables created up to that point, and typing `clear` removes them. The `format` command is also useful for changing how Matlab/Octave prints things on the screen (type `help format` for more information). In order to save the entire variable set computed in a session, type `save session` prior to quitting. At a later time the session may be resumed by typing `load session`.

Some additional (self-explanatory) functions include: `abs`, `conj`, `real`, `imag`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `exp`, `log`, `log10`. Some useful predefined constants include `pi`, `i`, `eps`; note that your code can change the values of these constants (it is particularly common to use `i` as an indexing variable) — be especially careful if you decide to do this!

Matlab and Octave are also distributed with many special additional functions to aid in linear problem-solving, control design, etc. In Octave (but, unfortunately, not in recent versions of Matlab), many of these advanced built-in functions are themselves saved as prewritten `m`-files (see below), and can easily be opened and accessed by the user for examination. *The present pedagogical text specifically avoids using most of these convenient* **black-box** *functions*, instead opting to work up many of them from scratch in order to remove the mystery that might otherwise be associated with using them.

Note that, in order to make a Octave **flush** (that is, print messages to the screen) before it either finishes running or encounters a `pause` statement in the code, you must insert the command `fflush(1);` before each `pause` command; such a flush command is not needed in Matlab. This is one of the few yet sometimes annoying differences between Octave and Maltab.

Matlab programming procedures: stay organized!

As an alternative to interactive mode, you can also save a series of Matlab/Octave commands as `m`-files, which are **ASCII** (**American Standard Code for Information Interchange**, a.k.a. **plain text**) files, with a descriptive filename ending in `.m`, containing a sequence of commands listed exactly as you would enter them if running interactively. When working on Matlab/Octave problems that take more than one line to complete (that is, essentially, all the time), *it is imperative to write and run* `m`*-files rather than working in interactive mode*. By so doing, it is self evident which calculation followed from which. Further, following this approach, the several commands typically required to perform a given calculation don't need to be retyped when the simulation needs to be rerun, which is generally more often than one would care to admit. *Staying organized with different versions of your* `m`*-files as the project evolves is essential; create new directories and subdirectories as appropriate for each problem you work on to stay organized, and to keep from accidentally overwriting previously-written and debugged codes.*

To execute the commands in an `m`-file named[6] `foo.m`, simply type `foo` at the Matlab/Octave prompt. Any text editor may be used to edit `m`-files. The `%` symbol is used in these files to indicate that the rest of that line is a comment. Entering `help foo` will print the first (commented) lines of the code to the screen, entering `type foo` will print the entire code to the screen[7]. *Comment all* `m`*-files clearly, sufficiently, and succinctly*, so that you can come back to the code later and understand how it works. Depending on the purpose of the code, you can also print to the screen (using the `disp` command) to update you on the code's progress as it runs; several codes presented in this text use a `verbose` flag to turn such updates on or off. It also helps to use descriptive variable names to indicate what the code is doing.

[6] Almost all texts describing computer programming make reference to simple example codes named `foo` and `bar`, though the etymology of this tradition is not precisely known.

[7] In the electronic version of this text, click View to see a plain text version of the production `m`-file in a web browser, and click Test to see the corresponding test code. To save space, the copyleft comments at the beginning of each code are not printed in the text.

The distinction between functions and scripts

There are two types of `m`-files: **scripts** and **functions**. A **script** is a set of Matlab/Octave commands which run exactly as if you had typed in each command interactively. A script has access to all previously-defined variables (that is, it inherits the **global workspace**).

A **function**, on the other hand, is a set of Matlab/Octave commands that begins with a `function` declaration, for example,

```
function [o1,o2] = bar(i1,i2,i3)
```

A function so defined may then be **called** (as in a compiler-based programming language) with the command

```
[c,d] = bar(a,b,c)
```

Note that some variables (in the above example, `c`) may be used as both inputs and outputs. As a function is running, it can only reference those variables in the input list with which it was called; in the present example, the function is only "aware" of the variables `a`, `b`, and `c`, which the function refers to internally as `i1`, `i2`, and `i3`. Conversely after the function is finished, the only variables that are modified as compared with before the function was called are those variables in the output list with which it was called; in the present example, the function ultimately only modifies the variables `c` and `d`, which the function refers to internally as `o1` and `o2`.

The special variables `nargin` and `nargout` identify the number of input and output arguments, respectively, that are used when any given function is called. Input and output arguments are assigned from the left to the right, so any missing variables are necessarily those at the end of each list. If some of the input arguments are omitted in the function call, these variables may be set to default values by the function. If some of the output arguments are omitted in the function call, the function may sometimes be accelerated by avoiding the explicit computation of these variables.

Functions are more easily extended to other problems than scripts, as functions make clear via their argument list what information from the calling function is used in, and returned by, the called function[8]. In complicated codes, unintentionally assigning a minor variables (like the index `i`) with different meanings in different scripts that call each other can lead to a bug that is very difficult to find. The proper use of functions, and the associated passing of only the relevant data back and forth (known as **handshaking**), goes a long way towards preventing such bugs from appearing in your production codes.

On the other hand, short test codes, such as those provided with each of the functions developed in this text, are usually convenient to write as scripts, so that the variables defined by the test script may be checked (for debugging purposes) after the test script runs. To illustrate, ten simple functions and one associated test script are listed in Algorithm A.1; in the comments at the beginning of each function is a brief description of what that function does. The coefficients of polynomials are represented frequently in this text as row vectors; the functions listed in Algorithm A.1 are useful to perform polynomial addition, convolution, division, etc. Algorithm A.3 is an example script illustrating the use of Matlab's built-in functions `eig` and `inv` to build matrices of various structure, as discussed further in §1 and 4.

The primary directory Matlab uses (for both saving new files and reading old ones) may be changed with the command `cd` *<directory name>*; in Matlab, there are also menu-driven ways of accomplishing this. If the necessary `m`-files to run your code are stored in more than one directory, which is often the case, the `path` command may be used to indicate to Matlab these additional directories in which to look for the appropriate `m`-files. A convenient startup script is given in Algorithm A.4 to extend the Matlab/Octave path to access all of the codes of the *Numerical Renaissance Codebase* (*NRC*); it is convenient (and, thus, recommended) to place a copy of this file, named `startup.m`, in the directory where Matlab/Octave normally starts up on your computer in order to initialize this path on startup (note that the variable `base` in this `startup.m` file must be updated appropriately in order to point at the directory where the *NRC* is stored on your computer.

[8] That is, in addition to those variables defined as global, as seen in Algorithm 11.8 and discussed in footnote 26 on page 375.

Algorithm A.1: Some simple example functions, and an example test script.

```
function p=Prod(a)
% Compute the product p of the elements in the vector a.
p=a(1); for i=2:length(a); p=p*a(i); end
end % function Prod
```
View

```
% script <a href="matlab:ProdTest">ProdTest</a>
disp('Compute the product of the elements in a'), a=[2 4 5], p=Prod(a), disp(' ')
% end script ProdTest
```
View

```
function a=Fac(b)
% Compute the factorial of each element of the matrix b.
[n,m]=size(b); for i=1:n,for j=1:m, a(i,j)=1; for k=2:b(i,j); a(i,j)=a(i,j)*k; end,end,end
end % function Fac
```
View Test

```
function p=Poly(r)
% Compute the coefficients of the polynomial with roots r.
n=length(r); p=1; for i=1:n; p=PolyConv(p,[1 -r(i)]); end
end % function Poly
```
View Test

```
function v=PolyVal(p,s)
% For n=length(p), compute p(1)*s(i)^(n-1) + ... + p(n-1)*s(i) + p(n) for each s(i) in s.
n=length(p); for j=1:length(s); v(j)=0; for i=0:n-1, v(j)=v(j)+p(n-i)*s(j)^i; end, end
end % function PolyVal
```
View Test

```
function a=PolyAdd(a,b,c,d,e,f,g,h,i,j)
% Add two to ten polynomials with right justification.
if nargin>9, a=PolyAdd(a,j); end, if nargin>8, a=PolyAdd(a,i); end
if nargin>7, a=PolyAdd(a,h); end, if nargin>6, a=PolyAdd(a,g); end
if nargin>5, a=PolyAdd(a,f); end, if nargin>4, a=PolyAdd(a,e); end
if nargin>3, a=PolyAdd(a,d); end, if nargin>2, a=PolyAdd(a,c); end
m=length(a); n=length(b); a=[zeros(1,n-m) a]+[zeros(1,m-n) b];
end % function PolyAdd
```
View Test

```
function p=PolyConv(a,b,c,d,e,f,g,h,i,j)
% Recursively compute the convolution of the two to ten polynomials, given as arguments.
if nargin>9, a=PolyConv(a,j); end, if nargin>8, a=PolyConv(a,i); end
if nargin>7, a=PolyConv(a,h); end, if nargin>6, a=PolyConv(a,g); end
if nargin>5, a=PolyConv(a,f); end, if nargin>4, a=PolyConv(a,e); end
if nargin>3, a=PolyConv(a,d); end, if nargin>2, a=PolyConv(a,c); end
m=length(a); n=length(b); p=zeros(1,n+m-1);
for k=0:n-1; p=p+[zeros(1,n-1-k) b(n-k)*a zeros(1,k)]; end
end % function PolyConv
```
View Test

```
function [d,b]=PolyDiv(b,a)
% Perform polynomial division of a into b, resulting in d with remainder in the modified b.
m=length(b); n=length(a); if m<n d=0; else
  if strcmp(class(b),'sym')|strcmp(class(a),'sym'), syms d, end
  for j=1:m-n+1, d(j)=b(1)/a(1); b(1:n)=PolyAdd(b(1:n),-d(j)*a); b=b(2:end); end, end
end % function PolyDiv
```
View Test

```
function b=PolyPower(p,n)
% Compute the convolution of the polynomial p with itself n times.
if n==0, b=1; else, b=p; for i=2:n, b=PolyConv(b,p); end
end % function PolyPower
```
View Test

```
function p=PolyDiff(p,d,n)
% Recursively compute the d'th derivative of a polynomial p of order n.
if nargin<2, d=1; end, if nargin<3, n=length(p)-1; end
if d>0, p=[n:-1:1].*p(1:n); if d>1, p=PolyDiff(p,d-1,n-1); end, end
end % function PolyDiff
```
View Test

```
function [b,a]=Swap(a,b)
% A curiously simple (empty!) function that swaps the contents of a and b.
end % function Swap
```
View Test

Algorithm A.2: (continued) Some more simple example functions.

View Test

```
function [b,a]=RationalSimplify(b,a)
% Simplify a rational function b(s)/a(s) [or b(z)/a(z)] by dividing out common factors,

f=1; while f; x=roots(a); y=roots(b); f=0; for i=1:length(x); for j=1:length(y);
  if abs(x(i)-y(j))<1e-6, f=1; a=PolyDiv(a,[1 -x(i)]); b=PolyDiv(b,[1 -x(i)]); break, end
end, if f, break, end, end, end
a=a(find(abs(a)>1e-10,1):end); b=b(find(abs(b)>1e-10,1):end); b=b/a(1); a=a/a(1);
end % function RationalSimplify
```

The function `SuDokuSolve` solves the well-known SuDoku problem. Note that Algorithm A.5 defines the auxiliary functions `PlaySuDoku`, `PrintSuDoku`, and `RecursiveSuDoku` that may only be executed when the main function `SuDokuSolve` is running; if it is desired to make these codes accessible outside of the function `SuDokuSolve`, they should each be saved in their own `m`-file.

The overhead associated with function calls

There is extra overhead associated with function calls, because they often **allocate** (that is, assign) new memory locations for variables passed in when called, then **deallocate** (that is, release) this memory upon exit. This behavior is referred to as **passing by value**. For small variables, the overhead associated with passing by value is minimal. For large arrays, however such overhead can be substantial. To avoid this overhead, one may either use globally-defined arrays, as done in Algorithms 11.8-11.10, or **pass by reference**, which means to pass a **pointer** to the original memory location of the array rather than copy the values in the array into a new memory location. This is the default for array passing in Fortran and is common in C. Matlab's default is essentially a pass-by-value approach, with memory allocation deferred to the first time the array is modified within the subprogram. However, a pass-by-reference mode is initiated in Matlab whenever a function calls another function with an identical argument in both the input and output lists (in both the calling function and the called function); this can lead to significant improvements in execution speed for large problems.

When passing several parameters back and forth between functions, it is often convenient to group these parameters as derived data types, as illustrated, for example, in Algorithm 10.3.

Plotting

Both 2D and 3D plots are easy to generate in Matlab and Octave, as shown below:

A sample 2D plot

```
x=[0:.1:10];
y1=sin(x);
y2=cos(x);
plot(x,y1,'-',x,y2,'x')
```

A sample 3D plot

```
[x,y] = meshgrid(-8:.5:8,-8:.5:8);
R = sqrt(x.^2 + y.^2) + eps;
Z = sin(R)./R;
mesh(x,y,Z)
```

The code segments listed above produce the figures shown in Figure A.1. Note that axis rescaling and labelling can be controlled with `loglog`, `semilogx`, `semilogy`, `title`, `xlabel`, `ylabel`, etc. (see the help pages on these and related commands for more information).

The plotting commands illustrated above produce plots in figure windows. Once your code is working correctly and the plot is as you like it, you will usually want to save it, so you can email it, make printouts of it, and/or include it in a report or paper you are writing. For such reports and papers, LaTeX invariably produces the best results, though you might find that **what-you-see-is-what-you-get** (**wysiwyg**) word processors such

Algorithm A.3: An m-file, written as a script, illustrating the use of Matlab's built-in functions eig and inv.

View

```
% script <a href="matlab:SampleMatlabBuiltInFns">SampleMatlabBuiltInFns</a>
% This sample script finds the eigenvalues and eigenvectors of a few random matrices
% constructed with special structure.  This sample script uses Matlab's built-in commands
% <a href="matlab:doc inv">inv</a> and <a href="matlab:doc eig">eig</a> (lower case).
% NR develops, from first principles, alternatives to matlab's built-in commands, such as
% <a href="matlab:help Inv">Inv</a> and <a href="matlab:help Eig">Eig</a> (mixed case).
clear, R = randn(4), eigenvalues = eig(R)
disp('As R has random real entries, it may have real or complex eigenvalues.')
disp('Press return to continue'), pause, R_plus_Rprime = R+R',  eigenvalues = eig(R+R')
disp('Notice that R+R'' is symmetric, with real eigenvalues.')
disp('Press return to continue'), pause, R_minus_Rprime = R-R', eigenvalues = eig(R-R')
disp('The matrix R-R'' is skew-symmetric, with pure imaginary eigenvalues.')
disp('Press return to continue'), pause, Rprime_times_R = R'*R, [S,Lambda] = eig(R'*R)
disp('This matrix R''*R is symmetric, with real POSITIVE eigenvalues.')
disp('Press return to continue'), pause, R_times_Rprime = R*R', [S,Lambda] = eig(R*R')
disp('R*R'' has the same eigenvalues as R''*R, but different eigenvectors.')
disp('Press return to continue'), pause, S = rand(4);  Lambda = diag([1 2 3 4]);
A = S * Lambda * inv(S),  eigenvalues_of_A = eig(A)
disp('You can also create a matrix with desired eigenvalues (say, 1, 2, 3, 4)')
disp('from A=S Lambda inv(S) for any invertible S.'), disp(' ')
% end script SampleMatlabBuiltInFns
```

Algorithm A.4: A script to extend the path of Matlab/Octave to include the Numerical Renaissance Codebase.

View

```
% script <a href="matlab:NRCpathsetup">NRCpathsetup</a>
% Initialize the path environment for using the Numerical Renaissance Codebase.
% Tip: set up a symbolic link in a convenient place to make it easy to call this script
% when firing up matlab or octave.  This can be done, e.g., in Mac OS X as follows:
%   ln -s /usr/local/lib/NRC/NRchapAA/NRCpathsetup.m ~/Documents/MATLAB/startup.m
% Be sure to modify "base" appropriately below if the NRC library is not in /usr/local/lib
base='/usr/local/lib/NRC/'; format compact, clc, close all, cd ~
addpath(strcat(base,'NRchap01'),strcat(base,'NRchap02'),strcat(base,'NRchap03'), ...
        strcat(base,'NRchap04'),strcat(base,'NRchap05'),strcat(base,'NRchap06'), ...
        strcat(base,'NRchap07'),strcat(base,'NRchap08'),strcat(base,'NRchap09'), ...
        strcat(base,'NRchap10'),strcat(base,'NRchap11'),strcat(base,'NRchap12'), ...
        strcat(base,'NRchap13'),strcat(base,'NRchap14'),strcat(base,'NRchap15'), ...
        strcat(base,'NRchap16'),strcat(base,'NRchap17'),strcat(base,'NRchap18'), ...
        strcat(base,'NRchap19'),strcat(base,'NRchap20'),strcat(base,'NRchap21'), ...
        strcat(base,'NRchap22'),strcat(base,'NRchapAA'),strcat(base,'NRchapAB'), ...
        strcat(base,'NRextra' ),strcat(base,'NRextra/exportfig'),base)
disp([' Path set for using Numerical Renaissance Codebase; ' ...
      'type <a href="matlab:help NRC">help NRC</a> to get started.' char(10)])
% end script NRCpathsetup
```

as Microsoft Word are, initially, somewhat easier to use[9]. The recommended format to save your figures for such purposes is **encapsulated postscript** (typically denoted with a **.eps** suffix). Encapsulated postscript is a robust, platform-independent standard for graphics files that saves lines as vectors (lines) rather than as bitmaps (pixels), and therefore looks sharp when printed. All major typesetting programs, including LaTeX and Microsoft Word, can import .eps files.

To produce a color .eps file, type the following command after executing the plotting commands:

```
print -depsc foo.eps
```

[9] That is, until you begin to care about how well the equations are typeset, at which point your best option (by far, in the opinion of the author) is to abandon Microsoft Word and switch to LaTeX...

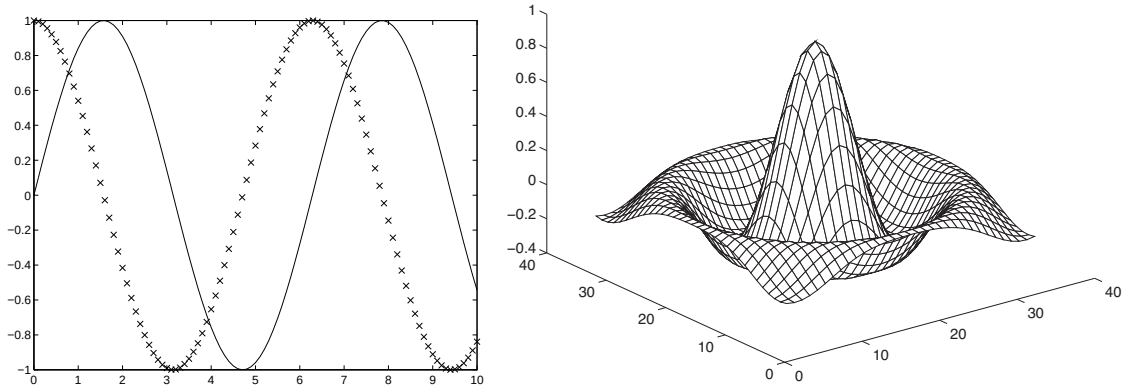


Figure A.1: Sample 2D and 3D plots.

Once you have created an .eps file, you may view it with a free piece of software called **Ghostview** (**gv**). **Adobe Illustrator** is a very useful commercial software package that may be used to further edit .eps files, which is often necessary when preparing scientific papers.

In Matlab, the contents of a figure window may also be saved with the command `saveas(1,'foo.fig')`, and later reopened and edited further with the command `open foo.fig`; this command does not exist in Octave. This command is not entirely recommended, however, because `.fig` files are often not portable from one platform to another. Instead, to send a figure via email, it is recommended that you create a `.eps` file (as described above) and send that. If you want the option to edit the figure later in Matlab or Octave, your flexibility is maximized if you save in an `m`-file the list of commands that created the figure (or, alternatively, save the session, as discussed previously) and recreate the figure from scratch later.

Printouts of the text appearing in the Matlab or Octave command window after a code is run is best achieved simply by copying this text and pasting it in to the editor of your choosing, then printing from there.

Advanced prepackaged numerical routines

Matlab and Octave have many many advanced prepackaged numerical routines built in. For example, a few prepackaged linear algebra commands which you might find useful (see the respective help pages for details) include: `lu`, `inv`, `hess`, `qr`, `orth`, `schur`, `eig`, `jordan`, `svd`, `chol`, `trace`, `norm`, `cond`, `pinv`, etc. These routines will certainly be useful for you as you learn how they work and why they are useful. However, the purpose of this text is *not* simply to catalog how to use these prepackaged routines (there are plenty of other texts that accomplish that), but rather to flush out, for some of the most important of these routines, *how* they actually work, and *why* they are useful. With this knowledge, the reader will be able to select and use these routines with much greater understanding and forethought. We thus, as mentioned previously, explicitly and intentionally avoid using *all* such advanced prepackaged routines in this text, instead learning to write these routines from scratch ourselves.

Exercises & References

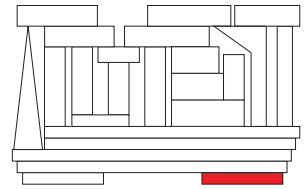
The only reference required to get up to speed in both Matlab and Octave is the online help, which is quite extensive. Programming is mostly an exercise in logical organization and is best learnt by example; by following through the many examples and exercises laid out in this text, you will quickly become proficient at writing clear, effective, efficient, and reusable numerical codes.

Algorithm A.5: A more complicated function that solves the SuDoku problem.

View
Test

```
function SuDokuSolve(F)
% An involved recursive code for solving SuDoku problems, emulating how a human plays.
% First, split SuDoku 9x9 array into a 3x3x3x3 array, which is easier for analysis
for i=1:3;for j=1:3;for k=1:3;for l=1:3; A(i,j,k,l)=F(i+(k-1)*3,j+(l-1)*3); end;end;end;end
PrintSuDoku(A), [A,B,flag]=PlaySuDoku(A);
PrintSuDoku(A), if flag==0; [A]=RecursiveSuDoku(A,B,flag); PrintSuDoku(A); end
end % function SuDokuSolve
%%%
```

```
%%%
```



Appendix B

Assorted mathematical foundations

B.1 Complex arithmetic

Complex arithmetic is based on the mathematical construct $\mathrm{i} \triangleq \sqrt{-1}$. Note that about half of the scientific literature refers to this construct as i, whereas the other half calls it j; this text calls it i. A **complex number** $z = a + b\,\mathrm{i}$ is a number with both a **real part**, $a = \Re(z)$, and an **imaginary part**, $b = \Im(z)$. If $b = 0$, z is said to be **real**; if $a = 0$, z is said to be **imaginary** or **pure imaginary**. Complex numbers may also be written in **polar form** $z = Re^{\mathrm{i}\theta}$, where $R = |z| = \sqrt{a^2 + b^2} \geq 0$ is referred to as the **magnitude** or **modulus** of z and[1] $\theta = \measuredangle z = \mathrm{atan2}(b,a) \in (-\pi, \pi]$ is referred to as the **phase** of z. Note that, for any integer j, $e^{\mathrm{i}\theta} = e^{\mathrm{i}(\theta + 2\pi j)} = \cos(\theta) + \mathrm{i}\sin(\theta)$; thus, $e^{\mathrm{i}\pi} + 1 = 0$ (this elegant equation is known as **Euler's identity**). Complex numbers are best understood in the **complex plane**, as illustrated in Figure B.1a. Note that complex numbers are added, subtracted, multiplied, and divided as if i were a normal algebraic variable, then simplified by leveraging the fact that $\mathrm{i}^2 = -1$. For example, $(a + b\,\mathrm{i})(c + d\,\mathrm{i}) = (ac - bd) + (ad + bc)\mathrm{i}$.

The ***n*'th roots** of any complex number z (written as $z = Re^{i\theta}$ for real R and θ) are easily found analytically:

$$\lambda^n = z \quad \Rightarrow \quad \lambda_j = R^{1/n} e^{\mathrm{i}(\theta + 2\pi j)/n} \quad \text{for } j \in [0, \ldots, n-1].$$

This formula is illustrated graphically by example in Figure B.1b. The **principal *n*'th root** of a real number z, denoted $\sqrt[n]{z}$, is defined here as the (positive) real root with $j = 0$ in the above formula if $z > 0$, and as the (negative) real root with $j = \frac{n-1}{2}$ in the above formula if both $z < 0$ and n is odd.

Note that, if $z_{k+1} = \sigma z_k$, then $z_k = \sigma^k z_0$ and thus $|z| \xrightarrow[k\to\infty]{} \infty$ if $|\sigma| > 1$, whereas $z \xrightarrow[k\to\infty]{} 0$ if $|\sigma| < 1$. If $dz(t)/dt = \lambda z(t)$, then $z(t) = e^{\lambda t} z(0)$ and thus $|z| \xrightarrow[t\to\infty]{} \infty$ if $\Re(\lambda) > 0$, whereas $z \xrightarrow[t\to\infty]{} 0$ if $\Re(\lambda) < 0$.

[1] Noting that $\mathrm{atan}\, x \in [0, \pi/2)$ if $x \geq 0$, the following definition is useful to remove ambiguity (see Figure B.1a):

$$\mathrm{atan2}(b,a) \triangleq \begin{cases} \mathrm{atan}\,|b/a| \cdot \mathrm{sgn}\, b & \text{if } b \neq 0,\ a > 0, \\ \pi/2 \cdot \mathrm{sgn}\, b & \text{if } b \neq 0,\ a = 0, \\ (\pi - \mathrm{atan}\,|b/a|) \cdot \mathrm{sgn}\, b & \text{if } b \neq 0,\ a < 0, \\ 0 & \text{if } b = 0,\ a \geq 0, \\ \pi & \text{if } b = 0,\ a < 0, \end{cases} \quad \text{where} \quad \mathrm{sgn}\, b = \begin{cases} -1 & \text{if } b < 0, \\ 0 & \text{if } b = 0, \\ 1 & \text{if } b > 0; \end{cases} \tag{B.1}$$

note that $\mathrm{atan2}(b,a) \in (-\pi, \pi]$.

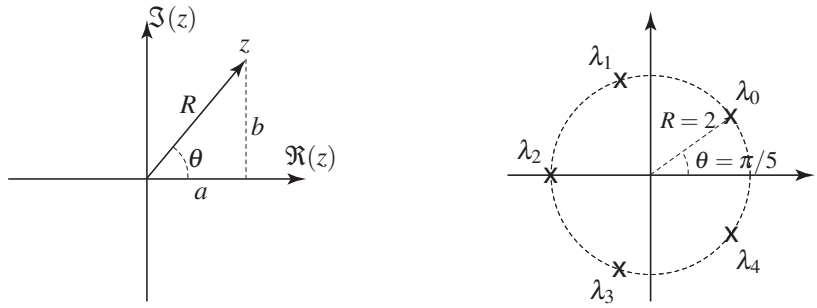


Figure B.1: (a) The relationship between the real part a and imaginary part b of the complex number $z = a + b\,\mathrm{i}$ and its polar components R and θ in the complex plane. (b) The fifth roots of $z = -32$: $\lambda_0 = 2e^{\mathrm{i}\pi/5}$, $\lambda_1 = 2e^{\mathrm{i}3\pi/5}$, $\lambda_2 = 2e^{\mathrm{i}\pi} = -2$, $\lambda_3 = 2e^{\mathrm{i}7\pi/5}$, $\lambda_4 = 2e^{\mathrm{i}9\pi/5}$; $\lambda_2 \triangleq \sqrt[5]{-32}$ is referred to as the principal root.

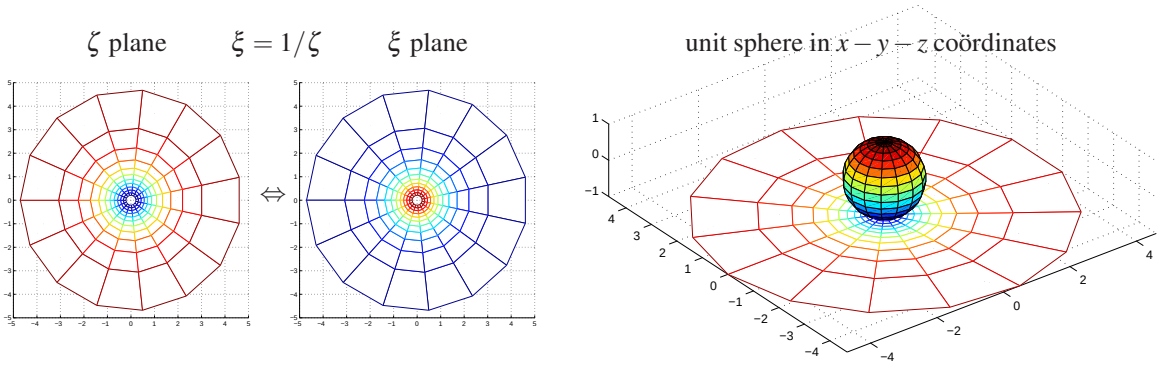


Figure B.2: (left) Two related complex planes which together introduce the concept of the **extended complex plane** (that is, the complex plane ζ together with a **point at infinity** which maps to the origin in the ξ plane). (right) The **Riemann sphere** obtained by mapping the complex plane ζ (illustrated by the "web" in the plane $z = -1$) onto the unit sphere; note that, with the origin of the ζ plane mapping to the south pole, the point at infinity maps to the north pole, and the "web" maps to corresponding longitude and latitude lines on the sphere. Alternatively, the Riemann sphere may be constructed by mapping from the complex plane ξ, with the origin of the ξ plane mapping to the north pole and its "point at infinity" mapping to the south pole.

The relationship between points far from the origin in the complex plane ζ may be understood by considering a related complex plane ξ, where all points except the origin in the two planes are related by the **transition maps** $\zeta = 1/\xi \;\Leftrightarrow \xi = 1/\zeta$. Writing $\zeta = Re^{i\theta}$ and $\xi = re^{i\phi}$, it follows that $r = 1/R$ and $\phi = -\theta$. Taking $R \to \infty$ for any θ in the ζ plane is equivalent to taking $r \to 0$ for $\phi = -\theta$ in the ξ plane. This introduces the notion of the **extended complex plane**: that is, the complex plane ζ together with the **point at infinity** (referred to in the singular), identified by the point in the ξ plane (i.e., the origin) to which ζ maps as $R \to \infty$ (a point which one might humorously identify as the location of the **restaurant at the end of the universe**).

A geometric interpretation of the extended complex plane (Figure B.2) is given by mapping the ζ and ξ planes to the unit sphere $x^2 + y^2 + z^2 = 1$ via the **stereographic projections** $\zeta = (x + \mathrm{i}y)/(1 - z)$, $\xi = (x - \mathrm{i}y)/(1 + z)$; approaching the origin in ζ (i.e., approaching the south pole of the sphere by taking $x \to 0$, $y \to 0$, and $z \to -1$) corresponds to moving toward infinity in the ξ plane, whereas approaching the origin in ξ (i.e., approaching the north pole of the sphere) corresponds to moving toward infinity in the ζ plane.

B.1.1 Counting zeros minus poles inside a contour: Cauchy's argument principle

Another result relating to complex arithmetic that we use in this book is Cauchy's argument principle, which is now stated and proved.

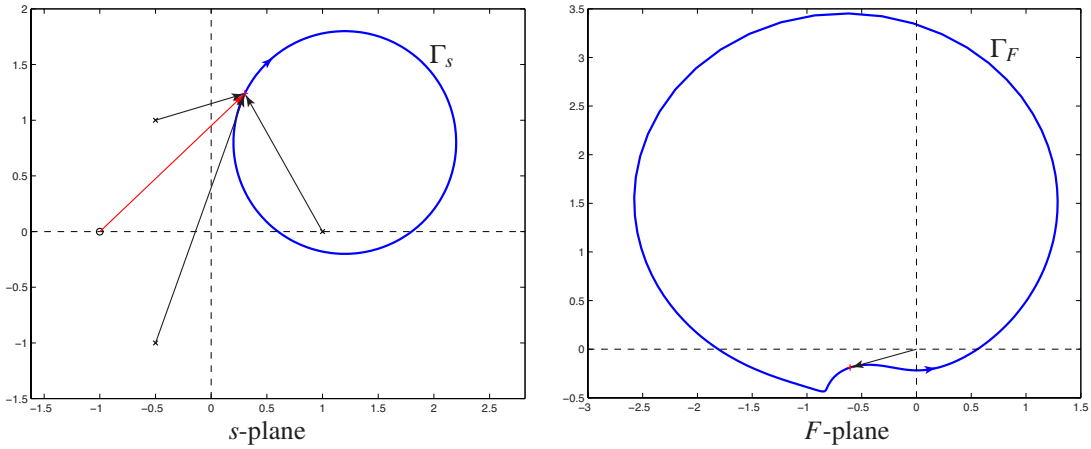


Figure B.3: Cauchy's argument principle (Fact B.1). Each point $\overline{F}$ on the contour Γ_F in the F-plane is found by applying the transform $F(s)$ to the corresponding point $\overline{s}$ on the contour Γ_s in the s-plane.

Fact B.1 (Cauchy's argument principle) *If a clockwise, closed contour Γ_s in the s-plane encircles Z zeros and P poles of a rational function $F(s)$, then the corresponding contour Γ_F in the F-plane [related to the s-plane by the mapping $F(s)$] makes $N = Z - P$ clockwise encirclements of the origin. Defining $L = F - 1$, the corresponding contour Γ_L in the L-plane makes $N = Z - P$ clockwise encirclements of the point $L = -1$.*

Proof: Consider first the change in the phase of $F(\overline{s})$ as $\overline{s}$ traverses the contour Γ_s (in the s-plane) one full circuit in the clockwise direction. Writing $F(s)$ [a **conformal mapping** from the s-plane to the F-plane] as

$$F(s) = K\frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)},$$

the phase of $F(\overline{s})$ at any given point $\overline{s}$ on the contour Γ_s is given by the sum of the phases of the vectors[2] from the m zeros to the point $\overline{s}$ minus the sum of the phases from the n poles to the point $\overline{s}$:

$$\angle F(\overline{s}) = [\angle(\overline{s}-z_1) + \angle(\overline{s}-z_2) + \ldots + \angle(\overline{s}-z_m)] - [\angle(\overline{s}-p_1) + \angle(\overline{s}-p_2) + \ldots + \angle(\overline{s}-p_n)].$$

As $\overline{s}$ traverses the contour Γ_s one full circuit in the clockwise direction, the contribution to $\angle F(\overline{s})$ from each zero and pole which are outside the closed contour Γ_s returns to its original value (before the circuit was begun). However, as $\overline{s}$ traverses the contour Γ_s one full circuit in the clockwise direction, the contribution to $\angle F(\overline{s})$ from each of the Z zeroes inside the closed contour Γ_s *decreases* by 360°, whereas the contribution to $\angle F(\overline{s})$ from each of the P poles inside the closed contour Γ_s *increases* by 360°. Thus, $\angle F(\overline{s})$ after the circuit is $(P-Z)\cdot 360°$ larger than $\angle F(\overline{s})$ before the circuit.

Now consider the change in the phase of $\overline{F} = F(\overline{s})$ as $\overline{F}$ traverses the contour Γ_F (in the F-plane) one full circuit as $\overline{s}$ traverses Γ_s one full circuit in the clockwise direction. The phase of the point $\overline{F}$ (in the F-plane) is measured simply by the angle of the vector from the origin of the F-plane to the point $\overline{F}$. This angle decreases [resp., increases] by 360° for every clockwise [resp., counterclockwise] encirclement of the origin executed by the contour Γ_F. Thus, $\angle\overline{F}$ after the circuit is $-N\cdot 360°$ larger than $\angle\overline{F}$ before the circuit, where N is the number of clockwise encirclements of the origin executed by the contour Γ_F.

Thus, for the phase of $F(\overline{s})$ as $\overline{s}$ traverses the clockwise, closed contour Γ_s in the s-plane to change by the same amount as does the phase of $\overline{F}$ as $\overline{F}$ traverses the closed contour Γ_F in the F-plane, it follows that the contour Γ_F makes $N = Z - P$ clockwise encirclements of the origin in the F-plane, which corresponds immediately to $N = Z - P$ clockwise encirclements of the point $L(s) = -1$ in the plane $L(s) = F(s) - 1$. □

[2]The phase of each vector is measured counterclockwise from horizontal, and is assumed to vary smoothly as the contour is traversed (that is, no `mod` 360° command is used to keep the angles in a prespecified range).

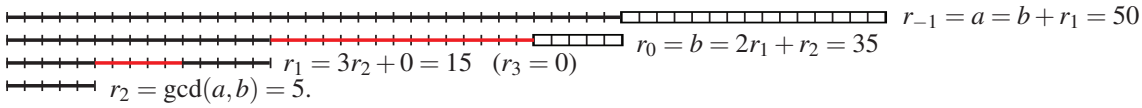


Figure B.4: Euclid's algorithm: Take $r_{-1} = a$ and $r_0 = b < a$. For $k \geq 1$, subtract the largest integer multiple of r_{k-1} from r_{k-2} that leaves a remainder $r_k \geq 0$, until some $k = n$ for which $r_n = 0$; then, $\gcd(a,b) = r_{n-1}$.

Algorithm B.1: Solution of the polynomial Diophantine equation $a(s)x(s) + b(s)y(s) = c(s)$ for $\{x(s), y(s)\}$.

View
Test

```
function [x,y,r,s] = Diophantine(a,b,c)
% Solve the polynomial Diophantine eqn a*x+b*y=c via the Extended Euclidean algorithm
% for coprime {a,b}.  The solution {x,y} returned is the solution with the lowest order
% for y; the general solution is given by {x+r*t,y+s*t} for any polynomial t.
n=max(2,abs(length(a)-length(b))+1); rm=a; r=b; for i=1:n+2
  r=r(find(r,1):end); [quo,rem]=PolyDiv(rm,r);  % Reduce (rm,r) to their GCD via Euclid's
  q(i,n+1-length(quo):n)=quo; rm=r; r=rem;      % algorithm, saving the quotients quo.
if norm(r,inf)<1e-13, g=rm; break, end, end
r=-PolyDiv(b,g); s=PolyDiv(a,g); y=PolyDiv(c,g);
x=0;  for j=i-1:-1:1                            % Using q as computed above, compute {x,y}
  t=x; x=y; y=PolyAdd(t,-PolyConv(q(j,:),y));    % via the Extended Euclidean algorithm
end, y=y(find(y,1):end); t=-PolyDiv(y,s);                % Find the solution {x,y} that
x=PolyAdd(x,PolyConv(r,t)); x=x(find(abs(x)>1e-8,1):end); % minimizes the order of y; this
y=PolyAdd(y,PolyConv(s,t)); y=y(find(abs(y)>1e-8,1):end); % is the most useful in practice
end % function Diophantine
```

B.2 Euclid's algorithm and the Diophantine equation

The **greatest common divisor** [**GCD**, a.k.a. **greatest common factor**] of two positive integers[3] a and b, denoted $\gcd(a,b)$, is the largest positive integer g such that a/g and b/g are both integers. The GCD may be computed using the **Euclidean algorithm** (a.k.a. **Euclid's algorithm**): define $r_{-1} = a$ and $r_0 = b < a$, and perform **integer division** of r_{k-2}/r_{k-1} to determine the quotient and remainder $\{q_k, r_k\}$ [i.e., find the largest positive integer q_k and associated positive integer r_k that solve $r_{k-2} = q_k r_{k-1} + r_k$] for $k = 1, 2, \ldots$ until some $k = n$ for which $r_n = 0$; it follows that $r_{n-1} = g$; graphical interpretation is given in Figure B.4. Note also that:

Fact B.2 (Bézout's identity) *If $g = gcd(a,b)$, then $g = ax + by$ for two integers $\{x,y\}$.*

A constructive proof of Bézout's identity is given by considering the **extended Euclidean algorithm**, which solves for $\{s,t\}$ in Bézout's identity simply by running through the quotients q_k computed in Euclid's algorithm in reverse order: define $z_{n+1} = 0$, $z_n = 1$, and compute $z_k = z_{k+2} - q_k z_{k+1}$ for $k = n-1, n-2, \ldots, 1$. It follows that $x = z_2$ and $y = z_1$.

The extended Euclidean algorithm may be verified by writing out the Euclidean algorithm with $g = r_{n-1}$:

$$a = q_1 b + r_1, \quad b = q_2 r_1 + r_2, \quad r_1 = q_3 r_2 + r_3 \quad \rightarrow \quad r_{n-4} = q_{n-2} r_{n-3} + r_{n-2}, \quad r_{n-3} = q_{n-1} r_{n-2} + g,$$

(note that $r_{n-2} = q_n g + 0$), then working backwards through this algorithm, solving each step for its last term:

$$g = r_{n-3} - q_{n-1} r_{n-2}, \quad r_{n-2} = r_{n-4} - q_{n-2} r_{n-3} \quad \rightarrow \quad r_3 = r_1 - q_3 r_2, \quad r_2 = b - q_2 r_1, \quad r_1 = a - q_1 b.$$

Starting with the first expression above, substituting in the second to eliminate r_{n-2}, substituting in the next to eliminate r_{n-3}, etc., ultimately leads to $g = sa + tb$, where s and t are the appropriate combinations of the integers q_i appearing in Euclid's algorithm.

The extended Euclidean algorithm applied to polynomials, as implemented in Algorithm B.1, solves the polynomial Diophantine equation $a(s)x(s) + b(s)y(s) = c(s)$.

[3] Note that $\gcd(a,b,c) = \gcd(a, \gcd(b,c)) = \gcd(b, \gcd(c,a)) = \gcd(c, \gcd(a,b))$.

B.3 Polynomials

B.3.1 Roots of a quadratic polynomial

Complex arithmetic facilitates the determination of the roots of a quadratic polynomial analytically. That is,

$$\lambda^2 + Q\lambda + R = 0 \quad \Rightarrow \quad \lambda_\pm = \frac{-Q \pm \sqrt{-D}}{2} \quad \text{where} \quad D = 4R - Q^2,$$

as may be verified by substitution. The quantity D, referred to as the **discriminant**, characterizes the nature of the solution. For real Q and R,

- if $D > 0$, there are two complex-conjugate roots,
- if $D = 0$, there are two identical, real roots, and
- if $D < 0$, there are two distinct, real roots.

B.3.2 Roots of a cubic polynomial

Similarly, we may also determine the roots of a cubic polynomial analytically. Starting with the **normal form**

$$\lambda^3 + P\lambda^2 + Q\lambda + R = 0,$$

we first define $\lambda = x - P/3$ and substitute, giving

$$x^3 + qx + r = 0,$$

where $q = Q - P^2/3$ and $r = R + 2P^3/27 - PQ/3$. This **reduced form** is solved first for its roots x_j for $j \in [1,2,3]$, after which the roots $\lambda_j = x_j - P/3$ solving the corresponding equation in normal form may be determined immediately. The **discriminant** in this case, $d = r^2/4 + q^3/27$, again characterizes the nature of the solution. For real q and r,

- if $d > 0$, there is one real and two complex-conjugate roots,
- if $d = 0$, there are three real roots, at least two of which are identical, and
- if $d < 0$, there are three distinct, real roots.

As may be verified by substitution, in the first two of these cases (with $d \geq 0$), the roots are given by **Cardano's formula** (note that the imaginary part of the complex roots goes to zero as $d \to 0$)

$$x_1 = u_+ + u_-, \quad x_{2,3} = -\frac{u_+ + u_-}{2} \pm \mathrm{i}\sqrt{3} \cdot \frac{u_+ - u_-}{2} \quad \text{where} \quad u_\pm = \sqrt[3]{-r/2 \pm \sqrt{d}}.$$

In the third case (with $d < 0$), the so-called **casus irreducibilis**, the three distinct, real roots are given by

$$x_j = v \cos(\theta/3 + 2\pi j/3) \quad \text{where} \quad v = 2\sqrt{-q/3}, \quad \theta = \operatorname{acos}\left(\frac{-r/2}{\sqrt{-q^3/27}}\right).$$

B.3.3 Roots of higher-order polynomials

A closed-form solution of the roots of a quartic polynomial, due originally to Ferrari, also exists, though it is a bit complicated. Unfortunately, as established in Fact 4.9, a general closed-form solution for quintic and higher-order polynomials does not exist, and must instead be found iteratively [for example, by finding the eigenvalues of a corresponding matrix in companion form, as illustrated in Algorithm 4.3.] In fact, as discussed in §4.4.1, the lack of a closed-form expression for solving the roots of the characteristic polynomial of a matrix is the reason that the eigenvalue problem must, in general, be solved iteratively.

B.3.4 Factoring polynomials: the fundamental theorem of algebra

Proofs of the fundamental theorem of algebra have a long and rich history. The first complete proof is often attributed to Gauss, though this is a matter of some debate (for a historical summary, see Fine & Rosenberger 1997). The proof provided below, summarized in Santos (2007), is particularly elementary.

Fact B.3 (The fundamental theorem of algebra) *Any n'th-order monic polynomial $p(s)$ with complex coefficients c_k has exactly n complex roots (including multiplicities), and thus may be written*

$$p(s) = s^n + c_{n-1}s^{n-1} + \ldots + c_1 s + c_0 = (s-p_1)(s-p_2)\cdots(s-p_{n-1})(s-p_n).$$

Proof: Let $P_m(s) = s^m + a_{m,m-1}s^{m-1} + \ldots + a_{m,1}s + a_{m,0}$, where $m \geq 2$, be an m'th-order monic polynomial with m complex coefficients $a_{m,k}$ for $k = 0,\ldots,m-1$. We first prove that $P_m(s)$ has at least one root, which we call p_m, and thus may be written $P_m(s) = (s-p_m)(s^{m-1} + a_{m-1,m-1}s^{m-2} + \ldots + a_{m-1,1}s + a_{m-1,0}) = (s-s_m)P_{m-1}(s)$. Starting with the n'th-order monic polynomial $P_n(s) = s^n + a_{n,n-1}s^{n-1} + \ldots + a_{n,1}s + a_{n,0}$, repeated application of this fact $n-1$ times, on the successively lower-order monic polynomials $P_{n-1}(s)$, $P_{n-2}(s)$, etc., thus proves Fact B.3.

To prove that $P_m(s) = s^m + a_{m,m-1}s^{m-1} + \ldots + a_{m,1}s + a_{m,0}$ has a root p_m, we simply assume that it doesn't (that is, that $P_m(s) \neq 0$ for all $s \in \mathbb{C}$) and show that this assumption leads to a contradiction, thereby establishing that $P_m(s)$ in fact has at least one complex root p_m. To accomplish this, for real r and ϕ, define

$$f_m(r,\phi) = \frac{1}{P_m(s)}\Big|_{s=re^{i\phi}} = \frac{1}{r^m e^{im\phi} + a_{m,m-1}r^{m-1}e^{i(m-1)\phi} + \ldots + a_{m,1}re^{i\phi} + a_{m,0}}. \tag{B.2}$$

Note that the denominator is continuously differentiable in both r and ϕ, and by assumption is never zero. Thus, $f_m(r,\phi)$ is also continuously differentiable in r and ϕ. Now define

$$F_m(r) = \int_0^{2\pi} f_m(r,\phi)\,d\phi. \tag{B.3}$$

By **Leibniz's integration rule**[4], $F_m(r)$ is continuously differentiable in r, and

$$\frac{dF_m(r)}{dr} = \int_0^{2\pi} \frac{\partial f_m(r,\phi)}{\partial r}\,d\phi.$$

Noting that

$$\frac{\partial f_m(r,\phi)}{\partial r} = -\frac{mr^{m-1}e^{im\phi} + (m-1)a_{m,m-1}r^{m-2}e^{i(m-1)\phi} + \ldots + a_{m,1}e^{i\phi}}{(r^m e^{im\phi} + a_{m,m-1}r^{m-1}e^{i(m-1)\phi} + \ldots + a_{m,1}re^{i\phi} + a_{m,0})^2},$$

$$\frac{\partial f_m(r,\phi)}{\partial \phi} = -\frac{imr^m e^{im\phi} + i(m-1)a_{m,m-1}r^{m-1}e^{i(m-1)\phi} + \ldots + ia_{m,1}re^{i\phi}}{(r^m e^{im\phi} + a_{m,m-1}r^{m-1}e^{i(m-1)\phi} + \ldots + a_{m,1}re^{i\phi} + a_{m,0})^2} = ir\frac{\partial f_m(r,\phi)}{\partial r},$$

it follows that

$$\frac{dF_m(r)}{dr} = \frac{1}{ir}\int_0^{2\pi} \frac{\partial f_m(r,\phi)}{\partial \phi}\,d\phi = \frac{1}{ir}f_m(r,\phi)\Big|_{\phi=0}^{\phi=2\pi} = \frac{1}{ir}\left[\frac{1}{P_m(re^{i2\pi})} - \frac{1}{P_m(re^{i0})}\right] = 0.$$

Thus, $F_m(r)$ is constant, and thus $F_m(r) = F_m(0) = \int_0^{2\pi} f(0,\phi)\,d\phi = 2\pi/P_m(0)$; that is, $F_m(r)$ is a nonzero constant. However, (B.2)-(B.3) imply that $F_m(r) \to 0$ as $r \to \infty$. This is a contradiction, which means that the assumption that $P_m(s)$ does not have a root p_m is false. □

[4] **Leibniz's integration rule** states simply that, if $f(r,s)$ and $\partial f(r,s)/\partial r$ are continuous, then $F(r) = \int_a^b f(r,s)\,ds$ is differentiable with respect to r, with $dF(r)/dr = \int_a^b \frac{\partial f(r,s)}{\partial r}\,ds$; that is, that the derivative with respect to r may be pulled outside the integral.

B.3.5 Continuity of a polynomial's roots as a function of its coefficients

Fact B.3 established that any polynomial can be factored via its roots. We now establish that these roots vary continuously as the coefficients of the polynomial are varied. We begin with a significant preliminary result.

Fact B.4 (Rouchés theorem) *Let $p(s)$ and $d(s)$ be complex polynomial functions*[5] *on the simply-connected domain Ω in the complex plane s. Let Γ be a* **Jordan curve** *(that is, a closed curve that has only a single multiple point where it closes upon itself) within Ω. Define P as the number of zeros of $p(s)$ on the interior of Γ, and Z as the number of zeros of $z(s) = p(s) + d(s)$ on the interior of Γ. If $|d(s)| < |p(s)|$ for all s on Γ, then $P = Z$.*

Proof: Define $F(s) = z(s)/p(s)$, and note that $|F(s) - 1| = |d(s)/p(s)| < 1$, and thus the real part of $F(s)$ is positive, for all s on Γ. By the same logic as in the proof of Cauchy's argument principle (Fact B.1), $\angle F(s)$ after a circuit over Γ is $(P - Z) \cdot 360°$ larger than $\angle F(s)$ before the circuit. As the real part of $F(s)$ is positive everywhere on Γ, $\angle F(s)$ does not change by more that $180°$ as s traverses Γ; thus, $P - Z = 0$. □

Fact B.5 (Continuity of polynomial roots) *Let $p(s)$ be an n'th-order polynomial which may be factored (Fact B.3) such that*

$$p(s) = s^n + a_{n-1}s^{n-1} + \ldots + a_0 = (s - p_1)^{q_1}(s - p_2)^{q_2} \cdots (s - p_P)^{q_P}. \tag{B.4a}$$

Let $\rho = \min_{i \neq j} |p_i - p_j|$. For any ε with $0 < \varepsilon < \rho/2$, there exists a $\delta > 0$ such that any n'th-order polynomial

$$z(s) = s^n + b_{n-1}s^{n-1} + \ldots + b_0 = (s - z_1)^{r_1}(s - z_2)^{r_2} \cdots (s - z_Z)^{r_Z} \tag{B.4b}$$

with $|b_j - a_j| < \delta$ for $j = 0, 1, \ldots, n-1$ has exactly q_i zeros in each disk $|s - p_i| < \varepsilon$ for $i = 1, 2, \ldots, P$. That is, each individual root moves in the complex plane s an amount less than ε, for any sufficiently small ε, if the coefficients of the polynomial a_i each changes by an amount less than a correspondingly small δ.

Proof (Henrici 1974): Note first that q_i is the multiplicity of the i'th root, p_i, of the polynomial $p(s)$, that r_i is the multiplicity of the i'th root, z_i, of the polynomial $z(s)$, and that $\sum_{i=1}^{P} q_i = \sum_{i=1}^{Z} r_i = n$. Note also that both $p(s)$ and $z(s)$ are taken above to be monic polynomials without loss of generality. Consider the contour Γ_i given by the circle in the s plane, around the root p_i, satisfying $|s - p_i| = \varepsilon$. Defining $d(s) = p(s) - z(s)$ and $c_j = a_j - b_j$, noting the assumption that $|c_j| = |a_j - b_j| < \delta$ for all j, we may write

$$d(s) = c_{n-1}s^{n-1} + c_{n-2}s^{n-2} + \ldots + c_0 \quad \Rightarrow \quad |d(s)| \leq \delta|s|^{n-1} + \delta|s|^{n-2} + \ldots + \delta;$$

it thus follows for all s on Γ_i that

$$|d(s)| \leq \delta\, \mu_i(\varepsilon) \quad \text{where} \quad \mu_i(\varepsilon) \triangleq \sum_{j=0}^{n-1} (|p_i| + \varepsilon)^j.$$

Further, for all s on Γ_i, it follows directly from the factored form of (B.4a) that

$$|p(s)| \geq \varepsilon^{q_i}(\rho - \varepsilon)^{n - q_i} \quad \text{for } i = 1, \ldots, P.$$

Thus, taking $\delta < \varepsilon^{q_i}(\rho - \varepsilon)^{n-q_i}/\mu_i(\varepsilon)$ for each $i = 1, \ldots, P$, it follows that $|d(s)| < |p(s)|$ for all s on each Γ_i, and thus Fact B.4 applies. That is, both $p(s)$ and $z(s)$ have exactly q_i roots inside the disk centered at p_i of radius ε, for each $i = 1, \ldots, P$, for any sufficiently small ε so long as δ is also made sufficiently small. □

[5] An **analytic function** is a function that is equal to its own Taylor series in some neighborhood of every point, as discussed further in §B.6. We mention analytic functions here only because Rouchés theorem (Fact B.4) may easily be extended from complex polynomial functions to complex analytic functions, though this extension is beyond the scope of the present discussion.

Algorithm B.2: The Routh Test (see §B.3.6) for computing the **inertia** $\{N_-(p), N_0(p), N_+(p)\}$ of $p(s)$.

View
Test

```
function inertia=PolyInertia(p)
% Find the number of roots of the polynomial p(s) that are in the LHP, on the imaginary
% axis, and in the RHP, referred to as the inertia of p(s). Algorithm due to Routh (1895).
i=find(abs(p)>1e-12,1); p=p(i:end); degree=length(p)-1;          % strip off leading zeros
inertia=[0 0 0]; flag=0; show('Routh',degree,p(1:2:end))
for n=degree:-1:1
  k=find(abs(p(2:2:n+1))>1e-14,1); show('Routh',n-1,p(2:2:end))  % Similar implementation
  if length(k)==0, flag=1;                                       % in Meinsma (SCL, 1995)
    if mod(n,2)==0, t='Even'; else, t='Odd'; end
    disp(['Case 3: ',t,' polynomial. Add its derivative.'])
    p(2:2:n+1)=p(1:2:n).*(n:-2:1); show('  NEW',n-1,p(2:2:end))
  elseif k>1
    if mod(k,2)==0, s=-1; t='Subtract'; else, s=1; t='Add'; end
    disp(['Case 2: p_{n-1}=0. ',t,' s^',num2str(2*(k-1)),' times row ',num2str(n-1),'.'])
    i=0:2:(n+1-2*k); p(i+2)=p(i+2)+s*p(i+2*k); show('  NEW',n-1,p(2:2:end))
  end
  eta=p(1)/p(2); if flag, inertia=inertia+[(eta<0) 0 (eta<0)];
                 else,    inertia=inertia+[(eta>0) 0 (eta<0)]; end
  p(3:2:n)=p(3:2:n)-eta*p(4:2:n+1); p=p(2:n+1);       % Update p, strip off leading element
end
inertia=inertia+[0 degree-sum(inertia) 0]; s='stable CT system';
if inertia(3)>0 s=['un',s]; elseif inertia(2)>0 s=['marginally ',s]; end, disp(s)
end % function PolyInertia
%%%
```

B.3.6 Location of a polynomial's roots with respect to the imaginary axis

Assume $p(s) = p_n s^n + p_{n-1} s^{n-1} + \ldots + p_1 s + p_0$ is a **real polynomial** (that is, a polynomials with real coefficients) of **degree** n (that is, $p_n \neq 0$); $p(s)$ is said to be **singular** if $p_{n-1} = 0$ and **nonsingular** otherwise. We will sometimes write $p(s) = [p(s)]_{even} + [p(s)]_{odd}$ where $[p(s)]_{even}$ & $[p(s)]_{odd}$ denote the terms of $p(s)$ with even & odd powers of s, respectively. The Routh test counts how many roots of $p(s)$ are in the LHP, on the imaginary axis, and in the RHP [denoted $\{N_-(p), N_0(p), N_+(p)\}$, respectively, and often referred to as the **inertia** of $p(s)$], without requiring the computation of the roots themselves, which can be computationally expensive. Following Meinsma (1995), we derive this test by first proving three preliminary lemmata.

Fact B.6 *Consider an n'th-degree, nonsingular, real polynomial* $p(s) = p_n s^n + p_{n-1} s^{n-1} + \ldots + p_1 s + p_0$. *Defining* $\eta^* = p_n / p_{n-1}$, $p(s)$ *has the same imaginary roots as the* $(n-1)$*'th-degree polynomial*

$$q(s) = p_{n-1} s^{n-1} + (p_{n-2} - \eta^* p_{n-3}) s^{n-2} + p_{n-3} s^{n-3} + (p_{n-4} - \eta^* p_{n-5}) s^{n-4} + p_{n-5} s^{n-5} + \ldots \tag{B.5}$$

Further, $p(s)$ *has the same inertia as* $q(s)$*, plus one additional root in the LHP if* $p_n/p_{n-1} > 0$*, and in the RHP if* $p_n/p_{n-1} < 0$.

Proof: Note that $p_{n-1} \neq 0$ because $p(s)$ is nonsingular. Define

$$q_\eta(s) = p(s) - \eta\, s\, (p_{n-1} s^{n-1} + p_{n-3} s^{n-3} + \ldots) = \begin{cases} ([p(s)]_{even} - \eta\, s\, [p(s)]_{odd}) + [p(s)]_{odd} & \text{even n,} \\ [p(s)]_{even} + ([p(s)]_{odd} - \eta\, s\, [p(s)]_{even}) & \text{odd n.} \end{cases} \tag{B.6}$$

Then $q_\eta(\mathrm{i}\omega) = [q_\eta(\mathrm{i}\omega)]_{even} + [q_\eta(\mathrm{i}\omega)]_{odd} = 0$ iff $[q_\eta(\mathrm{i}\omega)]_{even} = 0$ and $[q_\eta(\mathrm{i}\omega)]_{odd} = 0$, as $[q_\eta(\mathrm{i}\omega)]_{even}$ is real, and $[q_\eta(\mathrm{i}\omega)]_{odd}$ is imaginary. It follows from the expressions for both even and odd n at right in (B.6) that, for all η, $q_\eta(\mathrm{i}\omega) = 0$ iff both $p_{even}(\mathrm{i}\omega) = 0$ and $p_{odd}(\mathrm{i}\omega) = 0$; thus, $q(\mathrm{i}\omega) = q_{\eta^*}(\mathrm{i}\omega) = 0$ iff $p(\mathrm{i}\omega) = 0$.

Thus, $p(s) = q_0(s)$ and $q(s) = q_{\eta^*}(s)$ have the same imaginary roots. Further, by Fact B.5, the roots of $q_\eta(s)$ vary smoothly as η is varied from 0 up to (or down to), but not including, η^*; that is, roots do not "jump" between the LHP and the RHP as η is varied over this range. As $\eta \to \eta^*$, $q_\eta(s)$ in (B.6) approaches the $(n-1)$'th-degree polynomial $q(s)$ in (B.5); to see what happens in this limit, we may write

$$\begin{aligned} q_\eta(s) &= (p_n - \eta\, p_{n-1}) s^n + p_{n-1} s^{n-1} + (p_{n-2} - \eta\, p_{n-3}) s^{n-2} + p_{n-3} s^{n-3} + (p_{n-4} - \eta\, p_{n-5}) s^{n-4} + \ldots \\ &= \left(\frac{p_n - \eta\, p_{n-1}}{p_{n-1}} s + 1 \right) r_\eta(s) \end{aligned} \tag{B.7}$$

where $r_\eta(s) \to q(s)$ [see (B.5)] as $\eta \to \eta^*$. It is seen in (B.7) that exactly one root of $q_\eta(s)$ approaches infinity as $\eta \to \eta^*$; this root "starts out" [for $\eta = 0$, that is, for $q_0(s) = p(s)$] in the LHP if $p_n/p_{n-1} > 0$, and in the RHP if $p_n/p_{n-1} < 0$. Of the remaining roots of the n'th-degree polynomial $p(s)$, the number in the LHS, on the imaginary axis, and in the RHS are precisely the same as for the $(n-1)$'th-degree polynomial $q(s)$. □

Fact B.7 *Let* $p(s)$ *be an n'th-degree, singular, real polynomial that is neither odd nor even, and let* $a(s)$ *be a real even polynomial with* $a(i\omega) > 0$. *Define* $p_a(s)$ *and* $p_b(s)$ *such that*

$$p(s) = p_a(s) + p_b(s) \quad \textit{where} \quad \begin{cases} p_a(s) = [p(s)]_{even}, \quad p_b(s) = [p(s)]_{odd} & \textit{even } n, \\ p_a(s) = [p(s)]_{odd}, \quad p_b(s) = [p(s)]_{even} & \textit{odd } n. \end{cases} \tag{B.8}$$

If $a(s)$ *is chosen such that the degree of* $\{a(s)\, p_b(s)\}$ *is* $n-1$*, then the n'th-degree polynomials* $p(s)$ *and* $q(s) = p_a(s) + a(s)\, p_b(s)$ *have the same inertia, and* $q(s)$ *is nonsingular (i.e.,* $q_{n-1} \neq 0$*).*

Proof: If $p_a(s)$ is even (resp., odd), then $\{(1-\lambda)+\lambda\, a(s)\}\, p_b(s)$ is odd (resp., even). For $0 \le \lambda \le 1$, define the n'th-degree polynomial

$$q_\lambda(s) = p_a(s) + \{(1-\lambda)+\lambda\, a(s)\}\, p_b(s).$$

Noting that $a(\mathrm{i}\omega) > 0$, it follows as in the proof of Fact B.6 that $q_\lambda(\mathrm{i}\omega) = 0$ iff $p_a(\mathrm{i}\omega) = 0$ and $p_b(\mathrm{i}\omega) = 0$ [that is, iff $p(\mathrm{i}\omega) = 0$], independent of λ. Thus, $p(s)$ and $q_\lambda(s)$ have the same imaginary roots, independent of λ. Further, by Fact B.5, the n roots of $q_\lambda(s)$ vary smoothly as λ is varied; i.e., roots do not "jump" between the LHP and the RHP as λ is varied. Thus, $p(s) = q_0(s)$ and $q(s) = q_1(s)$ have the same inertia. □

Fact B.8 *Let $p(s)$ be a real n'th-degree odd or even polynomial. Defining $r(s) = p(s) + p'(s)$, it follows that $N_+(p) = N_-(p) = N_+(r)$, and $r(s)$ is nonsingular (i.e., $r_{n-1} \neq 0$).*

Proof: It follows from the fact that $p(s)$ is either odd or even that $r_{n-1} \neq 0$, and that $p(s)$ has the same number of RHP roots as LHP roots [that is, $N_+(p) = N_-(p)$]. Define $q_\varepsilon(s) = p(s) + \varepsilon p'(s)$; it follows from Fact B.7 [with $a(s) = \varepsilon$] that, for any $\varepsilon > 0$, $r(s)$ and $q_\varepsilon(s)$ have the same inertia. We thus focus on $q_\varepsilon(s)$ in the limit that $\varepsilon \to 0$. If $s = \mathrm{i}\omega$ is a root of multiplicity k of the polynomial $p(s) = q_0(s)$, then $s = \mathrm{i}\omega$ is a root of multiplicity $k-1$ of the polynomial $q_\varepsilon(s)$ for $\varepsilon > 0$, and exactly one root of $q_\varepsilon(s)$ moves (by Fact B.5, continuously) away from $s = \mathrm{i}\omega$ as ε is increased from zero. To quantify what direction this root moves as ε is increased from zero, denoting $p^{(k)} = d^k p(s)/ds^k$, we write the Taylor series expansion of $q_\varepsilon(s)$ around $s = \mathrm{i}\omega$:

$$\begin{aligned} q_\varepsilon(\mathrm{i}\omega+\delta) &= \frac{\delta^{k-1}}{(k-1)!}\left[q_\varepsilon^{(k-1)}(s)\right]_{s=\mathrm{i}\omega} + \frac{\delta^k}{k!}\left[q_\varepsilon^{(k)}(s)\right]_{s=\mathrm{i}\omega} + \ldots \\ &= \frac{\delta^{k-1}}{(k-1)!}\left[\cancelto{0}{p^{(k-1)}(\mathrm{i}\omega)} + \varepsilon p^{(k)}(\mathrm{i}\omega) + \ldots\right] + \frac{\delta^k}{k!}\left[p^{(k)}(\mathrm{i}\omega) + \varepsilon p^{(k+1)}(\mathrm{i}\omega) + \ldots\right] + \ldots \\ &= \frac{\delta^{k-1}}{k!}\, p^{(k)}(\mathrm{i}\omega)\left[\varepsilon\, k + \delta + \varepsilon\,\delta\,\frac{p^{(k+1)}(\mathrm{i}\omega)}{p^{(k)}(\mathrm{i}\omega)} + \ldots\right] \end{aligned} \tag{B.9}$$

where the "…" terms are higher order in ε, and thus negligible for small ε. By Fact B.5, δ is proportional to ε for small ε; the third term in brackets in (B.9) is thus also negligible compared to the first two terms. Solving for $q_\varepsilon(\mathrm{i}\omega+\delta) = 0$ thus shows that, in addition to the $k-1$ roots fixed at $\delta = 0$, the remaining root is given by $\delta \approx -k\varepsilon < 0$ for sufficiently small $\varepsilon > 0$ (that is, the remaining root moves into the LHP as ε is increased from zero). Since $p(s) = q_0(s)$ and $r(s) = q_1(s)$, it follows that $N_+(p) = N_+(r)$. □

Fact B.9 (The Routh Test) *The inertia $\{N_-(p), N_0(p), N_+(p)\}$ of an n'th-degree polynomial $p(s)$ may be found via application of the following three cases to polynomials successively smaller and smaller degree:*

Case 1: If $p(s)$ is nonsingular (that is, if $p_{n-1} \neq 0$), then define $q(s)$ as in (B.6). It follows that

$$\{N_-(p), N_0(p), N_+(p)\} = \begin{cases} \{N_-(q)+1, N_0(q), N_+(q)\} & \text{if } p_n/p_{n-1} > 0, \\ \{N_-(q), N_0(q), N_+(q)+1\} & \text{if } p_n/p_{n-1} < 0; \end{cases}$$

$\{N_-(q), N_0(q), N_+(q)\}$ may then be found by applying Fact B.9 to the $(n-1)$'th-degree polynomial $q(s)$.

Case 2: If $p(s)$ is singular (that is, if $p_{n-1} = 0$) but neither even nor odd, then define $p_a(s)$ and $p_b(s)$ as in (B.8). Since $p_{n-1} = 0$, $p_b(s)$ has degree $n-1-2k$ for some $k > 0$; define $a(s) = 1 + (-s^2)^k$. Defining $q(s) = p_a(s) + a(s)\, p_b(s)$, it follows that $\{N_-(p), N_0(p), N_+(p)\} = \{N_-(q), N_0(q), N_+(q)\}$, which may be found by applying Case 1 of Fact B.9 to the nonsingular n'th-degree polynomial $q(s)$.

Case 3: If $p(s)$ is either even or odd, then define $r(s) = p(s) + p'(s)$. It follows that $N_+(p) = N_-(p) = N_+(r)$, where $N_+(r)$ may be found by applying Case 1 of Fact B.9 to the nonsingular n'th-degree polynomial $r(s)$.

Proof: Case 1 follows immediately from Fact B.6. Case 2 follows from Fact B.7, noting that $a(\mathrm{i}\omega) > 0$ for real ω, and that the degree of $\{a(s)\, p_b(s)\}$ is $n-1$. Case 3 follows immediately from Fact B.8. □

B.3.7 Location of a polynomial's roots with respect to the unit circle

Assume $P_n(z) = p_n z^n + p_{n-1} z^{n-1} + \ldots + p_1 z + p_0$ is a real polynomial of degree n; $P_n(z)$ is said to be **regular** if $p_0 \neq 0$ (and, thus, $P_n(z)$ does not have any roots at the origin) and **irregular** otherwise. A polynomial $P_n(z)$ is said to be **symmetric** if $p_{n-j} = p_j$ for $j = 1, \ldots, n$, and **antisymmetric** if $p_{n-j} = -p_j$ for $j = 1, \ldots, n$. The Bistritz test counts how many roots of $P_n(z)$ are inside the unit circle, on the unit circle, and outside the unit circle [denoted $\{N_i(P), N_u(P), N_o(P)\}$, respectively, and which we will refer to as the **stationarity** of $P_n(z)$], without requiring the computation of the roots themselves, which can be computationally expensive. The Bistritz test is useful for determining the stability of discrete-time systems, much as the Routh test (see §B.3.6) is useful for determining the stability of continuous-time systems. Following loosely the related derivation in §B.3.6, we derive this test by first proving three preliminary lemmata.

Fact B.10 *Consider an n'th-degree real polynomial $P_n(z) = p_n z^n + p_{n-1} z^{n-1} + \ldots + p_1 z + p_0$ with $P_n(1) \neq 0$. Defining $P_n^r(z) \triangleq z^n P_n(1/z) = p_0 z^n + p_1 z^{n-1} + \ldots + p_{n-1} z + p_n$, $P_n(z)$ and $P_{n-1}(z)$ have the same roots on the unit circle as the symmetric n'th-degree and $(n-1)$'th-degree polynomials*

$$T_n(z) = P_n(z) + P_n^r(z) \quad \textit{and} \quad T_{n-1}(z) = \frac{P_n(z) - P_n^r(z)}{z-1}.$$

Proof: It follows immediately that T_n is symmetric; note also that $p_n \neq 0$ because $P_n(z)$ is of degree n, and that $P_n(z)$ has no roots at $z = 1$, because $P_n(1) \neq 0$. Writing

$$\begin{aligned} P_n(z) - P_n^r(z) &= (p_n - p_0) z^n + (p_{n-1} - p_1) z^{n-1} + \ldots + (p_1 - p_{n-1}) z + (p_0 - p_n) \\ &= (z-1)\left[q_{n-1} z^{n-1} + \ldots + q_{n-2} z^{n-2} + \ldots + q_1 z + q_0\right], \end{aligned}$$

it follows that the q_i may be determined by solving

$$\begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 \\ & & & & -1 \end{pmatrix} \begin{pmatrix} q_{n-1} \\ q_{n-2} \\ \vdots \\ q_1 \\ q_0 \end{pmatrix} = \begin{pmatrix} p_n - p_0 \\ p_{n-1} - p_1 \\ \vdots \\ p_1 - p_{n-1} \\ p_0 - p_n \end{pmatrix}$$

By the first and last rows of this matrix equation, it follows that $q_{n-1} = q_0 = p_n - p_0$; by the second and next-to-last rows, it follows that $q_{n-2} = q_1$, etc.; thus, T_{n-1} is also symmetric.

Since $P_n(z)$ is a real polynomial, $z_+ = e^{i\phi}$ is a root [on the unit circle] of $P_n(z)$ iff $e^{-i\phi} = 1/z_+$ is also root of $P_n(z)$, that is, iff $z_+ = e^{i\phi}$ is a root of $P_n^r(z) \triangleq z^n P_n(1/z) = p_0 z^n + p_1 z^{n-1} + \ldots + p_{n-1} z + p_n$. Thus, $P_n(z)$ and $P_n^r(z)$, and thus also $T_n(z)$ and $T_{n-1}(z)$, have the same roots on the unit circle. □

Fact B.11 *Consider an n'th-degree real polynomial $P_n(z) = \{(z-1)T_{n-1}(z) + T_n(z)\}/2$, with $P_n(1) \neq 0$ and either $T_{n-1}(z) \neq 0$ or $T_n(0) = 0$ (or both), such that $T_n(z)$ and $T_{n-1}(z)$ are symmetric, and $P_n(z)$, $T_n(z)$, and $T_{n-1}(z)$ have the same roots on the unit circle. Denote $T_n(z) = t_{n,n} z^n + \ldots + t_{n,0}$ and $T_{n-1}(z) = t_{n-1,n-1} z^n + \ldots + t_{n-1,0}$. Define λ as the smallest non-negative integer such that $t_{n-1,\lambda} \neq 0$. Consider the $(n-2)$'th-degree polynomial $T_{n-2}(z)$ defined such that*

$$z T_{n-2}(z) = \delta(z^{\lambda+1} + z^{-\lambda}) T_{n-1}(z) - T_n(z) \qquad \textit{where} \qquad \delta = \begin{cases} t_{n,0}/t_{n-1,\lambda} & \textit{if } T_{n-1}(z) \neq 0, \\ 0 & \textit{if } t_{n,0} = 0. \end{cases}$$

It follows that $P_n(z)$ has the same stationarity as $P_{n-1}(z) \triangleq \{(z-1)T_{n-2} + T_{n-1}(z)\}/2$, plus one additional root inside the unit circle if $T_n(1)/T_{n-1}(1) > 0$, and outside the unit circle if $T_n(1)/T_{n-1}(1) < 0$.

Proof:

$$P_n(z) = \{-zT_{n-2}(z) + [z - 1 + \delta(z^{\lambda+1} + z^{-\lambda})]T_{n-1}(z)\}/2 = \text{relation to } P_{n-1}(z)?$$

Now define

$$q_\eta(z) = p(z) - \eta\, p^r(z) = (p_n - \eta\, p_0)\, z^n + (p_{n-1} - \eta\, p_1)\, z^{n-1} + \ldots + (p_1 - \eta\, p_{n-1})\, z + (p_0 - \eta\, p_n), \quad \text{(B.10)}$$

$$\hat{q}_\lambda(z) = p^r(z) - \lambda\, p(z) = (p_0 - \lambda\, p_n)\, z^n + (p_1 - \lambda\, p_{n-1})\, z^{n-1} + \ldots + (p_{n-1} - \lambda\, p_1)\, z + (p_n - \lambda\, p_0). \quad \text{(B.11)}$$

If $|p_0/p_n| < 1$ [and thus the leading coefficient of $q_\eta(z)$ is nonzero for $\eta = 0$ through $\eta = \eta^* = p_0/p_n$], then, by Fact B.5, the n roots of $q_\eta(z)$ vary smoothly as η is varied from 0 up to (or down to) η^*; that is, roots do not "jump" between inside the unit circle and outside the unit circle as η is varied over this range. Since $q_{\eta^*}(z) = zq(z)$, it is seen that exactly one root of $q_\eta(z)$ goes to zero as $\eta \to \eta^*$. Of the remaining roots of the n'th-degree polynomial $p(z)$, the number inside the unit circle, on the unit circle, and outside the unit circle are precisely the same as for the $(n-1)$'th-degree polynomial $q(z)$.

If $|p_0/p_n| > 1$ [and thus the leading coefficient of $\hat{q}_\lambda(z)$ is nonzero for $\lambda = 0$ through $\lambda = \lambda^* = p_n/p_0$], then, by Fact B.5, the n roots of $\hat{q}_\lambda(z)$ vary smoothly as λ is varied from 0 up to (or down to) λ^*; that is, roots do not "jump" between inside the unit circle and outside the unit circle as λ is varied over this range. Since $\hat{q}_{\lambda^*}(z)/\lambda^* = q_{\eta^*}(z) = zq(z)$, it is seen that exactly one root of $\hat{q}_\lambda(z)$ goes to zero as $\lambda \to \lambda^*$. Of the remaining roots of the n'th-degree polynomial $p(z)$, the number inside the unit circle, on the unit circle, and outside the unit circle are precisely the same as for the $(n-1)$'th-degree polynomial $q(z)$.

□

Fact B.12 *Let $p(z)$ be an n'th-degree, irregular, real polynomial that is neither odd nor even, and let $a(z)$ be a real even polynomial with $a(i\omega) > 0$. Define $p_a(z)$ and $p_b(z)$ such that*

$$p(z) = p_a(z) + p_b(z) \quad \textit{where} \quad \begin{cases} p_a(z) = [p(z)]_{even}, \quad p_b(z) = [p(z)]_{odd} & \textit{even } n, \\ p_a(z) = [p(z)]_{odd}, \quad p_b(z) = [p(z)]_{even} & \textit{odd } n. \end{cases} \quad \text{(B.12)}$$

If $a(z)$ is chosen such that the degree of $\{a(z)\, p_b(z)\}$ is $n-1$, then the n'th-degree polynomials $p(z)$ and $q(z) = p_a(z) + a(z)\, p_b(z)$ have the same stationarity, and $q(z)$ is regular (i.e., $q_{n-1} \neq 0$).

Proof: If $p_a(z)$ is even (resp., odd), $\{(1-\lambda) + \lambda\, a(z)\}\, p_b(z)$ is odd (resp., even). For $0 \leq \lambda \leq 1$, define the n'th-degree polynomial

$$q_\lambda(z) = p_a(z) + \{(1-\lambda) + \lambda\, a(z)\}\, p_b(z).$$

Noting that $a(i\omega) > 0$, it follows as in the proof of Fact B.10 that $q_\lambda(i\omega) = 0$ iff $p_a(i\omega) = 0$ and $p_b(i\omega) = 0$ [that is, iff $p(i\omega) = 0$], independent of λ. Thus, $p(z)$ and $q_\lambda(z)$ have the same imaginary roots, independent of λ. Further, by Fact B.5, the n roots of $q_\lambda(z)$ vary smoothly as λ is varied; i.e., roots do not "jump" between the LHP and the RHP as λ is varied. Thus, $p(z) = q_0(z)$ and $q(z) = q_1(z)$ have the same stationarity. □

Fact B.13 *Let $p(z)$ be a real n'th-degree odd or even polynomial. Defining $r(z) = p(z) + p'(z)$, it follows that $N_o(p) = N_i(p) = N_o(r)$, and $r(z)$ is regular (i.e., $r_{n-1} \neq 0$).*

Proof: It follows from the fact that $p(z)$ is either odd or even that $r_{n-1} \neq 0$, and that $p(z)$ has the same number of RHP roots as LHP roots [that is, $N_+(p) = N_-(p)$]. Define $q_\varepsilon(z) = p(z) + \varepsilon p'(z)$; it follows from Fact B.11 [with $a(z) = \varepsilon$] that, for any $\varepsilon > 0$, $r(z)$ and $q_\varepsilon(z)$ have the same stationarity. We thus focus on $q_\varepsilon(z)$ in the limit that $\varepsilon \to 0$. If $z = i\omega$ is a root of multiplicity k of the polynomial $p(z) = q_0(z)$, then $z = i\omega$ is a root of multiplicity $k-1$ of the polynomial $q_\varepsilon(z)$ for $\varepsilon > 0$, and exactly one root of $q_\varepsilon(z)$ moves (by Fact B.5, continuously) away from $z = i\omega$ as ε is increased from zero. To quantify what direction this root moves as

ε is increased from zero, denoting $p^{(k)} = d^k p(z)/dz^k$, we write the Taylor series expansion of $q_\varepsilon(z)$ around $z = \mathrm{i}\omega$:

$$\begin{aligned} q_\varepsilon(\mathrm{i}\omega+\delta) &= \frac{\delta^{k-1}}{(k-1)!}\left[q_\varepsilon^{(k-1)}(z)\right]_{z=\mathrm{i}\omega} + \frac{\delta^k}{k!}\left[q_\varepsilon^{(k)}(z)\right]_{z=\mathrm{i}\omega} + \dots \\ &= \frac{\delta^{k-1}}{(k-1)!}\left[\cancelto{0}{p^{(k-1)}(\mathrm{i}\omega)} + \varepsilon p^{(k)}(\mathrm{i}\omega)\right] + \frac{\delta^k}{k!}\left[p^{(k)}(\mathrm{i}\omega) + \varepsilon p^{(k+1)}(\mathrm{i}\omega)\right] + \dots \\ &= \frac{\delta^{k-1}}{k!}\, p^{(k)}(\mathrm{i}\omega)\left[\varepsilon k + \delta + \varepsilon\,\delta\frac{p^{(k+1)}(\mathrm{i}\omega)}{p^{(k)}(\mathrm{i}\omega)} + \dots\right] \end{aligned} \tag{B.13}$$

where the "…" terms are higher order in ε, and thus negligible for small ε. By Fact B.5, δ is nearly proportional to ε for sufficiently small ε; the third term in brackets in (B.13) is thus also negligible compared to the first two terms. Solving for $q_\varepsilon(\mathrm{i}\omega+\delta) = 0$ thus shows that, in addition to the $k-1$ roots fixed at $\delta = 0$, the remaining root is given by $\delta \approx -k\varepsilon < 0$ for sufficiently small $\varepsilon > 0$ (that is, the remaining root moves into the LHP as ε is increased from zero). Since $p(z) = q_0(z)$ and $r(z) = q_1(z)$, it follows that $N_+(p) = N_+(r)$. □

Fact B.14 (The Bistritz Test) *The stationarity $\{N_i(p), N_u(p), N_o(p)\}$ of an n'th-degree polynomial $p(z)$ may be found via application of the following three cases to polynomials successively smaller and smaller degree:*

Case 1: If $p(z)$ is regular (that is, if $p_0 \neq 0$), then define $q(z)$ as in (B.10). It follows that

$$\{N_i(p), N_u(p), N_o(p)\} = \begin{cases} \{N_i(q)+1, N_u(q), N_o(q)\} & \text{if } |p_0/p_n| > 1, \\ \{N_i(q), N_u(q), N_o(q)\} & \text{if } |p_0/p_n| < 1; \end{cases}$$

$\{N_i(q), N_u(q), N_o(q)\}$ may then be found by applying Fact B.14 to the $(n-1)$'th-degree polynomial $q(z)$.

Case 2: If $p(z)$ is irregular (that is, if $p_{n-1} = 0$) but neither even nor odd, then define $p_a(z)$ and $p_b(z)$ as in (B.12). Since $p_{n-1} = 0$, $p_b(z)$ has degree $n-1-2k$ for some $k > 0$; define $a(z) = 1 + (-z^2)^k$. Defining $q(z) = p_a(z) + a(z)\,p_b(z)$, it follows that $\{N_i(p), N_u(p), N_o(p)\} = \{N_i(q), N_u(q), N_o(q)\}$, which may be found by applying Fact B.14 to the regular n'th-degree polynomial $q(z)$.

Case 3: If $p(z)$ is either even or odd, then define $r(z) = p(z) + p'(z)$. It follows that $N_i(p) = N_o(p) = N_o(r)$, where $N_o(r)$ may be found by applying Fact B.14 to the regular n'th-degree polynomial $r(z)$.

Proof: Case 1 follows immediately from Fact B.10. Case 2 follows from Fact B.11, noting that $a(\mathrm{i}\omega) > 0$ for real ω, and that the degree of $\{a(z)\,p_b(z)\}$ is $n-1$. Case 3 follows immediately from Fact B.12. □

B.4 Vector calculus: grad, div, curl, and Gauss, Stokes, Helmholtz

Vector calculus is the study of scalar and vector fields that vary in one or more spatial dimensions, e.g., $f(x_1,x_2,x_3)$ and $\vec{v}(x_1,x_2,x_3)$. The study of vector calculus is facilitated by the **del** operator (denoted by the **nabla** symbol ∇), which is defined using summation notation such that

$$\nabla = \mathbf{e}_j \frac{\partial}{\partial x_j}; \qquad \text{thus, in 3D,} \quad \nabla = \mathbf{e}_1 \frac{\partial}{\partial x_1} + \mathbf{e}_2 \frac{\partial}{\partial x_2} + \mathbf{e}_3 \frac{\partial}{\partial x_3}. \tag{B.14}$$

With this operator, the **gradient**, ∇f, the **divergence**, $\nabla \cdot \vec{v}$, and the **curl**, $\nabla \times \vec{v}$, are given in 3D as, respectively,

$$\nabla f = \mathbf{e}_1 \frac{\partial f}{\partial x_1} + \mathbf{e}_2 \frac{\partial f}{\partial x_2} + \mathbf{e}_3 \frac{\partial f}{\partial x_3}, \tag{B.15}$$

$$\nabla \cdot \vec{v} = \frac{\partial v_1}{\partial x_1} + \frac{\partial v_2}{\partial x_2} + \frac{\partial v_3}{\partial x_3}, \tag{B.16}$$

$$\nabla \times \vec{v} = \begin{vmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_3} \\ v_1 & v_2 & v_3 \end{vmatrix} = \Big(\frac{\partial v_3}{\partial x_2} - \frac{\partial v_2}{\partial x_3}\Big)\mathbf{e}_1 + \Big(\frac{\partial v_1}{\partial x_3} - \frac{\partial v_3}{\partial x_1}\Big)\mathbf{e}_2 + \Big(\frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}\Big)\mathbf{e}_3. \tag{B.17}$$

The **Laplacian**, Δf, is defined using summation notation such that

$$\Delta f = \nabla \cdot \nabla f = \frac{\partial^2 f}{\partial x_j^2}; \qquad \text{thus, in 3D,} \quad \Delta f = \Big(\frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \frac{\partial^2}{\partial x_3^2}\Big) f. \tag{B.18}$$

The **bilaplacian** (a.k.a. **biharmonic**) is defined as the Laplacian of the Laplacian, $\Delta\Delta f$; thus, in 3D,

$$\begin{aligned} \Delta\Delta f &= \Big(\frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \frac{\partial^2}{\partial x_3^2}\Big)\Big(\frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \frac{\partial^2}{\partial x_3^2}\Big) f \\ &= \Big(\frac{\partial^4}{\partial x_1^4} + \frac{\partial^4}{\partial x_2^4} + \frac{\partial^4}{\partial x_3^4} + 2\frac{\partial^4}{\partial x_1^2 \partial x_2^2} + 2\frac{\partial^4}{\partial x_1^2 \partial x_3^2} + 2\frac{\partial^4}{\partial x_2^2 \partial x_3^2}\Big) f. \end{aligned} \tag{B.19}$$

Identities related to the gradient, divergence, curl, and Laplacian [all easily verified via substitution] include:

$$\nabla(f\psi) = f\nabla\psi + \psi\nabla f, \tag{B.20}$$

$$\nabla \cdot (f\vec{v}) = f\nabla\cdot\vec{v} + \vec{v}\cdot\nabla f, \tag{B.21}$$

$$\nabla \times (\nabla f) = 0, \tag{B.22}$$

$$\nabla \cdot (\nabla \times \vec{v}) = 0, \tag{B.23}$$

$$\nabla(\vec{u}\cdot\vec{v}) = (\vec{u}\cdot\nabla)\vec{v} + (\vec{v}\cdot\nabla)\vec{u} + \vec{u}\times(\nabla\times\vec{v}) + \vec{v}\times(\nabla\times\vec{u}), \tag{B.24}$$

$$\nabla\cdot(\vec{u}\times\vec{v}) = \vec{v}\cdot(\nabla\times\vec{u}) - \vec{u}\cdot(\nabla\times\vec{v}), \tag{B.25}$$

$$\nabla\times(f\vec{v}) = (\nabla f)\times\vec{v} + f\nabla\times\vec{v}, \tag{B.26}$$

$$\nabla\times(\vec{u}\times\vec{v}) = \vec{u}(\nabla\cdot\vec{v}) - \vec{v}(\nabla\cdot\vec{u}) + (\vec{v}\cdot\nabla)\vec{u} - (\vec{u}\cdot\nabla)\vec{v}, \tag{B.27}$$

$$\nabla\times\nabla\times\vec{v} = \nabla(\nabla\cdot\vec{v}) - \Delta\vec{v}. \tag{B.28}$$

Gauss's theorem (a.k.a. the **divergence theorem**) relates the integral over a (3D or 2D) volume V of the *divergence* of a vector field $\vec{v}$ to the integral over the (2D or 1D) surface of the volume, ∂V, of the *normal component* of the vector field (note that $d\vec{A}$ is oriented as an outward-facing normal vector):

$$\int_V (\nabla\cdot\vec{v})\, dV = \int_{\partial V} \vec{v}\cdot d\vec{A}. \tag{B.29}$$

Stokes theorem relates the integral over a (2D) area A [which itself may be defined in $\mathbb{R}^3$] of the *curl* of a vector field $\vec{v}$ to the integral over the boundary of the area, ∂A, of the *tangential component* of the vector field (following the **right-hand rule**, $d\vec{s}$ is a counterclockwise-facing tangential vector when $d\vec{A}$ faces the viewer):

$$\int_A (\nabla\times\vec{v})\cdot d\vec{A} = \oint_{\partial A} \vec{v}\cdot d\vec{s}. \tag{B.30}$$

An important special case of Stoke's theorem, known as **Green's theorem**, is developed by taking $\vec{v} = \{\psi, \phi, 0\}$ and the (2D) area A as lying in the $x-y$ plane, in which case (B.30) reduces immediately to

$$\int_A \left(\frac{\partial \phi}{\partial x} - \frac{\partial \psi}{\partial y}\right) dx\,dy = \oint_{\partial A} (\psi\,dx + \phi\,dy). \tag{B.31}$$

The **Helmholtz decomposition** (a.k.a. the **fundamental theorem of vector calculus**), states that any vector field $\vec{v}$ whose curl and divergence vanish at infinity may be decomposed into an **irrotational** part, $-\nabla\phi$, and a **solenoidal** part, $\nabla \times \vec{\psi}$:

$$\vec{v} = -\nabla\phi + \nabla \times \vec{\psi}, \tag{B.32}$$

where ϕ is called a **scalar potential** and $\vec{\psi}$ is called a **vector potential**; by the identities summarized above, the irrotational part is curl free and the solenoidal part is divergence free.

In (3D) **cylindrical coördinates**, the gradient, divergence, curl, and Laplacian may be written

$$\nabla f = \mathbf{e}_r \frac{\partial f}{\partial r} + \mathbf{e}_\phi \frac{1}{r}\frac{\partial f}{\partial \phi} + \mathbf{e}_z \frac{\partial f}{\partial z}, \tag{B.33a}$$

$$\nabla \cdot \vec{v} = \frac{1}{r}\frac{\partial (r v_r)}{\partial r} + \frac{1}{r}\frac{\partial v_\phi}{\partial \phi} + \frac{\partial v_z}{\partial z}, \tag{B.33b}$$

$$\nabla \times \vec{v} = \mathbf{e}_r \left(\frac{1}{r}\frac{\partial v_z}{\partial \phi} - \frac{\partial v_\phi}{\partial z}\right) + \mathbf{e}_\phi \left(\frac{\partial v_r}{\partial z} - \frac{\partial v_z}{\partial r}\right) + \mathbf{e}_z \left(\frac{1}{r}\frac{\partial (r v_\phi)}{\partial r} - \frac{1}{r}\frac{\partial v_r}{\partial \phi}\right), \tag{B.33c}$$

$$\Delta f = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial f}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 f}{\partial \phi^2} + \frac{\partial^2 f}{\partial z^2}. \tag{B.33d}$$

In (3D) **spherical coördinates**, the gradient, divergence, curl, and Laplacian may be written

$$\nabla f = \mathbf{e}_r \frac{\partial f}{\partial r} + \mathbf{e}_\theta \frac{1}{r}\frac{\partial f}{\partial \theta} + \mathbf{e}_\phi \frac{1}{r\sin\theta}\frac{\partial f}{\partial \phi}, \tag{B.34a}$$

$$\nabla \cdot \vec{v} = \frac{1}{r^2}\frac{\partial (r^2 v_r)}{\partial r} + \frac{1}{r\sin\theta}\frac{\partial (v_\theta \sin\theta)}{\partial \theta} + \frac{1}{r\sin\theta}\frac{\partial v_\phi}{\partial \phi}, \tag{B.34b}$$

$$\nabla \times \vec{v} = \mathbf{e}_r \frac{1}{r\sin\theta}\left(\frac{\partial (v_\theta \sin\theta)}{\partial \theta} - \frac{\partial v_\theta}{\partial \phi}\right) + \mathbf{e}_\theta \frac{1}{r}\left(\frac{1}{\sin\theta}\frac{\partial v_r}{\partial \phi} - \frac{\partial (r v_\phi)}{\partial r}\right) + \mathbf{e}_\phi \frac{1}{r}\left(\frac{\partial (r v_\theta)}{\partial r} - \frac{\partial v_r}{\partial \theta}\right), \tag{B.34c}$$

$$\Delta f = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial f}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial \theta}\left(\sin\theta\frac{\partial f}{\partial \theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2 f}{\partial \phi^2}. \tag{B.34d}$$

In (2D) **polar coördinates**, the bilaplacian may be written

$$\begin{aligned} \Delta\Delta f &= \left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2}{\partial \phi^2}\right]\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2}{\partial \phi^2}\right] f \\ &= \frac{\partial^4 f}{\partial r^4} + \frac{2}{r^2}\frac{\partial^4 f}{\partial r^2 \partial \phi^2} + \frac{1}{r^4}\frac{\partial^4 f}{\partial \phi^4} + \frac{2}{r}\frac{\partial^3 f}{\partial r^3} - \frac{2}{r^3}\frac{\partial^3 f}{\partial r \partial \phi^2} - \frac{1}{r^2}\frac{\partial^2 f}{\partial r^2} + \frac{4}{r^4}\frac{\partial^2 f}{\partial \phi^2} + \frac{1}{r^3}\frac{\partial f}{\partial r}. \end{aligned} \tag{B.35}$$

B.5 Some useful expansions, sums, identities, and definitions

The **binomial coefficients** ${}_nC_k$ are the numerical coefficients in both the expansion of an n'th-degree binomial and the expansion of the n'th derivative of the product of two functions (known as **Leibnitz's rule**),

$$(x+y)^n = \sum_{k=0}^{n} {}_nC_k\, x^{n-k} y^k = {}_nC_0\, x^n + {}_nC_1\, x^{n-1} y^1 + \ldots + {}_nC_n\, y^n, \qquad (uv)^{(n)} = \sum_{k=0}^{n} {}_nC_k\, u^{(n-k)} v^{(k)}, \tag{B.36}$$

where $u^{(n)}$ denotes the n'th derivative of u with respect to its argument, and may be calculated as follows:

$$ {}_nC_k \triangleq \binom{n}{k} \triangleq \frac{n!}{k!\,(n-k)!} \quad \text{for } n \geq k \geq 0. \tag{B.37} $$

Note that ${}_nC_k$ is pronounced n **choose** k, as it represents the number of unordered collections of k objects that can be chosen from a set of n distinct objects. The binomial coefficients ${}_nC_k$ for $k = 0$ through $k = n$ appear on the n'th row (counting from zero) of what is commonly referred to as **Pascal's triangle**[6]:

						1								
					1		1							
				1		2		1						
			1		3		3		1					
		1		4		6		4		1				
	1		5		10		10		5		1			
1		6		15		20		15		6		1		
	7		21		35		35		21		7		1	
8		28		56		70		56		28		8		1

The following identities, derivatives, and sums are often useful[7]:

$$ e^{ix} = \cos x + i \sin x \tag{B.38} $$

$$ 1 = \cos^2 x + \sin^2 x \tag{B.39} $$

$$ \sin(x) = (e^{ix} - e^{-ix})/(2i) \tag{B.40} $$

$$ \cos(x) = (e^{ix} + e^{-ix})/2 \tag{B.41} $$

$$ \sinh(x) = (e^{x} - e^{-x})/2 \tag{B.42} $$

$$ \cosh(x) = (e^{x} + e^{-x})/2 \tag{B.43} $$

$$ \tan(x) = \sin(x)/\cos(x) \tag{B.44} $$

$$ \tanh(x) = \sinh(x)/\cosh(x) \tag{B.45} $$

$$ \sin(x+y) = \sin x \cos y + \cos x \sin y \tag{B.46} $$

$$ \cos(x+y) = \cos x \cos y - \sin x \sin y \tag{B.47} $$

$$ \cos(x) = 2\cos^2(x/2) - 1 = 1 - 2\sin^2(x/2) \tag{B.48} $$

$$ \tan(x+y) = \frac{\tan x + \tan y}{1 - \tan x \tan y} \tag{B.49} $$

$$ 2\,\sin(x)\,\cos(y) = \sin(x+y) + \sin(x-y) \tag{B.50} $$

$$ 2\,\cos(x)\,\cos(y) = \cos(x+y) + \cos(x-y) \tag{B.51} $$

$$ 2\,\sin(x)\,\sin(y) = \cos(x-y) - \cos(x+y) \tag{B.52} $$

$$ \sum_{m=1}^{M} \cos mx = \frac{\cos[(M+1)x/2]\sin(Mx/2)}{\sin(x/2)} \tag{B.53} $$

$$ \text{acos}\,(x) = \frac{\pi}{2} - \text{asin}\,x \tag{B.54} $$

$$ d(\sin x)/dx = \cos x; \;\; d(\cos x)/dx = -\sin x \tag{B.55} $$

$$ d(\tan x)/dx = 1/\cos^2 x \tag{B.56} $$

$$ d(\sinh x)/dx = \cosh x; \;\; d(\cosh x)/dx = \sinh x \tag{B.57} $$

$$ d(\tanh x)/dx = 1 - \tanh^2 x = 1/\cosh^2(x) = \text{sech}^2(x) \tag{B.58} $$

$$ d(\text{asin}\,x)/dx = 1/\sqrt{1-x^2} \tag{B.59} $$

$$ d(\text{acos}\,x)/dx = -1/\sqrt{1-x^2} \tag{B.60} $$

$$ d(\text{atan}\,x)/dx = 1/(1+x^2) \tag{B.61} $$

$$ d(\ln x)/dx = 1/x \tag{B.62} $$

$$ \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \triangleq T_n \tag{B.63} $$

$$ \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{B.64} $$

$$ \sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4} \tag{B.65} $$

$$ \sum_{k=1}^{n} k^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \tag{B.66} $$

$$ \sum_{k=1}^{n} T_n = \frac{n(n+1)(n+2)}{6} = \frac{(n+1)^3 - (n+1)}{6} \tag{B.67} $$

[6] The triangular table of binomial coefficients is, in the west, commonly (and incorrectly) attributed by many, via the use of this name, to Blaise Pascal (b. 1623), though it dates back to several earlier sources, the earliest being Pingala's Sanskrit work *Chandas Shastra*, written in the fifth century BC.

[7] A further bit of history: Pythagoras (Greece, b. 572 BC) is credited with determining (B.63) [known as the *triangular numbers* T_n, as it gives the number of objects (e.g., rocks or coins) in a triangular pack; the Pythagoreans placed a particular mystic significance on the triangular pack with four objects on a side, corresponding to $T_4 = 10$], Archimedes (Greece, b. 287 BC) with (B.64), Abu Bakr al-Karaji (Baghdad, d. 1019) with (B.65), Abu Ali al-Hasan ibn al-Hasan ibn al-Haytham (Egypt, b. 965) with (B.66), and Aryabhata (India, b. 476) with (B.67) [which gives the number of spheres (e.g., cannonballs) in a stack with a trangular base].

The **polylogarithm** is defined, for any complex s and for $|z|<1$, by the infinite sum

$$Li_s(z) = \sum_{k=1}^{\infty} z^k/k^s; \tag{B.68a}$$

for integer nonpositive arguments, the polylogarithm may be written

$$Li_{-n}(z) = \left(z\frac{d}{dz}\right)^n \frac{z}{1-z} = \sum_{k=0}^{n} k!\,S(n+1,k+1)\left(\frac{z}{1-z}\right)^{k+1} \tag{B.68b}$$

where the $S(n,k)$, known as the **Stirling numbers of the second kind**, are defined such that

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j} j_k C_j\, j^n.$$

Noting the definitions of δ_{di} in §1.2.3, $\delta^\sigma(t)$ in §5.3.3 (and it's peculiar $\sigma\to 0$ limit), and $\delta^{\lambda,m}(t)$ in §5.3.4 for integer $m\geq 0$ (and it's peculiar $\lambda\to\infty$ limit), the **continuous-time Heaviside step function** $h_a(t)$, with $a=0$, $a=1$, or $a=1/2$, and **discrete-time Heaviside step function** h_{dk} are defined, respectively, as

$$h_a(t-d) = \begin{cases} 0 & t<d, \\ a & t=d, \\ 1 & t>d, \end{cases} \quad \Rightarrow \quad \begin{cases} h_{1/2}(t-d) = \lim\limits_{\sigma\to 0}\displaystyle\int_{-\infty}^{t} \delta^\sigma(t'-d)\,dt' \\ h_0(t-d) = \lim\limits_{\lambda\to\infty}\displaystyle\int_{-\infty}^{t} \delta^{\lambda,m}(t'-d)\,dt' \end{cases} \tag{B.69a}$$

$$h_{dk} = \begin{cases} 0 & k<d, \\ 1 & k\geq d. \end{cases} \quad \Rightarrow \quad h_{dk} = \sum_{i=-\infty}^{k}\delta_{di}; \tag{B.69b}$$

$h_0(t)$ is said to be **left-continuous** at $t=0$, as $h_0(-\varepsilon)\to h_0(0)$ as $\varepsilon\to 0$ with $\varepsilon>0$, whereas $h_1(t)$ is said to be **right-continuous** at $t=0$, as $h_1(\varepsilon)\to h_1(0)$ as $\varepsilon\to 0$ with $\varepsilon>0$. Note also that h_{0k} is often denoted as simply h_k. Finally, the following Taylor-series expansions, each valid for sufficiently small ε, are often useful:

$$e^\varepsilon = 1+\varepsilon+\frac{\varepsilon^2}{2!}+\frac{\varepsilon^3}{3!}+\frac{\varepsilon^4}{4!}+\ldots \tag{B.70}$$

$$\ln(1+\varepsilon) = \varepsilon-\frac{\varepsilon^2}{2}+\frac{\varepsilon^3}{3}-\frac{\varepsilon^4}{4}+\ldots \tag{B.71}$$

$$\sin(\varepsilon) = \varepsilon-\frac{\varepsilon^3}{3!}+\frac{\varepsilon^5}{5!}-\frac{\varepsilon^7}{7!}+\ldots \tag{B.72}$$

$$\cos(\varepsilon) = 1-\frac{\varepsilon^2}{2!}+\frac{\varepsilon^4}{4!}-\frac{\varepsilon^6}{6!}+\ldots \tag{B.73}$$

$$\tan(\varepsilon) = \varepsilon+\frac{\varepsilon^3}{3}+\frac{2\varepsilon^5}{15}+\frac{17\varepsilon^7}{315}+\ldots \tag{B.74}$$

$$\tanh(\varepsilon) = \varepsilon-\frac{\varepsilon^3}{3}+\frac{2\varepsilon^5}{15}-\frac{17\varepsilon^7}{315}+\ldots \tag{B.75}$$

$$\operatorname{atan}(\varepsilon) = \varepsilon-\frac{\varepsilon^3}{3}+\frac{\varepsilon^5}{5}-\frac{\varepsilon^7}{7}+\ldots \tag{B.76}$$

$$\frac{1}{1-\varepsilon} = 1+\varepsilon+\varepsilon^2+\varepsilon^3+\varepsilon^4\ldots \tag{B.77}$$

$$\frac{1}{(1-\varepsilon)^2} = 1+2\varepsilon+3\varepsilon^2+4\varepsilon^3+5\varepsilon^4+\ldots \tag{B.78}$$

$$\frac{1}{(1-\varepsilon)^3} = 1+3\varepsilon+6\varepsilon^2+10\varepsilon^3+15\varepsilon^4+\ldots \tag{B.79}$$

$$\frac{1}{(1-\varepsilon)^n} = {}_nC_n+{}_{n+1}C_n\,\varepsilon+{}_{n+2}C_n\,\varepsilon^2+\ldots \tag{B.80}$$

$$\sqrt{1+\varepsilon} = 1+\frac{\varepsilon}{2}-\frac{\varepsilon^2}{8}+\frac{\varepsilon^3}{16}-\frac{5\varepsilon^4}{128}+\ldots \tag{B.81}$$

$$\frac{1}{\sqrt{1-\varepsilon}} = 1+\frac{1}{2}\cdot\varepsilon+\frac{1\cdot 3}{2\cdot 4}\cdot\varepsilon^2+\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\cdot\varepsilon^3+\ldots \tag{B.82}$$

$$\operatorname{asin}(\varepsilon) = \varepsilon+\frac{1}{2}\cdot\frac{\varepsilon^3}{3}+\frac{1\cdot 3}{2\cdot 4}\cdot\frac{\varepsilon^5}{5}+\frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}\cdot\frac{\varepsilon^7}{7}+\ldots \tag{B.83}$$

B.6 Complex analysis†

As mentioned in §B.3.5, a function is said to be **analytic** in some open domain if it is equal to its own Taylor series in a neighborhood of every point in the domain. It follows immediately from this definition that all

analytic functions are **infinitely differentiable** (a.k.a. C^∞ or **smooth**), with the derivatives of the function at a point z_0 appearing in each of the coefficients of its Taylor series about this point,

$$f(z) = f(z_0) + \Delta z f'(z_0) + \frac{(\Delta z)^2}{2!} f''(z_0) + \frac{(\Delta z)^3}{3!} f'''(z_0) + \frac{(\Delta z)^4}{4!} f''''(z_0) + \dots \tag{B.84}$$

There are two important types of analytic functions, **real analytic** functions (that is, real functions of a real argument) and **complex analytic**[8] functions (that is, complex functions of a complex argument).

If the first derivative of a complex function is uniquely defined, the function is said to be **holomorphic**, as discussed further in Fact B.15.

As illustrated in (B.91) below, not all once-differentiable real functions (or, for that matter, even infinitely differentiable real functions) are real analytic. However, as established in Fact B.18 below, *all holomorphic functions are complex analytic, and all complex analytic functions are holomorphic*. This remarkable fact, combined with a variety of other useful properties satisfied by complex analytic functions, lead to a powerful body of theory known as **complex analysis**. By design, the present text does *not* leverage extensively this elegant body of theory; the main text is founded on more basic principles. However, it is instructive to summarize here the key results of complex analysis, which set the stage for a few useful algorithms and derivations presented in the remainder of this appendix.

B.6.1 Key results of complex analysis

Fact B.15 *If a complex-valued function $f(x+iy) = \phi(x,y) + i\psi(x,y)$, where $\phi(x,y)$ and $\psi(x,y)$ are real differentiable functions, satisfy the* **Cauchy-Riemann equations**

$$\frac{\partial \phi}{\partial x} = \frac{\partial \psi}{\partial y} \qquad \text{and} \qquad \frac{\partial \phi}{\partial y} = -\frac{\partial \psi}{\partial x} \tag{B.85}$$

over an open domain $\Omega \subseteq \mathbb{R}^2$, then the complex function $f(z)$ of the complex argument $z = x+iy$ is **complex differentiable** *everywhere in Ω, meaning that the* **complex derivative**

$$\frac{df}{dz}\Big|_{z=z_0} = \lim_{z\to z_0} \frac{f(z)-f(z_0)}{z-z_0} = \lim_{\Delta z\to 0} \frac{f(z_0+\Delta z)-f(z_0)}{\Delta z} \tag{B.86}$$

is uniquely defined [i.e., the limit gives the same value no matter from what direction z approaches z_0] for all $z_0 \in \Omega$, and the complex function $f(z)$ is said to be **holomorphic**. *The converse statement [i.e., the real and imaginary components of any complex differentiable function $f(x+iy) = \phi(x,y) + i\psi(x,y)$ obey the Cauchy-Riemann equations] is also true.*

Proof: We first consider (B.86) evaluated on a path with *real* $\Delta z = z - z_0$ [i.e., with $\Delta z = \Delta x$ and $\Delta y = 0$]:

$$\begin{aligned}\frac{df}{dz}\Big|_{z=z_0} &= \lim_{\Delta x\to 0} \frac{f(z_0+\Delta x)-f(z_0)}{\Delta x} = \lim_{\Delta x\to 0} \frac{[\phi(z_0+\Delta x)+\mathrm{i}\psi(z_0+\Delta x)]-[\phi(z_0)+\mathrm{i}\psi(z_0)]}{\Delta x} \\ &= \Big[\frac{\partial \phi}{\partial x} + \mathrm{i}\frac{\partial \psi}{\partial x}\Big]_{z=z_0}.\end{aligned}$$

We then consider (B.86) evaluated on a path with *imaginary* $\Delta z = z - z_0$ [i.e., with $\Delta z = \mathrm{i}\Delta y$ and $\Delta x = 0$]:

$$\begin{aligned}\frac{df}{dz}\Big|_{z=z_0} &= \lim_{\Delta y\to 0} \frac{f(z_0+\mathrm{i}\Delta y)-f(z_0)}{\mathrm{i}\Delta y} = \lim_{\Delta y\to 0} \frac{[\phi(z_0+\mathrm{i}\Delta y)+\mathrm{i}\psi(z_0+\mathrm{i}\Delta y)]-[\phi(z_0)+\mathrm{i}\psi(z_0)]}{\mathrm{i}\Delta y} \\ &= \Big[-\mathrm{i}\frac{\partial \phi}{\partial y} + \frac{\partial \psi}{\partial y}\Big]_{z=z_0}.\end{aligned}$$

[8] Complex analytic functions are often, sometimes confusingly, referred to simply as analytic functions.

Setting the real and imaginary components of these two evaluations of $(df/dz)_{z=z_0}$ as equal then leads immediately to (B.85). Conversely, if these two evaluations of $(df/dz)_{z=z_0}$ are not equal, then (B.85) is not satisfied, and (B.86) is not uniquely defined. □

Note that it follows by taking $\partial/\partial x$ and $\partial/\partial y$ of the two equations in (B.85) and combining that the real functions $\phi(x,y)$ and $\psi(x,y)$ themselves obey **Laplace's equation** on Ω:

$$\Big(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\Big)\phi = 0, \qquad \Big(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\Big)\psi = 0. \tag{B.87}$$

Note also that there is a natural interpretation of the ϕ and ψ components of any analytic function $f(x+\mathrm{i}y) = \phi(x,y)+\mathrm{i}\psi(x,y)$ as the **velocity potential** and **streamfunction**, respectively, of a **2D steady incompressible irrotational fluid flow** (a.k.a. **potential flow**): by defining

$$u = -\frac{\partial\phi}{\partial x}, \quad v = -\frac{\partial\phi}{\partial y} \qquad \text{and} \quad u = -\frac{\partial\psi}{\partial y}, \quad v = \frac{\partial\psi}{\partial x}, \tag{B.88a}$$

it follows immediately from the Cauchy-Riemann equations (B.85) that

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y} = 0, \tag{B.88b}$$

$$\frac{\partial v}{\partial x}-\frac{\partial u}{\partial y} = 0; \tag{B.88c}$$

note that (B.88b) is the **incompressibility condition** and (B.88c) is the **irrotationality condition** of the 2D flow. In this representation, the flow moves with a velocity vector proportional to the negative gradient of the **velocity potential** ϕ [see (B.88a)], and along lines of constant values of the **streamfunction** ψ, thus motivating these names. It follows that lines of constant velocity potential ϕ and lines of constant streamfunction ψ are necessarily *orthogonal* to each other (and, thus, which are the lines of constant ϕ and which are the lines of constant ψ in a given potential flow solution, and which direction the flow actually moves, are matters of interpretation, and may be swapped); this property is explored further in §B.6.2.

Fact B.16 (Cauchy's integral theorem) *If $f(z)$ is holomorphic on some simply-connected domain D as well as on its boundary Γ, then*

$$\oint_\Gamma f(z)\,dz = 0. \tag{B.89}$$

Proof: Taking $f(z) = \phi(z)+\mathrm{i}\psi(z)$ and $z = x+\mathrm{i}y$, expanding, applying Green's theorem (B.31), then applying the Cauchy-Riemann conditions (B.85) of Fact B.15 leads to

$$\begin{aligned}\oint_\Gamma f(z)\,dz &= \int_\Gamma(\phi+\mathrm{i}\psi)(dx+\mathrm{i}\,dy) = \int_\Gamma(\phi\,dx-\psi\,dy)+\mathrm{i}\int_\Gamma(\psi\,dx+\phi\,dy)\\ &= -\int_D\Big(\frac{\partial\psi}{\partial x}+\frac{\partial\phi}{\partial y}\Big)dx\,dy+\mathrm{i}\int_D\Big(\frac{\partial\phi}{\partial x}-\frac{\partial\psi}{\partial y}\Big)dx\,dy = 0.\end{aligned}$$

□

Fact B.17 (Cauchy's integral formula) *If $f(z)$ is holomorphic on some simply-connected domain D as well as on its boundary Γ, taken counterclockwise around D, then for every z_0 on the interior of D,*

$$f(z_0) = \frac{1}{2\pi i}\oint_\Gamma \frac{f(z)}{z-z_0}\,dz. \tag{B.90}$$

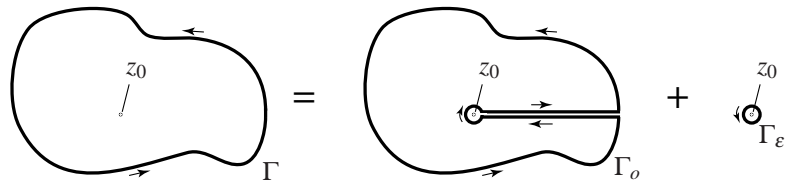


Figure B.5: Decomposition of the integral (B.90) over the (counterclockwise) contour Γ into two parts, a contour over a domain Γ_o inside of which the integrand is holomorphic, plus a contour Γ_ε (a counterclockwise circle of radius ε) which surrounds the point z_0 at which the integrand is nonholomorphic.

Proof: By the equivalence sketched in Figure B.5 and Cauchy's integral theorem (Fact B.16), noting that the function $f(z)/(z-z_0)$ is holomorphic inside Γ_o, the integral over Γ in (B.90) is equivalent to (i.e., the contour integral may be "shrunk" to) the same integral over Γ_ε, defined to be a circle of radius ε around the point z_0. Parameterize the contour Γ_ε as $z = z_0 + \varepsilon e^{it}$ for $t = 0$ to 2π [and, thus, $dz = \varepsilon\,\mathrm{i}\,e^{it}dt$] and note that, since $f(z)$ is holomorphic inside Γ, it follows that $f(z) \to f(z_0)$ everywhere on Γ_ε in the limit that $\varepsilon \to 0$. Thus,

$$\oint_\Gamma \frac{f(z)}{z-z_0}\,dz = \lim_{\varepsilon\to 0}\oint_{\Gamma_\varepsilon}\frac{f(z)}{z-z_0}\,dz = f(z_0)\lim_{\varepsilon\to 0}\int_0^{2\pi}\frac{1}{\varepsilon\,e^{it}}\,[\varepsilon\,\mathrm{i}\,e^{it}dt] = f(z_0)\int_0^{2\pi}\mathrm{i}\,dt = f(z_0)[2\pi\mathrm{i}]. \qquad \square$$

Fact B.18 (Analyticity of holomorphic functions) *All complex analytic functions are holomorphic, and all holomorphic functions are complex analytic.*

Proof: By definition, the Taylor Series of an analytic function converges in a finite neighborhood of every point in the domain, which implies that the coefficients of the Taylor Series itself are uniquely defined and finite [see (B.84)]; thus, all complex analytic functions are holomorphic. To establish that all holomorphic functions are complex analytic, let $f(z)$ be holomorphic within an open simply-connected domain Ω containing the point $z = a$, let z_0 be any point in Ω with $|z_0 - a| < \varepsilon$ for some small ε, and let Γ be a closed contour within Ω which encircles z_0 counterclockwise such that $|z-a| > \varepsilon$ for all points z on the contour Γ (which is always possible if ε is sufficiently small). Then, by Fact B.17, rearranging, and applying (B.77),

$$f(z_0) = \frac{1}{2\pi\mathrm{i}}\oint_\Gamma \frac{f(z)}{z-z_0}\,dz = \frac{1}{2\pi\mathrm{i}}\oint_\Gamma \frac{f(z)}{z-a}\cdot\frac{1}{1-\frac{z_0-a}{z-a}}\,dz = \frac{1}{2\pi\mathrm{i}}\oint_\Gamma \frac{f(z)}{z-a}\cdot\sum_{n=0}^{\infty}\left(\frac{z_0-a}{z-a}\right)^n dz.$$

The sum on the RHS converges uniformly for all z on the contour Γ, because $|z_0-a|/|z-a| < 1$ everywhere on Γ; thus, the integral and the series may be swapped, leading to

$$f(z_0) = \sum_{n=0}^{\infty}(z_0-a)^n\frac{1}{2\pi\mathrm{i}}\oint_\Gamma\frac{f(z)}{(z-a)^{n+1}}\,dz = \sum_{n=0}^{\infty}c_n(z_0-a)^n \quad \text{where} \quad c_n = \frac{1}{2\pi\mathrm{i}}\oint_\Gamma\frac{f(z)}{(z-a)^{n+1}}\,dz,$$

thus establishing that all points z_0 within a radius ε from the point a may be expanded via a Taylor series. $\square$

Notwithstanding Fact B.18, note that not all once differentiable or even infinitely differentiable *real* functions (of a real argument) are real analytic; two simple examples are

$$f_1(x) = |x|^k \;\text{ for odd } k \geq 3 \qquad \text{and} \qquad f_2(x) = \begin{cases} e^{-1/(1-z^2)} & \text{for } |x| < 1, \\ 0 & \text{otherwise.} \end{cases} \tag{B.91}$$

The function $f_1(x)$ is only $k-1$ times differentiable at $x = 0$, and is thus nonanalytic. The function $f_2(x)$, sometimes called a **bump** function, is an example of an infinitely differentiable function with **compact support**. Note that $f_2(x) > 0$ for $-1 < x < 1$; however, *all* derivatives of $f_2(x)$ at $x = \pm 1$ are exactly zero; thus, the function $f_2(x)$ does *not* match its Taylor series in any finite neighborhood of $x = 1$ or $x = -1$.

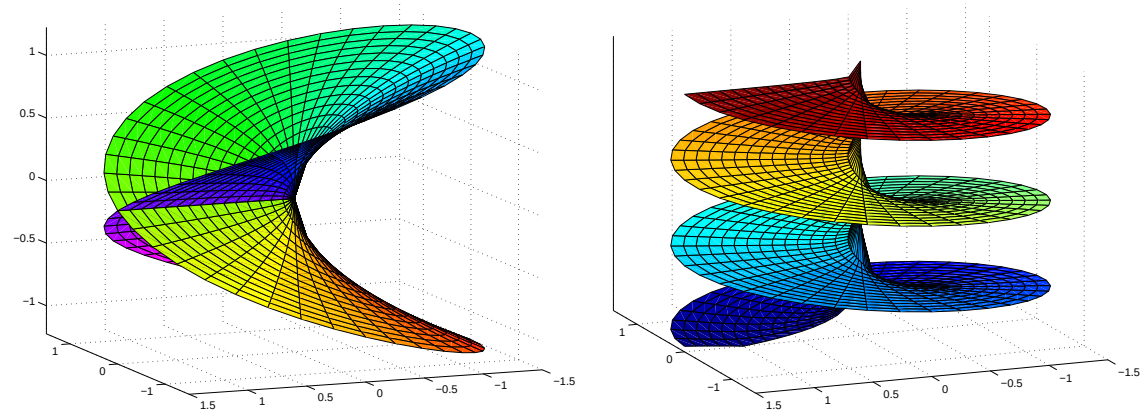


Figure B.6: Riemann surfaces of (a) $f(z) = z^{1/2}$, indicating the real part of $f(z)$ by the vertical axis and the argument of $f(z)$ by the color, and (b) $f(z) = ln(z)$, indicating the absolute value of $f(z)$ by the vertical axis and the argument of $f(z)$ by the color.

By the chain rule for differentiation, it follows immediately that:

Fact B.19 *The composition of analytic functions is analytic everywhere that said composition is finite.*

The following facts are also easily established:

Fact B.20 *Polynomials are analytic over the entire complex plane. Indeed, the Taylor series of an n'th-order polynomial written around any point in the complex plane vanishes after the n'th term.*

Fact B.21 *Rational functions (that is, polynomials divided by polynomials) are analytic everywhere in the complex plane except at their poles, where the denominator vanishes.*

Fact B.22 *The exponential function is analytic over the entire complex plane. Thus, by* (B.40)-(B.43)*, so are* $\sin()$, $\cos()$, $\sinh()$, *and* $\cosh()$.

Many smooth complex functions $f(z)$ are **multivalued**; that is, until specifically restricted otherwise, $f(z)$ can take multiple values for any particular value of z. Simple examples include the n'th root [introduced in Figure B.1b for $n = 5$] and the logarithm [writing $z = re^{i\phi}$, it follows that $\ln(z) = \ln r + i\phi$]. The multiple values of such functions for any given value of z may be understood geometrically via appropriately-defined **Riemann surfaces**, as illustrated in Figure B.6. The restriction of such multivalued functions to single-valued functions generally involves the definition of a **branch cut**. Two useful examples follow:

Fact B.23 *When restricting the n'th root function* $f(z) = z^{1/n} = re^{i\phi}$ *to have an argument* ϕ *in the range* $\pi/n < \phi \leq \pi/n$*, a branch cut in the z-plane is formed as a ray from the origin out the negative real axis. Away from this branch cut, the n'th root function restricted to this sheet of the corresponding Riemann surface (see Figure B.6a for the* $n = 2$ *case) is single valued and analytic.*

Fact B.24 *An identical branch cut as in Fact B.23 (a ray from the origin out the negative real axis) may be used for the logarithm* $\ln(z) = \ln r + i\phi$ *where* $z = re^{i\phi}$*; away from this branch cut, the logarithm function restricted to this sheet of the corresponding Riemann surface (see Figure B.6b) is also single valued and analytic.*

Returning now to Fact B.16, Cauchy's integral theorem may be extended as follows:

Fact B.25 (Cauchy's differentiation formula) *Denoting the n'th derivative of $f(z)$, for $n > 0$, as $f^{(n)}(z)$ and assuming that $f^{(n)}(z)$ is holomorphic,*

$$f^{(n)}(z_0) = \frac{n!}{2\pi i} \oint_\Gamma \frac{f(z)}{(z-z_0)^{n+1}}\, dz. \tag{B.92}$$

Proof: Applying integration by parts n times to $\oint_\Gamma f(z)/(z-z_0)^{n+1}\, dz$, noting that the contour Γ is closed and thus the integrand is the same at its "beginning" and its "end", then applying Fact B.17 gives

$$\oint_\Gamma \frac{f(z)}{(z-z_0)^{n+1}}\, dz = \frac{1}{n} \oint_\Gamma \frac{f^{(1)}(z)}{(z-z_0)^n}\, dz = \ldots = \frac{1}{n!} \oint_\Gamma \frac{f^{(n)}(z)}{z-z_0}\, dz = \frac{1}{n!} f^{(n)}(z_0)\,[2\pi \mathrm{i}]. \qquad \square$$

An important consequence of Fact B.25 with $f(z) = 1$, together with Fact B.16, is that

$$\oint_\Gamma \frac{1}{(z-z_0)^k}\, dz = 0 \text{ if } k \neq 1. \tag{B.93}$$

If a function $f(z)$ is holomorphic in an *open annulus* A around z_0, defined as all points z such that $r < |z-z_0| < R$, but is nonholomorphic at one or more points with $|z-z_0| \leq r$ and/or with $|z-z_0| \geq R$, then, everywhere in this open annulus A, $f(z)$ may be represented with the more general **Laurent series expansion**

$$f(z) = \sum_{n=-\infty}^{\infty} a_n (z-z_0)^n. \tag{B.94a}$$

That is, Laurent series expansions (B.94a) may be used to express functions which are holomophic in an *annulus* around z_0 (not including the point z_0), much as Taylor series expansions (B.84) may be used to express functions which are holomophic in a *disk* around z_0 (including the point z_0). By the same argument as that which leads from the Taylor series expansion in (B.84) to Cauchy's differentiation formula (B.92), it follows that the coefficients of the Laurent series expansion in (B.94a) may be computed as

$$a_n = \frac{1}{2\pi \mathrm{i}} \oint_\Gamma \frac{f(z)}{(z-z_0)^{n+1}}\, dz \tag{B.94b}$$

for some curve Γ which encircles z_0 counterclockwise once and is everywhere contained in A. The **principal part** of a Laurent series is the sum of the terms in (B.94a) with negative degree $n < 0$.

Many (but by no means all[9]) nondifferentiable functions of interest in fact fail to be differentiable only at a finite collection of distinct points $\{z_1, z_2, \ldots, z_p\}$, called **singular points** or **singularities**, in the domain of interest. Such functions may be represented with a Laurent series expansion in an annulus around any of these points z_i as described above, with $r = 0$ and $R > 0$ being the distance to the closest singularity z_j, $j \neq i$. If the principal part of such a Laurent series is a finite sum, then $f(z)$ is said to have a **pole** at z_i, of **order** equal to the negative of the degree of the highest term; a pole of order one is said to be a **simple pole**. If, on the other hand, the principal part of such a Laurent series is an infinite sum, then $f(z)$ is said to have an **essential singularity** at z_i; if a function does not have any essential singularities in the domain of interest, it is said to be **meromorphic**. In either case:

Fact B.26 (Residues) *The coefficient a_1 of the Laurent series expansion about z_i of the function $f(z)$ is referred to as the* **residue** *of $f(z)$ at z_i, and is denoted $Res(f, z_i)$. Everywhere $f(z)$ is analytic, its residue is zero.*

[9] An example of a uniformly continuous function that is nowhere differentiable is the **Weierstrass function** $W(x) = \sum_{n=0}^{\infty} a^n \cos(b^n \pi x)$ for $0 < a < 1$ and an odd integer $b > (1 + 3\pi/2)/a$.

The above arguments lead directly to the following straightforward but powerful generalization of Cauchy's integral formula (Fact B.17):

Fact B.27 (Cauchy's residue theorem) *For any simply-connected domain D with boundary Γ taken counterclockwise around D, if $f(z)$ is analytic everywhere inside D and on Γ, except for a finite number of singular points $\{z_1, z_2, \ldots, z_p\}$ inside D (not on the boundary Γ), then*

$$\oint_\Gamma f(z)\,dz = 2\pi i \sum_{i=1}^{p} Res(f, z_i).$$

Cauchy's integral theorem (Fact B.16) and Cauchy's residue theorem (Fact B.27) are immensely useful for calculating the integral of functions around closed contours (in the complex plane) inside of which the function is analytic almost everywhere; such contour integrals are often virtually impossible to compute without using these convenient tools. [Note that the integral along certain portions of the closed contour of interest in such problems is often identically zero.] A representative and physically quite important example is given below.

Example B.1 *Proof of Bode's Integral Theorem*

Bode's integral theorem, in the form given in Fact 18.2, may now be proved. Consider the function

$$S(s) = 1/[1 + L(s)]$$

where $L(s)$ is a rational function with relative degree $n_r > 0$ and all poles in the LHP. Define

$$\kappa = \lim_{s\to\infty} sL(s);$$

note that κ is finite if $n_r = 1$, and zero if $n_r > 1$. Consider the contour integral

$$I_R = \oint_{C_R} \ln S(s)\,ds,$$

where C_R is a closed D-shaped contour consisting of two parts, a line segment $C_{R,1}$ that extends from $s = -iR$ to $s = iR$ on the imaginary axis, and an arc $C_{R,2}$ that forms a half-circle of radius R in the RHP. Note that

$$\begin{aligned} I_R &= \int_{C_{R,1}} \ln S(s)\,ds && + \int_{C_{R,2}} \ln S(s)\,ds \\ &= \int_{-R}^{R} \ln S(i\omega)\,i\,d\omega && - \int_{\pi/2}^{-\pi/2} \ln[1 + L(Re^{i\phi})]\,[Re^{i\phi}\,i\,d\phi] \\ &= i\int_{-R}^{R} [\ln|S(i\omega)| + i\angle S(i\omega)]\,d\omega + i\int_{-\pi/2}^{\pi/2} Re^{i\phi}\ln[1 + L(Re^{i\phi})]\,d\phi. \end{aligned}$$

Since all poles of $L(s)$ are in the LHP, it follows from Facts B.16, B.19, B.21, and B.24 that $I_R = 0$ for any $R > 0$. Thus, taking the limit of I_R as $R \to \infty$ and applying (B.71) gives

$$\begin{aligned} 0 = \lim_{R\to\infty} I_R &= 2i\int_0^\infty \ln|S(i\omega)|\,d\omega + i\int_{-\pi/2}^{\pi/2} \lim_{R\to\infty} Re^{i\phi}\ln[1 + L(Re^{i\phi})]\,d\omega \\ &= 2i\int_0^\infty \ln|S(i\omega)|\,d\omega + i\kappa\pi; \qquad \Rightarrow \qquad \int_0^\infty \ln|S(i\omega)|\,d\omega = -\kappa\pi/2. \end{aligned}$$

B.6.2 Conformal mappings

A complex function $w = f(z)$ [in this setting, considered as a **mapping** from all points in some **domain**, $z \in \Omega_z$, to the points in the corresponding **image** (a.k.a. **range**), $w \in \Omega_w$] is said to be **conformal** (that is, **angle-preserving**) if, given any two smooth curves C_z and D_z passing through a point $z_0 \in \Omega_z$ and mapping these curves into Ω_w via the mapping $w = f(z)$, both of the mapped curves, denoted C_w and D_w, are rotated by the same amount in the vicinity of the mapped point $w_0 = f(z_0)$, and thus C_w and D_w intersect at the same angle in Ω_w as do C_z and D_z in Ω_z. Due to the following fact, coupled with Fact B.18, complex analytic functions that are nonzero over a given domain Ω_z are immensely useful for generating such mappings.

Fact B.28 *If $f(z)$ is both holomorphic and nonzero for all z in an open subset Ω_z of the complex plane, then the mapping given by $f(z)$ is* **conformal**.

Proof: Let the curves C_z and D_z in the domain Ω_z be parameterized by the variables ε and δ in the vicinity of the point z_0 (that is, in the vicinity of $\varepsilon = 0$ and $\delta = 0$):

$$C_z: \;\; z = z_0 + c_1\varepsilon + c_2\varepsilon^2 + c_3\varepsilon^3 + \ldots, \qquad D_z: \;\; z = z_0 + d_1\delta + d_2\delta^2 + d_3\delta^3 + \ldots$$

Note that, in the above parameterizations, the *phase* of the complex numbers c_1 and d_1 represent the angles of the curves C_z and D_z, respectively, in the complex plane z at the point z_0. Noting the definition of the complex derivative in (B.86), the mappings of these curves into Ω_w are given by

$$C_w: \;\; w = f(z) = f(z_0 + c_1\varepsilon + \ldots) = f(z_0) + \tilde{c}_1\varepsilon + \ldots \quad \text{where} \quad \tilde{c}_1 = c_1 \left.\frac{df}{dz}\right|_{z=z_0},$$

$$D_w: \;\; w = f(z) = f(z_0 + d_1\delta + \ldots) = f(z_0) + \tilde{d}_1\delta + \ldots \quad \text{where} \quad \tilde{d}_1 = d_1 \left.\frac{df}{dz}\right|_{z=z_0}.$$

The phase of $\tilde{c}_1$ and $\tilde{d}_1$ represent the angles of the curves C_w and D_w, respectively, in the complex plane w at the point $w_0 = f(z_0)$; note that the phases of $\tilde{c}_1$ and $\tilde{d}_1$ are simply the phases of c_1 and d_1 plus a rotation given by the phase of $(df/dz)_{z=z_0}$. □

B.6.2.1 Simple orthogonal grids and power laws

From high school geometry, the reader is probably already familiar with several easily-constructed locally-orthogonal grids, such as those implemented in Algorithm B.4 and depicted in Figure B.7. Note that, writing $z = \phi + \mathrm{i}\psi$ and $w = f(z) = re^{\mathrm{i}\theta}$, the **power law**

$$z = w^n \quad \Leftrightarrow \quad w = z^{1/n} \quad \Leftrightarrow \quad \left\{ \begin{array}{l} \phi = r^n \cos(n\theta) \\ \psi = r^n \sin(n\theta) \end{array} \right\} \tag{B.95}$$

is itself, away from the origin, a simple nonzero conformal mapping[10] that maps curves of constant ϕ and curves of constant ψ in the z-plane, as depicted by the **Cartesian grid** in Figure B.7a, to the corresponding curves in the w-plane in Figures B.7c, e, f, and g for $n = 1/2$, 2, 3, and -1, respectively (see Exercise B.3).

[10] Note that, in the last form shown in (B.95), it is easy to determine an expression for $r(\theta)$ in the w-plane corresponding to curves of constant ϕ and curves of constant ψ.

Algorithm B.4: A simple code to generate a few convenient locally-orthogonal grids.

View
Test

```
function z=OrthGrid(II,JJ,type,g,c0x,c1x,c0y,c1y)
% Generate a few convenient locally-orthogonal 2D grids.
x=Stretch1DMesh([0:1/(II-1):1],'p',0,1,c0x,c1x);
y=Stretch1DMesh([0:1/(JJ-1):1],'p',0,1,c0y,c1y);
switch type % Set up the grid in the z plane
  case 'Cartesian'
    for I=1:II, for J=1:JJ, z(I,J)=g.x0+x(I)*g.x1+i*y(J)*g.y1; end, end
  case 'ConfocalParabola'
    for I=1:II, for J=1:JJ, z(I,J)=x(I)*g.x1+i*y(J)*g.y1; end, end, z=z.^2+g.x0;
  case 'EllipseHyperbola'
    for I=1:II/2, a=2*x(II/2+I)-1; for J=1:JJ, b=y(J)*g.y1;
      if a==1, zr=sqrt(1+b^2); zi=0; else
        zr=sqrt((b^2+(1-a^2))/(b^2/(1+b^2)+(1-a^2)/a^2)); zi=sqrt(b^2-zr^2*b^2/(1+b^2));
      end, z(II/2+I,J)=zr+i*zi; z(II/2+1-I,J)=-zr+i*zi;
    end, end
end
end % function OrthGrid
```

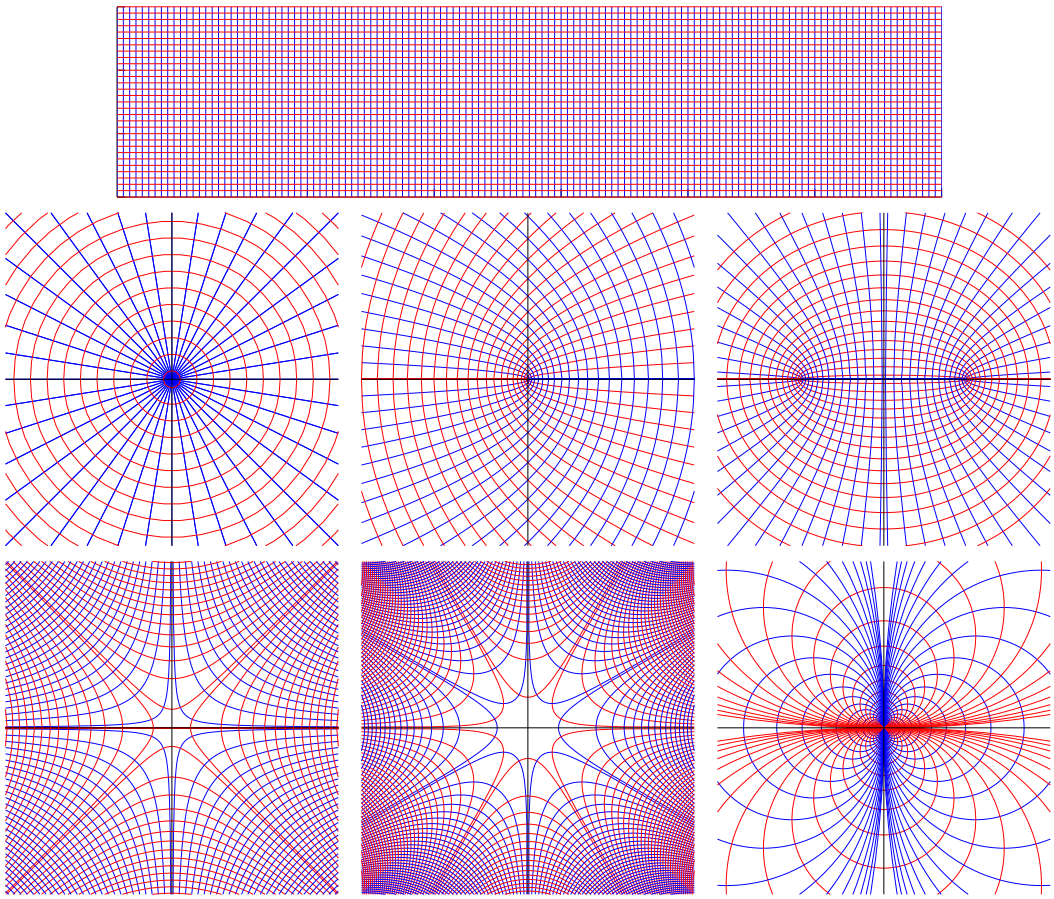


Figure B.7: Some locally-orthogonal grids: (a) Cartesian; (b) circles & rays; (c) sets of confocal parabolae with opposite curvature; (d) sets of confocal ellipses & hyperbolae; (e) sets of hyperbolae, with $\pm 45^\circ$ asymptotes, rotated 45° from each other; (f) similar to d, but with third-order curvature and $\pm 30^\circ$ asymptotes, rotated 30° from each other; and (g) sets of circles tangent to the real or imaginary axis at the origin. Portions of such grids may be mapped conformally (see §B.6.2.2-B.6.2.3) to make other locally-orthogonal grids.

Algorithm B.5: A conformal map of a Cartesian grid in the upper-half plane to the region above a step.

View

```
% script <a href="matlab:CMGridTest">CMGridTest</a>
% Compute a conformal mapping from a Cartesian grid in the upper half-plane to the region
% above a unit step.
clear; close all; g.x0=-9; g.x1=13; g.y1=3; II=131; JJ=31;
z=OrthGrid(II,JJ,'Cartesian',g,0,0,0,0); Plot2DMesh(z,1,II,JJ)
w=(sqrt(z-1).*sqrt(z+1)+acosh(z))/pi;     Plot2DMesh(w,2,II,JJ)
% end script CMGridTest
```

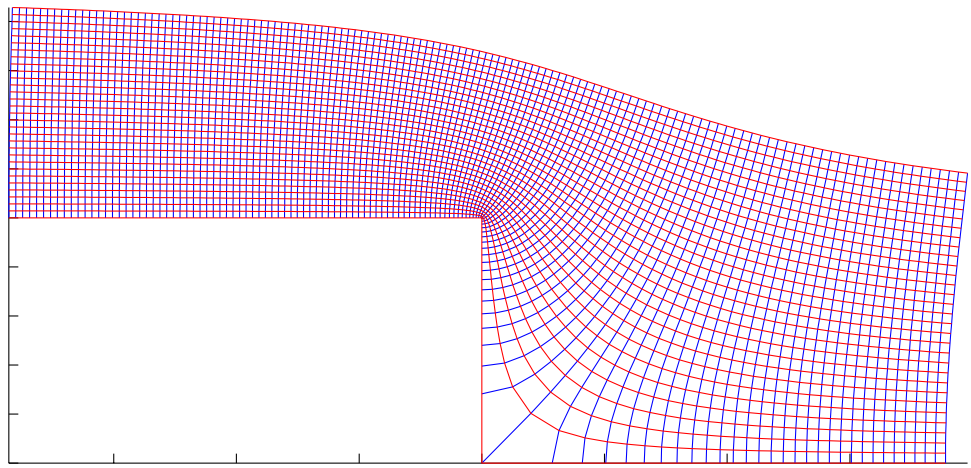


Figure B.8: A conformal mapping of a Cartesian grid in the upper-half plane (see Figure B.7a) to the region above a step. The red lines may be interpreted as the streamlines of a 2D potential flow, and the blue lines may be interpreted as equal-spaced velocity potential lines. In this interpretation, the flow travels in a path parallel to the streamlines and moves at a rate inversely proportional to the distance between the potential lines (i.e., fast near the outside corner, and slow near the inside corner); the same picture is valid for both flow from left to right and flow from right to left. As a consequence of the incompressibility condition (B.88b), the streamlines get farther apart wherever the flow slows down; thus, the aspect ratio of each of the cells formed by adjacent streamlines and adjacent potential lines is unity (actually, "nearly" unity in any finite realization).

B.6.2.2 A conformal mapping from the upper-half plane to the region above a unit step

Conformal mappings are useful for the generation of more general locally-orthogonal structured 2D grids than those depicted in Figure B.7—which, in turn, are valuable for accurate finite-difference computations (§11) in addition to the computation of 2D potential flow solutions (§B.6.1). Such locally-orthogonal grids may be created by first generating a simple orthogonal grid (such as one of those depicted Figure B.7) in the domain Ω_z, then mapping a portion of this grid via one or more[11,12] nonzero complex analytic function(s).

As just one example, consider the simple function

$$w = \left[\sqrt{z-1}\,\sqrt{z+1} + \operatorname{acosh}(z)\right]/\pi,$$

which is nonzero and complex analytic for all z in the upper half plane, and maps to the region above a unit step, as implemented in Algorithm B.5 and illustrated in Figure B.8. Noting (B.88) and the surrounding discussion, the resulting families of locally-orthogonal curves may be interpreted as lines of constant velocity potential and lines of constant streamfunction (a.k.a. **streamlines**) in a 2D potential flow.

[11] Note that, if $w = f(z)$ is analytic and $s = g(w)$ is analytic, then the **composite function** $s = g(f(z))$ is also analytic.

[12] A wide range of nonzero complex analytic functions are available and, along with their domain & image, tabulated on the web.

B.6.2.3 The Schwartz-Christoffel transformation

Perhaps the most flexible conformal mapping is given by the **Schwartz-Christoffel transformation**, which may be used to map conformally from the upper half of the z-plane onto a region in the w-plane bounded by a straight line with n corners. This mapping may be defined in derivative form as follows:

$$\frac{dw}{dz} = f'(z) = M \prod_{i=1}^{n} (z - x_i)^{\alpha_i/\pi} \qquad \text{with} \qquad \alpha_{sum} = \sum_{i=1}^{n} \alpha_i, \tag{B.96}$$

where the points x_1 to x_n lie (in order) on the real axis in the z-plane, and map to corresponding corners c_1 to c_n in the w-plane, with corresponding exterior angles α_1 to α_n, respectively. If $\alpha_{sum} = 2\pi$, the upper half of the z-plane is mapped to the *exterior* of a closed polygon in the w-plane, and the points c_i are enumerated around this polygon in *clockwise* order (Figure B.9a). If $\alpha_{sum} = -2\pi$, the upper half of the z-plane is mapped to the *interior* of a closed polygon in the w-plane, and the points c_i are enumerated around this polygon in *counterclockwise* order (Figure B.9b). If $-\pi \le \alpha_{sum} \le \pi$, the upper half of the z-plane is mapped to an open region in the w-plane that extends to infinity (Figure B.9c).

A second-order-accurate approximation of (B.96), which is valid even near the points x_i in the z-plane (and the corresponding corners c_i in the w-plane), may be written as follows (see Exercise B.4):

$$w_{m+1} - w_m = \frac{M}{(z_{m+1} - z_m)^{n-1}} \prod_{i=1}^{n} \left[\left(\frac{(z - x_i)^{1+\alpha_i/\pi}}{1 + \alpha_i/\pi} \right)_{z_m}^{z_{m+1}} \right]. \tag{B.97}$$

Given specified corners c_1 to c_n in the w-plane, with corresponding exterior angles α_1 to α_n, the appropriate locations of the points x_1 to x_n in the z-plane, as well as the complex constant M, are initially unknown; however, two of these points (denoted x_{i_1} and x_{i_2} where $1 \le i_1 < i_2 \le n$) may, without loss of generality, be specified arbitrarily. The following straightforward algorithm (Davis 1979) iteratively refines both M and the location of the remaining $n-2$ points x_i, for $i \notin \{i_1, i_2\}$, until the transformation is adjusted to map the points x_i in the z-plane onto the specified corners c_i in the w-plane:

(a) Fix two of the points x_i; for example, set $x_{i_1} = -1$ and $x_{i_2} = 1$. Assign initial values along the real axis in the z-plane (in order) for the remaining $n-2$ points, denoted $x_{i,guess}$, for $i \notin \{i_1, i_2\}$.

(b) Given x_{i_1} and x_{i_2}, the assumed values of $x_{i,guess}$ for $i \notin \{i_1, i_2\}$, and the angles α_1 to α_n, and initially taking $M = 1$, march (B.97) from $z = x_{i_1}$ to $z = x_{i_2}$, and then compute the (complex) value of M necessary to move correspondingly from c_{i_1} and c_{i_2} in the w-plane.

(c) Using α_1 to α_n, x_{i_1} and x_{i_2}, the assumed values of $x_{i,guess}$ for $i \notin \{i_1, i_2\}$, and the value of M determined in step b, compute the locations in the w-plane, denoted $c_{i,guess}$, corresponding to the various $x_{i,guess}$, for $i \notin \{i_1, i_2\}$, by marching (B.97) from $z = x_{i_1}$ appropriately (note: due to the selection of M in step b, we get $c_{i_2,guess} = c_{i_2}$ by construction).

(d) Starting from $x_{i_1,new} = x_{i_1}$ and initially taking $K = 1$, compute values of the $x_{i,new}$ for $i \ne i_1$ from the following:

$$\frac{x_{i,new} - x_{i-1,new}}{x_{i,guess} - x_{i-1,guess}} = K \left\| \frac{c_i - c_{i-1}}{c_{i,guess} - c_{i-1,guess}} \right\|, \tag{B.98}$$

then compute the (real) value of K necessary to give $x_{i_2,new} = x_{i_2}$ in (B.98).

(e) Recompute the $x_{i,new}$ in (B.98) with the corrected value of K selected in step d, call these corrected values $x_{i,guess}$ for $i \notin \{i_1, i_2\}$, and repeat from step b until the $c_{i,guess}$ converge to the specified c_i.

Once the points x_i and the constant M are determined using the above algorithm, (B.97) may then be marched appropriately to map any locally-orthogonal grid in the upper half of the z-plane (see Figure B.7) onto the specified region, as illustrated in Figure B.10 and implemented in Algorithm B.6. A wide variety of possible grid stretching strategies (see Algorithm 8.2) may be applied when generating the orthogonal grid in the z plane (see Algorithm B.4) in order to cluster gridpoints appropriately in regions of interest in the w plane.

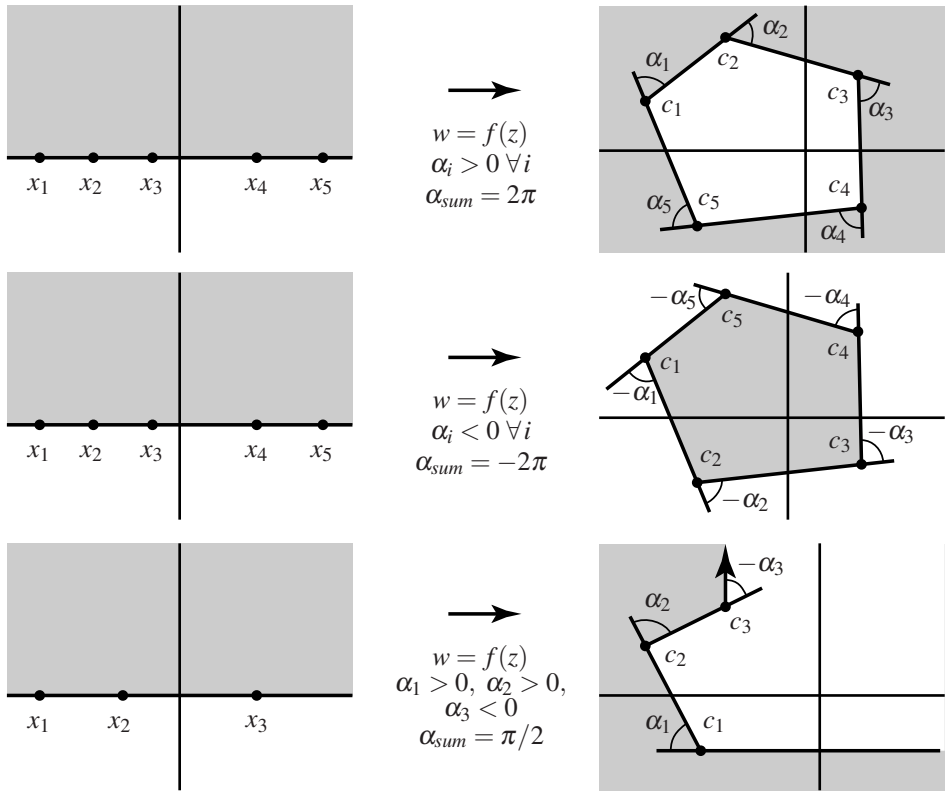


Figure B.9: The Schwartz-Christoffel transformation from the upper half of the z-plane to: (a) the outside of a closed polygon in the w-plane for $\alpha_{sum} = 2\pi$, (b) the inside of a closed polygon in the w-plane for $\alpha_{sum} = -2\pi$, and (c) an open region that extends to infinity in the w-plane for $-\pi \leq \alpha_{sum} \leq \pi$.

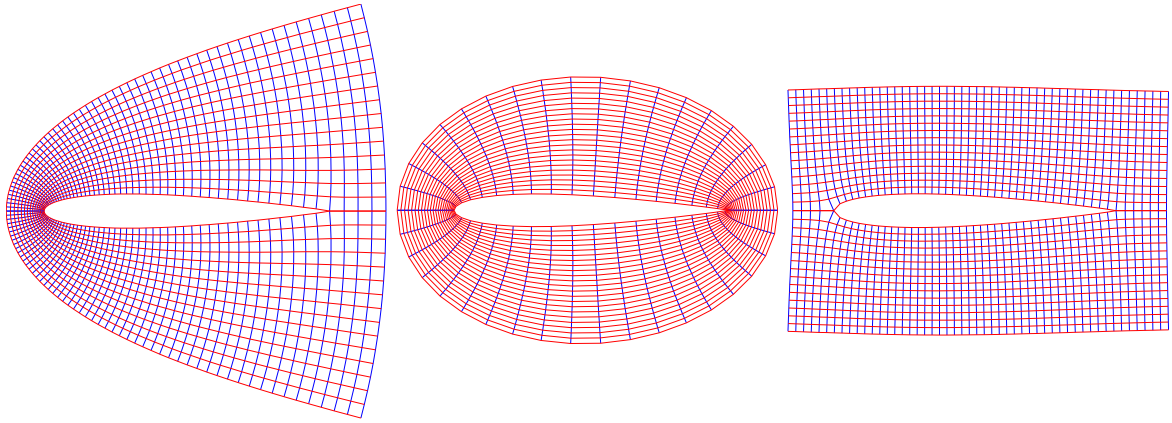


Figure B.10: A conformal map of a confocal parabola grid (Figure B.7b), a confocal ellipse/hyperbola grid (Figure B.7c), and a Cartesian grid (Figure B.8a) in the upper half of the z-plane to the semi-infinite domain in the w-plane above a curve defined by the real axis together with 65 points selected on the upper surface of a NACA0012 airfoil, thereby (respectively) generating (upon subsequent reflection of the grid so generated to the domain below the airfoil) what are commonly referred to as (a) a C-grid [with C-shaped gridlines], (b) an O-grid [with O-shaped gridlines], and (c) an H-grid [that is, a grid which is neither a C-grid nor an O-grid].

Algorithm B.6: The Schwartz-Christoffel transformation.

View
Test

```
function w=SCGrid(c,n,cin,cout,i1,i2,x,steps,z,II,JJ)
% Optimize the x_i and M of the Schwartz-Christoffel transformation in order to map onto
% the specified corners c_i, then map (conformally) the specified z grid to the w plane.
d=[c, cout]-[cin, c]; d(:)=-atan2(imag(d(:)),real(d(:))); cg(i1)=c(i1); xn(i1)=x(i1);
for i=1:n, a(i)=d(i+1)-d(i); a(i)=rem(a(i)/pi+3,2)-1; end % (-1<=a(i)<1, alpha(i)=a(i)*pi)
for iteration=1:50     % OPTIMIZE THE x_i (FOR i NOT EQUAL TO i1 OR i2) AND M
  % step (b): compute M
  dw=0; for i=i1+1:i2, dw=dw+MarchSC(x(i-1),x(i),n,1,x,a,steps); end, M=(c(i2)-c(i1))/dw;
  % step (c): compute the c_guess
  for i=i1+1:n,     cg(i)=cg(i-1)+MarchSC(x(i-1),x(i),n,M,x,a,steps); end
  for i=i1-1:-1:1, cg(i)=cg(i+1)+MarchSC(x(i+1),x(i),n,M,x,a,steps); end
  % step (d): compute the new K
  for i=i1+1:i2,    xn(i)=xn(i-1)+Adjustx(i,1,c,cg,x); end, K=(x(i2)-x(i1))/(xn(i2)-xn(i1));
  % step (e): adjust the x
  for i=i1+1:n,     xn(i)=xn(i-1)+Adjustx(i,K,c,cg,x);    end
  for i=i1-1:-1:1, xn(i)=xn(i+1)-Adjustx(i+1,K,c,cg,x); end, x=xn;
end
for i=1:II % USING THE OPTIMIZED x_i AND M, MAP AN ORTHOGONAL GRID FROM z-PLANE TO w-PLANE
   if i==1, w(i,1)=0.0; else, w(i,1)=w(i-1,1)+MarchSC(z(i-1,1),z(i,1),n,M,x,a,steps); end
   for j=2:JJ, w(i,j)=w(i,j-1)+MarchSC(z(i,j-1),z(i,j),n,M,x,a,steps); end
end
end % function SCGrid
%%%
```

s^0 in the numerator of the LHS equal to the corresponding coefficients in the numerator on the RHS. This results in n linear equations for the n unknown coefficients, which is easily solved using the techniques of §2.

In the case that a root p_i of $a(s)$ is not repeated (even if other roots are), an easy direct formula may be used instead to determine the coefficient $d_{i,1}$. Writing $Y(s) = \frac{d_{i,1}}{(s-p_i)} + H_i(s)$, where [as in (B.99b)] it is easily seen that $H_i(p_i)$ is finite, and multiplying by $(s-p_i)$ gives

$$(s-p_i)Y(s) = d_{i,1} + (s-p_i)H_i(s); \tag{B.100a}$$

subsequently evaluating this expression at $s = p_i$, noting that $[(s-p_i)H_i(s)]_{s=p_i} = 0$, results immediately in

$$d_{i,1} = \Big[(s-p_i)Y(s)\Big]_{s=p_i}. \tag{B.100b}$$

In this expression, the factor of $(s-p_i)$ in the brackets exactly cancels the corresponding factor in the denominator of $Y(s)$; $d_{i,1}$ is then determined by evaluating the remaining factors of $Y(s)$ at $s = p_i$. This direct method of computing one of the coefficients of a partial fraction expansion is the key to understanding the simple method of constructing a Bode plot, as described in §17.5.1.

In the case that a root p_i of $a(s)$ is repeated $r > 1$ times, as illustrated in (B.99b), the corresponding coefficients, $d_{i,1}$ to $d_{i,r}$, may also be determined analytically. First, following the same approach as that used in (B.100a)-(B.100b), we multiply (B.99b) by $(s-p_i)^r$ and subsequently evaluate the resulting expression at $s = p_i$, noting that $[(s-p_i)^r H_i(s)]_{s=p_i} = 0$, resulting [cf. (B.100b)] in

$$d_{i,r} = \Big[(s-p_i)^r Y(s)\Big]_{s=p_i}. \tag{B.101}$$

Now assume that k coefficients $d_{i,r-k+1}$ to $d_{i,r}$ are already known for some k, where $0 \le k \le r-1$. To determine the next coefficient, $d_{i,r-k}$, multiply (B.99b) by $a(s)/(s-a)^{k+1}$ and rearrange to give

$$\frac{b(s)}{(s-p_i)^{k+1}} - \frac{a(s)}{(s-p_i)^{r+k+1}}\Big(d_{i,r} + d_{i,r-1}(s-p_i) + \ldots + d_{i,r-k+1}(s-p_i)^{k-1}\Big) =$$
$$\frac{a(s)}{(s-p_i)^{r+1}}d_{i,r-k} + \underbrace{\frac{a(s)}{(s-p_i)^r}\Big(d_{i,r-k-1} + \ldots + d_{i,1}(s-p_i)^{r-k-2}\Big) + H_i(s)\frac{a(s)}{(s-p_i)^{k+1}}}.$$

Define Γ as a counterclockwise circular contour centered at $s = p_i$ with sufficiently small radius that it doesn't contain or touch any of the other poles p_j for $j \neq i$. Taking the integral of the above equation over Γ, noting Fact B.16, that $a(s) = (s-p_i)^r a_i(s)$ where $a_i(p_i)$ is finite, and that the underbraced terms on the RHS of the above expression are analytic everywhere on and within Γ thus gives

$$d_{i,r-k}\oint_\Gamma \frac{a_i(s)}{s-p_i}\,dz = \oint_\Gamma \frac{b(s)}{(s-p_i)^{k+1}}\,dz - \oint_\Gamma \frac{a_i(s)}{(s-p_i)^{k+1}}\Big(d_{i,r} + d_{i,r-1}(s-p_i) + \ldots + d_{i,r-k+1}(s-p_i)^{k-1}\Big)\,dz.$$

Applying Fact B.17 to the LHS and Fact B.25 to the RHS thus leads immediately to

$$d_{i,r-k}\,a_i(p_i) = \frac{1}{k!}\frac{d}{ds^k}\Big[b(s) - a_i(s)\Big(d_{i,r} + d_{i,r-1}(s-p_i) + \ldots + d_{i,r-k+1}(s-p_i)^{k-1}\Big)\Big]_{s=p_i}.$$

Finally, applying Leibnitz's rule (B.36) to the derivative of a product, denoting $b^{(k)}(s)$ as the k'th derivative of $b(s)$ with respect to its argument, and noting that ${}_kC_\ell = k!/[\ell!\,(k-\ell)!]$ gives

$$d_{i,r-k} = \frac{1}{a_i(p_i)}\Big[\frac{b^{(k)}(s)}{k!} - \sum_{\ell=0}^{k}\frac{a_i^{(k-\ell)}(s)}{\ell!\,(k-\ell)!}\Big(d_{i,r} + d_{i,r-1}(s-p_i) + \ldots + d_{i,r-k+1}(s-p_i)^{k-1}\Big)^{(\ell)}\Big]_{s=p_i}.$$

Algorithm B.7: Perform a partial fraction expansion, $Y(s) = \frac{b(s)}{a(s)} = d_1/(s-p_1)^{k_1} + d_2/(s-p_2)^{k_2} + \ldots$

View
Test

```
function [p,d,k,n]=PartialFractionExpansion(num,den,eps)
% Compute {p,d,k,n} so that Y(s)=num(s)/den(s)=d(1)/(s-p(1))^k(1) +...+ d(n)/(s-p(n))^k(n),
% where order(num)<=order(den) and eps is tolerance when finding repeated roots.
n=length(den)-1; m=length(num)-1; flag=0; if n<1, p=1; k=0; d=num/den; n=1; return, end
if m==n, flag=1; [div,rem]=PolyDiv(num,den); m=m-1; else, rem=num; end
k=ones(n,1); p=roots(den); if n>1, p=SortComplex(p); end, if nargin<3, eps=1e-3; end
for i=1:n-1, if abs(p(i+1)-p(i))<eps, k(i+1)=k(i)+1; end, end, k(n+1,1)=0;
for i=n:-1:1
  if k(i)>=k(i+1), r=k(i); a=1;
    for j=1:i-k(i), a=PolyConv(a,[1 -p(j)]); end
    for j=i+1:n,    a=PolyConv(a,[1 -p(j)]); end
    for j=1:k(i)-1, ad{j}=PolyDiff(a,j); end
  end
  q=r-k(i); d(i,1)=PolyVal(PolyDiff(rem,q),p(i))/Fac(q);
  for j=q:-1:1, d(i)=d(i)-d(i+j)*PolyVal(ad{j},p(i))/Fac(j); end
  d(i)=d(i)/PolyVal(a,p(i));
end, if ~flag, k=k(1:n); else, n=n+1; p(n,1)=1; d(n,1)=div; end
end % function PartialFractionExpansion
```

Noting that

$$\left[\frac{d}{ds^\ell}(s-p_i)^h\right]_{s=p_i} = \begin{cases} 0 & \ell \neq h \\ \ell! & \ell = h \end{cases}$$

reduces this expression to

$$d_{i,r-k} = \frac{1}{a_i(p_i)}\left[\frac{b^{(k)}(p_i)}{k!} - \sum_{\ell=0}^{k-1} \frac{a_i^{(k-\ell)}(s)}{(k-\ell)!}\right], \tag{B.102}$$

as implemented in Algorithm B.7; in particular,

$$d_{i,r} = [b(p_i)/0!]/a_i(p_i) \qquad \text{[note: consistent with (B.101)]} \tag{B.103a}$$

$$d_{i,r-1} = [b^{(1)}(p_i)/1! - d_{i,r}a_i^{(1)}(p_i)/1!]/a_i(p_i), \tag{B.103b}$$

$$d_{i,r-2} = [b^{(2)}(p_i)/2! - d_{i,r}a_i^{(2)}(p_i)/2! - d_{i,r-1}a_i^{(1)}(p_i)/1!]/a_i(p_i), \tag{B.103c}$$

$$d_{i,r-3} = [b^{(3)}(p_i)/3! - d_{i,r}a_i^{(3)}(p_i)/3! - d_{i,r-1}a_i^{(2)}(p_i)/2! - d_{i,r-2}a_i^{(1)}(p_i)/1!]/a_i(p_i), \quad \text{etc.} \tag{B.103d}$$

This is but one demonstration of the extensive practical utility of the theory of complex analysis of §B.6.1.

Exercises

Exercise B.1 Modify Algorithm 4.3 to calculate the roots of quadratic and cubic polynomials in closed form (rather than iterateively), using the formulae presented in this appendix. Extra credit: look up the corresponding analytic formulae for the roots of a quartic polynomial, and implement that as well.

Exercise B.2 **(a)** Draw the line segment $[-1,1]$ in the complex plane ξ. Considering the mapping $\xi = 1/\zeta$, draw the corresponding contour in the complex plane ζ. **(b)** Now considering the stereographic projection $\zeta = (x+\mathrm{i}y)/(1-z)$ where $x^2+y^2+z^2=1$, compute the $\{x,y,z\}$ coordinates of the beginning point, the midpoint, and the end point on the unit sphere of the contour in the complex plane ζ considered in part (a). Sketch this contour on the unit sphere and discuss.

Exercise B.3 Modify Algorithm B.5 to incorporate the power law discussed in §B.6.2.1, and generate the families of curves illustrated in Figures B.7c, e, f, and g by taking $n = 1/2$, 2, 3, and -1, respectively.

Exercise B.4 Verify that (B.96) is a second-order accurate approximation of (B.97), even near the points x_i in the z-plane. Hint: see Davis (1979).

References

Adams, D (1980) *The Restaurant at the End of the Universe*, Pan Books.[13]
Davis, RT (1979) Numerical Methods for Coordinate Generation Based on Schwarz-Christoffel Transformations, *AIAA Paper 79-1463*.

Fine, B, & Rosenberger, G (1997) *The Fundamental Theorem of Algebra*, Springer-Verlag.

Henrici, P (1974) *Applied and computational complex analysis, Volume 1*, Wiley.

Meinsma, G (1995) Elementary proof of the Routh-Hurwitz test. *Systems & Control Letters*, **25**, 237-242.

Santos, JC (2007) The fundamental theorem of algebra deduced from elementary calculus. *The Mathematical Gazette*, **91**, 302-303.

Schey, HM (2005) *Div, grad, curl, and all that: an informal text on vector calculus*. WW Norton & Co.

[13] **Circular Reasoning**: see explanation of **Reasoning, Circular** in footnote on Page v.