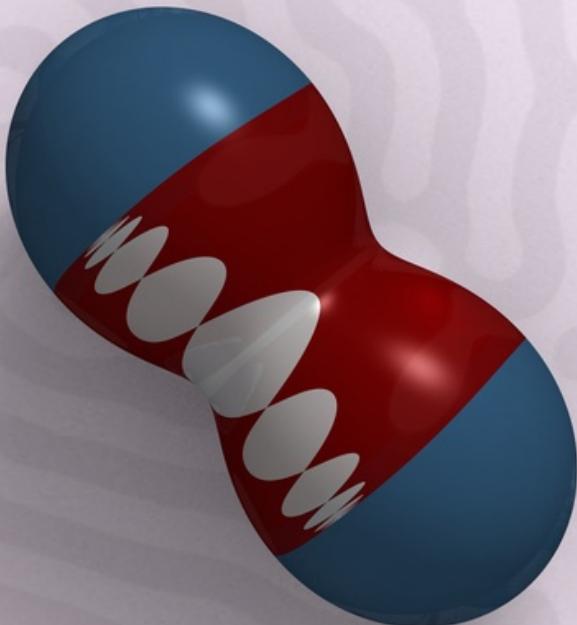


NUMERICAL METHODS FOR SCIENTIFIC COMPUTING

Second Edition



KYLE NOVAK

NUMERICAL METHODS FOR SCIENTIFIC COMPUTING

NUMERICAL METHODS FOR SCIENTIFIC COMPUTING

THE DEFINITIVE MANUAL FOR MATH GEEKS

Second Edition

KYLE NOVAK



Copyright ©2022 by Kyle A. Novak

ISBN 979-8-9854218-0-4

Library of Congress Control Number: 2021924724

Second Edition, March 2022

Version 2-220322

Cover illustration: The Rayleigh quotient with its three basins of attraction for a symmetric three-dimensional matrix. The Rayleigh quotient method, an iterative technique for finding eigenvalue–eigenvector pairs, is developed on page 98. The solution to the Swift–Hohenberg equation with random initial conditions, which models Rayleigh–Bénard convection, is discussed on page 457.

Pardon the dust: This book is in continuous development, published using an agile mindset of making it available sooner so that it can be of use and providing steady improvements. Great efforts have been made to identify and correct mistakes. Nonetheless to borrow a Piet Hein grook: “The road to wisdom? Well, it’s plain and simple to express: err and err and err again but less and less and less.” Your comments, suggestions, corrections, and criticisms are appreciated: kyle.novak@equalsharepress.com. The latest digital version of this book is available at <https://www.equalsharepress.com/media/NMFSC.pdf>.

Equal Share Press
1919 Clarendon Boulevard
Arlington, Virginia 22201
www.equalsharepress.com



Your feedback is important. Use this QR code to leave a comment, make a suggestion, or report errata.

Dedicated to Laura

$$\begin{aligned}(x^2 + y^2 - 1)^3 - x^2 y^3 &= 0 \\ (x^2 + y^2)^2 - 2(x^2 - y^2) &= 0\end{aligned}$$

Contents

Preface *xiii*

Introduction *xix*

Numerical Linear Algebra

1 A Review of Linear Algebra 3

- 1.1 Vector spaces 3
- 1.2 Eigenspaces 6
- 1.3 Measuring vectors and matrices 13
- 1.4 Stability 21
- 1.5 Geometric interpretation of linear algebra 24
- 1.6 Exercises 25

2 Direct Methods for Linear Systems 29

- 2.1 Gaussian elimination 30
- 2.2 Cholesky decomposition 35
- 2.3 Linear programming and the simplex method 37
- 2.4 Sparse matrices 45
- 2.5 Exercises 55

3 Inconsistent Systems 57

3.1	Overdetermined systems	57
3.2	Underdetermined systems	67
3.3	Singular value decomposition	70
3.4	Non-negative matrix factorization	83
3.5	Exercises	84
4	Computing Eigenvalues	89
4.1	Eigenvalue sensitivity and estimation	89
4.2	The power method	93
4.3	The QR method	101
4.4	Implicit QR	107
4.5	Getting the eigenvectors	109
4.6	Arnoldi method	110
4.7	Singular value decomposition	115
4.8	Exercises	117
5	Iterative Methods for Linear Systems	119
5.1	Jacobi and Gauss–Seidel methods	121
5.2	Successive over relaxation	124
5.3	Multigrid method	127
5.4	Solving the minimization problem	130
5.5	Krylov methods	137
5.6	Exercises	141
6	The Fast Fourier Transform	143
6.1	Discrete Fourier transform	143
6.2	Cooley–Tukey algorithm	145
6.3	Toeplitz and circulant matrices	149
6.4	Bluestein and Rader factorization	153
6.5	Applications	157
6.6	Exercises	162

Numerical Analysis

7	Preliminaries	167
7.1	Well-behaved functions	167
7.2	Well-posed problems	170
7.3	Well-posed methods	173
7.4	Floating-point arithmetic	177

7.5	Computational error	183
7.6	Exercises	186
8	Solutions to Nonlinear Equations	189
8.1	Bisection method	189
8.2	Newton's method	190
8.3	Fixed-point iterations	194
8.4	Dynamical systems	198
8.5	Roots of polynomials	204
8.6	Systems of nonlinear equations	208
8.7	Homotopy continuation	212
8.8	Finding a minimum (or maximum)	215
8.9	Exercises	218
9	Interpolation	221
9.1	Polynomial interpolation	222
9.2	How good is polynomial interpolation?	228
9.3	Splines	231
9.4	Bézier curves	238
9.5	Exercises	242
10	Approximating Functions	245
10.1	Least squares approximation	246
10.2	Orthonormal systems	249
10.3	Fourier polynomials	252
10.4	Chebyshev polynomials	255
10.5	Wavelets	259
10.6	Neural networks	272
10.7	Data fitting	277
10.8	Exercises	288
11	Differentiation and Integration	291
11.1	Numerical differentiation	291
11.2	Automatic differentiation	295
11.3	Newton–Cotes quadrature	299
11.4	Gaussian quadrature	307
11.5	Monte Carlo integration	315
11.6	Exercises	321

Numerical Differential Equations

12 Ordinary Differential Equations 327

- 12.1 Well-posed problems 327
- 12.2 Single-step methods 329
- 12.3 Lax equivalence theorem 337
- 12.4 Multistep methods 339
- 12.5 Runge–Kutta methods 347
- 12.6 Nonlinear equations 356
- 12.7 Stiff equations 357
- 12.8 Splitting methods 360
- 12.9 Symplectic integrators 364
- 12.10 Practical implementation 366
- 12.11 Exercises 369

13 Parabolic Equations 373

- 13.1 Method of lines 375
- 13.2 Von Neumann analysis 380
- 13.3 Higher-dimensional methods 385
- 13.4 Nonlinear diffusion equations 388
- 13.5 Exercises 391

14 Hyperbolic Equations 395

- 14.1 Linear hyperbolic equations 395
- 14.2 Methods for linear hyperbolic equations 396
- 14.3 Numerical diffusion and dispersion 402
- 14.4 Linear hyperbolic systems 404
- 14.5 Hyperbolic systems of conservation laws 414
- 14.6 Methods for nonlinear hyperbolic systems 418
- 14.7 Exercises 423

15 Elliptic Equations 425

- 15.1 A one-dimensional example 425
- 15.2 A two-dimensional example 428
- 15.3 Stability and convergence 433
- 15.4 Time-dependent problems 435
- 15.5 Practical implementation 436
- 15.6 Exercises 439

16 Fourier Spectral Methods 441

16.1 Discrete Fourier transform 441

16.2 Nonlinear stiff equations 448

16.3 Incompressible Navier–Stokes equation 452

16.4 Exercises 456

Back Matter**A Solutions 461**

A.1 Linear algebra 461

A.2 Analysis 483

A.3 Differential equations 503

B Computing in Python and Matlab 535

B.1 Scientific programming languages 535

B.2 Python 541

B.3 Matlab 590

References 637**Index 651****Julia Index 665****Python Index 669****Matlab Index 671**

Preface

This book was born out of a set of lecture notes I wrote for a sequence of three numerical methods courses taught at the Air Force Institute of Technology. The courses Numerical Linear Algebra, Numerical Analysis, and Numerical Methods for Partial Differential Equations were taken primarily by mathematics, physics, and engineering graduate students. My goals in these courses were to present the foundational principles and essential tools of scientific computing, provide practical applications of the principles being taught, and generate interest in the topic. These notes were themselves inspired by lectures from a two-sequence numerical analysis course taught by my doctoral advisor Shi Jin at the University of Wisconsin.

The purpose of my notes was first to guide the lectures and discussion, and second, to provide students with a bridge to more rigorous and complete, but also more mathematically dense textbooks and references. To this end, the notes acted as a summary to help students learn the key mathematical ideas and explain the principles intuitively. I favored simple picture proofs and explanations over more rigorous but abstruse analytic derivations. Students who wanted the details could find them in any number of other numerical mathematics texts.

In moving from a set of supplementary lecture notes to a stand-alone book, I wondered whether to make them into a handbook, a guidebook, or a textbook. My goal in writing and subsequently revising this book was to provide a concise treatment of the core ideas, algorithms, proofs, and pitfalls of numerical methods for scientific computing. I aimed to present the topics in a way that might be consumed in bits and pieces—a handbook. I wanted them weaved together into a grand mathematical journey, with enough detail to elicit an occasional “aha” but not so much as to become overbearing—a guidebook. Finally, I wanted to present the ideas using a pedagogical framework to help anyone with a good understanding of multivariate calculus and linear algebra learn valuable mathematical skills—a textbook. Ultimately, I decided on a tongue-in-cheek

“definitive manual for math geeks.” To be clear, it’s definitely not definitive. And, it need not be. If a person knows the right questions to ask, they can find a dozen answers through a Google search, a Github tutorial, a Wikipedia article, a Stack Exchange snippet, or a YouTube video.

When I published the first edition of this book several years ago, I did so with an agile development mindset. Get it out quickly with minimal bugs so that it can be of use to others. Then iterate and improve. Make it affordable. When textbooks often sell for over a hundred dollars, keep the print version under twenty and the electronic one free. Publish it independently to be able to make rapid improvements and to keep costs down. While I had moderate aspirations, the Just Barely Good Enough (JBGE) first edition was panned. A reviewer named Ben summarized it on Goodreads: “Typos, a terrible index. Pretty sure it was self published. It does a good job as a primer, but functionally speaking, it’s a terrible textbook.” Another reviewer who goes by Nineball stated on Amazon: “This text is riddled with typos and errors. You can tell the author had great objectives for making concise book for numerical methods, however the number of errors significantly detracts from the message and provides a substantial barrier to understanding.” This JBGE++ edition fixes hundreds of typos, mistakes, and coding glitches. Still, there are undoubtedly errors I missed and ones I inadvertently introduced during revision. Truth be told, I’d rather spend my time watching *The Great British Bake Off* with my beautiful wife than hunting down every last typo. I also learned that designing a good index is a real challenge. The index in this book is still a work in progress. I apologize to anyone who struggled with the JBGE edition. And I apologize in advance for any mistakes that appear in this one. I understand that Donald Knuth would personally send a check for \$2.56 to anyone who found mistakes in his books. I can’t do that, but if you ever find yourself in my town, I’ll happily buy you a beer to ease your frustration. That offer goes out to you especially, Ben and Nineball.

Paul Halmos once said about writing mathematics to “pretend that you are explaining the subject to a friend on a long walk in the woods, with no paper available.” I wonder what he would have said about writing about numerical methods. Would he have said “with no computer available?” I believe so—understanding Newton’s method, for instance, has more to do with the Banach fixed-point theorem than addition assignment operators. While a mathematics book should be agnostic about scientific programming languages, snippets of code can help explain the underlying theory and make it more practical. In the first edition of this book, I focused entirely on Matlab. With this edition, I’ve embraced Julia. To help understand the switch, one need only consider that Matlab was first released forty years ago, placing its development closer to the 1940s ENIAC than today’s grammar of graphics, dataframes, reactive notebooks, and cloud computing. Python’s NumPy and SciPy are twenty years old. Julia is barely ten. Still, Matlab and Python are both immensely important languages. Michael Mol’s Rosetta Code (<http://www.rosettacode.org>), a wiki he describes

as a “program chrestomathy,” inspired me to include both in the back matter.

In designing this book, I repurposed the first five aphorisms of Tim Peter’s “The Zen of Python.” *Beautiful is better than ugly*. I’ve chosen the printed page because it provides the most expressive mathematical typography and consistency in notation. I’ve rendered graphics in this book almost entirely within the L^AT_EX environment to maintain visual unity. *Explicit is better than implicit*. I’ve provided solutions to most exercises because they are invaluable to self-study. And I’ve including working Julia, Python, and Matlab code to encourage tinkering and exploration. All of the code is available as Jupyter notebooks. *Simple is better than complex*. I’ve kept the code to minimum working examples to enable the underlying mathematics to more readily be seen—something that mathematician Nick Trefethen has dubbed “ten-digit algorithms.” These are programs that are “ten digits, five seconds, and just one page.” Such programs must run fast enough to permit students to explore, iterate, adjust, and experiment. They must be concise enough to communicate the how the algorithm works. And, they must have scientific accuracy, i.e., at least ten digits. I’ve tried to keep unity in notation throughout the book wherever possible. I’ve favored intuitive explanations over rigorous ones that require unnecessary mathematical machinery. *Complex is better than complicated*. But, I’ve introduced heavy mathematical machinery when necessary. *Universal is better than specialized*. I’ve focused on the mathematics over the algorithm and the algorithm over the code.

Let me express a few words of gratitude. This book has been an ongoing struggle—at times demoralizing and annoying and at others meditative and therapeutic. I am most grateful for the support and patience of my mind-blowingly awesome wife. Thank you, Laura. I am grateful for the open-source community for developing tools such as tools Octave, R, Python, Julia, L^AT_EX, and Inkscape. I am grateful for the countless authors who volunteer answers on Wikipedia, StackExchange, Reddit, and GitHub. Finally, I am grateful for the tinkers and the teachers. The world is a better place because of people like you. This book project is me paying it forward.

KYLE A. NOVAK
Washington, D.C.
February 2022

About the Author

Kyle Novak is an applied mathematician, data scientist, and policy advisor with twenty-five years of experience in finding solutions to real problems. Kyle examined the national security threats of autonomous air systems while at the Air Force Research Laboratory, served as a cryptologic mathematician at the National Security Agency, taught graduate students at the Air Force Institute of Technology, and provided decision analysis to senior military leaders at the Pentagon. As a science and technology policy fellow at the U.S. Agency for International Development, Kyle explored the use of digital technologies and data science toward ending extreme poverty and promoting resilient, democratic societies. He subsequently served as an advisor on science and technology, national security, defense, and foreign policy in the U.S. Senate. In his most recent position at the National Institute of Justice, Kyle guides research on the application of artificial intelligence and mathematical modeling in criminal justice and policing.

Kyle Novak has been featured in the American Mathematical Society's *101 Careers in Mathematics*. His book *Special Functions of Mathematical Physics: A Tourist's Guidebook* explores the quintessential special functions that arise from solving differential equations and develops the mathematical tools and intuition for working with them. Kyle holds a PhD in mathematics from the University of Wisconsin–Madison.



Introduction

Scientific computing is the study and use of computers to solve scientific problems. Numerical methods are techniques designed to solve those problems efficiently and accurately, using numerical approximation. Numerical methods have been around for centuries. Indeed, some four thousand years ago, Babylonians used a technique for approximating square roots. And over two thousand years ago, Archimedes developed a method for approximating π by inscribing and circumscribing a circle with polygons. With the progress of mathematical thought, with the discovery of new scientific problems requiring novel and efficient approaches, and more recently with the proliferation of cheap and powerful computers, numerical methods have worked their way into all aspects of our lives, although mostly hidden from view.

Eighteenth-century mathematicians Joseph Raphson and Thomas Simpson used Newton's then recently invented Calculus to develop numerical methods for finding the zeros of functions. Their approaches are at the heart of gradient descent, making today's deep learning algorithms possible. In solving problems of heat transfer, nineteenth-century mathematician Joseph Fourier discovered that any continuous function could be written as an infinite series of sines and cosines. At the same time, Carl Friedrich Gauss invented though never published a numerical method to compute the coefficients of Fourier's series recursively. It wasn't until the 1960s that mathematicians James Cooley and John Tukey reinvented Gauss' fast Fourier transform, this time spurred by a Cold War necessity of detecting nuclear detonations. Computers themselves have enabled mathematical discovery. Shortly after World War I, French mathematicians Pierre Fatou and Gaston Julia developed a new field of mathematics that examined the dynamical structure of iterated complex functions. Sixty years later, Benoit Mandelbrot, a researcher at IBM, used computers to discover the intricate fractal worlds hidden in these simple recursive formulas.

Today, numerical methods are accelerating the convergence of different

scientific disciplines, in which artificial intelligence uses techniques of nonlinear optimization and dimensionality reduction to generate implicit solutions to complex systems. In the early nineteenth century, Ada Lovelace envisioned the first computer program to compute Bernoulli numbers for a hypothetical invention of Charles Babbage, an invention that wouldn't be built until almost a hundred years later. In the 1980s, physicist Richard Feynman postulated that solving some physics problems such as simulating quantum systems would require an entirely new kind of computer, a quantum computer. A decade later, mathematician Peter Shor developed an algorithm of prime factorization that could only be run on such a hypothetical computer. Today, quantum computing is being realized. One can wonder what numerical methods are yet to be developed to solve complex scientific problems on future biocomputers.

This book examines many of the essential numerical methods used today, the mathematics behind these methods, and the scientific problems they were developed to solve. The book is structured into three parts—numerical methods for linear algebra, numerical methods for analysis, and numerical methods for differential equations.

▷ Numerical methods for linear algebra

There are two fundamental problems in linear algebra: solving a system of linear equations and solving the eigenvalue problem. Succinctly, the first problem says

1. Given an operator \mathbf{A} and a vector \mathbf{b} , find the vector \mathbf{x} such that $\mathbf{Ax} = \mathbf{b}$.

Such a problem frequently arises in science and engineering applications. For example, find the polynomial $p(x) = \sum_{i=0}^n c_i x^i$ passing through the points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$. Another example, solve the Poisson equation, which models the steady-state heat distribution $u(x, y)$,

$$\begin{aligned} -\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)u(x, y) &= f(x, y) \quad \text{for } (x, y) \in \Omega \\ u(x, y) &= g(x, y) \quad \text{for } (x, y) \in \partial\Omega \end{aligned}$$

where $f(x, y)$ is the source term and $g(x, y)$ is the boundary value. Numerically, we can solve this problem by first considering a discrete approximation and then solving the often large resultant system of linear equations. When solving the problem, getting the solution is primarily left up to a black box. Part One of this book breaks open that black box.

Numerically, one can solve the problem $\mathbf{Ax} = \mathbf{b}$ either directly or iteratively. The basic direct solver is Gaussian elimination. We can get more efficient methods such as banded solvers, block solvers, and Cholesky decomposition by taking advantage of properties of the matrix such as low bandwidth and symmetry. Chapter 2 looks at direct methods. Large, sparse matrices often arise in the

numerical methods for partial differential equations and machine learning. We can efficiently solve such systems by using iterative methods. Iterative methods rely on matrix multiplication to find an approximate solution. Chapter 5 looks at iterative methods such as the Jacobi, Gauss–Seidel, SOR, and Krylov methods like the conjugate gradient method. It also examines multigrid methods.

In some cases, a solution to $\mathbf{Ax} = \mathbf{b}$ may not exist. Or, if one does exist, it may not be unique. In the real world, such an answer is often unacceptable. Therefore, we can consider the modified problem

1'. Find the “best” \mathbf{x} that satisfies $\mathbf{Ax} = \mathbf{b}$.

Of course, what “best” means needs to be rigorously defined. Chapter 3 does this by looking at the least squares problem. In most applications, the best solution results from an orthogonal projection. A few different algorithms get us the orthogonal decomposition of a matrix, namely the Gram–Schmidt process, Givens rotations, and Householder reflections. Singular value decomposition provides yet another way of finding the least squares solution.

We can state the second fundamental problem of linear algebra as

2. Given an operator \mathbf{A} , find the scalar λ and the vector \mathbf{x} such that $\mathbf{Ax} = \lambda\mathbf{x}$.

Eigenvalue problems arise in the study of stability, steady-state behavior, and ordinary differential equations. In general, it is impossible to solve the eigenvalue problem directly, and we must instead use iterative methods. In Chapter 4, we take a closer look at how to do this.

Finally, Chapter 6 looks at the fast Fourier transform, which has revolutionized not only computational mathematics but also signal processing, imaging, and a host of other fields.

► Numerical methods for analysis

Numerical analysis develops tools and methods of problem-solving in such a manner that they can be implemented *efficiently* and *accurately* using a computer. Because of this, one can study numerical analysis from a rather theoretical framework, heavy in functional analysis. But one can also study it by looking at its application in solving problems—an approach often called scientific computing. As mathematics research becomes more and more interdisciplinary, numerical analysis is leaning more towards scientific computing.

The general strategy of scientific computing is to find a simple and efficient method to solve a difficult problem. Infinite dimensions are replaced with finite dimensions, nonlinear problems are localized as linear problems, and continuous models are exchanged for discrete models. In this way, numerical analysis is foremost a study of numerical approximation. Chapter 7 gives a quick discussion of the preliminaries: convergence, stability, and sources of errors of a numerical method. Chapter 8 introduces iterative methods to find

the approximate solutions to nonlinear equations and the roots of polynomials and extends the results to nonlinear systems. Chapter 9 considers methods of interpolating data using polynomial expansion, splines, and so forth. Chapter 10 looks at methods of approximating whole functions using basis functions such as orthogonal polynomials, wavelets, and neural nets. It also discusses the nonlinear least squares problem. Finally, Chapter 11 provides an overview of numerical differentiation and integration.

► Numerical methods for differential equations

Part Three examines ordinary differential equations and three classifications of partial differential equations: parabolic, hyperbolic and elliptic. Parabolic equations include the heat or diffusion equation. Solutions are always smooth and grow smoother over time. An important related set of problems includes the reaction-diffusion equations which model ice melting in a pool of water, the patterning of stripes on a zebra or spots on a leopard, and the aggregation of bacteria or tumor cells in response to chemical signals. Hyperbolic equations are often used to model low viscosity gas and particle dynamics. Nonlinear hyperbolic equations are synonymous with shock waves and are used to model anything from supersonic flight to bomb blasts to tsunamis to traffic flow. Without a smoothing term, discontinuities may appear but do not disappear. An alternative weak solution is needed to handle these equations mathematically. Adding viscosity to the equation regularizes the solution and brings us back to parabolic equations. In some regards, elliptic equations may be viewed as the steady-state solutions to parabolic equations. They include the Laplace equation and the Poisson equation. Unlike parabolic and hyperbolic equations, which are typically time-dependent, elliptic equations are often time-independent. They are used to model strain in materials, steady-state distribution of electric charge, and so forth. Often, a problem does not neatly fit into any given category. And sometimes, the problem behaves very differently over different length and time scales. For example, the Navier–Stokes equation describing fluid flow is predominantly diffusive on small length and time scales, but it is predominately advective on large scales. The melting of ice occurs only at the thin interface region separating the liquid and solid. Deflagration (and detonation) has two timescales—a slow diffusion and a fast chemical reaction timescale. These so-called stiff problems require careful consideration.

Part Three also examines three essential numerical tools for solving partial differential equations: finite difference methods, finite element methods, and Fourier spectral methods. Finite difference (and finite volume) methods are the oldest. They were employed in the 1950s and developed throughout the twentieth century (and before that). Their simple formulation allows them to be applied to a large class of problems. However, finite difference methods become cumbersome when the boundaries are complicated. Finite element methods were

developed in the 1960s to solve problems for which finite difference methods were not well adapted to, such as handling complicated boundaries. Hence, finite element methods provide attractive solutions to engineering problems. Finite element methods often require much more numerical and mathematical machinery than finite difference methods. Fourier spectral methods were also developed in the 1960s following the (re)discovery of the fast Fourier transform. They are important in fluid modeling and analysis where boundary effects can be assumed to be periodic.

Finally, Part Three explores three mathematical requirements for a numerical method: consistency, stability, and convergence. Consistency says that the discrete numerical method is a correct approximation to the continuous problem. Stability says that the numerical solution does not blow up. Finally, convergence says you can always reduce error in the numerical solution by refining the mesh. In other words, convergence says you get the correct solution.

► Doing mathematics

Mathematician Paul Halmos once remarked, “the only way to learn mathematics is to do mathematics.” Doing mathematics involves visualizing complex data structures, thinking logically and creatively, and gaining conceptual understanding and insight. Doing mathematics is about problem-solving and pattern recognition. It is recognizing that the same family of equations that models the response of tumor cells to chemical signals also models an ice cube melting in a glass of water and the patterning of spots on a leopard. It is appreciating that the equations of fluid dynamics can apply in one instance to tsunamis, in the next to traffic flow, and in a third to supersonic flight. Ultimately, mathematics is about understanding the world, and doing mathematics is learning to see that. Of course, one must learn the mechanics and structure of mathematics to have the familiarization, technique, and confidence in order to start doing mathematics. Each chapter concludes with a set of problems. Solutions to problems marked with a  are provided in the Back Matter.

► Julia, Python, Matlab

Not surprisingly, programming plays a starring role in scientific computing. There are several scientific programming languages which one might use—Julia, Python with SciPy and NumPy, Matlab or its open-source alternatives Octave and Scilab, R, Mathematica and Maple, SageMath, C, Fortran, and perhaps even JavaScript. This book emphasizes Julia because of the language’s freshness, versatility, simplicity, growing popularity, and notation and syntax that accurately mirror mathematical expressions. That’s not to say that the book ignores other scientific programming languages. Indeed, the Back Matter includes a chapter devoted to Python and Matlab/Octave. Every Julia commentary—identified with the  glyph—has a matching commentary for Python and Matlab. Every snippet

of Julia code has a matching snippet of Python and Matlab code. The code may not be entirely Julian, Pythonic, or Matlab-esque, as some effort was taken to bridge all of the languages with similar syntax to elucidate the underlying mathematical concepts. The code is likely not the fastest implementation—in some cases, it is downright slow. Furthermore, the code may overlook some coding best practices such as exception handling in favor of brevity. And, let's be honest, long blocks of code are dull. All of the code is available as a Jupyter notebook:

<https://nbviewer.jupyter.org/github/nmfsc/julia/blob/main/julia.ipynb>

To cut down on redundancy, the `LinearAlgebra.jl` and `Plots.jl` packages are implicitly assumed always to be available, while other packages will be explicitly imported in the code blocks. Additionally, we'll use the variable `bucket` to reference the GitHub directory of data files.

```
using LinearAlgebra, Plots  
bucket = "https://raw.githubusercontent.com/nmfsc/data/master/";
```

► QR links

When I was a young boy, I would thumb through my father's copy of *The Restless Universe* by physicist Max Born. (Max Born is best known as a Nobel laureate for his contributions to fundamental research in quantum mechanics and lesser known as a grandfather to 80s pop icon Olivia–Newton John.) While I didn't understand much of the book, I marveled at the illustrations. The book, first published in 1936, featured along its margins a set of what Born called “films,” what the publisher called “mutoscopic pictures,” and what we today would call flipbooks. By flicking the pages with one's thumb, simple animations emerged from the pages that helped the reader visualize the physics a little bit better. In designing this book, I wanted to repurpose Max Born's idea. I decided to use QR codes at the footers of several pages as a hopefully unobtrusive version of a digital flipbook to animate an illustration or concept discussed on that page. As a starting example, the QR code on this page contains one of Max Born's original films animating gas molecules. Simply unlock the code using your smartphone—I promise no Rick Astley. But you can also ignore them altogether.



Numerical Methods for Linear Algebra

Chapter 1

A Review of Linear Algebra



We'll start with a review of some important concepts and definitions of linear algebra. This chapter is brief, so please check out other linear algebra texts such as Peter Lax's book *Linear Algebra and Its Applications*, Gilbert Strang's identically titled book, or David Watkin's book *Fundamentals of Matrix Computations* for missing details.

1.1 Vector spaces

Simply stated, a *vector space* is a set V that is closed under linear combinations of its elements called *vectors*. If any two vectors of V are added together, the resultant vector is also in V ; and if any vector is multiplied by a scalar, the resultant vector is also in V . The scalar is often an element of the real numbers \mathbb{R} or the complex numbers \mathbb{C} , but it could also be an element of any other field \mathbb{F} . To remove the ambiguity, we often say that V is a vector space over the field \mathbb{F} . Once we have the vector space's basis—which we'll come to shortly—we can express and manipulate vectors as arrays of elements of the field with a computer.

What are some examples of vector spaces? The set of points in an n -dimensional space is a vector space. The set of polynomials $p_n(x) = \sum_{k=0}^n a_k x^k$ is another vector space. The set of piecewise linear functions over the interval $[0, 1]$ is yet another vector space—this one is important for the finite-element method. The set of all differentiable functions over the interval $[0, 1]$ that vanish at the endpoints is also a vector space. Another is the vector space over the Galois field $\text{GF}(p^n)$ with characteristic p and n terms. For example, $\text{GF}(2^8)$

gives byte xor arithmetic,

$$\{11011011\} \text{ xor } \{10001101\} = \{01010110\}.$$

If V is a vector space over \mathbb{F} , then a subset W of V is a *subspace* if W is a vector space over \mathbb{F} . Consider a system of n vectors $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\} \in V$. The set of all linear combinations of the system *generates* a subspace W of V . We call W the *span* of the system and denote it by $W = \text{span}\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$. The system $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ is called the *spanning set* or the *generators* of W .

The system of vectors $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ of a vector space V is said to be *linearly independent* if all linear combinations are unique. That is,

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \cdots + a_n\mathbf{v}_n = 0$$

if and only if the scalars a_1, a_2, \dots, a_n are all zero. Otherwise, we say that the system is *linearly dependent*.

A *basis* of V is a system of linearly independent generators of V . For example, the monomials $\{1, x, x^2, \dots, x^k\}$ are a basis for the space of k th-order polynomials. The number of elements in a basis of a vector space V is the *dimension* of V . Not every vector space can be generated by a finite number of elements. Take, for instance, the space of smooth functions over the interval $(0, 1)$ that vanish at the endpoints. Functions in this space can be represented as a Fourier sine series, and the vector space is spanned by the vectors $\sin m\pi x$ with $m = 1, 2, 3, \dots$. Such a vector space is said to be infinite-dimensional. If $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_n\}$ is a basis of V , then any vector in $\mathbf{v} \in V$ has a unique decomposition with respect to the basis:

$$\mathbf{v} = v_1\mathbf{u}_1 + v_2\mathbf{u}_2 + \cdots + v_n\mathbf{u}_n.$$

This decomposition can be expressed in matrix representation as the vector

$$\mathbf{v} = (v_1, v_2, \dots, v_n)$$

where the basis is implicitly understood. Hence, any n -dimensional vector space over \mathbb{R} is isomorphic to \mathbb{R}^n . The standard basis of \mathbb{R}^n is $\{\xi_1, \xi_2, \dots, \xi_n\}$ where

$$\xi_i = (\underbrace{0, \dots, 0}_{i-1}, 1, 0, \dots, 0).$$

It should be emphasized that the choice of a basis is not unique, although the representation of a vector in that basis is unique. For example, consider the space of quadratic polynomials \mathbb{P}_2 . One basis for this space is the set of monomials $\{1, x, x^2\}$. Another basis is the first three Legendre polynomials $\{1, x, \frac{1}{2}(3x^2 - 1)\}$. Both of these sets span \mathbb{P}_2 and the elements of each set are linearly independent—we can't form x^2 by combining x and 1. Given a basis, any vector in \mathbb{P}_2 has a unique representation in \mathbb{R}^3 . The vector $1 + x^2$ can be represented as $(1, 0, 1)$ in the basis $\{1, x, x^2\}$. The same vector $1 + x^2$ can be represented $(\frac{4}{3}, 0, \frac{2}{3})$ in the Legendre basis $\{1, x, \frac{1}{2}(3x^2 - 1)\}$.

► Matrices

Let V and W be vector spaces over \mathbb{C} . A *linear map* from V into W is a function $f : V \rightarrow W$ such that $f(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y})$ for all $\alpha, \beta \in \mathbb{C}$ and $\mathbf{x}, \mathbf{y} \in V$. For any linear map f , there exists a unique matrix $\mathbf{A} \in \mathbb{C}^{m \times n}$ such that $f(\mathbf{x}) = \mathbf{Ax}$ for all $\mathbf{x} \in \mathbb{C}^n$. Every n -dimensional vector space over \mathbb{C} is isomorphic to \mathbb{C}^n . And every linear map from an n -dimensional vector space into an m -dimensional vector space can be represented by an $m \times n$ matrix.

● A column vector can be formed using the syntax `[1,2,3,4]`, `[1;2;3;4]`, `[(1:4)...]`, or `[i for i ∈ 1:4]`. A row vector has the syntax `[1 2 3 4]`.

Example. Consider the derivative operator defined over the space of quadratic polynomials ($\frac{d}{dx} : \mathbb{P}_2 \mapsto \mathbb{P}_2$). It's easy to confirm that the derivative operator is a linear operator. Let's determine its matrix representation. First, we need to assign a basis for \mathbb{P}_2 . Let's take the monomial basis $\{1, x, x^2\}$. Note that $\frac{d}{dx}(a + bx + cx^2) = b + 2cx$. It follows from

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} b \\ 2c \\ 0 \end{bmatrix} \quad \text{that} \quad \frac{d}{dx} \equiv \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

for quadratic polynomials using the monomial basis. ◀

A matrix represents a mapping from one vector space to another one—or possibly the same one. And matrix multiplication represents the composition of mappings from one vector space to another one to a subsequent one. Two matrices can be multiplied together only if their dimensions are compatible, i.e., the number of columns of the first equals the number of rows of the second. Matrices can also be combined through element-wise operations like addition or the Hadamard product if they are compatible, i.e., their dimensions agree.

● Julia uses the “dot” operation to broadcast arithmetic operators by expanding scalars and arrays to compatible sizes. If $\mathbf{A} = \text{rand}(4,4)$ and $\mathbf{B} = \text{rand}(4,4)$, then $\mathbf{A} * \mathbf{B}$ is matrix multiplication, while $\mathbf{A} . * \mathbf{B}$ is the component-wise Hadamard product. If $\mathbf{x} = \text{rand}(4,1)$, then $\mathbf{A} . * \mathbf{x}$ is computed by first implicitly replicating the column vector \mathbf{x} to first produce a 4×4 array. The dot syntax can also be applied to functions. For example, `sin.(A)` will take the sine of each element of \mathbf{A} and return a 4×4 array. Furthermore, the `@.` macro applies dots to all operations in an expression.

The *transpose* \mathbf{A}^\top of an $m \times n$ matrix \mathbf{A} is the $n \times m$ matrix obtained by interchanging the rows of \mathbf{A} with columns of \mathbf{A} . That is, $[\mathbf{A}^\top]_{ij} = [\mathbf{A}]_{ji}$. The transpose has a few well-known and easily proved properties: $(\mathbf{A}^\top)^\top = \mathbf{A}$,

$(\mathbf{A} + \mathbf{B})^\top = \mathbf{A}^\top + \mathbf{B}^\top$, $(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$, and $(\mathbf{A}^{-1})^\top = (\mathbf{A}^\top)^{-1}$. The *conjugate transpose* or *adjoint* \mathbf{A}^H of a matrix $\mathbf{A} \in \mathbb{C}^{m \times n}$ is the $n \times m$ matrix obtained by exchanging the row of \mathbf{A} with complex conjugates of columns of \mathbf{A} .

• The transpose of \mathbf{A} is $\text{transpose}(\mathbf{A})$, and the conjugate transpose is \mathbf{A}' .

Consider a matrix $\mathbf{A} \in \mathbb{C}^{m \times n}$. The *column space* (or range) of \mathbf{A} is the subspace of \mathbb{C}^n generated by the columns of \mathbf{A} . The dimension of the column space of \mathbf{A} equals the dimension of the rows space of \mathbf{A} and is called the *rank* of \mathbf{A} . The *null space* or *kernel* of \mathbf{A} is the subspace of \mathbb{C}^n generated by the vectors $\mathbf{x} \in \mathbb{C}^n$ such that $\mathbf{Ax} = \mathbf{0}$. The dimension of the null space is called the *nullity* of \mathbf{A} . The *row space* is a subspace of \mathbb{C}^m generated by the rows of \mathbf{A} (the columns of \mathbf{A}^\top). The *left null space* is a subspace of \mathbb{C}^m generated by the vectors $\mathbf{x} \in \mathbb{C}^m$ with $\mathbf{x}\mathbf{A} = \mathbf{0}$. In other words, the left null space of \mathbf{A} is the null space of \mathbf{A}^\top . For a system $\mathbf{Ax} = \mathbf{b}$, we have¹

$$\mathbf{A}(\mathbf{x}_{\text{row}} + \mathbf{x}_{\text{null}}) = \mathbf{b}_{\text{column}} + \mathbf{b}_{\text{left null}}.$$

A nonzero null space leads to an undetermined system with infinite solutions. A nonzero left null space leads to an inconsistent solution with zero solutions.

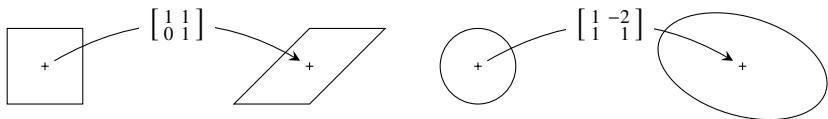
Example. Consider the derivative operator over the space of quadratic polynomials using the monomial basis $\{1, x, x^2\}$. The derivative operator maps constants to zero, so the null space of the derivative operator is $\text{span}\{1\}$ or equivalently $\text{span}\{(1, 0, 0)\}$. The derivative of a quadratic polynomial spans $\{1, x\}$, so the column space is $\text{span}\{1, x\}$ or equivalently $\text{span}\{(1, 0, 0), (0, 1, 0)\}$. The left null space is the complement to the column space, so it follows that the left null space is $\text{span}\{x^2\}$ or $\text{span}\{(0, 0, 1)\}$. ◀

Several important types of matrices are listed on page 8. Remember, all vector spaces over \mathbb{R}^n are isomorphic to \mathbb{R}^n . So by considering maps in \mathbb{R}^n , we have a nice geometric interpretation of linear maps. Take \mathbb{R}^2 . A circle (centered at the origin) is mapped to an ellipse (also centered at the origin), and a square is mapped to a parallelogram. In higher dimensions, a sphere is mapped to an ellipsoid, and a cube is mapped to a parallelipiped.

1.2 Eigenspaces

A number λ is called an *eigenvalue* of a square matrix \mathbf{A} if there exists a nonzero vector \mathbf{x} , called the *eigenvector*, such that $\mathbf{Ax} = \lambda\mathbf{x}$. The set of eigenvectors

¹These set of statements relating the four fundamental subspaces (the row space, null space, column space, and left null space) is often called the fundamental theorem of linear algebra, a term popularized by Gilbert Strang.

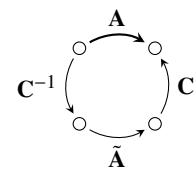


associated with an eigenvector λ is called the *eigenspace*. A subspace S is said to be *invariant* with respect to a square matrix \mathbf{A} if any vector in S stays in S under mapping by \mathbf{A} . In other words $\mathbf{AS} \subset S$. An eigenspace is invariant, and specifically, an eigenvector is directionally invariant. The set of eigenvalues $\{\lambda_1, \lambda_2, \dots, \lambda_n\}$ of \mathbf{A} is called its *spectrum* and denoted by $\lambda(\mathbf{A})$. The *spectral radius* of \mathbf{A} is the largest absolute value of its eigenvalues: $\rho(\mathbf{A}) = \max\{|\lambda_1|, |\lambda_2|, \dots, |\lambda_n|\}$.

Eigenvectors have a simple geometric interpretation. Let's take \mathbb{R}^2 . Start with two unit eigenvectors for \mathbf{A} . They fit in a unit circle. The matrix \mathbf{A} will stretch the eigenvectors by their respective eigenvalues, and our unit circle becomes an ellipse. The vectors that lie along the semi-major and semi-minor axes of our new ellipse are called *singular vectors*. The semi-major and semi-minor radii are called *singular values*. If the matrix \mathbf{A} happens to be a normal matrix, i.e., $\mathbf{A}^\top \mathbf{A} = \mathbf{A} \mathbf{A}^\top$, then the eigenvectors are all mutually orthogonal. In this case, the eigenvectors are the singular vectors and the eigenvalues are the singular values.

► Similar matrices

Two square matrices \mathbf{A} and $\tilde{\mathbf{A}}$ are *similar* if there exists a matrix \mathbf{C} such that $\mathbf{A} = \mathbf{C}\tilde{\mathbf{A}}\mathbf{C}^{-1}$. Such a transformation is called a *similarity transform*. The mapping by $\tilde{\mathbf{A}}$ is the same transformation in the standard basis as the mapping by \mathbf{A} in the basis given by the columns of \mathbf{C} . That is, $\mathbf{C}\tilde{\mathbf{A}}\mathbf{C}^{-1}$ means simply “change to the basis given by the columns of \mathbf{C} , apply $\tilde{\mathbf{A}}$ and then change back to the standard basis.”



If \mathbf{C} is a unitary matrix, we say that $\tilde{\mathbf{A}}$ and \mathbf{A} are *unitarily similar*. Similarity transformations can simplify a problem by transforming a matrix into a diagonal or triangular matrix. By analogy, it is often faster to take two detours and hook up with a superhighway than take the direct route.

► Diagonalization

A matrix that has a complete set of eigenvectors is said to be *semisimple*. Otherwise, it is *defective*. For semisimple matrices, we can rewrite the eigenvalue problem $\mathbf{Ax} = \lambda x$ as $\mathbf{AS} = \mathbf{S}\Lambda$ where \mathbf{S} is a matrix of the eigenvectors and Λ is a diagonal matrix of eigenvalues. This formulation leads to the *diagonalization* of \mathbf{A} as $\mathbf{A} = \mathbf{S}\Lambda\mathbf{S}^{-1}$, which says that \mathbf{A} is similar to the diagonal matrix Λ in its

Some important types of matrices

Symmetric: $\mathbf{A}^T = \mathbf{A}$.

Hermitian or self-adjoint: $\mathbf{A}^H = \mathbf{A}$.

Positive definite: a Hermitian matrix \mathbf{A} such that $\mathbf{x}^H \mathbf{A} \mathbf{x} > 0$ for any non-zero vector \mathbf{x} .

Orthogonal: $\mathbf{Q}^T \mathbf{Q} = \mathbf{I}$.

Unitary: $\mathbf{U}^H \mathbf{U} = \mathbf{I}$. Columns of orthogonal matrices and unitary matrices are mutually orthonormal. Orthogonal and unitary matrices are geometrically equivalent to rotations and reflections.

Permutation: A permutation matrix is an orthogonal matrix whose columns are permutations of the identity matrix. Geometrically, a permutation matrix is a type of reflection.

Normal: $\mathbf{A}^H \mathbf{A} = \mathbf{A} \mathbf{A}^H$. Examples of normal matrices include unitary, Hermitian, and skew-Hermitian ($\mathbf{A}^H = -\mathbf{A}$) matrices.

Projection: $\mathbf{P}^2 = \mathbf{P}$.

Orthogonal projection: $\mathbf{P}^2 = \mathbf{P}$ and $\mathbf{P}^T = \mathbf{P}$. The matrix $\mathbf{P} = \mathbf{A}(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T$ maps vectors into the column space of \mathbf{A} but does not change vectors already in that subspace. Any vector orthogonal to the column space of \mathbf{A} is mapped to the zero vector, i.e., if $\mathbf{A}^T \mathbf{B} = \mathbf{0}$, then $\mathbf{PB} = \mathbf{0}$.

Diagonal: $a_{ij} = 0$ for $i \neq j$. We denote it as $\text{diag}(a_{11}, a_{22}, \dots, a_{nn})$.

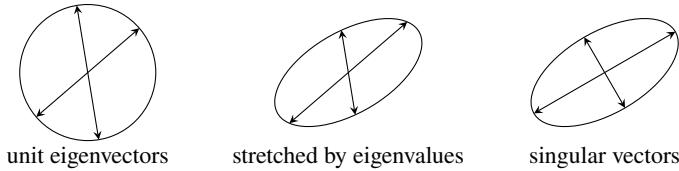
Upper triangular: $a_{ij} = 0$ for $i > j$.

Tridiagonal: $a_{ij} = 0$ if $|i - j| > 1$.

Banded: $a_{ij} = 0$ if $i - j > m_l$ or $j - i < m_u$. The number $m_u + m_l + 1$ is called the *bandwidth*.

Upper Hessenberg: $a_{ij} = 0$ for $i > j + 1$.

diagonal	tridiagonal	upper Hessenberg	upper triangular	block
$\begin{bmatrix} \cdot & & \\ & \ddots & \\ & & \cdot \end{bmatrix}$	$\begin{bmatrix} \cdot & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \ddots \end{bmatrix}$	$\begin{bmatrix} \cdot & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \ddots \end{bmatrix}$	$\begin{bmatrix} \cdot & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \ddots \end{bmatrix}$	$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$



eigenvector basis. The system is uncoupled in this eigenvector basis and much easier to manipulate.

Example. We can evaluate a function of a matrix $f(\mathbf{A})$ by using the Taylor series expansion of f and the diagonalization of the matrix:

$$f(\mathbf{A}) = \sum_{k=0}^{\infty} c_k \mathbf{A}^k = \sum_{k=0}^{\infty} c_k (\mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1})^k = \sum_{k=0}^{\infty} c_k \mathbf{S}\mathbf{\Lambda}^k \mathbf{S}^{-1} = \mathbf{S} \left(\sum_{k=0}^{\infty} c_k \mathbf{\Lambda}^k \right) \mathbf{S}^{-1}.$$

So $f(\mathbf{A})$ is $\mathbf{S} \operatorname{diag}(f(\lambda_1), f(\lambda_2), \dots, f(\lambda_n)) \mathbf{S}^{-1}$. For instance, the system of differential equations

$$\frac{d}{dt} \mathbf{u} = \mathbf{A} \mathbf{u} \quad \text{has the formal solution} \quad \mathbf{u}(t) = e^{t\mathbf{A}} \mathbf{u}(0).$$

The solution can be evaluated as $\mathbf{u}(t) = \mathbf{S} \operatorname{diag}(e^{\lambda_1 t}, e^{\lambda_2 t}, \dots, e^{\lambda_n t}) \mathbf{S}^{-1} \mathbf{u}(0)$ or simply as $\mathbf{v}(t) = \operatorname{diag}(e^{\lambda_1 t}, e^{\lambda_2 t}, \dots, e^{\lambda_n t}) \mathbf{v}(0)$ where $\mathbf{v}(t) = \mathbf{S}^{-1} \mathbf{u}(t)$, completely decoupling the system. If \mathbf{A} happens to be a circulant matrix (the type of matrix that often arises when solving a partial differential equation with periodic boundary conditions), \mathbf{S} is easy and fast to compute—it's a discrete Fourier transform. We'll come back to the discrete Fourier transform in Chapter 6. ◀

► Schur and spectral decompositions

Every square matrix is unitarily similar to an upper triangle matrix whose diagonal elements are the eigenvalues of the original matrix, a representation called the Schur decomposition. Furthermore, every normal matrix is unitarily similar to a diagonal matrix whose elements are the eigenvalues of the original matrix, a representation called spectral decomposition.

Theorem 1 (Schur decomposition). *Every square matrix is unitarily similar to an upper triangular matrix. In other words, given \mathbf{A} , there exists a unitary matrix \mathbf{U} and an upper triangular matrix \mathbf{T} such that $\mathbf{T} = \mathbf{U}^H \mathbf{A} \mathbf{U}$. Furthermore, the diagonal elements of \mathbf{T} are the eigenvalues of \mathbf{A} .*

Proof. We can prove this by induction. Take $\mathbf{A} \in \mathbb{C}^{n \times n}$. When $n = 1$, the hypothesis is clearly true: $\mathbf{A} = \lambda$ is already an upper triangular 1×1 matrix with

$\mathbf{U} = \mathbf{1}$. Now, assume that the hypothesis holds for $n - 1$ and show that it also holds for n . Let λ be an eigenvalue of \mathbf{A} and \mathbf{v} be the associated eigenvector with $\|\mathbf{v}\|_2 = 1$. Let \mathbf{U}_1 be a unitary matrix whose first column is \mathbf{v} :

$$\mathbf{U}_1 = [\mathbf{v} \quad \mathbf{W}]$$

where \mathbf{W} is the $n \times (n - 1)$ matrix of remaining columns. Let $\mathbf{A}_1 = \mathbf{U}_1^H \mathbf{A} \mathbf{U}_1$. Then

$$\mathbf{A}_1 = \begin{bmatrix} \mathbf{v}^H \\ \mathbf{W}^H \end{bmatrix} \mathbf{A} \begin{bmatrix} \mathbf{v} & \mathbf{W} \end{bmatrix} = \begin{bmatrix} \mathbf{v}^H \mathbf{A} \mathbf{v} & \mathbf{v}^H \mathbf{A} \mathbf{W} \\ \mathbf{W}^H \mathbf{A} \mathbf{v} & \mathbf{W}^H \mathbf{A} \mathbf{W} \end{bmatrix} = \begin{bmatrix} \lambda & \mathbf{v}^H \mathbf{A} \mathbf{W} \\ \mathbf{0} & \mathbf{W}^H \mathbf{A} \mathbf{W} \end{bmatrix} = \left[\begin{array}{c|cccc} \lambda & * & \cdots & * \\ \hline 0 & & & & \\ \vdots & & \hat{\mathbf{A}} & & \\ 0 & & & & \end{array} \right]$$

where $\hat{\mathbf{A}} = \mathbf{W}^H \mathbf{A} \mathbf{W} \in \mathbb{C}^{(n-1) \times (n-1)}$. By the induction hypothesis, there exists a unitary matrix $\hat{\mathbf{U}}_1$ and an upper triangular matrix $\hat{\mathbf{T}}$ such that $\hat{\mathbf{T}} = \hat{\mathbf{U}}_1^H \hat{\mathbf{A}} \hat{\mathbf{U}}_1$. Let

$$\mathbf{U}_2 = \left[\begin{array}{c|ccccc} \lambda & * & \cdots & * \\ \hline 0 & & & & & \\ \vdots & & \hat{\mathbf{U}}_1 & & & \\ 0 & & & & & \end{array} \right].$$

Then \mathbf{U}_2 is unitary and

$$\mathbf{U}_2^H \mathbf{A}_1 \mathbf{U}_2 = \left[\begin{array}{c|ccccc} \lambda & * & \cdots & * \\ \hline 0 & & & & & \\ \vdots & & \hat{\mathbf{U}}_1^H \hat{\mathbf{A}} \hat{\mathbf{U}}_1 & & & \\ 0 & & & & & \end{array} \right] = \left[\begin{array}{c|ccccc} \lambda & * & \cdots & * \\ \hline 0 & & & & & \\ \vdots & & \hat{\mathbf{T}} & & & \\ 0 & & & & & \end{array} \right]$$

is an upper triangular matrix. Call it \mathbf{T} . So

$$\mathbf{T} = \mathbf{U}_2^H \mathbf{A}_1 \mathbf{U}_2 = \mathbf{U}_2^H \mathbf{U}_1^H \mathbf{A} \mathbf{U}_1 \mathbf{U}_2 = \mathbf{U}^H \mathbf{A} \mathbf{U}. \quad \square$$

Theorem 2 (Spectral theorem). *If \mathbf{A} is a normal matrix ($\mathbf{A}^H \mathbf{A} = \mathbf{A} \mathbf{A}^H$), then $\mathbf{A} = \mathbf{U} \Lambda \mathbf{U}^H$ where \mathbf{U} is a unitary matrix whose columns are eigenvectors of \mathbf{A} and Λ is a diagonal matrix whose elements are the eigenvalues of \mathbf{A} . Furthermore, the eigenvalues of a Hermitian matrix are real.*

Proof. As a consequence of the Shur decomposition theorem, $\mathbf{A} = \mathbf{U} \Lambda \mathbf{U}^H$ where Λ is an upper triangular matrix whose diagonal elements are eigenvalues of \mathbf{A} and \mathbf{U} is a unitary matrix. It follows that

$$\mathbf{A}^H \mathbf{A} = \mathbf{U} \Lambda^H \mathbf{U}^H \mathbf{U} \Lambda \mathbf{U}^H = \mathbf{U} \Lambda^H \Lambda \mathbf{U}^H \quad \text{and}$$

$$\mathbf{A} \mathbf{A}^H = \mathbf{U} \Lambda \mathbf{U}^H \mathbf{U} \Lambda^H \mathbf{U}^H = \mathbf{U} \Lambda \Lambda^H \mathbf{U}^H.$$

$\Lambda^H \Lambda = \Lambda \Lambda^H$ looks like

$$\begin{bmatrix} \times & & & & \\ \times & \times & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & & & & \\ \times & \times & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \end{bmatrix}.$$

Let's look just at the diagonal elements of the product $\Lambda^H \Lambda = \Lambda \Lambda^H$. Starting with the $(1, 1)$ -element:

$$t_{11}^2 = \sum_{k=1}^n t_{1k}^2.$$

So $t_{1k} = 0$ for $k > 1$. Hence $\Lambda^H \Lambda = \Lambda \Lambda^H$ looks like

$$\begin{bmatrix} \times & & & & \\ \times & \times & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & & & & \\ \times & \times & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \end{bmatrix}.$$

Now, move to the $(2, 2)$ -element:

$$t_{22}^2 = \sum_{k=2}^n t_{2k}^2.$$

So $t_{2k} = 0$ for $k > 2$. Hence $\Lambda^H \Lambda = \Lambda \Lambda^H$ looks like

$$\begin{bmatrix} \times & & & & \\ \times & \times & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \begin{bmatrix} \times & & & & \\ \times & \times & & & \\ \times & \times & \times & & \\ \times & \times & \times & \times & \\ \times & \times & \times & \times & \times \end{bmatrix}.$$

Continuing like this, it follows that Λ is a diagonal matrix. When \mathbf{A} is also a Hermitian matrix, $\mathbf{A} = \mathbf{A}^H$, from which it follows that $\Lambda = \Lambda^H$. So the eigenvalues of a Hermitian matrix are real. \square

By the spectral theorem, a Hermitian matrix \mathbf{A} is unitarily similar to a diagonal matrix $\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^H$ where \mathbf{Q} is a unitary matrix of the eigenvectors. Geometrically, this says that a real, symmetric matrix behaves like a diagonal operator in a rotated (and reflected) basis. Also,

$$\mathbf{A} = \lambda_1 \mathbf{q}_1 \mathbf{q}_1^H + \lambda_2 \mathbf{q}_2 \mathbf{q}_2^H + \cdots + \lambda_n \mathbf{q}_n \mathbf{q}_n^H$$

where $\mathbf{q}_k \mathbf{q}_k^H$ is an orthogonal projection matrix onto the unit eigenvector \mathbf{q}_k . This decomposition is known as the *spectral decomposition* of \mathbf{A} .

Example. A circulant matrix

$$\mathbf{C} = \begin{bmatrix} c_1 & c_2 & \cdots & c_{n-1} & c_n \\ c_n & c_1 & c_2 & & c_{n-1} \\ \vdots & c_n & c_1 & \ddots & \vdots \\ c_3 & & \ddots & \ddots & c_2 \\ c_2 & c_3 & \cdots & c_n & c_1 \end{bmatrix}$$

is a normal matrix. Circulant matrices often appear in practice as the discrete approximation to the Laplacian and as convolution operators on a periodic domain. They are unitarily similar to diagonal matrices $\mathbf{A} = \mathbf{FCF}^H$ where \mathbf{F} is the discrete Fourier transform. By making a change of basis using \mathbf{F} , we trade a complicated operator \mathbf{C} for a simple diagonal one $\mathbf{\Lambda}$. With the invention of the fast Fourier transform, the two changes of the bases \mathbf{F} and \mathbf{F}^H are relatively quick steps. \blacktriangleleft

► Singular value decomposition

We can generalize spectral decomposition to non-symmetric and even non-square matrices. Such a decomposition is called the *singular value decomposition* or simply the SVD. Let $\mathbf{A} \in \mathbb{C}^{m \times n}$. There exist two unitary matrices $\mathbf{U} \in \mathbb{C}^{m \times m}$ and $\mathbf{V} \in \mathbb{C}^{n \times n}$ such that

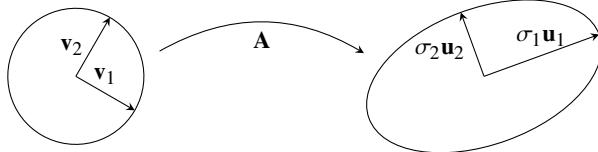
$$\mathbf{U}^H \mathbf{A} \mathbf{V} = \Sigma = \text{diag}(\sigma_1, \dots, \sigma_n)$$

with $\sigma_1 \geq \dots \geq \sigma_n \geq 0$. The numbers σ_i are called the *singular values* of \mathbf{A} and are given by

$$\sigma_i(\mathbf{A}) = \sqrt{\lambda_i(\mathbf{A}^H \mathbf{A})}.$$

The unitary matrix \mathbf{U} is the matrix of eigenvectors of $\mathbf{A}\mathbf{A}^H$ and the unitary matrix \mathbf{V} is the matrix of eigenvectors of $\mathbf{A}^H\mathbf{A}$.

Singular value decomposition has an intuitive geometric interpretation: $\mathbf{AV} = \mathbf{U}\Sigma$. Any matrix \mathbf{A} maps a sphere with axes along the columns of \mathbf{V} into an ellipsoid $\mathbf{U}\Sigma$ with some dimensions possibly zero. The singular values give the radii of the ellipsoid. The right singular vectors are mapped into the left singular vectors, which form the semiprincipal axes of the ellipsoid. The largest singular value gives the maximum magnification of the matrix. The smallest singular value gives the smallest magnification of the matrix.



Example. A 2×2 matrix maps points on the unit circle to points on an ellipse. Such a mapping can have two, one, or no real eigenvectors. See Figure 1.1. Here we consider equivalent matrices with singular values $\sigma_1 = 2$ and $\sigma_2 = 1$ formed by changing the right singular vectors of the singular value decomposition. Think of twisting the unit circle by different angles. Eigenvectors lie along the radial

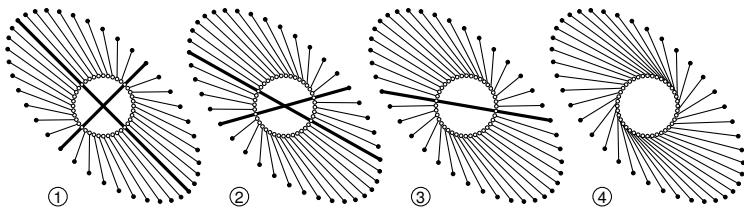


Figure 1.1: A matrix maps points on the unit circle to points on an ellipse. Such a mapping has two, one, or no real eigenvectors (thick line segments) running along the radial directions.

direction. When the right and left singular matrices are the same, the matrix is symmetric and has a pair of orthogonal eigenvectors that are identical to the left and right singular vectors ①. If we adjust the right singular vector slightly by twisting the unit circle, the eigenvectors move toward one another ② until they are colinear ③, and finally they become complex ④. A matrix that does not have a complete basis of eigenvectors is called defective. ▶

1.3 Measuring vectors and matrices

► Inner products

An *inner product* on the vector space V is any map (\cdot, \cdot) from $V \times V$ into \mathbb{R} or \mathbb{C} with the three properties—linearity, Hermiticity, and positive definiteness:

1. $(a\mathbf{u} + b\mathbf{v}, \mathbf{w}) = a(\mathbf{u}, \mathbf{w}) + b(\mathbf{v}, \mathbf{w})$ for all $\mathbf{u}, \mathbf{v}, \mathbf{w} \in V$ and $a, b \in \mathbb{C}$;
2. $(\mathbf{u}, \mathbf{v}) = \overline{(\mathbf{v}, \mathbf{u})}$; and
3. $(\mathbf{u}, \mathbf{u}) \geq 0$ and $(\mathbf{u}, \mathbf{u}) = 0$ if and only if $\mathbf{u} = 0$.

Two vectors \mathbf{u} and \mathbf{v} are *orthogonal* if $(\mathbf{u}, \mathbf{v}) = 0$. Important examples of inner products include the *Euclidean inner product* $(\mathbf{u}, \mathbf{v})_2 = \mathbf{u}^\top \mathbf{v}$ and the *A-energy inner product* $(\mathbf{u}, \mathbf{v})_A = \mathbf{u}^\top \mathbf{A} \mathbf{v}$ where \mathbf{A} is symmetric, positive definite. Vectors that are orthogonal under the A-energy inner product are said to be A-orthogonal or A-conjugate. We can visualize two vectors \mathbf{u} and \mathbf{v} that are orthogonal under the Euclidean inner product as being perpendicular in space. What can we make of A-orthogonal vectors? Because \mathbf{A} is a symmetric, positive-definite matrix, it has the spectral decomposition $\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^{-1}$, where \mathbf{Q} is an orthogonal matrix of eigenvectors. Then $(\mathbf{u}, \mathbf{v})_A$ is simply $(\Lambda^{1/2}\mathbf{Q}^{-1}\mathbf{u})^\top (\Lambda^{1/2}\mathbf{Q}^{-1}\mathbf{v})$. That is, the \mathbf{u} and \mathbf{v} are perpendicular in the scaled eigenbasis of \mathbf{A} .

► Vector norms

A *vector norm* on the vector space V is any map $\|\cdot\|$ from V into \mathbb{R} with the three properties—positivity, homogeneity, and subadditivity:

1. $\|\mathbf{v}\| \geq 0$ for all $\mathbf{v} \in V$ and $\|\mathbf{v}\| = 0$ if and only if $\mathbf{v} = 0$;
2. $\|\alpha\mathbf{v}\| = |\alpha|\|\mathbf{v}\|$; and
3. $\|\mathbf{v} + \mathbf{w}\| \leq \|\mathbf{v}\| + \|\mathbf{w}\|$.

The p -norm is an important class of vector norms:²

$$\|\mathbf{u}\|_p = \left(\sum_{i=1}^n |u_i|^p \right)^{1/p} \quad \text{for } 1 \leq p < \infty.$$

Specific cases of the p -norm include the 1-norm

$$\|\mathbf{u}\|_1 = \sum_{i=1}^n |u_i|,$$

the *Euclidean norm* or 2-norm

$$\|\mathbf{u}\|_2 = \left(\sum_{i=1}^n |u_i|^2 \right)^{1/2} = \sqrt{(\mathbf{u}, \mathbf{u})_2},$$

and the ∞ -norm (taking $p \rightarrow \infty$)

$$\|\mathbf{u}\|_\infty = \max_{1 \leq i \leq n} |u_i|.$$

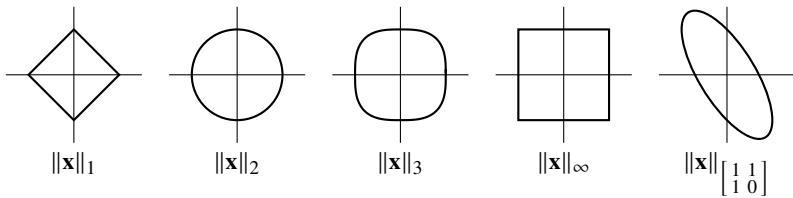
Another important class of vector norms is the *energy norm*

$$\|\mathbf{u}\|_{\mathbf{A}} = \sqrt{(\mathbf{u}, \mathbf{u})_{\mathbf{A}}} \quad \text{where } \mathbf{A} \text{ is symmetric, positive definite.}$$

Let's make a couple of observations about vector norms. First, every inner product *induces* a vector norm: $\|\mathbf{u}\|^2 = (\mathbf{u}, \mathbf{u})$. All such induced vector norms satisfy the *Cauchy–Schwarz inequality* $(\mathbf{x}, \mathbf{y}) \leq \|\mathbf{x}\|\|\mathbf{y}\|$. To see this, note that for any value α we have

$$0 \leq \|\mathbf{x} - \alpha\mathbf{y}\|^2 = \|\mathbf{x}\|^2 - 2\alpha(\mathbf{x}, \mathbf{y}) + |\alpha|^2\|\mathbf{y}\|^2.$$

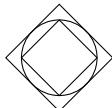
By taking $\alpha = (\mathbf{x}, \mathbf{y})/\|\mathbf{y}\|^2$, it follows that $0 \leq \|\mathbf{x}\|^2\|\mathbf{y}\|^2 - (\mathbf{x}, \mathbf{y})^2$. Second, the Euclidean norm is invariant under orthogonal transformations $\|\mathbf{Q}\mathbf{x}\|_2 = \|\mathbf{x}\|_2$. This result should be clear from the geometric interpretation of the Euclidean

Figure 1.2: Unit circles in different norms on \mathbb{R}^2 .

norm and orthogonal transformation, but you can crosscheck it with simple algebra: $\|\mathbf{Q}\mathbf{x}\|_2^2 = \|\mathbf{x}^\top \mathbf{Q}^\top \mathbf{Q}\mathbf{x}\|_2 = \|\mathbf{x}^\top \mathbf{x}\|_2 = \|\mathbf{x}\|_2^2$.

Two norms $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ on a vector space V are *equivalent* if there are positive constants c and C such that $c\|\mathbf{u}\|_\alpha \leq \|\mathbf{u}\|_\beta \leq C\|\mathbf{u}\|_\alpha$ for all vectors $\mathbf{u} \in V$. All norms on finite-dimensional vector spaces are equivalent. This means that we can often choose a convenient norm with which to work and the results will hold with respect to the other norms.

Example. Let's first show that the 1- and 2-norms are equivalent and then show that the 2- and ∞ -norms are equivalent. We'll take $V = \mathbb{R}^n$.



The hypercube $\|\mathbf{x}\|_1 = 1$ can be inscribed in the hypersphere $\|\mathbf{x}\|_2 = 1$, which can itself be inscribed in the hypercube $\|\mathbf{x}\|_1 = \sqrt{n}$. So $\frac{1}{\sqrt{n}}\|\mathbf{x}\|_1 \leq \|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1$.



The hypercube $\|\mathbf{x}\|_\infty = 1/\sqrt{n}$ can be inscribed in the hypersphere $\|\mathbf{x}\|_2 = 1$, which can be inscribed in the hypercube $\|\mathbf{x}\|_\infty = 1$. So $\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_\infty$. \blacktriangleleft

► Matrix norms

A *matrix norm* is map $\|\cdot\|$ from $\mathbb{R}^{m \times n}$ into \mathbb{R} with the three properties—positivity, homogeneity, and subadditivity:

1. $\|\mathbf{A}\| \geq 0$ and $\|\mathbf{A}\| = 0$ if and only if $\mathbf{A} = 0$;
2. $\|\alpha\mathbf{A}\| = |\alpha|\|\mathbf{A}\|$; and
3. $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$.

A matrix norm is said to be *submultiplicative* if it has the additional property

²The p -norm is the discrete, finite-dimensional analog to an L^p -norm for an infinite-dimensional vector space and an L^p -norm for a continuous vector space.

$$4. \|\mathbf{AB}\| \leq \|\mathbf{A}\| \|\mathbf{B}\|.$$

A matrix norm $\|\cdot\|$ is said to be *compatible* or *consistent* with a vector norm $\|\cdot\|$ if $\|\mathbf{Ax}\| \leq \|\mathbf{A}\| \|\mathbf{x}\|$ for all $\mathbf{x} \in \mathbb{R}^n$. One might ask “what is the smallest matrix norm that is compatible with a vector norm?” We call such a norm the *induced matrix norm*:

$$\|\mathbf{A}\| = \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} = \sup_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\|.$$

The induced norm is the most that a matrix can stretch a vector in a vector norm.

Theorem 3. Let $\|\cdot\|$ be a matrix norm induced by the vector norm $\|\cdot\|$. Then 1. $\|\cdot\|$ is compatible with $\|\cdot\|$, 2. $\|\mathbf{I}\| = 1$, and 3. $\|\cdot\|$ is submultiplicative.

Proof.

1. $\|\mathbf{Ax}\| = \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} \|\mathbf{x}\| \leq \left(\sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} \right) \|\mathbf{x}\| = \|\mathbf{A}\| \|\mathbf{x}\|$
2. $\|\mathbf{I}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{x}\| = 1$
3. $\|\mathbf{AB}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{ABx}\| \leq \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}\| \|\mathbf{Bx}\| = \|\mathbf{A}\| \|\mathbf{B}\|$ □

Arguably, the most important induced matrix norms are the p -norms

$$\|\mathbf{A}\|_p = \sup_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\|_p$$

In particular, the 2-, 1-, and ∞ -norms arise frequently in numerical methods. So, it's good to get a geometric intuition about each one. See the Figure 1.3 on the next page.

We'll start with the 2-norm or the *spectral norm*. A circle (\mathbf{x}) is mapped to an ellipse (\mathbf{Ax}) under a linear transformation (\mathbf{A}), and $\sup \|\mathbf{Ax}\|_2$ corresponds to the largest radius of the ellipse. So $\|\mathbf{A}\|_2 = \sigma_{\max}(\mathbf{A})$.

Next, consider the unit circle of the 1-norm. The square (\mathbf{x}) is mapped to a parallelogram (\mathbf{Ax}). The supremum of $\|\mathbf{Ax}\|_1$ must come from one of the vertices of the parallelogram. The vertices of the parallelogram are mapped from the corners of the square, each of which corresponds to one of the standard basis elements $\pm \xi_j$, e.g., in three-dimensions $(\pm 1, 0, 0)$, $(0, \pm 1, 0)$ or $(0, 0, \pm 1)$. Then $\mathbf{A}\xi_j$ returns (plus or minus) the j th column of \mathbf{A} . Therefore,

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|.$$

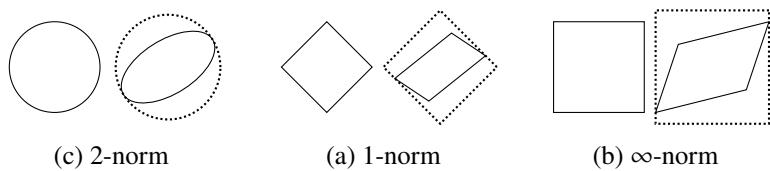


Figure 1.3: Mapping of unit circles by $\frac{1}{4} \begin{bmatrix} 1 & 1 \\ 1 & 3 \end{bmatrix}$.

Finally, consider the unit circle for the ∞ -norm. Again, a square (\mathbf{x}) is mapped to a parallelogram (\mathbf{Ax}). The supremum of $\|\mathbf{Ax}\|_1$ must originate from the corners of the square. This time the corners are at $(\pm 1, \pm 1, \dots, \pm 1)$. Explicitly,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \pm 1 \\ \pm 1 \\ \vdots \\ \pm 1 \end{bmatrix} = \begin{bmatrix} \pm a_{11} \pm a_{12} \pm a_{13} \\ \pm a_{21} \pm a_{22} \pm a_{23} \\ \pm a_{31} \pm a_{32} \pm a_{33} \end{bmatrix}.$$

By taking the largest possible element, we have

$$\|\mathbf{A}\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^m |a_{ij}|.$$

Note that $\|\mathbf{A}\|_\infty = \|\mathbf{A}^\top\|_1$. You can use the superscripts of the 1- and ∞ -norms as a mnemonic to remember in which direction to take the sums. The “1” runs vertically, so take the maximum of the sums down the columns. The “ ∞ ” lies horizontally, so take the maximum of the sums along the rows. For example,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 1 \\ 1 & 4 & 0 \end{bmatrix}, \quad \|\mathbf{A}\|_1 = 7, \quad \|\mathbf{A}\|_\infty = 6.$$

The `LinearAlgebra.jl` function `opnorm(A, p)` computes the p -norm of a matrix \mathbf{A} , where the optional argument p can be either 1, 2 (default), or Inf. The `LinearAlgebra` function `norm(A)` returns the Frobenius norm.

Theorem 4. $\|\mathbf{A}\|_2$ equals the largest singular value of \mathbf{A} . If \mathbf{A} is Hermitian, then $\|\mathbf{A}\|_2$ equals the spectral radius of \mathbf{A} . If \mathbf{A} is unitary, then $\|\mathbf{A}\|_2 = 1$.

Proof. This theorem has the easy picture proof mentioned above and depicted in Figure 1.3 above. A circle is mapped to an ellipse, and the induced 2-norm corresponds to the largest radius of the ellipse. If the matrix is Hermitian, then the singular values are the same as the eigenvalues. If the matrix is unitary, the ellipse is simply another unit circle.

If you're not satisfied with this picture proof, here's an algebraic proof. Take $\mathbf{A} \in \mathbb{C}^{n \times n}$. $\mathbf{A}^H \mathbf{A}$ is Hermitian, so there exists a unitary matrix \mathbf{U} such that

$$\mathbf{U}^H \mathbf{A}^H \mathbf{A} \mathbf{U} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$$

where λ_i are the nonnegative eigenvalues of $\mathbf{A}^H \mathbf{A}$. Let $\mathbf{y} = \mathbf{U}^H \mathbf{x}$. Then

$$\begin{aligned}\|\mathbf{A}\|_2 &= \sup_{\|\mathbf{x}\|_2=1} \sqrt{\mathbf{x}^H \mathbf{A}^H \mathbf{A} \mathbf{x}} = \sup_{\|\mathbf{x}\|_2=1} \sqrt{\mathbf{y}^H \mathbf{U}^H \mathbf{A}^H \mathbf{A} \mathbf{U} \mathbf{y}} \\ &= \sup_{\|\mathbf{y}\|_2=1} \sqrt{\sum_{i=1}^n \lambda_i |y_i|^2} = \sqrt{\max_{1 \leq i \leq n} \lambda_i}.\end{aligned}$$

Furthermore, if \mathbf{A} is Hermitian, then the singular values of \mathbf{A} equal its eigenvalues. If \mathbf{A} is unitary, $\mathbf{A}^H \mathbf{A} = \mathbf{I}$. \square

Theorem 5. $\rho(\mathbf{A}) \leq \|\mathbf{A}\|$ where $\|\cdot\|$ is an induced norm. Furthermore, for symmetric matrices, $\|\mathbf{A}\|_{\max} \leq \rho(\mathbf{A})$ where the max norm $\|\mathbf{A}\|_{\max} = \max_{i,j} |a_{ij}|$.

Proof. If λ is an eigenvalue of \mathbf{A} , then $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ for some vector \mathbf{x} . So $\|\mathbf{A}\mathbf{x}\| = |\lambda| \|\mathbf{x}\|$. From this it follows that $|\lambda| = \|\mathbf{A}\mathbf{x}\| / \|\mathbf{x}\|$, and hence

$$\rho(\mathbf{A}) = \max_{\lambda \in \lambda(\mathbf{A})} |\lambda| \leq \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|} = \|\mathbf{A}\|.$$

A standard basis vector ξ_j is mapped to the j th column of \mathbf{A} . So $\|\mathbf{A}\|_{\max}$ must be equal to or smaller than the largest component $\mathbf{A}\xi_j$ for some ξ_j . The largest component of $\mathbf{A}\xi_j$ must be equal to or smaller than the length of $\mathbf{A}\xi_j$. So $\|\mathbf{A}\|_{\max} \leq \max_j \|\mathbf{A}\xi_j\|_1 \leq \sigma_{\max}(\mathbf{A})$. \square

Example. We can put a lower and upper bound on the spectral radius of a symmetric matrix using $\|\mathbf{A}\|_{\max} \leq \rho(\mathbf{A}) \leq \|\mathbf{A}\|_{\infty}$. For the following matrix

$$\begin{bmatrix} 5 & 1 & 2 & 1 \\ 1 & 2 & 2 & 3 \\ 2 & 2 & 4 & 2 \\ 1 & 3 & 2 & 6 \end{bmatrix}$$

we find that $6 \leq \rho(\mathbf{A}) \leq 12$. The computed value is $\rho(\mathbf{A}) \approx 10.03$. \blacktriangleleft

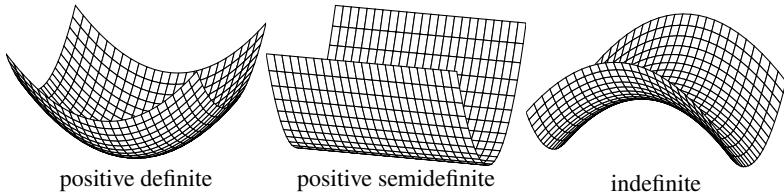
► Quadratic forms

The quadratic form of a real, symmetric $n \times n$ matrix \mathbf{A} is defined as $q(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$. In component form,

$$q(x_1, x_2, \dots, x_n) = \sum_{i,j=1}^n a_{ij} x_i x_j.$$

In the case of a 2×2 matrix,

$$q(x, y) = [x \ y] \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = ax^2 + 2bxy + cy^2 :$$



A matrix \mathbf{A} is said to be *positive definite*, if $\mathbf{x}^\top \mathbf{A} \mathbf{x} > 0$ for all $\mathbf{x} \neq 0$. Likewise, a matrix \mathbf{A} is said to be negative definite, if $\mathbf{x}^\top \mathbf{A} \mathbf{x} < 0$ for all $\mathbf{x} \neq 0$. A matrix \mathbf{A} is positive semidefinite if $\mathbf{x}^\top \mathbf{A} \mathbf{x} \geq 0$ and negative semidefinite if $\mathbf{x}^\top \mathbf{A} \mathbf{x} \leq 0$. Otherwise, the matrix is indefinite.

Theorem 6. *A symmetric matrix is positive definite if and only if it has only positive eigenvalues.*

Proof. Suppose that \mathbf{A} is positive definite. Then $\mathbf{x}^\top \mathbf{A} \mathbf{x} > 0$ for any nonzero vector \mathbf{x} . If \mathbf{x} is an eigenvector, $\mathbf{x}^\top \mathbf{A} \mathbf{x} = \mathbf{x}^\top \lambda \mathbf{x} = \lambda \|\mathbf{x}\|^2$. It follows that $\lambda > 0$. On the other hand, suppose that \mathbf{A} has only positive eigenvalues λ_i . Because \mathbf{A} is symmetric, any vector \mathbf{x} can be written as a linear combination of orthogonal unit eigenvectors $\mathbf{x} = c_1 \mathbf{q}_1 + c_2 \mathbf{q}_2 + \cdots + c_n \mathbf{q}_n$. Then the quadratic form

$$\begin{aligned} \mathbf{x}^\top \mathbf{A} \mathbf{x} &= (c_1 \mathbf{q}_1 + c_2 \mathbf{q}_2 + \cdots + c_n \mathbf{q}_n)^\top (c_1 \lambda_1 \mathbf{q}_1 + c_2 \lambda_2 \mathbf{q}_2 + \cdots + c_n \lambda_n \mathbf{q}_n) \\ &= c_1^2 \lambda_1 \|\mathbf{q}_1\|^2 + c_2^2 \lambda_2 \|\mathbf{q}_2\|^2 + \cdots + c_n^2 \lambda_n \|\mathbf{q}_n\|^2 = \sum_{i=1}^n \lambda_i c_i^2 \end{aligned}$$

by orthogonality of the eigenvectors. Because the $\lambda_i > 0$ for all i , it follows that $\mathbf{x}^\top \mathbf{A} \mathbf{x} > 0$. \square

A matrix \mathbf{A} is said to be *congruent* to a matrix \mathbf{B} if there is an invertible matrix \mathbf{S} such that $\mathbf{B} = \mathbf{S}^\top \mathbf{A} \mathbf{S}$. Congruent matrices have the same quadratic forms in different bases:

$$q_B(\mathbf{x}) = \mathbf{x}^\top \mathbf{B} \mathbf{x} = \mathbf{x}^\top \mathbf{S}^\top \mathbf{A} \mathbf{S} \mathbf{x} = (\mathbf{S} \mathbf{x})^\top \mathbf{A} (\mathbf{S} \mathbf{x}) = \mathbf{y}^\top \mathbf{A} \mathbf{y} = q_A(\mathbf{y}) \text{ for } \mathbf{y} = \mathbf{S} \mathbf{x}.$$

Example. Linear algebra used in quantum physics has its own unique notation. When quantum theory was developed in the early twentieth century, linear algebra was not widely taught in physics (or mathematics) departments, so the quantum pioneer Paul Dirac invented his own notation. A solution $\psi(x)$ to Schrödinger's

equation $\mathbf{H}\psi = E\psi$, with the Hamiltonian operator $\mathbf{H} = -(\hbar^2/2m)\Delta + V(x)$ and the real-valued energy E , is a vector of an infinite-dimensional vector space W . In Dirac or bra–ket notation, a vector ψ is denoted by a special symbol $| \rangle$. We can then express the vector $\alpha\psi$ as $\alpha| \rangle$. Because this notation is ambiguous when working with more than one vector, say ψ_1 and ψ_2 , we use an additional label, say 1 and 2, to write the vectors as $|1\rangle$ and $|2\rangle$. Note that the label is simply enclosed in $|$ and \rangle . Such an object is called a “ket.” We can then express $\alpha\psi_1$ as $\alpha|1\rangle$. Bear in mind that anything between $|$ and \rangle is simply a label, so $|\alpha 1\rangle \neq \alpha|1\rangle$ and $|1+2\rangle \neq |1\rangle + |2\rangle$. The space of kets spans the vector space W , and we can choose a basis of kets.

A linear functional mapping $W \rightarrow \mathbb{C}$ is often called a *covector*. The space of linear functionals, which forms the dual space to the vector space, is itself a vector space. A vector in the dual space of Hermitian operators is denoted by $\langle |$. Define a linear functional $\langle | = (\phi, \cdot) = \int_{-\infty}^{\infty} \phi^* \cdot dx$, where $\phi^*(x)$ is the complex conjugate of a function $\phi(x)$. We can now write an inner product $(\phi, \psi) = \int_{-\infty}^{\infty} \phi^* \psi dx$ as $\langle | \rangle$. Just as we used labels to make $| \rangle$ unambiguous, we can do the same for $\langle |$. To denote a specific operator $\langle |$, we can similarly enclose a label a between $\langle |$ and $|$ to get $\langle |a|$. We call such a linear operator a “bra.” Let’s take an orthonormal basis for W : $\{|0\rangle, |1\rangle, |2\rangle, \dots\}$. For each ket there is a corresponding bra $\{\langle 0|, \langle 1|, \langle 2|, \dots\}$, and the inner product $\langle i|j\rangle = \delta_{ij}$. (By convention, it is not necessary to write $|$ twice when combining a bra with a ket.) Furthermore, the projection operator is simply the outer product $\mathbf{P}_i = |i\rangle \langle i|$. This allows us to decompose an arbitrary vector

$$\psi = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \dots + \alpha_i |i\rangle + \dots$$

into its orthonormal basis vector components

$$\mathbf{P}_i\psi = |i\rangle \langle i| \psi = \alpha_0 |i\rangle \langle i|0\rangle + \alpha_1 |i\rangle \langle i|1\rangle + \dots + \alpha_i |i\rangle \langle i|i\rangle + \dots = \alpha_i |i\rangle .$$

States are the solutions to the Schrödinger equation $\mathbf{H}\psi = E\psi$. Recognize that the Schrödinger equation is simply an eigenvalue equation (an eigenequation). The eigenvalue solutions of it are called eigenstates. The complex-valued solutions ψ are called wave functions, and by convention are normalized over space $\int_{-\infty}^{\infty} \psi^* \psi dx = 1$. This gives a wavefunction ψ the interpretation of a probability distribution $\rho(x) = |\psi(x)|^2$, the probability density of a particle at position x . Because the Schrödinger equation is an eigenvalue problem, we can find the spectral decomposition of the Hamiltonian operator

$$\mathbf{H} = E_0 |0\rangle \langle 0| + E_1 |1\rangle \langle 1| + E_2 |2\rangle \langle 2| + \dots$$

The Schrödinger equation can be solved in terms of its eigenvector components. Recall that the eigenvalues of Hermitian operators are real and the eigenvectors are orthogonal. When a physicist says that ψ a superposition of states, they are

saying that the vector ψ is a linear combination of the orthonormal basis vectors $|0\rangle, |1\rangle, |2\rangle, \dots$.

Hermitian operators and unitary operators both play key roles in quantum physics. The Hermitian operators that have starring roles include the position operator $\mathbf{x} = x$, the momentum operator $\mathbf{p} = -i\hbar\nabla$, and the Hamiltonian operator \mathbf{H} . For example, applying the position operator is the same as multiplying by x : $\langle x \rangle = x\langle \psi \rangle$. The composition $\langle x \rangle$ represents the expected value of the probability distribution $\int_{-\infty}^{\infty} \psi^* x \psi dx = \int_{-\infty}^{\infty} x |\psi|^2 dx$. The composition $\langle \mathbf{p} \rangle$ represents the expected value of the momentum $-i\hbar \int_{-\infty}^{\infty} \psi^* (\partial/\partial x) \psi dx$. And $\langle \mathbf{H} \rangle$ represents the expected value of the energy. ◀

1.4 Stability

It's convenient to solve problems by using black-box methods—letting an algorithm in a computer, for example, take care of the tedious computation. This book is largely about prying open those black boxes to inspect and tinker with their algorithmic cogwheels. Still, regardless of the algorithm: garbage in, garbage out. How can we know whether a problem will have a meaningful solution? The problem may have no solution. The problem may have multiple solutions. Perhaps the problem is so sensitive to change that a method cannot replicate the output if the input changes by even the slightest amount. To address this concern, French mathematician Jacques Hadamard introduced the concept of mathematical well-posedness in 1923. A problem is said to be *well-posed* if

1. a solution for the problem *exists*;
2. the solution is *unique*; and
3. this solution is *stable* under small perturbations in the data.

If any of these conditions fails to hold, the problem is *ill-posed*. A solution \mathbf{x} to the linear problem $\mathbf{Ax} = \mathbf{b}$ exists if \mathbf{b} is in the column space of \mathbf{A} . The solution \mathbf{x} is unique if the null space of \mathbf{A} is only the zero vector. As long as the matrix \mathbf{A} is invertible, there exists a unique solution. We will examine the case when \mathbf{A} is not invertible in Chapter 3. In this section, we look at stability.

Let $\mathbf{b} = \mathbf{Ax}$. Suppose that we perturb the input data \mathbf{x} , perhaps due to round-off error or errors in measurement. How does this perturbation affect our numerical computation of \mathbf{b} ? Alternatively, how does perturbing \mathbf{A} affect our computation? Or, if we are solving $\mathbf{Ax} = \mathbf{b}$, what is the effect of perturbing \mathbf{b} ?

As we will see, the first and third questions are equivalent. Put more precisely, how does the relative change in the input vector $\|\delta\mathbf{x}\|/\|\mathbf{x}\|$ affect the relative change in the output vector $\|\delta\mathbf{b}\|/\|\mathbf{b}\|$? Take

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

By linearity, $\delta\mathbf{b} = \mathbf{A}\delta\mathbf{x}$, from which it follows that

$$\|\delta\mathbf{b}\| = \|\mathbf{A}\delta\mathbf{x}\| \leq \|\mathbf{A}\| \|\delta\mathbf{x}\|.$$

Furthermore, from $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ we have

$$\|\mathbf{x}\| = \|\mathbf{A}^{-1}\mathbf{b}\| \leq \|\mathbf{A}^{-1}\| \|\mathbf{b}\|.$$

Combining these two inequalities,

$$\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \leq \|\mathbf{A}^{-1}\| \|\mathbf{A}\| \frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|}.$$

The factor $\|\mathbf{A}^{-1}\| \|\mathbf{A}\|$ is called the *condition number* of the matrix \mathbf{A} and is denoted by $\kappa(\mathbf{A})$. The condition number of the p -norm of \mathbf{A} is denoted by $\kappa_p(\mathbf{A})$. When the condition number is small, a small perturbation $\delta\mathbf{x}$ on \mathbf{x} results in a small change $\delta\mathbf{b}$ in \mathbf{b} . In this case, we say that \mathbf{b} *depends continuously* on the data \mathbf{x} .

Let's look at the condition number another way. For any induced norm, the condition number $\kappa(\mathbf{A}) \geq 1$:

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\| \geq \|\mathbf{A}\mathbf{A}^{-1}\| = \|\mathbf{I}\| = 1.$$

Furthermore, since

$$\begin{aligned} \|\mathbf{A}^{-1}\| &= \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}^{-1}\mathbf{x}\| = \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{A}^{-1}\mathbf{x}\|}{\|\mathbf{x}\|} = \sup_{\mathbf{y} \neq 0} \frac{\|\mathbf{A}^{-1}\mathbf{y}\|}{\|\mathbf{y}\|} \text{ where } \mathbf{y} = \mathbf{Ax} \\ &= \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{A}^{-1}\mathbf{Ax}\|}{\|\mathbf{Ax}\|} = \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{x}\|}{\|\mathbf{Ax}\|} = \sup_{\|\mathbf{x}\|=1} \frac{1}{\|\mathbf{Ax}\|} = \frac{1}{\inf_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\|}, \end{aligned}$$

it follows that

$$\kappa(\mathbf{A}) = \frac{\sup_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\|}{\inf_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\|}.$$

Therefore, a geometric interpretation of the condition number is

$$\kappa(\mathbf{A}) = \frac{\text{maximum magnification}}{\text{minimum magnification}}.$$

Specifically, the condition number in the Euclidean norm is $\kappa_2(\mathbf{A}) = \sigma_{\max}/\sigma_{\min}$. In other words, the condition number is the ratio of radii of the semi-major to semi-minor axes of an ellipsoid image of a matrix. See the figure on the next page.

Another way to think of the condition number is as a measure of linear dependency of the columns. A matrix \mathbf{A} is *ill-conditioned* if $\kappa(\mathbf{A}) \gg 1$. If \mathbf{A}

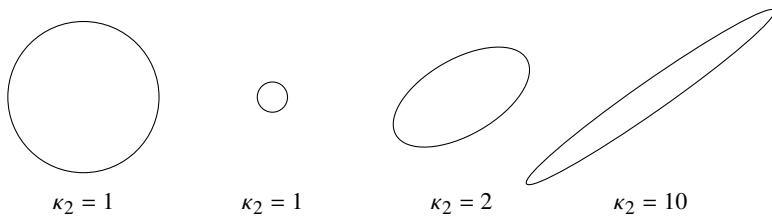


Figure 1.4: Images of unit circles under different linear transformations and their associated condition numbers.

is singular, $\kappa(\mathbf{A}) = \infty$. If \mathbf{A} is a unitary or orthogonal matrix, then $\kappa_2(\mathbf{A}) = 1$. This makes orthogonal matrices numerically very appealing because multiplying by them does not introduce numerical instability.

Now, the second question: how does the perturbation $\mathbf{A} + \delta\mathbf{A}$ affect our computation? Take the singular value decomposition of $\Sigma = \mathbf{U}^H \mathbf{A} \mathbf{V}$. Perturbation of \mathbf{A} says $\mathbf{U}^H (\mathbf{A} + \delta\mathbf{A}) \mathbf{V} = \Sigma + \delta\Sigma$ and hence $\mathbf{U}^H \delta\mathbf{A} \mathbf{V} = \delta\Sigma$. Because \mathbf{U} and \mathbf{V} are unitary, they are 2-norm preserving. So $\|\delta\mathbf{A}\|_2 = \|\delta\Sigma\|_2$. Perturbations in a matrix cause perturbations of roughly the same size in its singular values. See Watkins [2014] for a detailed analysis of simultaneous perturbations in both the matrix and the input vectors.

Example. A Hilbert matrix is an example of an ill-conditioned matrix. The 4×4 Hilbert matrix is

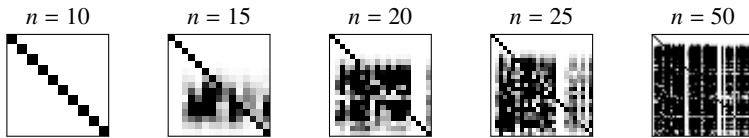
$$\mathbf{H} = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}$$

and the elements of a general Hilbert matrix are $h_{ij} = (i + j - 1)^{-1}$. We can construct a Hilbert matrix using Julia with the function

```
hilbert(n) = [1/(i+j-1) for i=1:n, j=1:n]
```

and display the density plot of $\mathbf{H}^{-1} \mathbf{H}$ by explicitly converting it to type Gray:

```
using Images
[Gray.(1 .- abs.(hilbert(n)\hilbert(n))) for n ∈ (10,15,20,25,50)]
```



Zero values are white, values of one or greater are black, and intermediate values are shades of gray. An identity matrix is represented by a diagonal of black in a field of white. While the numerical computation appears to be more-or-less correct for $n \leq 10$, it produces substantial round-off error for even moderately low dimensions such as $n = 15$. \blacktriangleleft

- The `LinearAlgebra.jl` function `cond(A, p)` returns the p -condition number of A for p either 2 (default), 1, or `Inf`.

- The `SpecialMatrices.jl` function `hilbert(n)` returns a Hilbert matrix as rationals.

1.5 Geometric interpretation of linear algebra

This chapter reviewed several fundamental concepts of linear algebra. Many of these concepts have simple geometric analogs that help develop an intuition of abstract and sometimes abstruse notions.

Consider the space \mathbb{R}^n . A vector is a point in space, and a unit vector is a point on a unit circle (n -sphere). The unit vector is the direction of a vector. The zero vector is the origin. A matrix, as a linear transformation, maps a circle (n -sphere) centered at the origin to an ellipse (ellipsoid) centered at the origin. An affine transformation adds translation, but a linear transformation keeps the origin fixed. Similarly, a square is mapped to a parallelogram, and an n -cube is mapped to a parallelepiped. An $m \times n$ matrix maps vectors from \mathbb{R}^n to \mathbb{R}^m . The rank is the number of dimensions of the resultant ellipsoid. The null space is the space of the vectors that are mapped by the matrix to the zero vector. The column space is the space spanned by the ellipsoid.

A diagonal matrix stretches along coordinate axes. A unit upper triangular matrix or unit lower triangular matrix shears an image like a deck of cards sliding on top of one another. A projection matrix squashes all of the vectors in some direction to a pancake. A vector in the column space is already in the pancake and doesn't move. Orthogonal projection squashes perpendicularly. An orthogonal matrix is a generalized rotation or reflection. A permutation reorders the coordinate axes by successively swapping rows of a matrix. If there is an even number of exchanges, the signature is even and orientation is right-handed. Otherwise, it is left-handed. A determinant is the signed volume of the parallelepiped spanned by the column vectors of a square matrix. The

absolute value of the determinant is the relative change in volume between the n -cube and the associated parallelepiped. The sign of the determinant is the orientation of the associated permutation. The Jacobian determinant (familiar from calculus) measures the relative, local change in volume caused by a change of variables.

An eigenvector of a matrix is any vector whose direction is not changed by the matrix. An eigenvalue is the amount by which an eigenvector is stretched. A matrix maps a unit circle into an ellipse, and the singular values are the lengths of the radii of the ellipsoid. A basis is the underlying set of vectors on which we create a coordinate system. The standard basis consists of an orthogonal unit vectors. Sometimes, it is more convenient to choose another basis, like the eigenbasis.

A vector norm is the length of a vector—that is, the distance to the origin. Often, we think of distance in terms of a Euclidean norm (the 2-norm). But other norms are useful, particularly the 1-norm and the ∞ -norm. The 1-norm is the sum of the absolute values of the coordinates of a vector in the standard basis, and the ∞ -norm is the magnitude of the largest coordinate of a vector in the standard basis. The 2-norm is invariant under an orthogonal change of basis, while the 1- and ∞ -norms are not. We can extend the concept of a norm to a matrix. An induced norm is the most that any vector could be stretched by the matrix. Norms also help us measure the condition of a matrix. An ill-conditioned matrix maps a circle to a very eccentric ellipse. The 2-condition number is the ratio of the semi-major to semi-minor radii.

1.6 Exercises

- 1.1. Show that if \mathbf{x} is an eigenvector of a nonsingular matrix \mathbf{A} with eigenvalue λ , then \mathbf{x} is also an eigenvector of \mathbf{A}^{-1} with eigenvalue $1/\lambda$. Also, show that \mathbf{x} is an eigenvector of $\mathbf{A} - c\mathbf{I}$ with eigenvalue $\lambda - c$.
- 1.2. Prove that similar matrices have the same spectrum of eigenvalues.
- 1.3. Krylov subspaces are important tools in the computation of eigenvalues of large matrices. An order- r Krylov subspace $\mathcal{K}_r(\mathbf{A}, \mathbf{x})$ generated by the $n \times n$ matrix \mathbf{A} and a vector \mathbf{x} is the subspace spanned by $\{\mathbf{x}, \mathbf{Ax}, \mathbf{A}^2\mathbf{x}, \dots, \mathbf{A}^{r-1}\mathbf{x}\}$.
 - (a) What is the dimension of the Krylov subspace if \mathbf{x} is an eigenvector of \mathbf{A} ?
 - (b) What is the dimension of the Krylov subspace if \mathbf{x} is the sum of two linearly independent eigenvectors of \mathbf{A} ?
 - (c) What is the maximum possible dimension that the Krylov subspace can have if \mathbf{A} is a projection operator?

- (d) What is the maximum possible dimension that the Krylov subspace can have if the nullity of \mathbf{A} is m ?

4 1.4. A $(0,1)$ -matrix is a matrix whose elements are either zero or one. Such matrices are important in graph theory and combinatorics. How many of them are invertible? This is an easy problem when the matrix is small. For example, there are two 1×1 $(0,1)$ -matrices, namely, $[0]$ and $[1]$. So half are invertible. There are sixteen 2×2 matrices whose entries are ones and zeros:

$$\text{Singular: } [0\ 0] \ [0\ 1] \ [1\ 0] \ [1\ 1] \ [0\ 0] \ [0\ 1] \ [0\ 0] \ [0\ 1] \ [1\ 0] \ [1\ 1]$$

$$\text{Invertible: } [1\ 0] \ [0\ 1] \ [1\ 1] \ [1\ 0] \ [1\ 0] \ [0\ 1]$$

So three-eighths are invertible. Estimate how many $n \times n$ $(0,1)$ -matrices are invertible. Plot the ratio of invertible matrices as a function of size n for $n = 1, 2, \dots, 20$ and explain the results. *Note:* There are 2^n $n \times n$ $(0,1)$ -matrices—for instance, roughly 10^{19} 8×8 matrices. To check each 8×8 matrix using a 100 petaflop supercomputer would take over 100 years. Instead, we can approximate the number of invertible matrices by using a random sample of $(0,1)$ -matrices. Test a large number (perhaps 10,000) of such matrices for each $n = 1, 2, \dots, 20$.

4 1.5. Consider the $n \times n$ matrix used to approximate a second-derivative

$$\mathbf{D} = \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}.$$

- (a) Find the eigenvalues of \mathbf{D} . Hint: One approach is using the method of cofactors to derive a three-term recurrence for the characteristic polynomial of $\mathbf{D} + 2\mathbf{I}$. Use the recurrence to show that the characteristic polynomial is an n th-order Chebyshev polynomial. An easier but less satisfying approach is to show that vectors of the form

$$\mathbf{x}_k = \left(\sin\left(k \frac{\pi}{n+1}\right), \sin\left(k \frac{2\pi}{n+1}\right), \dots, \sin\left(k \frac{n\pi}{n+1}\right) \right)^T$$

are eigenvectors of \mathbf{D} where $k = 1, 2, \dots, n$. (The column vectors \mathbf{x}_k divided by $\sqrt{(n+1)/2}$ form an orthogonal matrix \mathbf{F} . The spectral decomposition $\mathbf{DF} = \Lambda\mathbf{F}$ gives us the discrete sine transform of \mathbf{D} .)

- (b) Compute the spectral radius $\rho(\mathbf{D} + 2\mathbf{I})$.

- (c) Determine $\kappa_2(\mathbf{D})$ and discuss the behavior as $n \rightarrow \infty$.
 (d) Confirm your answers numerically.

1.6. Prove that the induced norm is a matrix norm.

1.7. Consider the matrix \mathbf{A} that maps \mathbb{R}^3 into \mathbb{R}^2 :

$$\mathbf{A} = \begin{bmatrix} 4 & -3 \\ -2 & 6 \\ 4 & 6 \end{bmatrix}.$$

- (a) Compute the SVD of \mathbf{A} by hand and use it to find the null space of \mathbf{A} .
 (b) What vector is magnified the most in the 2-norm? By how much and what is the resultant vector?
 1.8. Let \mathbf{P} be an orthogonal projection matrix. Prove that $\mathbf{I} - \mathbf{P}$ is also a projection matrix and that $\mathbf{I} - 2\mathbf{P}$ is a symmetric, orthogonal matrix. Describe geometrically what each operator does. Determine the eigenvalues.

1.9. The Frobenius norm is defined as $\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} a_{ij}^2}$. This norm is particularly useful in image processing.

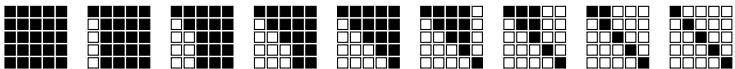
- (a) Prove that the Frobenius norm is a matrix norm by showing that the three properties of matrix norms hold.
 (b) Determine $\|\mathbf{Q}\|_F$ where \mathbf{Q} is an $n \times n$ orthogonal matrix.
 (c) Prove that the Frobenius norm is invariant under orthogonal transformations $\|\mathbf{Q}\mathbf{A}\|_F = \|\mathbf{A}\|_F$.
 (d) Prove that $\|\mathbf{A}\|_F = \sqrt{\sum_i \sigma_i^2}$, where σ_i is the i th singular value of \mathbf{A} .
 (e) Prove that $\|\mathbf{A}\|_F^2$ equals the trace of $\mathbf{A}^\top \mathbf{A}$.
 (f) Prove that the Frobenius norm is compatible with the Euclidean vector norm.
 (g) Prove that the Frobenius norm is also submultiplicative.

Hint: Don't solve this problem sequentially. Spoiler: Start with (e).

1.10. Cleve Moler and Charles van Loan's article "Nineteen dubious ways to compute the exponential of a matrix" presents several methods of computing $e^\mathbf{A}$ for a matrix \mathbf{A} . Discuss or implement one of the methods. How is a matrix exponent computed in practice?

Chapter 2

Direct Methods for Linear Systems



Finding solutions to linear systems is a core problem of scientific computing. The problem is so fundamental that functions are widely available in optimized software libraries for numerical linear algebra such as LAPACK (Linear Algebra Package) and the Intel Math Kernel Library. So it's unlikely that one would ever need to explicitly program any of these algorithms. Needless to say, it is worthwhile to have an understanding of the mathematics underpinning these packages to be aware of potential pitfalls. In this chapter, we discuss direct methods for solving the problem $\mathbf{Ax} = \mathbf{b}$ where \mathbf{A} is invertible. We discuss methods for solving $\mathbf{Ax} = \mathbf{b}$ where \mathbf{A} is not invertible in Chapter 3. And we discuss iterative methods for solving $\mathbf{Ax} = \mathbf{b}$ in Chapter 5.

There are several ways to solve $\mathbf{Ax} = \mathbf{b}$ in Julia. For a 2000×2000 matrix, $\mathbf{A}\backslash\mathbf{b}$ takes about 0.15 seconds and $\text{inv}(\mathbf{A})\ast\mathbf{b}$ takes 0.42 seconds. The `(\)` function solves the system by first determining the structure of \mathbf{A} and then applying specialized routines: dividing by diagonal elements of diagonal matrices, using backward or forward substitution on upper or lower triangular matrices, or applying general LU decomposition on square matrices. The `LinearAlgebra.jl` package provides further optimized methods to factorize special matrix types, such as `Tridiagonal` and `Cholesky`. Such routines are significantly faster than those for general arrays. For example, computing $\mathbf{A}\backslash\mathbf{b}$ using a 2000×2000 tridiagonal matrix that was explicitly cast as type `Tridiagonal` is about a thousand times faster than performing the same operation on one that was not.

• The macro `@less` shows the Julia code for a method. The command can help you know what's going on inside the black box.

• The macros `@time` and `@elapsed` provide run times. Use the `begin...end` block for compound statements. Julia precompiles your code on the first run, so run it twice and take the time from the second run.

2.1 Gaussian elimination

There are three elementary row operations that leave a matrix in row equivalent form, i.e., that do not fundamentally change the underlying system of equations but instead return an equivalent system of equations. The three elementary row operations are

1. multiply a row by a scalar,
2. add a scalar multiple of a row to another row, and
3. interchange two rows.

We can solve $\mathbf{Ax} = \mathbf{b}$ by using elementary row operations on the augmented matrix $[\mathbf{A} | \mathbf{b}]$ to get a reduced row echelon matrix.

With paper and pencil, there is a clear benefit to interchanging two rows to simplify computation. To a computer, on the other hand, every floating-point computation takes just as much work as any other floating-point computation. Let's start by writing an algorithm that does not interchange rows. Getting a matrix into reduced row echelon form entails a two-stage procedure.

1. Forward elimination. Starting with the first column, successively march through the columns zeroing out the elements below the diagonal.

$$\left[\begin{array}{ccccc|c} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \right] \rightarrow \left[\begin{array}{ccccc|c} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \right] \rightarrow \left[\begin{array}{ccccc|c} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \right] \rightarrow \left[\begin{array}{ccccc|c} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \right]$$

2. Backward elimination. Starting with the last column, successively march through the columns zeroing out the elements above the diagonal.

$$\left[\begin{array}{ccccc|c} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \right] \rightarrow \left[\begin{array}{ccccc|c} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \right] \rightarrow \left[\begin{array}{ccccc|c} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \right] \rightarrow \left[\begin{array}{ccccc|c} x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \\ x & x & x & x & x & x \end{array} \right]$$

• The `RowEchelon.jl` function `rref` returns the reduced row echelon form.

The first stage (forward elimination) requires the most effort because we need to compute using entire rows. We need to operate on n^2 elements to zero out the elements below the first pivot, $(n - 1)^2$ elements to zero out the elements below the second pivot, and so forth. The second stage (backward elimination) is relatively fast because we only need to work on one column at each iteration.

Zeroing out the elements above the last pivot requires changing n elements, zeroing out the elements above the second pivot requires $n - 1$ elements, and so forth. By saving the intermediate terms of forward elimination, we can express a matrix \mathbf{A} as the product $\mathbf{A} = \mathbf{LU}$ where \mathbf{L} is a unit lower triangular matrix (with ones along the diagonal) and \mathbf{U} is an upper triangular matrix. If such a decomposition exists, it is unique. Such a decomposition allows us to solve the problem $\mathbf{Ax} = \mathbf{b}$ in three steps:

1. Compute \mathbf{L} and \mathbf{U} .
2. Solve $\mathbf{Ly} = \mathbf{b}$ for \mathbf{y} .
3. Solve $\mathbf{Ux} = \mathbf{y}$ for \mathbf{x} .

This process is called *Gaussian elimination*. Contrary to its name, Gaussian elimination did not originate with Carl Friedrich Gauss. The technique is well over two thousand years old and was commonly known as just “elimination” until the 1950s, when the influential mathematician George Forsythe misattributed it to Gauss. Gaussian elimination in its modern matrix form was developed by John von Neumann and Herman Goldstine as “the combination of two tricks” in their foundational 1947 article “Numerical Inverting of Matrices of High Order.” (Grcar [2011]) The first trick, specified in step 1 to decompose a matrix into a product of two triangular matrices, is called *LU decomposition*. The second trick, specified in steps 2 and 3, solves the triangular systems inductively. Let’s look at each of these tricks, starting with the second.

► Solving triangular systems

Steps 2 and 3 are relatively straightforward. We’ll start with them and then come back to step 1. The lower triangular system

$$\begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

can be solved using forward elimination, starting from the top and working down. From the i th row

$$l_{11}y_1 + \dots + l_{i,i-1}y_{i-1} + y_i = b_i,$$

we have

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij}y_j,$$

where each y_i is determined from the y_j coming before it. To save computer memory, it is often convenient to overwrite the array \mathbf{b} with the values \mathbf{y} .

Now, let's implement step 3 to solve the upper triangular system

$$\begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ u_{22} & \dots & u_{2n} \\ \ddots & \ddots & \vdots \\ & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

using backward elimination (starting from the bottom and working up). From the i th row

$$u_{ii}x_i + u_{i,i+1}x_{i+1} + \dots + u_{in}x_n = y_i,$$

we have

$$x_i = \frac{1}{u_{ii}} \left(y_i - \sum_{j=1+i}^n u_{ij}x_j \right).$$

As before, we can overwrite the array **b** with the values **x**.

Note that we never need to change the elements of **L** or **U** when solving the triangular systems. So, once we have the LU decomposition of a matrix, we can use it over and over again.

► LU decomposition

Now, let's implement the LU decomposition itself (step 1). On a practical note, computer memory is valuable (especially when n is large). So, it's important to make efficient use of it. To store the full matrices **A**, **L**, and **U** in computer memory takes $3n^2$ floats. **A** really has the same information as **L** and **U**, so storing both is redundant. Also, **L** and **U** are almost half-filled with zeros—wasted memory. Now, the smart idea! Note that except along the diagonal, the non-zero elements of **L** and **U** are mutually exclusive. In fact, since the diagonal of **L** is defined to be all ones, we can store the information in these two matrices in the same array as

$$\begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ l_{21} & u_{22} & \dots & u_{2n} \\ \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & u_{nn} \end{bmatrix}.$$

Now, the really smart idea! Overwrite **A** with the elements of **L** and **U**. When we perform LU decomposition, we work from the top-left to the bottom-right. We zero out the elements of the i th column below the i th row and fill up the corresponding elements in **L**. There is no conflict if we store this information in the same matrix. Furthermore, since we are moving diagonally down the matrix **A**, there is no conflict if we simply overwrite the elements of **A**. For example,

$$\left[\begin{array}{cccc} 2 & 4 & 2 & 3 \\ -2 & -5 & -3 & -2 \\ 4 & 7 & 6 & 8 \\ 6 & 10 & 1 & 12 \end{array} \right] \rightarrow \left[\begin{array}{cccc} 2 & 4 & 2 & 3 \\ -1 & -1 & -1 & 1 \\ 2 & -1 & 2 & 2 \\ 3 & -2 & -5 & 3 \end{array} \right] \rightarrow \left[\begin{array}{cccc} 2 & 4 & 2 & 3 \\ -1 & -1 & -1 & 1 \\ 1 & 3 & 1 & 1 \\ 3 & 2 & -3 & 1 \end{array} \right] \rightarrow \left[\begin{array}{cccc} 2 & 4 & 2 & 3 \\ -1 & -1 & -1 & 1 \\ 2 & 1 & 3 & 1 \\ 3 & 2 & -1 & 2 \end{array} \right].$$

Let's implement this algorithm. Starting with the first column, we move right. We zero out each element in the j th column below the j th row using elementary row operations. And here's the switch—we backfill those zeros with the values from \mathbf{L} .

```

for j = 1, ..., n
    for i = j + 1, ..., n
        [ aij ← aij / ajj
          for k = j + 1, ..., n
              [ aik ← aik - aij ajk ] ]
    for i = 2, ..., n
        [ bi ← bi - ∑j=1i-1 aij bj ] } forward elimination
    for i = n, n - 1, ..., 1
        [ bi ← (bi - ∑j=i+1n aij bj) / aii ] } backward elimination
}

```

The corresponding Julia code is

```

function gaussian_elimination(A,b)
    n = size(A,1)
    for j in 1:n
        A[j+1:n,j] /= A[j,j]
        A[j+1:n,j+1:n] -= A[j+1:n,j:j].*A[j:j,j+1:n]
    end
    for i in 2:n
        b[i:i] -= A[i:i,1:i-1]*b[1:i-1]
    end
    for i in n:-1:1
        b[i:i] = ( b[i] .- A[i:i,i+1:n]*b[i+1:n] )/A[i,i]
    end
    return b
end

```

Julia's built-in LAPACK functions for Gaussian elimination are substantially faster than the above script. The code above takes about 4 seconds to solve a 1000×1000 system, whereas Julia's built-in function takes about 0.02 seconds.

► Operation count

We can count the number of operations to gauge the complexity of Gaussian elimination. To produce the LU decomposition of an $n \times n$ matrix requires

$$\sum_{j=1}^n \sum_{i=j+1}^n \left(1 + \sum_{k=j+1}^n 2 \right) = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n$$

additions and multiplications. Forward and backward elimination each require

$$\sum_{i=1}^n \left(2 + \sum_{j=1}^{i-1} 2 \right) = n^2 + n$$

operations. When n is large, Gaussian elimination requires about $\frac{2}{3}n^3$ operations to get the initial decomposition and $2n^2$ to solve the two triangular systems.

Breaking Gaussian elimination into an LU decomposition step and forward-backward elimination steps is especially useful when we solve a system like $\mathbf{Ax}(t) = \mathbf{b}(t)$ where the term $\mathbf{b}(t)$ changes in time. For example, we might want to determine the changing heat distribution $\mathbf{x}(t)$ in a room with a heat source $\mathbf{b}(t)$. Computing the LU decomposition is the most expensive step requiring $\frac{2}{3}n^3$ operations. But we only need to do it once. After that, we only need to perform $2n^2$ operations at each time step to solve the two triangular systems.

► Pivoting

If the pivot a_{ii} is zero at any step, Gaussian elimination fails because we are dividing by zero. In practice, even if a pivot is close to zero, Gaussian elimination may be unstable because roundoff errors may be amplified. Consider the following matrix and its LU decomposition without row exchanges

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2+\varepsilon & 0 \\ 4 & 14 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 10/\varepsilon & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 2 \\ 0 & \varepsilon & -4 \\ 0 & 0 & 40/\varepsilon - 4 \end{bmatrix}.$$

Let $\varepsilon = 10^{-15}$, or about five times machine epsilon. The condition number of this matrix is $\kappa_2 \approx 10.3$, so it is numerically pretty well-conditioned. But because of round-off error, the computed LU decomposition is

$$\tilde{\mathbf{A}} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 10/\varepsilon & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 2 \\ 0 & \varepsilon & -4 \\ 0 & 0 & 40/\varepsilon \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2+\varepsilon & 0 \\ 4 & 14 & 8 \end{bmatrix}.$$

The original matrix \mathbf{A} and the effective matrix $\tilde{\mathbf{A}}$ differ in the bottom-right elements. In truth, the bottom-right element of \mathbf{A} cannot easily be determined

because of rounding error in $8 + 40/\varepsilon - 40/\varepsilon$. For $\mathbf{b} = (-5, 10, 0)^\top$, $\mathbf{A}^{-1}\mathbf{b} \approx (5, 0, -5)^\top$ whereas $\tilde{\mathbf{A}}^{-1}\mathbf{b} = (3, 2, -5)^\top$, an error of about $(2, -2, 0)$. The instability is the result of dividing by a small pivot. To avoid this instability, we should instead divide by a larger number by using partial or complete pivoting in each iteration.

In *partial pivoting*, we permute the rows of the matrix to put the largest element in the pivot position a_{ii} . With each step of LU factorization, we find $r = \arg \max_{k \geq i} |a_{kj}|$ and interchange row i and row r . To permute the rows, we left-multiply by a permutation matrix \mathbf{P} that keeps track of all the row interchanges $\mathbf{PA} = \mathbf{LU}$. As a result of pivoting, the magnitude of all of the values of matrix \mathbf{L} are less than or equal to one.

In *complete pivoting*, we permute both the rows *and* the columns to put the maximum element in the pivot position a_{ii} . With each step of LU factorization, find $r, c = \arg \max_{k, l \geq i} |a_{kl}|$, and interchange row i and row r and interchange column i and column c . To interchange rows we left-multiply by a permutation matrix \mathbf{P} and to interchange columns we right-multiply by a permutation matrix \mathbf{Q} , i.e., $\mathbf{PAQ} = \mathbf{LU}$. Applied to the problem $\mathbf{Ax} = \mathbf{b}$, complete pivoting yields $\mathbf{LU}(\mathbf{Q}^{-1}\mathbf{x}) = \mathbf{P}^{-1}\mathbf{b}$. Generally, complete pivoting is unnecessary and partial pivoting is sufficient.

2.2 Cholesky decomposition

A symmetric, positive-definite matrix \mathbf{A} has the factorization $\mathbf{R}^\top \mathbf{R}$ where \mathbf{R} is an upper triangular matrix. This factorization is called the Cholesky decomposition. Let's first prove this property and then examine how to implement it numerically.

Theorem 7. *A symmetric, positive-definite matrix \mathbf{A} has the decomposition $\mathbf{R}^\top \mathbf{R}$ where $\mathbf{R} = \mathbf{LD}^{1/2}$.*

Proof. The proof has several steps. First, We show that the inverse of a unit lower triangular matrix is a unit lower triangular matrix. It is easy to confirm the block matrix identity

$$\begin{bmatrix} \mathbf{A} & 0 \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & 0 \\ -\mathbf{D}^{-1}\mathbf{CA}^{-1} & \mathbf{D}^{-1} \end{bmatrix}.$$

The hypothesis is certainly true for 1×1 matrix $[1]$. Suppose that the hypothesis is true for an $n \times n$ unit lower triangular matrix. By letting \mathbf{D} be such a matrix and $\mathbf{A} = [1]$, then the claim follows for an $(n+1) \times (n+1)$ matrix.

Next, we show that if \mathbf{A} is symmetric and invertible, then \mathbf{A} has the decomposition $\mathbf{A} = \mathbf{LDL}^\top$, where \mathbf{L} is a unit lower triangular matrix and \mathbf{D} is a diagonal matrix. \mathbf{A} has the decomposition \mathbf{LDM}^\top , where \mathbf{M} is a unit lower triangular

matrix. We only need to show that $\mathbf{M} = \mathbf{L}$. We left multiply $\mathbf{A} = \mathbf{LDM}^T$ by \mathbf{M}^{-1} and right multiply it by \mathbf{M}^{-T} giving us

$$\mathbf{M}^{-1}\mathbf{AM}^{-T} = \mathbf{M}^{-1}\mathbf{LD}.$$

Note that $\mathbf{M}^{-1}\mathbf{AM}^{-T}$ is a symmetric matrix, and by theorem 7, the matrix $\mathbf{M}^{-1}\mathbf{LD}$ is lower triangular with diagonal \mathbf{D} . Hence, $\mathbf{M}^{-1}\mathbf{AM}^{-T}$ must be the diagonal matrix \mathbf{D} . So,

$$\mathbf{A} = \mathbf{LDM}^T = \mathbf{M}^{-1}\mathbf{AM}^{-T}\mathbf{M}^T = \mathbf{LM}^{-1}\mathbf{A}$$

from which it follows that $\mathbf{LM}^{-1} = \mathbf{I}$, or equivalently $\mathbf{L} = \mathbf{M}$.

Now, we show that if \mathbf{A} is an $n \times n$ symmetric, positive-definite matrix and \mathbf{X} is an $n \times k$ matrix with $\text{rank}(\mathbf{X}) = k$, then $\mathbf{B} = \mathbf{X}^T\mathbf{AX}$ is symmetric, positive definite. (We say that \mathbf{B} is *congruent* to \mathbf{A} if there exists an invertible \mathbf{X} such that $\mathbf{B} = \mathbf{X}^T\mathbf{AX}$.) Take $\mathbf{x} \in \mathbb{R}^k$. Then

$$\mathbf{x}^T\mathbf{Bx} = (\mathbf{X}\mathbf{x})^T\mathbf{A}(\mathbf{X}\mathbf{x}) \geq 0$$

and it equals 0 if and only if $\mathbf{X}\mathbf{x} = \mathbf{0}$. Since the $\text{rank}(\mathbf{X}) = k$, \mathbf{x} must be $\mathbf{0}$. Therefore, \mathbf{B} is positive definite.

Finally, we finish the proof. Because $\mathbf{A} = \mathbf{LDL}^T$ where \mathbf{L} is a unit lower triangular matrix and $\mathbf{D} = \mathbf{L}^{-1}\mathbf{AL}^{-T}$ is positive definite, it follows that the eigenvalues of \mathbf{D} (the diagonal elements of \mathbf{D}) are positive. So we can define $\mathbf{R}^T = \mathbf{LD}^{1/2}$. \square

As one might expect, the algorithm for Cholesky decomposition is similar to the algorithm for LU decomposition without pivoting. We'll formulate the algorithm by looking at how \mathbf{R}^T multiplies with \mathbf{R} to give us \mathbf{A} :

$$\begin{bmatrix} r_{11} & & & \\ r_{12} & r_{22} & & \\ \vdots & & \ddots & \\ r_{1n} & r_{2n} & \dots & r_{nn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ & r_{22} & & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{12} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{bmatrix}$$

Starting with a_{11} :

$$r_{11}r_{11} = a_{11} \quad \text{from which} \quad r_{11} = \sqrt{a_{11}}$$

$$\begin{bmatrix} \bullet & & & \\ \circ & \circ & & \\ \circ & \circ & \circ & \\ \circ & \circ & \circ & \\ \circ & \circ & \circ & \circ \end{bmatrix} \begin{bmatrix} \bullet & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}$$

where \bullet is the now known and can be used in subsequent steps and \circ are still unknowns. Once we have r_{11} we can find all other elements of the first column:

$$r_{11}r_{1i} = a_{1i} \quad \text{from which} \quad r_{1i} = a_{1i}/r_{11}$$

$$\begin{bmatrix} \bullet & \circ & & \\ \bullet & \circ & \circ & \\ \bullet & \circ & \circ & \\ \bullet & \circ & \circ & \circ \end{bmatrix} \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}$$

We continue for each remaining $n - 1$ columns of \mathbf{R} by first finding the i th diagonal element:

$$\sum_{k=1}^i r_{kj}r_{ki} = a_{ii} \quad \text{from which} \quad r_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} r_{ki}^2 \right)^{\frac{1}{2}}$$

$$\begin{bmatrix} \bullet & \circ & & \\ \bullet & \circ & \circ & \\ \bullet & \circ & \circ & \\ \bullet & \circ & \circ & \circ \end{bmatrix} \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}$$

Once we have the diagonal, we fill in the remaining elements from that column:

$$\sum_{k=1}^i r_{kj}r_{ki} = a_{ji} \quad \text{from which} \quad r_{ji} = \frac{1}{r_{ii}} \left(a_{ji} - \sum_{k=1}^{i-1} r_{kj}r_{ki} \right)$$

$$\begin{bmatrix} \bullet & \circ & & \\ \bullet & \circ & \circ & \\ \bullet & \circ & \circ & \\ \bullet & \circ & \circ & \circ \end{bmatrix} \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}$$

We can write this algorithm as the following pseudocode:

```
for i = 1, ..., n
     $r_{ii} \leftarrow \left( a_{ii} - \sum_{k=1}^{i-1} a_{ik}^2 \right)^{1/2}$ 
    for j = 2, ..., i
         $r_{ij} \leftarrow \frac{1}{r_{ii}} \left( a_{ji} - \sum_{k=1}^{i-1} r_{kj}r_{ki} \right)$ 
```

Just as in LU decomposition, we can overwrite the matrix \mathbf{A} successively. Cholesky decomposition requires $\frac{1}{3}n^3$ operations as opposed to $\frac{2}{3}n^3$ operations that general LU decomposition requires. Furthermore, pivoting is not needed in Cholesky decomposition, because the matrix is symmetric, positive definite.

2.3 Linear programming and the simplex method

Linear programming was developed in the 1940s by mathematicians George Dantzig, John von Neumann, and Leonid Kantorovich to solve problems of large-scale industrial and military production, management, and logistics. This development coincided with the development of digital computers that were needed to implement such solutions. A “program” is a schedule or plan of things to do.¹ By analyzing a system, determining the objective to be fulfilled, and

¹The word “program” has found its way into many different contexts referring to schedules or plans: an economic program, a theater program, a television program, a computer program.

constructing a statement of actions to perform (a program), the system can be represented by a mathematical model. The use of such mathematical models is called mathematical programming. That a number of military, economic, and industrial problems can be approximated by mathematical systems of linear inequalities and equations gives rise to linear programming (LP).

As a typical example of an LP problem, imagine that you are tasked with shipping canned tomatoes from several factories to different warehouses all scattered around the United States. Each of the factories can fill a certain number of cases per day and similarly each of the warehouses can sell a fixed number of cases per day. The cost of shipment from each factory to each warehouse varies by location. How do you schedule or program shipment to minimize the total transportation cost? An LP problem consists of decision variables (the quantities that the decision-maker controls), a linear objective function (the quantity to be maximized or minimized), and linear constraints (conditions that the solution must satisfy). In the cannery problem, the decision variables are the number of cases to ship from each factory to each warehouse, the linear objective function is the cost of shipping all of the cases, and the constraints are the number of cases each factory can fill and each store can sell.

Example. The Stigler diet problem is another typical LP problem. In 1944 economist George Stigler wondered what was the least amount of money a typical person would need to spend on food to maintain good health. To do this he examined 77 different foods along with nine nutrients (calories, protein, calcium, etc.). In this problem, the decision variables are the quantities of foods, the objective function is the total dollars spent, and the constraints are the minimum nutritional requirements. (Stigler [1945]) The simplex method, which we'll discuss below, wouldn't be developed for a few more years. Instead, Stigler found a solution heuristically by eliminating all but fifteen foods and then searching for the answer: \$39.93 per year (in 1939 prices or around \$730 today).

In 1947 Jack Laderman, at the National Bureau of Standards, revisited the Stigler diet problem using the recently developed simplex method and a team of nine clerks each armed with a hand-operated desk calculator. The team took approximately 120 person-days to obtain the optimal solution: \$39.69 per year, just a little better than Stigler.

George Dantzig revisited the problem in the 1950s, this time with the aid of a computer, to find himself a personal diet in an effort to lose weight. He adjusted the objective function to “maximize the feeling of feeling full,” which he then interpreted as the weight of food minus the weight of water content. He collected data on over 500 different foods, punched onto cards and fed into an IBM 701 computer. He recalled that the optimal diet was “a bit weird but conceivable” except that it also included several hundred gallons of vinegar. Reexamining the data, he noted that vinegar was listed as a weak acid with zero water content.

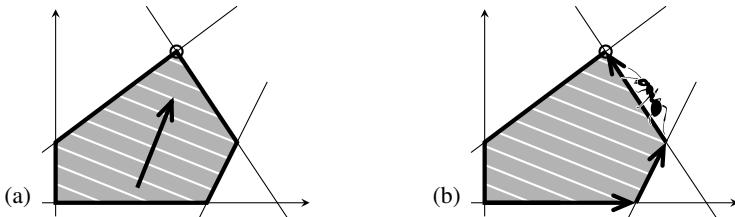


Figure 2.1: A two-dimensional simplex. (a) The objective function reaches its maximum and minimum at vertices of the simplex. (b) The simplex method steps along edges from vertex to vertex in a direction that increases the objective function, just as an ant might crawl, until it finally reaches a maximum.

Fixing the data, he tried to solve the problem again. This time it called for hundreds of bouillon cubes per day. No one at the time had thought to put upper limits on sodium intake. So, he added an upper constraint for salt. But now the program demanded two pounds of bran each day. And when he added an upper constraint on bran, it prescribed an equal amount of blackstrap molasses. At this point, he gave up. (Dantzig [1990]) ◀

The standard form of an LP problem is

Find the maximum of the objective function $z = \mathbf{c}^T \mathbf{x}$ subject to constraint $\mathbf{Ax} \leq \mathbf{b}$ and non-negativity restriction $\mathbf{x} \geq 0$.

We'll take \mathbf{A} to be an $m \times n$ matrix. If a problem is not already in standard form, there are several things we can do to put it there. Instead of finding the minimum of the objective function $\mathbf{c}^T \mathbf{x}$, we find the maximum of $-\mathbf{c}^T \mathbf{x}$. In place of a constraint $\mathbf{a}^T \mathbf{x} \geq b$, we write the constraint as $-\mathbf{a}^T \mathbf{x} \leq -b$. In place of an equation constraint $\mathbf{a}^T \mathbf{x} = b$, we use two inequality constraints $\mathbf{a}^T \mathbf{x} \leq b$ and $-\mathbf{a}^T \mathbf{x} \leq -b$. If a decision variable x is negative, we can take $-x$; and if the variable x is not restricted, we can redefine it $x = x^{(1)} - x^{(2)}$ where $x^{(1)}, x^{(2)} \geq 0$.

We can get a better understanding of the structure of an LP problem by sketching out the feasible region—the set of all possible points that satisfy the constraints. See the figure above. The nonnegativity condition restricts us to the positive orthant (the upper right quadrant for two dimensions). The system $\mathbf{Ax} = \mathbf{b}$ describes a set of hyperplanes (a set of lines in the two dimensions). So, assuming that the constraints are not inconsistent, the set $\mathbf{Ax} \leq \mathbf{b}$ carves out a convex n -polytope, also known as a simplex, in the positive orthant (a convex polygon in the upper right quadrant in two dimensions). The objective function $\mathbf{c}^T \mathbf{x}$ is itself easy enough to describe. The objective function increases linearly in the direction of its gradient \mathbf{c} . Imagine water slowly filling a convex polyhedral

vessel in the direction opposite of a gravitational force vector \mathbf{c} . The surface of the water is a level set given by $z = \mathbf{c}^\top \mathbf{x}$. As the water finally fills the vessel, its surface will either come to a vertex of the polyhedron—a unique solution—or it will come to a face or edge of the polyhedron—infinitely many solutions. This is often called the fundamental theorem of linear programming.

Theorem 8. *The maxima of a linear functional over a convex polytope occur at its vertices. If the values are at k vertices, then they must be along the k -cell between them.*

Proof. Let \mathbf{x}^* be a maximum of $\mathbf{c}^\top \mathbf{x}$ subject to $\mathbf{Ax} \leq \mathbf{b}$. Suppose that \mathbf{x}^* is in the interior of the polytope. Then we can move a small distance ε from \mathbf{x}^* in any direction and still be in the interior. Take the direction $\mathbf{c}/\|\mathbf{c}\|$. Then $\mathbf{c}^\top(\mathbf{x}^* + \varepsilon \mathbf{c}/\|\mathbf{c}\|) = \mathbf{c}^\top \mathbf{x}^* + \varepsilon \|\mathbf{c}\| > \mathbf{c}^\top \mathbf{x}^*$. But this says that \mathbf{x}^* is not a maximum. It follows that \mathbf{x}^* must be on the boundary of the polytope. If \mathbf{x}^* is not a vertex, then it is the convex combination of vertices: $\mathbf{x}^* = \sum_{i=1}^k \lambda_i \mathbf{x}_i$ with $\sum_{i=1}^k \lambda_i = 1$ and $\lambda_i \geq 0$. Then

$$0 = \mathbf{c}^\top \left(\mathbf{x}^* - \sum_{i=1}^k \lambda_i \mathbf{x}_i \right) = \sum_{i=1}^k \lambda_i (\mathbf{c}^\top \mathbf{x}^* - \mathbf{c}^\top \mathbf{x}_i).$$

Because \mathbf{x}^* is a maximum, $\mathbf{c}^\top \mathbf{x}^* \geq \mathbf{c}^\top \mathbf{x}_i$ for all i . And it follows that each term of the sum is nonnegative. The sum itself is zero, so all the terms must be zero. Hence, $\mathbf{c}^\top \mathbf{x}_i = \mathbf{c}^\top \mathbf{x}^*$ for each \mathbf{x}_i , i.e., every \mathbf{x}_i is maximal. And therefore all the points on the k -cell whose vertices are $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_p\}$ are maximal. \square

The diet problem that George Stigler solved had 15 decision variables along with 9 constraints. So, there are as many as $\binom{15}{9} = 5005$ possible vertices from which to choose a solution. That's not so many that the problem couldn't be solved using clever heuristics. But, the original problem with 77 decision variables has as many as $\binom{77}{9}$ or over 160 billion possibilities. How can we systematically pick the solution among all of these possibilities? George Dantzig's simplex solution was to pick any vertex and examine the edges leading away from that vertex, choosing any edge along which the cost variable was positive. If the edge is finite, then it connects to another vertex. Here you examine its edges and choose yet another one along which the cost variable is positive. Now continue like that, traversing from vertex to vertex along edges much as an ant might do as it climbs the simplex until arriving at a vertex for which no direction is strictly increasing. This vertex is the maximum. See Figure 2.1.

We can rewrite the system of constraints $\mathbf{Ax} \leq \mathbf{b}$ as a system of equations by introducing nonnegative slack variables \mathbf{s} such that $\mathbf{Ax} + \mathbf{s} = \mathbf{b}$. For “greater than” inequalities, slack variables are called surplus variables and are subtracted. From this, we have the standard equational form of the LP problem:

Find the maximum of the objective function $z = \mathbf{c}^T \mathbf{x}$ subject to constraint $\mathbf{Ax} \pm \mathbf{s} = \mathbf{b}$ and non-negativity restriction $\mathbf{x}, \mathbf{s}, \mathbf{b} \geq 0$.

If any elements of \mathbf{b} are negative, we can simply multiply the corresponding rows of \mathbf{A} by minus one to make \mathbf{b} nonnegative. The \pm operator is applied componentwise—i.e., to enforce nonnegativity of both \mathbf{b} and \mathbf{s} we either add or subtract the slack or surplus variables accordingly. We can write the objective function $z = \mathbf{c}^T \mathbf{x}$ as $\mathbf{c}^T \mathbf{x} + \mathbf{0}^T \mathbf{s} - z = 0$ where z is our still unknown cost and $\mathbf{0}$ is an m -dimensional vector of zeros called the reduced cost. The problem is now one of simplifying the system

$$\begin{bmatrix} \mathbf{A} & \mathbf{I} & \mathbf{0} \\ \mathbf{c}^T & \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{s} \\ -z \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \quad \text{where } \mathbf{x}, \mathbf{s} \geq 0.$$

The matrix \mathbf{I} is a diagonal matrix of $\{+1, -1\}$ depending on whether the initial constraint was a “less than” inequality (leading to a slack variable) or a “greater than” inequality (leading to a surplus variable). Equality constraints lead to an equation with a slack variable and an equation with a surplus variable. Consider the identical system

$$\begin{bmatrix} \bar{\mathbf{A}} & \mathbf{0} \\ \bar{\mathbf{c}} & 1 \end{bmatrix} \begin{bmatrix} \bar{\mathbf{x}} \\ -z \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \quad \text{where } \bar{\mathbf{A}} = [\mathbf{A} \quad \mathbf{I}], \bar{\mathbf{x}} = [\mathbf{x} \quad \mathbf{s}], \bar{\mathbf{c}} = [\mathbf{c} \quad \mathbf{0}], \text{ and } \bar{\mathbf{x}} \geq \mathbf{0}.$$

To help organize calculations, Dantzig introduced a tableau, which is updated at each iteration of the simplex method:

$$\begin{array}{c|c|c} \bar{\mathbf{A}} & \mathbf{0} & \mathbf{b} \\ \hline \bar{\mathbf{c}}^T & 1 & 0 \end{array} \tag{2.1}$$

The matrix $\bar{\mathbf{A}}$ has m rows and $m+n$ columns, so its rank is at most m . We’ll assume that the rank is n to keep the discussion simpler. Otherwise, we need to consider a few extra steps. We can choose m linearly independent columns as the basis for the column space of $\bar{\mathbf{A}}$. Let \mathbf{A}_B be the submatrix formed by these columns. The variables \mathbf{x}_B associated with these columns are called *basic variables*. The remaining n columns form a submatrix \mathbf{A}_N , and the variables \mathbf{x}_N associated with it are called the *nonbasic variables*. Altogether, we have $\bar{\mathbf{A}}\bar{\mathbf{x}} = \mathbf{A}_B\mathbf{x}_B + \mathbf{A}_N\mathbf{x}_N = \mathbf{b}$. By setting all nonbasic variables to zero, the solution is simply given by the basic variables. Such a solution is called a *basic feasible solution* when the nonnegativity restriction $\bar{\mathbf{x}} \geq 0$ is enforced. How do we ensure that it is enforced?

Our strategy will be to start with the basic feasible solution equal to the slack variables and then systematically swap nonbasic and basic variables so that the cost function increases and the nonnegativity restrictions are not broken. At each

step, we choose an entering variable from \mathbf{x}_N and a leaving variable from \mathbf{x}_B , along with their respective columns in \mathbf{A}_N and \mathbf{A}_B . We can use elementary row operations to convert the newly added column in \mathbf{A}_B to a standard basis vector (a column of the identity matrix). In this way, by taking nonbasic vectors $\mathbf{x}_N = \mathbf{0}$, we will always keep $\mathbf{x}_B = \mathbf{b}$, up to a permutation. If \mathbf{b} is nonnegative, then we know that \mathbf{x} is nonnegative. So, we need to choose prospective pivots accordingly. The entering basic variable determines the pivot column and the leaving basic variable determines the pivot row. At each iteration, we want to increase the objective function. So choose any column j for which c_j is positive. Next, we want to ensure that none of the elements of \mathbf{b} ever become negative when we perform row reduction. So, we will choose the row i for which a_{ij} is positive and the ratio b_i/a_{ij} is the smallest. Because we will always set nonbasic variables to zero, using row operations to zero out the cost function on the entering basic variable will automatically update the objective value: $-z + \bar{\mathbf{c}}^T \bar{\mathbf{x}} = 0$. We can summarize the simplex algorithm as

1. Build a simplex tableau (2.1) using \mathbf{A} , \mathbf{b} , and \mathbf{c} .
2. Take the basic variables to initially be the set of slack variables.
3. Repeat the following steps until no positive elements of $\bar{\mathbf{c}}$ remain:
 - a) Choose a pivot column j with positive \bar{c}_j
 - b) Choose the pivot row $i = \arg \min_i \{b_i/a_{ij} \text{ with } a_{ij} > 0\}$.
 - c) Use elementary row operations to set the pivot to one and to zero out all other elements in the column.

The following Julia code implements a naïve simplex method. We start by defining the function used for pivot selection and row reduction. We don't need to explicitly track entering and leaving variables, because we can sort it out in the end. The $[0 \ 1]^T$ column of the tableau (2.1) doesn't change, so we don't need to include it in the code. We can write the row reduction function as an in-place assignment operator.

```
function row_reduce!(tableau)
    (i,j) = get_pivot(tableau)
    G = tableau[i:i,:]/tableau[i,j]
    tableau[:,:] -= tableau[:,j:j]*G
    tableau[i,:] = G
end

function get_pivot(tableau)
    j = argmax(tableau[end,1:end-1])
    a, b = tableau[1:end-1,j], tableau[1:end-1,end]
    k = findall(a.>0)
    i = k[argmin(b[k]./a[k])]
    return(i,j)
```

```
end
```

Now we can write the simplex algorithm:

```
function simplex(c,A,b)
(m,n) = size(A)
tableau = [[A I b] ; [c' zeros(1,m) 0]]
while (any(tableau[end,1:n].>0))
    row_reduce!(tableau)
end
p = findall(tableau[end,1:n].==0)
x = zeros(n,1)
[x[i]=tableau[:,i]*tableau[:,end] for i in p]
z = -tableau[end,end]
y = -tableau[end,n.+(1:m)]
return((z=z, x=x, y=y))
end
```

Example. Consider the following LP problem: “Find the maximum of the objective function $2x + y + z$ subject to the constraints $2x + z \leq 3$, $4x + y + 2z \leq 2$, and $x + y \leq 1$.” The program can be solved using

```
A = [2 0 1;4 1 2;1 1 0]; c = [2;1;1]; b = [3;2;1]
solution = simplex(c,A,b)
```

To better illustrate the simplex method, it will be helpful to add the command

```
display(round.(tableau,digits=2))
```

to the while loop. The initial tableau consists of six blocks \mathbf{A} , \mathbf{I} , \mathbf{b} , \mathbf{c}^T , $\mathbf{0}$ and 0 :

2.00	0.00	1.00	1.00	0.00	0.00	3.00
4.00	1.00	2.00	0.00	1.00	0.00	2.00
1.00	1.00	0.00	0.00	0.00	1.00	1.00
2.00	1.00	1.00	0.00	0.00	0.00	0.00

The simplex algorithm scans the bottom row for the largest value and chooses that column as pivot column. It next determines the pivot row by selecting the largest positive element in that column relative to the matching element in the last column.

2.00	0.00	1.00	1.00	0.00	0.00	3.00
4.00	1.00	2.00	0.00	1.00	0.00	2.00
1.00	1.00	0.00	0.00	0.00	1.00	1.00
2.00	1.00	1.00	0.00	0.00	0.00	0.00

The algorithm then uses row reduction to zero out the elements in the pivot column and set the coefficient of the pivot row to a one. Now, it repeats the loop choosing the next pivot column and pivot row.

0.00	-0.50	0.00	1.00	-0.50	0.00	2.00
1.00	0.25	0.50	0.00	0.25	0.00	0.50
0.00	0.75	-0.50	0.00	-0.25	1.00	0.50
0.00	0.50	0.00	0.00	-0.50	0.00	-1.00

The algorithm repeats row reduction and pivot column and row selection.

0.00	0.00	-0.33	1.00	-0.67	0.67	2.33
1.00	0.00	0.67	0.00	0.33	-0.33	0.33
0.00	1.00	-0.67	0.00	-0.33	1.33	0.67
0.00	0.00	0.33	0.00	-0.33	-0.67	-1.33

After row reduction, there are more remaining positive values in the bottom row. The simplex algorithm has found a solution.

0.50	0.00	0.00	1.00	-0.50	0.50	2.50
1.50	0.00	1.00	0.00	0.50	-0.50	0.50
1.00	1.00	0.00	0.00	0.00	1.00	1.00
-0.50	0.00	0.00	0.00	-0.50	-0.50	-1.50

The negative value of the cost variable `solution.z` is in the lower left corner. The second and third columns are reduced. They correspond to basic variables. The first column is not reduced. It corresponds to a nonbasic variable, which we set to zero. From the values in the right-most column, we have that the solution `solution.x` is $(x, y, z) = (0, 1.0, 0.5)$. ◀

The bottom row of the reduced tableau also gives us the solution to a related problem. An LP problem has two related formulations—the primal problem and the dual problem. Every variable of the primal LP is a constraint in the dual LP, every constraint of the primal LP is a variable in the dual LP, and a maximum objective function in the primal is a minimum in the dual (and vice versa). Until now, we've only considered the primal problem:

Find the maximum of the objective function $z = \mathbf{c}^T \mathbf{x}$ subject to constraint $\mathbf{Ax} \leq \mathbf{b}$ and non-negativity restriction $\mathbf{x}, \mathbf{b} \geq 0$.

The dual of this problem is

Find the minimum of the objective function $z = \mathbf{b}^T \mathbf{y}$ subject to constraint $\mathbf{A}^T \mathbf{y} \geq \mathbf{c}$ and non-negativity restriction $\mathbf{y}, \mathbf{c} \geq 0$.

The strong duality theorem states four possibilities: 1. both the primal and dual are infeasible (no feasible solutions); 2. the primal is infeasible and the dual is unbounded; 3. the dual is infeasible and the primal is unbounded; or 4. both the primal and dual have feasible solutions and their values are the same. This means that we can solve the dual problem instead of the primal problem.

Example. The LP problem “find the minimum of the objective function $3x + 2y + z$ subject to the constraints $2x + 4y + z \leq 2$, $y + z \leq 1$, and $x + 2y \leq 1$ ” is the dual to the primal LP problem in the previous example. Its solution is given by `solution.y`, or $(x, y, z) = (0, 0.5, 0.5)$ where cost function has a minimum value of 1.5. ◀

• JuMP.jl (“Julia for Mathematical Programming”) is a modeling language for formulating optimization problems in Julia. JuMP interfaces with a number of external solvers such as GLPK.jl, COSMO.jl, and Tulip.jl.

2.4 Sparse matrices

James H. Wilkinson defines a *sparse matrix* as “any matrix with enough zeros that it pays to take advantage of them” in terms of memory and computation time. If we are smart about our implementation of Gaussian elimination, each zero in a matrix means potentially many saved computations. So, it is worthwhile to use algorithms that maintain sparsity by preventing fill-in. In Chapter 5, we’ll look at iterative methods for sparse matrices.

• The SparseArrays.jl package includes several utilities for constructing sparse matrices.

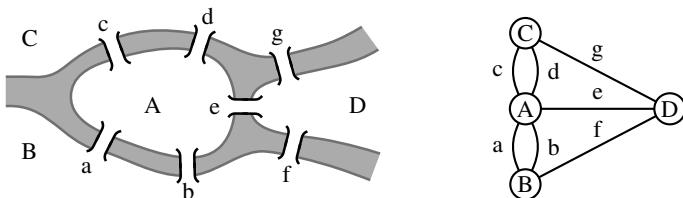
• Julia’s backslash and forwardslash operators use UMPACK (pronounced “Umph Pack”) to directly solve unsymmetric, sparse systems.

• The Plots.jl function `spy(A)` returns the sparsity plot of a matrix `A`.

► Graphs

In 1735 mathematician Leonhard Euler solved a puzzle asking whether it was possible to walk about the city of Königsberg, crossing each of the seven bridges

connecting the banks of the Pregel River and its two islands exactly once—no backtracking or double crossing.²



Euler found the solution by replacing the islands and mainland river banks with points called *vertices* or *nodes* and the bridges with line segments called *edges* that linked the vertices together. By counting the *degree* of each vertex—that is, the number of edges associated with each vertex—Euler was able to confidently answer “no.” A *path* or *walk* is a sequence of edges that join a sequence of vertices. An Eulerian path is a path that visits every edge exactly once, the type of path that the Königsberg bridge puzzle sought to find. An Eulerian path can have at most two vertices with odd degree—one for the start of the walk and one for the end of it. The vertices of the Königsberg graph have degrees 5, 3, 3, and 3. This puzzle laid the foundation for the mathematical discipline called graph theory, which examines mathematical structures that model pairwise relationships between objects.

Nowadays, graph theory is used to describe any number of things. When describing the internet, vertices are websites and edges are hyperlinks connecting those websites together. In social networks, vertices are people and edges are the social bonds between pairs of individuals. In an electrical circuit, vertices are junctions and edges are resistors, inductors, capacitors, and other components.

We can use sparse matrices to analyze graph structure. Similarly, we can use graphs to visualize the structure of sparse matrices. An *adjacency matrix* \mathbf{A} is a symmetric $(0, 1)$ -matrix indicating whether or not pairs of vertices are adjacent. The square of an adjacency matrix \mathbf{A}^2 tells us the number of paths of length two joining a particular vertex with another particular one. The k th power of an adjacency matrix \mathbf{A}^k tells us the number of paths of length k joining a particular vertex with another particular one. A graph is said to be *connected* if there exists a path between every two vertices. Otherwise, the graph consists of multiple components, the sets of all vertices connected by paths and connected to no other vertices of the graph.

²Euler's paper was published in 1741. A hundred years later Königsberg native Gustav Kirchhoff published his eponymous electrical circuits laws. One can wonder if walking along the bridges inspired him. Königsberg is now Kalingrad, Russia. Only two of the original bridges remain today. One was completely rebuilt in 1935, two were destroyed during the bombing of Königsberg in World War II, and another two others were later demolished to make way for a modern highway.

One way to measure the connectivity of a graph is by examining the Laplacian matrix of a graph. The *Laplacian matrix* (sometimes called the graph Laplacian or Kirchhoff matrix) for a graph is $\mathbf{L} = \mathbf{D} - \mathbf{A}$, where the *degree matrix* \mathbf{D} is a diagonal matrix that counts the degrees of each vertex and \mathbf{A} is the adjacency matrix. The matrix \mathbf{L} is symmetric and positive semidefinite. Because \mathbf{L} is symmetric, its eigenvectors are all orthogonal. The row and column sums of \mathbf{L} are zero. From this it follows that the vector of all ones is in the nullspace, i.e., the vector of all ones is an eigenvector. Finally, the number of connected components equals the dimension of the nullspace, i.e., the multiplicity of the 0 eigenvalue.

As its name might suggest, the Laplacian matrix also has a physical interpretation. Imagine an initial heat distribution \mathbf{u} across a graph where u_i is the temperature at vertex i . Suppose that heat moves from vertex i to vertex j proportional to $u_i - u_j$ if the two vertices share an edge. We can model the heat flow in the network as

$$\frac{d\mathbf{u}}{dt} = -(\mathbf{D} - \mathbf{A})\mathbf{u} = -\mathbf{Lu},$$

which is simply the heat equation if we replace $-\mathbf{L}$ with the Laplacian Δ . A solution $\mathbf{u}(t)$ can be written using the eigenvector basis \mathbf{L} . Namely,

$$\mathbf{u}(t) = \sum_{i=1}^n c_i(t)\mathbf{v}_i$$

for eigenvalues \mathbf{v}_i , where $c_i(0)$ is the spectral projection of the initial state $\mathbf{u}(0)$. The solution to the decoupled system is $c_i(t) = c_i(0)e^{-\lambda_i t}$, where the eigenvalues of \mathbf{L} are $0 = \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$. What happens to the solution as $t \rightarrow \infty$? If the graph is simply connected, the equilibrium solution is given by the eigenvector associated with the zero eigenvalue, i.e., the null space of \mathbf{L} . In other words, the distribution across the vertices converges to the average value of the initial state across all vertices—what we should expect from the heat equation.³

In moving from the a perturbed initial state to the equilibrium state, the large-eigenvalue components decay rapidly and the small-eigenvalue components linger. The zero eigenvector is the smoothest and corresponds to all constants. The next eigenvector is next smoothest, and so on. Near equilibrium, the solution can be approximated using the smallest eigenvalues λ_2 and λ_3 . This gives us a way to draw the graph in two-dimensions in a meaningful way—compute the two smallest nonzero eigenvalues of the graph Laplacian and the unit eigenvectors associates the these eigenvalues. Place vertex i at the coordinates given by the i th component of \mathbf{v}_2 and the i th component of \mathbf{v}_3 . This approach is called spectral

³ Alternatively, we can imagine the graph as a set of nodes linked together by springs. In this case, the governing equation is the wave equation $\mathbf{u}_{tt} = -\mathbf{Lu}$. The solution that minimizes the Dirichlet energy is the one that minimizes oscillations making the solution as smooth as possible.

layout drawing. To dig deeper, see Fan Chung and Ronald Graham's *Spectral Graph Theory*.

Another popular method for drawing graphs also uses an equilibrium solution to a physical problem. The force-directed graph drawing (developed by Thomas Fruchterman and Edward Reingold in 1991) models edges as springs that obey Hooke's law and vertices as electrical charges that obey Coulomb's law. The force at each node is given by $f_{ij} = a_{ij} + r_{ij}$ where

$$\begin{aligned} a_{ij} &= k^{-1} \|\mathbf{x}_i - \mathbf{x}_j\|, && \text{if } i \text{ connects to } j \\ r_{ij} &= -\alpha k^2 \|\mathbf{x}_i - \mathbf{x}_j\|^{-2}, && \text{if } i \neq j \end{aligned}$$

where k is a parameter adjusts the optimal distance between vertices and α is an additional adjustment parameter. To keep the graph to a two-dimensional unit square, we might choose the parameter k to be one over the square root of the number of vertices. Because we are simply trying to get a visually pleasing graph with separated vertices, we don't need to solve the problem precisely. We start with a random placement of the nodes. Then we iterate, updating the position of each vertex sequentially

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t \sum_{i=1}^n f_{ij} \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|},$$

where the step size Δt is a chosen sufficiently small to ensure stability. Because the force on each vertex is affected by the position of every other vertex and because it may take potentially hundreds of iterations to reach the equilibrium solution, force-directed graph drawing may require significant computation time.

- The LightGraphs.jl and GraphPlot.jl libraries provide utilities for constructing, analyzing, and visualizing graphs.

Example. The bottlenose dolphins living in Doubtful Sound, a fjord in the far south west of New Zealand, form a small population that has decreased significantly over the past thirty years. Researchers studying this community of 62 dolphins observed that they appeared to have greater cooperation and group stability to survive in the ecological constraints of the southern fjord. (Lusseau et al. [2003]) Their data set provides a glimpse into the social dynamics of the dolphins. The adjacency matrix is available from the SuiteSparse Matrix Collection (<https://sparse.tamu.edu>) maintained by Timothy Davis.⁴ Let's draw the graph of the frequent associations of the dolphins. We'll first define a helper

⁴The collection is a curated set of around 3000 sparse matrices that occur in applications such as computation fluid dynamics, financial modeling, and structural engineering. SuiteSparse also maintains the software library of sparse matrix routines such as UMFPACK, SSMULT, and SPQR used in Julia, Python, and Matlab.

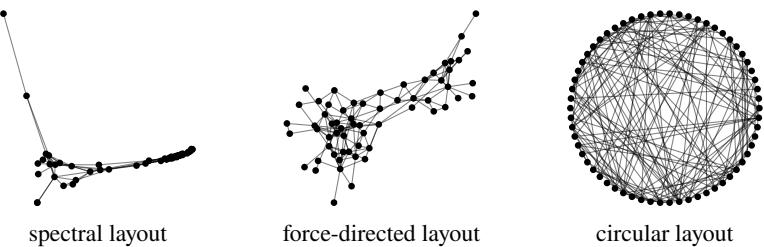


Figure 2.2: Graph drawing layouts for the bottlenose dolphin network.

function to download and build a sparse adjacency matrix.

```
using DelimitedFiles, SparseArrays
function get_adjacency_matrix(filename)
    ij = readdlm(download(bucket*filename*.csv"), ',', Int)
    sparse(ij[:,1], ij[:,2], one.(ij[:,1]))
end
```

Then, we'll plot the graph

```
using LightGraphs, GraphPlot
g = get_adjacency_matrix("dolphins") |> Graph
gplot(g, layout=spectral_layout)
```

The graphs using `spectral_layout`, `spring_layout`, and `circular_layout` are shown in the figure above. We can see in the force-directed layout how the dolphin group appear to consist of two smaller two pods. The circular layout draw the vertices at equally spaced points along the circumference of a circle. It is difficult to interpret the associations between the dolphins using this layout because there are so many crossings. In the next section, we'll develop method of sorting the vertices to minimize the bandwidth. ◀

► Preserving sparsity

Suppose that we have the 4×6 full matrix

$$\mathbf{A} = \begin{bmatrix} 11 & 0 & 0 & 14 & 0 & 16 \\ 0 & 22 & 0 & 0 & 25 & 26 \\ 0 & 0 & 33 & 34 & 0 & 36 \\ 41 & 0 & 43 & 44 & 0 & 46 \end{bmatrix}.$$

If the number of nonzero elements is small we could store the matrix more efficiently by just recording the locations and values of the nonzero elements.

For example,

column	1	1	2	3	3	4	4	4	5	6	6	6	6
row	1	4	2	3	4	1	3	4	2	1	2	3	4
value	11	41	22	33	43	14	34	44	25	16	26	36	46

To store this array in memory, we can go a step further and use *compressed sparse column* (CSC) format. Rather than saving explicit column numbers, compressed column format data structure only saves pointers to where the new columns begin.

index	1	3	4	6	9	10		14
row	1	4	2	3	4	1	3	4
value	11	41	22	33	43	14	34	44

Similarly, the compressed sparse row (CSR) format indexes arrays by rows instead of columns. For languages like Julia, Matlab, R, and Fortran that use a column-major order convention for storing data, a CSC format is most efficient. For languages like Python and C that use a row-major convention, a CSR format is most efficient.

A matrix \mathbf{A} is called a *banded matrix* with lower bandwidth p and upper bandwidth q if $a_{ij} = 0$ whenever $i > j + p$ or $i < j - q$. For example, a diagonal matrix has lower and upper bandwidths of 0 and a tridiagonal matrix has lower and upper bandwidths of 1. Banded matrices often arise in finite difference and finite element approximations to one-dimensional partial differential equations using either

If a banded matrix \mathbf{A} has a lower bandwidth p and an upper bandwidth q , then the LU decomposition of \mathbf{A} produces a lower triangular matrix \mathbf{L} with a lower bandwidth of p and an upper triangular matrix \mathbf{U} with an upper bandwidth of q . Furthermore, the number of multiplications and additions is approximately $2npq$. Similarly, forward and backward substitution requires $2np + 2nq$ operations.

Consider the simple example of the following sparse matrix:

$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$ has the LU decomposition

with a tremendous amount of fill-in. But, if we reorder the rows and columns first by moving the first row to the bottom and the first column to the end, then

$\begin{bmatrix} \bullet & & \\ \bullet & \ddots & \\ & \ddots & \bullet \end{bmatrix}$ has the LU decomposition $\begin{bmatrix} \bullet & & \\ \bullet & \ddots & \\ & \ddots & \bullet \end{bmatrix} \begin{bmatrix} I & & \\ & \ddots & \\ & & I \end{bmatrix}$.

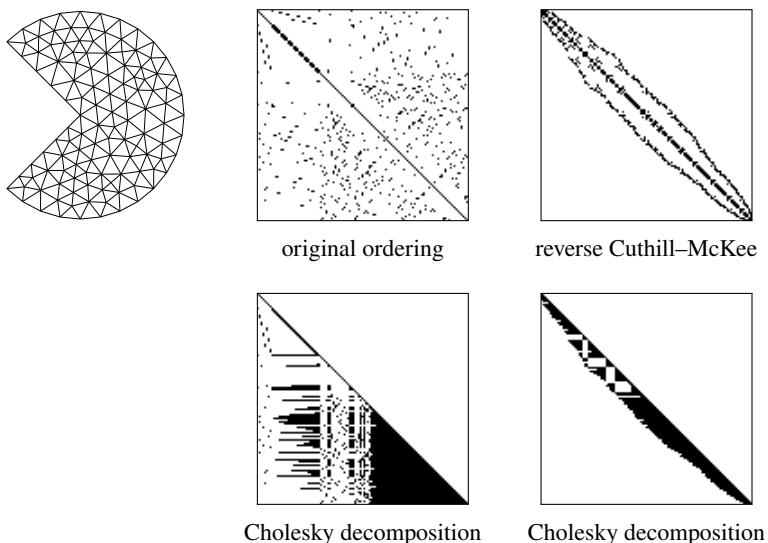


Figure 2.3: The original ordering and the reverse Cuthill–McKee ordering of the sparsity plot for the Laplacian operator in a finite element solution for a pacman domain. The Cholesky decompositions for each ordering are directly below.

In the $n \times n$ case, the original matrix requires $O(n^3)$ operations to compute the LU decomposition plus another $O(n^2)$ operations to solve the problem. Furthermore, we need to store all n^2 terms of the LU decomposition. The reordered matrix, on the other hand, suffers from no fill-in. It only needs $4n$ operations (additions and multiplications) for the LU decomposition and $8n$ operations to solve the system.

We can lower the bandwidth of a sparse matrix and reduce the amount of fill-in by reordering the rows and columns to get the nonzero elements as close to the diagonal as possible. The Cuthill–McKee algorithm does this for symmetric matrices by using a breadth-first search to traverse the associated graph.⁵ The algorithm builds a permutation sequence starting with an empty set:

1. Choose a vertex with the lowest degree and add it to the permutation set.
2. For each vertex added to the permutation set, progressively add its adjacent vertices in order from lowest degree to highest degree, skipping any vertex that has already been included.

⁵Dijkstra's algorithm also uses breadth-first search—this time to determine the shortest path between nodes on a graph. Imagine that nodes represent intersections on a street map and the weighted edges are the travel times between intersections. Then, Dijkstra's algorithm can help route the fastest travel from one part of a city to another. In contrast, a depth-first search explores a path as far as possible before backtracking to take different path—a strategy one might use to solve mazes.

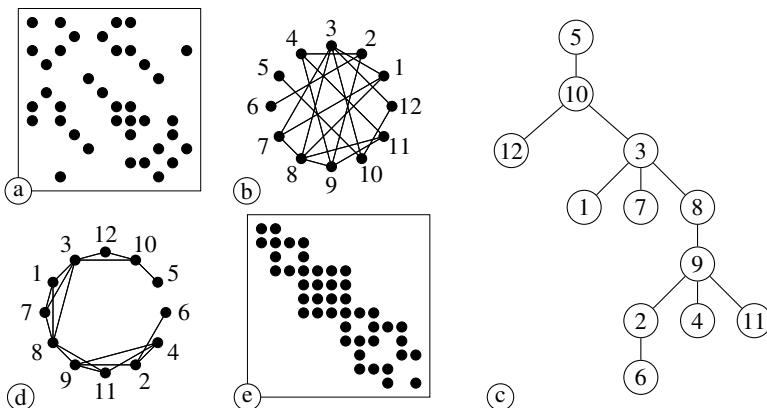


Figure 2.4: Cuthill–McKee sorting: (a) original sparsity pattern; (b) associated graph; (c) completed breadth-first search tree; (d) reordered graph; (e) sparsity pattern of column-and-row-permuted matrix.

3. If the graph is connected, repeating step 2 will eventually run through all vertices—at which point we’re done. If the graph is disconnected, repeating step 2 will terminate without reaching all of the vertices. In this case, we’ll need to go back to step 1 with the remaining vertices.

Consider the 12×12 matrix in the figure above. We start the Cuthill–McKee algorithm by selecting a column of \mathbf{A} that has the fewest off-diagonal nonzero elements—column 5 which has one nonzero element in row 10. We now find the uncounted nonzero elements in column 10—rows 3 and 12. We repeat the process, looking at column 3 and skipping any vertices that we’ve already counted. We continue in this fashion until we’ve run through all of the columns to build a tree. To get the permutation, start with the root node and subsequently collect vertices at each lower level to get $\{5, 10, 12, 3, 1, 7, 8, 9, 2, 4, 11, 6\}$. In practice, one typically uses the *reverse Cuthill–McKee* algorithm, which simply reverses the ordering of the relabeled vertices $\{6, 11, 4, 3, 9, 8, 7, 1, 3, 12, 10, 5\}$. The new graph is isomorphic to the original graph—the labels have changed but nothing else. Relabeling the vertices of the graph is identical to permuting the rows and columns of the symmetric matrix. Other resorting methods include minimum degree and nested dissection.

• Finite element packages such as JuliaFEM.jl and FinEtools.jl include functions for the Cuthill–McKee algorithm.

► Revised simplex method

Linear programming problems often involve large, sparse systems. Unfortunately, the simplex tableau, which may start with very few nonzero entries, can fill in with each iteration. In practice, it's not necessary to store or compute with the entire tableau at each iteration. We only need to select the basic variables in such a way that the objective function increases. The revised simplex method does exactly that. Remember the LP problem:

Find the maximum of the objective function $z = \bar{\mathbf{c}}^T \bar{\mathbf{x}}$ subject to constraint $\bar{\mathbf{A}}\bar{\mathbf{x}} = \mathbf{b}$ and non-negativity restriction $\bar{\mathbf{x}}, \mathbf{b} \geq 0$,

which has the tableau

$$\begin{array}{c|c|c} \bar{\mathbf{A}} & \mathbf{0} & \mathbf{b} \\ \hline \bar{\mathbf{c}}^T & 1 & 0 \end{array}.$$

At its simplest level, the simplex method involves sorting the basic variables from the nonbasic variables. We can decompose the augmented matrix into basis and non-basis columns $\bar{\mathbf{A}} = [\mathbf{A}_N \quad \mathbf{A}_B]$ such that $\mathbf{A}_N \mathbf{x}_N + \mathbf{A}_B \mathbf{x}_B = \mathbf{b}$. Although we represent them as grouped together, the columns of \mathbf{A}_N and the columns of \mathbf{A}_B may come from anywhere in $\bar{\mathbf{A}}$ as long as they are mutually exclusive. As with the regular simplex method, we'll again assume that $\bar{\mathbf{A}}$ has full rank. In this case, \mathbf{A}_B is invertible and

$$\mathbf{A}_B^{-1} \mathbf{A}_N \mathbf{x}_N + \mathbf{x}_B = \mathbf{A}_B^{-1} \mathbf{b}. \quad (2.2)$$

By setting the non-basic variables \mathbf{x}_N to zero, we have the solution $\mathbf{x}_B = \mathbf{A}_B^{-1} \mathbf{b}$. We choose the basic variables, and hence the columns of \mathbf{A}_B , to maximize the objective function

$$z = \mathbf{c}_N^T \mathbf{x}_N + \mathbf{c}_B^T \mathbf{x}_B = (\mathbf{c}_N^T - \mathbf{c}_B^T \mathbf{A}_B^{-1} \mathbf{A}_N) \mathbf{x}_N + \mathbf{c}_B^T \mathbf{A}_B^{-1} \mathbf{b}. \quad (2.3)$$

We can build a tableau from (2.2) and (2.3):

$$\begin{array}{c|c|c|c} \mathbf{A}_B^{-1} \mathbf{A}_N & \mathbf{I} & \mathbf{0} & \mathbf{A}_B^{-1} \mathbf{b} \\ \hline \mathbf{c}_N^T - \mathbf{c}_B^T \mathbf{A}_B^{-1} \mathbf{A}_N & \mathbf{0} & 1 & -\mathbf{c}_B^T \mathbf{A}_B^{-1} \mathbf{b} \end{array}.$$

At each iteration we need to keep track of which variables are basic and which ones are non-basic, but it will save us quite a bit of computing time if we also keep track of \mathbf{A}_B^{-1} .

We start with $\mathbf{A}_N = \mathbf{A}$ and $\mathbf{A}_B = \mathbf{I}$. At each iteration, we choose a new entering variable—associated with the pivot column—by looking for any (the first) positive element among the reduced cost variables $\mathbf{c}_N^T - \mathbf{c}_B^T \mathbf{A}_B^{-1} \mathbf{A}_N$. Let \mathbf{q} be that j th column of \mathbf{A}_N and $\hat{\mathbf{q}}$ be j th column of $\mathbf{A}_B^{-1} \mathbf{A}_N$, i.e. $\hat{\mathbf{q}} = \mathbf{A}_B^{-1} \mathbf{A}_N \xi_j$, where ξ_j is the j th standard basis vector. With our pivot column in hand, we now look for a pivot row i that will correspond to our leaving variable. We take

$i = \arg \min_i x_i/q_i$ with $q_i > 0$ to ensure that $\mathbf{x}_B = A_B^{-1}\mathbf{b}$ remains nonnegative. Let \mathbf{p} be the i th column of A_B , the column associated with the leaving variable.

After swapping columns \mathbf{p} and \mathbf{q} between A_B and A_N , we'll need to update A_B^{-1} . Fortunately, we don't need to recompute the inverse entirely. Instead, we can use the Sherman–Morrison formula, which gives a rank-one perturbation of the inverse of a matrix:

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}.$$

By taking $\mathbf{u} = \mathbf{q} - \mathbf{p}$ and $\mathbf{v} = \xi_i$, we have

$$(A_B + (q - p)\xi_i^T)^{-1} = A_B^{-1} - \hat{q}_i^{-1}(\hat{q} - \xi_i)\xi_i^TA_B^{-1}$$

where $\hat{\mathbf{q}} = A_B^{-1}\mathbf{q}$, $\hat{q}_i = \xi_i^T A_B^{-1} \mathbf{q}$, and $A_B^{-1}\mathbf{p} = \xi_i$. Note that this is equivalent to row reduction.

Let's implement the revised simplex method in Julia. We'll take $ABinv$ to be A_B^{-1} and B and N to be the list of columns of \bar{A} in A_B and A_N , respectively. We define a method $\xi(i)$ to return a unit vector. In practice, we can save and update $[A_B^{-1} \quad A_B^{-1}\mathbf{b}]$ instead of simply \mathbf{B}^{-1} and stop as soon as we find the first j .

```
using SparseArrays
function revised_simplex(c,A,b)
    (m,n) = size(A)
    ξ = i -> (z=spzeros(m);z[i]=1;z)
    N = Vector(1:n); B = Vector(n .+ (1:m))
    A = [A sparse(I, m, m)]
    ABinv = sparse(I, m, m)
    c = [c;spzeros(m,1)]
    while(true)
        j = findfirst(x->x>0,(c[N]' .-(c[B]'*ABinv)*A[:,N])[:,])
        if isnan(j); break; end
        q = ABinv*A[:,N[j]]
        k = findall(q.>0)
        i = k[argmin(ABinv[k,:]*b./q[k])]
        B[i], N[j] = N[j], B[i]
        ABinv -= ((q - ξ(i))/q[i])*ABinv[i:i,:]
    end
    i = findall(B.≤n)
    x = zeros(n,1)
    x[B[i]] = ABinv[i,:]*b
    return((z=c[1:n]'*x, x=x, y=c[B]'*ABinv))
end
```

2.5 Exercises

2.1. Modify the program on page 33 to include partial pivoting. Compare both algorithms to solve the system $\mathbf{Ax} = \mathbf{b}$ for

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 + \varepsilon & 0 \\ 4 & 14 & 4 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} -5 \\ 10 \\ 0 \end{bmatrix}$$

with $\varepsilon = 10^{-15}$ (or about five times machine epsilon). Note that because of machine rounding errors, you should not necessarily expect the solutions provided in the example on page 34.

2.2. Write the LU decomposition of the block matrix: $\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$.

- ✍ 2.3. LU decomposition can be used to efficiently compute the determinant of a matrix. (This is what Julia, Python, and Matlab do.) Write a program to find a determinant using LU decomposition.
- ✍ 2.4. Write a program to implement the reverse Cuthill–McKee algorithm for symmetric matrices.
- ✍ 2.5. Use the Cuthill–McKee algorithm to reorder the vertices in the circular graph layout of the bottlenose dolphin network in the example on page 48.
- 2.6. The nematode *Caenorhabditis elegans* is the first and only organism whose complete connectome, consisting of 277 interconnected neurons, has been mapped. The adjacency matrix for the connectome (from Choe et al. [2004]) is available at <https://github.com/nmfsc/data/celegans.csv> as edges of a bipartite graph. Use the Cuthill–McKee algorithm to order the nodes to minimize the bandwidth. See Reid and Scott’s article “Reducing the total bandwidth of a sparse unsymmetric matrix” for different approaches.
- ✍ 2.7. Use the simplex method to solve the Stigler diet problem to find the diet that meets minimum nutritional requirements at the lowest cost. The data from Stigler’s 1945 paper is available as “diet.csv” at <https://github.com/nmfsc/data/> (nutritional values of foods are normalized per dollar of expenditure).
- ✍ 2.8. In 1929 Hungarian author Frigyes Karinthy hypothesized that any person could be connected to any other person through a chain consisting of at most five intermediary acquaintances, a hypothesis now called six degrees of separation.⁶

⁶In Karinthy’s short story “Láncszemek” (“Chains”), the author explains how he is two steps removed from Nobel laureate Selma Lagerlöf and no more than four steps away from an anonymous riveter at the Ford Motor Company.

In a contemporary version of Karp's game called “six degrees of Kevin Bacon,” players attempt to trace an arbitrary actor to Kevin Bacon by identifying a chain of movies coappearances.⁷ The length of the shortest path is called the Bacon number. Kevin Bacon's Bacon number is zero. Anyone who has acted in a movie with him has Bacon number of one. Anyone who has acted with someone who has acted with Kevin Bacon has Bacon number of two. For example, John Belushi starred in the movie *Animal House* with Kevin Bacon, so his Bacon number is one. Danny DeVito never appeared in a movie with Kevin Bacon. But he did appear in the movie *Goin' South* with John Belushi, so his Bacon number is two. The path connecting Danny DeVito to Kevin Bacon isn't unique. Danny DeVito also acted in *The Jewel of the Nile* with Holland Taylor, who acted with Kevin Bacon in *She's Having a Baby*.

Write a graph search algorithm to find the path connecting two arbitrary actors. You can use the corpus of narrative American feature films from 1972 to 2012 to test your algorithm. The data, scraped from Wikipedia, is available at <https://github.com/nmfsc/data/>. The file “actors.txt” is a list of the actors, the file “movies.txt” is a list of the movies, and the CSV file “actor-movie.csv” identifies the edges of the bipartite graph. Because Wikipedia content is crowd-sourced across thousands of editors, there are inconsistencies across articles.

⁷Mathematicians sometimes play a nerdier game by tracking the chain of co-authors on research papers to the prolific Hungarian mathematician Paul Erdős.

Chapter 3

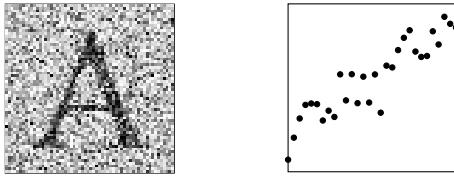
Inconsistent Systems



In the last chapter, we examined solutions to square, consistent systems. Now, we turn our attention to non-square matrices. Like the square system, we can solve a non-square system $\mathbf{Ax} = \mathbf{b}$ in Julia using the `(\)` function. Computing $\mathbf{A}\backslash\mathbf{b}$ for a 2000×1999 matrix takes 1.54 seconds, considerably longer than the 0.15 seconds to solve a 2000×2000 matrix. Alternatively, solving the same non-square system using the pseudoinverse `pinv(A)*b` takes 5.6 seconds—the inverse `inv(A)*b` takes 0.42 seconds for the square system. Both approaches—the backslash operator and the pseudoinverse—use orthogonal projection to reduce a problem to a lower-dimensional subspace in which it can be solved uniquely. In this chapter, we examine how QR decomposition and singular value decomposition are used to generate such subspaces. We also look at several applications of low-rank approximation.

3.1 Overdetermined systems

Often we want to solve a problem with noisy input data. Such problems include curve fitting, image recognition, and statistical modeling. We may have abundant data, but they are inconsistent, so the problem is ill-posed. The solution is to filter out the noise leaving us with consistent data. It is not difficult for us to mentally filter out the noise in the figures on the following page and recognize the letter on the left or the slope of the line on the right.



In this chapter, we'll develop the mathematical tools to instruct a computer to do the same. Consider the overdetermined system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{b} \in \mathbb{R}^m$ with $m > n$. The linear system does not have a solution unless \mathbf{b} is in the column space of \mathbf{A} , i.e., unless \mathbf{b} is a linear combination of the columns of \mathbf{A} . While we may not be able to find “the” solution \mathbf{x} , we will often be able to find a “best” solution $\tilde{\mathbf{x}}$, so that $\mathbf{A}\tilde{\mathbf{x}}$ is closest to \mathbf{b} . This means that we want to choose $\tilde{\mathbf{x}}$ that minimizes the norm of the *residual* $\tilde{\mathbf{r}} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}$.

So, which norm? The 1-, 2- and ∞ -norms are all relevant norms, but the 2-norm is Goldilocks’ baby bear of norms—not too hard and not too soft. The 1-norm minimizes the importance of outliers, the ∞ -norm only looks at the outliers, and the 2-norm takes the middle of the road. Also, some problems are ill-posed in the 1- and ∞ -norms. So, the 2-norm seems to be a natural choice. But wait, there's more! Perhaps the nicest property of the 2-norm is that the problem of finding a solution that minimizes the 2-norm of the residual is a linear problem. Recently, the 1-norm has gained some popularity in research and applications, because it handles noisy data better than the 2-norm by minimizing the influence of outliers (Candès and Wakin [2008]). But because the minimization problem in the 1-norm is nonlinear, more complex solvers must be used.¹

► The normal equation

Suppose that $\tilde{\mathbf{x}}$ minimizes $\|\mathbf{Ax} - \mathbf{b}\|_2^2$. Then for any arbitrary vector \mathbf{y}

$$\|\mathbf{A}(\tilde{\mathbf{x}} + \mathbf{y}) - \mathbf{b}\|_2^2 \geq \|\mathbf{A}\tilde{\mathbf{x}} - \mathbf{b}\|_2^2.$$

By rearranging the terms on the left-hand side, we have

$$\|(\mathbf{A}\tilde{\mathbf{x}} - \mathbf{b}) + \mathbf{Ay}\|_2^2 \geq \|\mathbf{A}\tilde{\mathbf{x}} - \mathbf{b}\|_2^2.$$

And then expanding,

$$\|\mathbf{A}\tilde{\mathbf{x}} - \mathbf{b}\|_2^2 + \|\mathbf{Ay}\|_2^2 + 2\mathbf{y}^T \mathbf{A}^T \mathbf{Ax} - 2\mathbf{y}^T \mathbf{A}^T \mathbf{b} \geq \|\mathbf{A}\tilde{\mathbf{x}} - \mathbf{b}\|_2^2,$$

which says

$$\|\mathbf{Ay}\|_2^2 + 2\mathbf{y}^T \mathbf{A}^T \mathbf{A}\tilde{\mathbf{x}} - 2\mathbf{y}^T \mathbf{A}^T \mathbf{b} \geq 0.$$

¹In statistics and machine learning applications, the 2-norm arises from the maximum likelihood estimation using a Gaussian distribution $\exp(-x^2)$. The 1-norm comes from maximum likelihood estimation using the Laplace distribution $\exp(-|x|)$.

or equivalently

$$2\mathbf{y}^T(\mathbf{A}^T\mathbf{A}\tilde{\mathbf{x}} - \mathbf{A}^T\mathbf{b}) \geq -\|\mathbf{Ay}\|_2^2.$$

Because this expression holds for any \mathbf{y} and because $\mathbf{A}^T\mathbf{A}\tilde{\mathbf{x}} - \mathbf{A}^T\mathbf{b}$ is independent of \mathbf{y} , it must follow that

$$\mathbf{A}^T\mathbf{A}\tilde{\mathbf{x}} - \mathbf{A}^T\mathbf{b} = 0.$$

From this, we get the *normal equation*

$$\mathbf{A}^T\mathbf{A}\tilde{\mathbf{x}} = \mathbf{A}^T\mathbf{b}.$$

Here's another way to get this equation. The residual $\|\mathbf{r}(\tilde{\mathbf{x}})\|_2$ is minimized when the gradient of $\|\mathbf{r}(\tilde{\mathbf{x}})\|_2$ is zero—equivalently, when the gradient of $\|\mathbf{r}(\tilde{\mathbf{x}})\|_2^2$ is zero. We have that

$$\|\mathbf{r}(\tilde{\mathbf{x}})\|_2^2 = \|\mathbf{Ax} - \mathbf{b}\|_2^2 = \|\mathbf{A}\tilde{\mathbf{x}}\|_2^2 - 2\tilde{\mathbf{x}}^T\mathbf{A}^T\mathbf{b} + \|\mathbf{b}\|_2^2,$$

and taking the gradient of this expression with respect to $\tilde{\mathbf{x}}$ gives us

$$0 = 2\mathbf{A}^T\mathbf{A}\tilde{\mathbf{x}} - 2\mathbf{A}^T\mathbf{b}.$$

Therefore, $\mathbf{A}^T\mathbf{A}\tilde{\mathbf{x}} = \mathbf{A}^T\mathbf{b}$.

Let's examine the normal equation by looking at the error $\mathbf{e} = \mathbf{x} - \tilde{\mathbf{x}}$ and the residual $\mathbf{r} = \mathbf{Ae} = \mathbf{b} - \delta\mathbf{b}$ where $\delta\mathbf{b} = \mathbf{Ae}$. The solution to the normal equation is

$$\tilde{\mathbf{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$$

from which

$$\tilde{\mathbf{b}} = \mathbf{A}\tilde{\mathbf{x}} = \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} = \mathbf{P}_A\mathbf{b},$$

where \mathbf{P}_A is the orthogonal projection matrix into the column space of \mathbf{A} . The residual is simply

$$\mathbf{r} = \mathbf{b} - \tilde{\mathbf{b}} = (\mathbf{I} - \mathbf{P}_A)\mathbf{b} = \mathbf{P}_{N(\mathbf{A}^T)}\mathbf{b}$$

where $\mathbf{P}_{N(\mathbf{A}^T)}$ is the orthogonal projection matrix into the left null space of \mathbf{A} . The normal equation gets its name because the residual $\mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}$ is normal to the column space of \mathbf{A} . Recall that we can write $\mathbf{Ax} = \mathbf{b}$ as

$$\mathbf{A}(\mathbf{x}_{\text{row}} + \mathbf{x}_{\text{null}}) = \mathbf{b}_{\text{column}} + \mathbf{b}_{\text{left null}}.$$

The system is inconsistent because $\mathbf{b}_{\text{left null}} \neq 0$. By simply zeroing out $\mathbf{b}_{\text{left null}}$, we get a solution. Put another way, we want to solve the problem $\mathbf{Ax} = \mathbf{b}$. But we can't, because it has no solutions. So, instead we adjust the problem to be $\mathbf{Ax} = \mathbf{b} - \delta\mathbf{b}$ where the $\delta\mathbf{b}$ puts the right-hand side into the column space of \mathbf{A} . The residual $\delta\mathbf{b}$ is none other than the left null space component of \mathbf{b} . Pretty remarkable! By furthermore zeroing out \mathbf{x}_{null} , we'll get a unique solution. We'll come back to this idea later in the chapter.

Direct implementation of the normal equation can make a difficult problem even worse because the condition number $\kappa(\mathbf{A}^\top \mathbf{A})$ equals $\kappa(\mathbf{A})^2$ and the condition number $\kappa((\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top)$ may be even larger. A moderately ill-conditioned matrix can lead to a very ill-conditioned problem. Instead, let's look at an alternative means of solving an overdetermined system by applying an orthogonal matrix to the original problem. Such an approach is more robust (numerically stable) and is almost as efficient as solving the normal equation.

► QR decomposition

Every real $m \times n$ matrix \mathbf{A} can be written as the product of an $m \times m$ orthogonal matrix \mathbf{Q} and an $m \times n$ upper triangular matrix \mathbf{R} . We'll constructively prove this later. This decomposition, known as *QR decomposition*, is the key idea of this chapter and will be used extensively in the next chapter as well. Let's see how we can use it.

We'll start with the normal equation $\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b}$. Taking $\mathbf{A} = \mathbf{QR}$, then

$$(\mathbf{QR})^\top \mathbf{QR}\mathbf{x} = (\mathbf{QR})^\top \mathbf{b},$$

which reduces to

$$\mathbf{R}^\top \mathbf{R}\mathbf{x} = \mathbf{R}^\top \mathbf{Q}^\top \mathbf{b}.$$

Note that on the left-hand side we simply have the Cholesky decomposition $\mathbf{R}^\top \mathbf{R}$ of $\mathbf{A}^\top \mathbf{A}$. By rearranging the terms of this equality we have

$$\mathbf{R}^\top (\mathbf{R}\mathbf{x} - \mathbf{Q}^\top \mathbf{b}) = \mathbf{0}$$

which says $\mathbf{R}\mathbf{x} - \mathbf{Q}^\top \mathbf{b}$ is in the null space of \mathbf{R}^\top (the left null space of \mathbf{R}):

$$\begin{bmatrix} \bullet & 0 & 0 & 0 & 0 & 0 \\ \bullet & 0 & 0 & 0 & 0 & 0 \\ \bullet & \bullet & 0 & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

So, $\mathbf{R}\mathbf{x}$ must equal $\mathbf{Q}^\top \mathbf{b}$ in the first m rows. In the other $m - n$ rows, it can be anything. Let's come at this again. Suppose that we have the overdetermined system $\mathbf{Ax} = \mathbf{b}$:

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \mathbf{x} = \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}.$$

The matrix $\mathbf{A} = \mathbf{QR}$ for some \mathbf{Q} . By applying $\mathbf{Q}^\top = \mathbf{Q}^{-1}$ to both sides of the equation, we get the triangular system $\mathbf{Rx} = \mathbf{Q}^\top \mathbf{b}$:

$$\begin{bmatrix} 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \\ 0 & 0 & 0 & \bullet \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{x} = \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}.$$

We discard the bottom inconsistent equations and solve the system

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ 0 & \bullet & \bullet & \bullet \\ 0 & 0 & \bullet & \bullet \\ 0 & 0 & 0 & \bullet \end{bmatrix} \mathbf{x} = \begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}.$$

In the next chapter, we'll use orthogonal matrices to help us find eigenvalues. One reason why orthogonal matrices are nice is that they don't change the lengths of vectors in the 2-norm. The 2-condition number of an orthogonal matrix is one. So, the errors don't blow up when we iterate. How do we find the orthogonal matrix \mathbf{Q} ? Typically using Gram–Schmidt orthogonalization, a series of Givens rotations, or a series of Householder reflections. Let's examine each of these methods of QR decomposition.

► Gram–Schmidt orthogonalization

The Gram–Schmidt process finds an orthonormal basis for a subspace by successively projecting out the nonorthogonal components. The orthogonal projection matrix for a general matrix \mathbf{A} is $\mathbf{P}_\mathbf{A} = \mathbf{A}(\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top$. For a vector \mathbf{v} , the projection matrix is $\mathbf{P}_\mathbf{v} = (\mathbf{v}\mathbf{v}^\top)/(\mathbf{v}^\top \mathbf{v})$. And, for a unit length vector \mathbf{q} , the projection matrix is simply $\mathbf{P}_\mathbf{q} = \mathbf{q}\mathbf{q}^\top$.

Suppose that we are given the vectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$. Let's find an orthonormal basis $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n$ for the subspace spanned by $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$.

1. Take $\mathbf{w}_1 = \mathbf{v}_1$ and normalize it to get $\mathbf{q}_1 = \mathbf{w}_1/\|\mathbf{w}_1\|_2$.
2. We want to find \mathbf{q}_2 such that $(\mathbf{q}_2, \mathbf{q}_1)_2 = 0$. We can do this by subtracting the \mathbf{q}_1 component of \mathbf{v}_2 from \mathbf{v}_2 . Taking $\mathbf{w}_2 = \mathbf{v}_2 - \mathbf{P}_{\mathbf{q}_1} \mathbf{v}_2$ and noting that $\mathbf{P}_{\mathbf{q}_j} \mathbf{v}_k = \mathbf{q}_j^\top \mathbf{q}_j \mathbf{v}_k = (\mathbf{q}_j, \mathbf{v}_k)_2 \mathbf{q}_j$ we have

$$\mathbf{w}_2 = \mathbf{v}_2 - (\mathbf{q}_1, \mathbf{v}_2)_2 \mathbf{q}_1.$$

This step finds the closest vector to \mathbf{v}_2 (in the 2-norm) that is orthogonal to \mathbf{w}_1 . Now, normalize \mathbf{w}_2 to get $\mathbf{q}_2 = \mathbf{w}_2/\|\mathbf{w}_2\|_2$.

3. To get \mathbf{q}_3 , subtract the $\text{span}\{\mathbf{q}_1, \mathbf{q}_2\}$ components of \mathbf{v}_3 from \mathbf{v}_3 and normalize:

$$\mathbf{w}_3 = \mathbf{v}_3 - \mathbf{P}_{\mathbf{q}_1} \mathbf{v}_3 - \mathbf{P}_{\mathbf{q}_2} \mathbf{v}_3 = \mathbf{v}_3 - (\mathbf{q}_1, \mathbf{v}_3)_2 \mathbf{q}_1 - (\mathbf{q}_2, \mathbf{v}_3)_2 \mathbf{q}_2.$$

This step finds the closest vector to \mathbf{v}_3 which is orthogonal to both \mathbf{w}_1 and \mathbf{w}_2 . Now, normalize \mathbf{w}_3 to get $\mathbf{q}_3 = \mathbf{w}_3/\|\mathbf{w}_3\|_2$.

4. In general, at the k th step we compute $\mathbf{q}_k = \mathbf{w}_k/\|\mathbf{w}_k\|_2$ with

$$\mathbf{w}_k = \mathbf{v}_k - \sum_{j=1}^{k-1} r_{jk} \mathbf{q}_j \quad \text{where} \quad r_{jk} = (\mathbf{q}_j, \mathbf{v}_k).$$

We continue to get the remaining $\mathbf{q}_4, \mathbf{q}_5, \dots, \mathbf{q}_n$.

The classical Gram–Schmidt method that subtracts the projection all at once can be numerically unstable because unless the vectors are already close to orthogonal, the orthogonal component is small. A better implementation is the modified Gram–Schmidt process that subtracts the projections successively:

```
for  $k = 1, \dots, n$ 
   $\mathbf{q}_k \leftarrow \mathbf{v}_k$ 
  for  $j = 1, \dots, k - 1$ 
     $r_{jk} = (\mathbf{q}_j, \mathbf{v}_k)$ 
     $\mathbf{q}_k \leftarrow \mathbf{q}_k - r_{jk} \mathbf{q}_j$ 
   $r_{kk} \leftarrow \|\mathbf{q}_k\|_2$ 
   $\mathbf{q}_k \leftarrow \mathbf{q}_k / r_{kk}$ 
```

► Givens rotations

A faster way to get the QR decomposition of \mathbf{A} is to use a series of rotation matrices applied to \mathbf{A} . Let's see how we can get an upper triangular matrix this way. A rotation in the plane is given by

$$\mathbf{Q} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}.$$

By choosing θ appropriately, we can rotate a vector to zero out either the first or the second components. For example, for a vector $\mathbf{x} = (x_1, x_2)$ we can get

$$\hat{\mathbf{x}} = \mathbf{Q}\mathbf{x} = \begin{bmatrix} \sqrt{x_1^2 + x_2^2} \\ 0 \end{bmatrix}$$

by taking

$$\cos \theta = \frac{x_1}{\sqrt{x_1^2 + x_2^2}} \quad \text{and} \quad \sin \theta = -\frac{x_2}{\sqrt{x_1^2 + x_2^2}}.$$

By applying a right rotation \mathbf{Q} to a 2×2 matrix \mathbf{A} , we can get

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} * & * \\ 0 & * \end{bmatrix}$$

where for ease of notation we take $c \equiv \cos \theta$ and $s \equiv \sin \theta$.

We can similarly rotate a vector in a plane in higher dimensions. Consider the rotation of the 5×5 matrix \mathbf{A} in the 2–5 plane

$$\mathbf{Q}_{52} = \begin{bmatrix} 1 & & & & \\ & c & & & -s \\ & & 1 & & \\ & & & 1 & \\ s & & & & c \end{bmatrix}$$

which zeros out element 5-2. $\mathbf{Q}_{52}\mathbf{A}$ equals

$$\begin{bmatrix} 1 & & & & \\ & c & & -s & \\ & & 1 & & \\ & & & 1 & \\ s & & & & c \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ * & * & * & * & * \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ * & 0 & * & * & * \end{bmatrix}$$

where rows two and five of the new matrix are simply linear combinations of rows two and five of the original matrix. By starting from the left and working to the right, we can create an upper triangular matrix.

For example, consider a 4×4 matrix

$$\mathbf{A} = \begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}.$$

By applying three Givens rotations, we can zero out the elements below the first pivot:

$$\mathbf{Q}_{41}\mathbf{Q}_{31}\mathbf{Q}_{21}\mathbf{A} = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix}.$$

By applying another two Givens rotations, we can zero out the elements below the second pivot:

$$\mathbf{Q}_{42}\mathbf{Q}_{32}\mathbf{Q}_{41}\mathbf{Q}_{31}\mathbf{Q}_{21}\mathbf{A} = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}.$$

And by applying one more Givens rotations, we can zero out the element below the third pivot:

$$\mathbf{Q}_{43}\mathbf{Q}_{42}\mathbf{Q}_{32}\mathbf{Q}_{41}\mathbf{Q}_{31}\mathbf{Q}_{21}\mathbf{A} = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} = \mathbf{R}.$$

So, formally

$$\mathbf{A} = \mathbf{Q}\mathbf{A} = (\mathbf{Q}_{43}\mathbf{Q}_{42}\mathbf{Q}_{32}\mathbf{Q}_{41}\mathbf{Q}_{31}\mathbf{Q}_{21})^{-1}\mathbf{R}.$$

To solve the problem $\mathbf{Ax} = \mathbf{b}$, we instead solve

$$\mathbf{Rx} = \mathbf{Q}_{43}\mathbf{Q}_{42}\mathbf{Q}_{32}\mathbf{Q}_{41}\mathbf{Q}_{31}\mathbf{Q}_{21}\mathbf{b}.$$

 The LinearAlgebra.jl function `(G,r)=givens(x,i,j)` computes the Givens rotation matrix G and scalar r such that $(G*x)[i]$ equals r and $(G*x)[j]$ equals 0 .

► Householder reflections

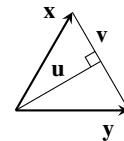
While Givens rotations zero out one element at a time, *Householder reflections* speed things up by changing whole columns all at once

$$\begin{matrix} \mathbf{A} \\ \left[\begin{array}{cccc} x & x & x & x \\ x & x & x & x \end{array} \right] \end{matrix} \rightarrow \begin{matrix} \mathbf{Q}_1\mathbf{A} \\ \left[\begin{array}{cccc} x & x & x & x \\ 0 & x & x & x \end{array} \right] \end{matrix} \rightarrow \begin{matrix} \mathbf{Q}_2\mathbf{Q}_1\mathbf{A} \\ \left[\begin{array}{cccc} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & x & x \\ 0 & 0 & x & x \end{array} \right] \end{matrix} \rightarrow \begin{matrix} \mathbf{Q}_3\mathbf{Q}_2\mathbf{Q}_1\mathbf{A} \\ \left[\begin{array}{cccc} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ 0 & 0 & x & x \\ 0 & 0 & 0 & x \end{array} \right] \end{matrix}.$$

We start with the first column \mathbf{x} of the $n \times n$ matrix. Let's find an orthogonal matrix \mathbf{Q}_1 such that $\mathbf{Q}_1\mathbf{x} = \mathbf{y}$, where $\mathbf{y} = \pm\|\mathbf{x}\|_2\xi$ with $\xi = (1, 0, \dots, 0)$ and the sign \pm taken from the first component of \mathbf{x} . We can determine \mathbf{Q}_1 with a little geometry.

Let $\mathbf{u} = (\mathbf{x} + \mathbf{y})/2$ and $\mathbf{v} = (\mathbf{x} - \mathbf{y})/2$. Then \mathbf{u} is orthogonal to \mathbf{v} and $\text{span}\{\mathbf{u}, \mathbf{v}\} = \text{span}\{\mathbf{x}, \mathbf{y}\}$, and we can get \mathbf{y} by subtracting twice \mathbf{v} from \mathbf{x} . That is to say, we can get \mathbf{y} by subtracting twice the projection of \mathbf{x} in the \mathbf{v} direction: $\mathbf{y} = (\mathbf{I} - 2\mathbf{P}_v)\mathbf{x}$. In general, for any vector in the direction of \mathbf{v} , we define the *Householder reflection matrix*:

$$\mathbf{H}_n = \mathbf{I} - 2\mathbf{P}_v = \mathbf{I} - 2\mathbf{v}\mathbf{v}^T/\|\mathbf{v}\|_2^2.$$



Because we want \mathbf{y} to be $\pm\|\mathbf{x}\|_2\xi$, we'll take $\mathbf{v} = \mathbf{x} \pm \|\mathbf{x}\|_2\xi$. (Technically, we ought to take $\mathbf{v} = (\mathbf{x} \pm \|\mathbf{x}\|_2\xi)/2$, but the one-half scaling factor is simply divided out when we compute \mathbf{P}_v .) Geometrically, the Householder matrix \mathbf{H}_n reflects a vector across the subspace $\text{span}\{\mathbf{v}\}^\perp$.

In the second step, we want to zero out the subdiagonal elements in the second column leaving the first column unaltered. To do this, we left multiply by the Householder matrix

$$\mathbf{Q}_2 = \begin{bmatrix} 1 & 0 \\ 0 & \mathbf{H}_{n-1} \end{bmatrix}$$

where \mathbf{H}_{n-1} is the $(n - 1) \times (n - 1)$ Householder matrix that maps an $(n - 1)$ -dimensional vector \mathbf{x} to the $(n - 1)$ -dimensional vector $\pm\|\mathbf{x}\|_2\xi$.

In the k th step, we use the Householder matrix

$$\mathbf{Q}_k = \begin{bmatrix} \mathbf{I}_k & 0 \\ 0 & \mathbf{H}_{n-k} \end{bmatrix}$$

where \mathbf{H}_{n-k+1} is the Householder matrix that maps an $(n - k + 1)$ -dimensional vector \mathbf{x} to an $(n - k + 1)$ -dimensional vector $\pm\|\mathbf{x}\|_2\xi$.

• The `LinearAlgebra.jl` function `qr` computes the QR factorization of a matrix using Householder reflection, returning an object that stores an orthogonal matrix and a sequence of Householder reflectors as an upper triangular matrix.

► Least-squares problem (again)

Let's reexamine the least squares problem in the context of QR decomposition without first deriving the normal equation. Suppose that $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$ with $m \geq n$. We want to find the $\mathbf{x} \in \mathbb{R}^n$ that minimizes

$$\|\mathbf{r}\|_2 = \|\mathbf{Ax} - \mathbf{b}\|_2.$$

Using QR decomposition, we can rewrite the 2-norm of the residual as

$$\|\mathbf{r}\|_2 = \|\mathbf{QRx} - \mathbf{b}\|_2$$

where \mathbf{R} is an upper triangular $m \times n$ matrix. Applying an orthogonal matrix to \mathbf{r} doesn't change its length, so

$$\|\mathbf{r}\|_2 = \|\mathbf{Q}^\top \mathbf{r}\|_2 = \|\mathbf{Rx} - \mathbf{Q}^\top \mathbf{b}\|_2.$$

This says that we can instead consider the equivalent upper triangular system $\mathbf{Rx} = \mathbf{c}$ where $\mathbf{c} = \mathbf{Q}^\top \mathbf{b}$. The residual \mathbf{s} of this system

$$\mathbf{s} = \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix} - \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{c}_1 - \mathbf{R}_1 \mathbf{x} \\ \mathbf{c}_2 \end{bmatrix}$$

where $\mathbf{c}_1 \in \mathbb{R}^n$, $\mathbf{c}_2 \in \mathbb{R}^{m-n}$, and $\mathbf{R}_1 \in \mathbb{R}^{n \times n}$. So,

$$\|\mathbf{s}\|_2^2 = \|\mathbf{c}_1 - \mathbf{R}_1 \mathbf{x}\|_2^2 + \|\mathbf{c}_2\|_2^2.$$

The term $\|\mathbf{c}_2\|_2^2$ is independent of \mathbf{x} —it doesn't change $\|\mathbf{s}\|_2^2$ by any amount regardless of the value of \mathbf{x} . So, the \mathbf{x} that minimizes $\|\mathbf{c}_1 - \mathbf{R}_1 \mathbf{x}\|_2$ also minimizes $\|\mathbf{s}\|_2$ and by equivalence $\|\mathbf{r}\|_2$. And, if \mathbf{R}_1 has full rank, then $\|\mathbf{c}_1 - \mathbf{R}_1 \mathbf{x}\|_2$ is minimized when precisely

$$\mathbf{R}_1 \mathbf{x} = \mathbf{c}_1.$$

On the other hand, what if \mathbf{R}_1 does not have full rank? In this case, $\mathbf{R}_1 \mathbf{x} = \mathbf{c}_1$ is

$$\begin{bmatrix} \mathbf{R}_{11} & \mathbf{R}_{12} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{bmatrix}$$

where \mathbf{R}_{11} is an upper triangular matrix. Then

$$\|\mathbf{s}\|_2^2 = \|\mathbf{c}_1 - \mathbf{R}_{11} \mathbf{x}_1 - \mathbf{R}_{12} \mathbf{x}_2\|_2^2 + \|\mathbf{c}_2\|_2^2,$$

and hence $\|\mathbf{s}\|_2^2$ is minimized for any vector \mathbf{x}_2 for which

$$\mathbf{R}_{11} \mathbf{x}_1 = \mathbf{c}_1 - \mathbf{R}_{12} \mathbf{x}_2.$$

In this case, we don't have a unique solution. Instead, we'll need to choose which of the solutions will be the “best” solution. We develop a choosing methodology in the next section.

 The \ method chooses different algorithms based on the structure of matrices. Solutions to overdetermined systems are computed using pivoted QR factorization.

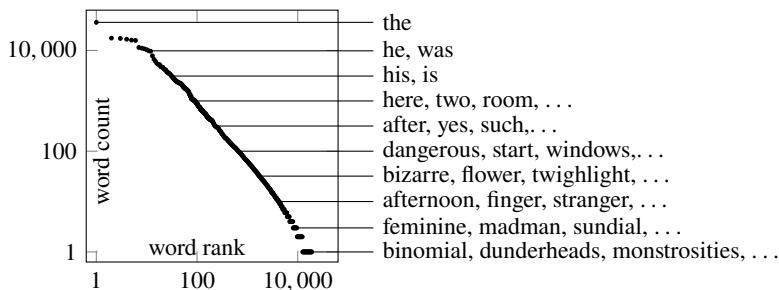


Figure 3.1: Frequency versus rank of words in Sherlock Holmes.

Example. George Zipf was a linguist who noticed that a few words, like *the*, *is*, *of*, and *and*, are used quite frequently while most words, like *anhydride*, *embryogenesis*, and *jackscrew*, are rarely used at all. By examining word frequency rankings across several corpora, Zipf made a statistical observation, now called Zipf's law. The empirical law states that in any natural language corpus the frequency of any word is inversely proportional to its rank in a frequency table. That is, the most common word is twice as likely to appear as the second most common word, three times as likely to appear as the third most common word, and n times as likely as the n th most common word. For example, 18,951 of the 662,817 words that appear in the canon of Sherlock Holmes by Sir Arthur Conan Doyle are unique. The word *the* appears 36,125 times. *Holmes* appears 3051 times, *Watson* 1038 times, and *Moriarty*² appears 54 times. And there are 6610 words like *binomial*, *dunderheads*, *monstrosities*, *sunburnt*, and *vendetta* that appear only once. See Figure 3.1 above

Zipf's power law states that the frequency of the k th most common word is approximately $n_k = n_1 k^{-s}$ where s is some parameter and n_1 is the frequency of the most common word. Let's find the power s for the words in Sherlock Holmes using ordinary least squares and total least squares. First, we'll write the power law in a more obvious form: $\log n_k = -s \log k + \log n_1$. The Julia code is

```
using DelimitedFiles
T = readdlm(download(bucket*"sherlock.csv"), '\t')[:,2]
n = length(T)
A = [ones(n,1) log.(1:n)]
B = log.(T)
c = A\B
```

²Professor James Moriarty—criminal mastermind and archenemy of Sherlock Holmes—was a brilliant mathematician who at age of twenty-one wrote a treatise on the binomial theorem winning him the mathematical chair at a small university. His book on asteroid dynamics is described as one “which ascends to such rarefied heights of pure mathematics that it is said that there was no man in the scientific press capable of criticizing it.”

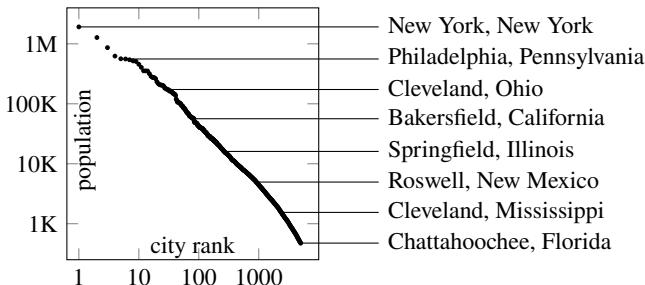


Figure 3.2: Population versus rank of cities in the United States.

We find that s is -1.165 using least squares. Zipf's law applies to more than just words. We can use it to model the size of cities, the magnitude of earthquakes, and even the distances between galaxies. For example, Figure 3.2 shows the log-log plot of the population of U.S. cities against their associated ranks.³ We find that s is -0.770 . ▶

3.2 Underdetermined systems

Up until now we've assumed that if $\mathbf{Ax} = \mathbf{b}$ has a least squares solution, then it has a unique least squares solution. This assumption is not always true. If the null space of \mathbf{A} has more than the zero vector, then $\mathbf{A}^T\mathbf{A}$ is not invertible. And the submatrix \mathbf{R}_1 from the previous section does not have full rank. In this case, the problem has infinitely many solutions. So then, how do we choose which of the infinitely many solutions is the “best” solution? One way is to take the solution \mathbf{x} with the smallest norm $\|\mathbf{x}\|$. This approach is known as *Tikhonov regularization* or *ridge regression*. It is especially good in problems where \mathbf{x} has a zero mean, which happens frequently in statistics by standardizing variables. Let's change our original problem

$$\text{Find the } \mathbf{x} \in \mathbb{R}^n \text{ that minimizes } \Phi(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|_2^2$$

to the new problem

$$\text{Find the } \mathbf{x} \in \mathbb{R}^n \text{ that minimizes } \Phi(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \alpha^2 \|\mathbf{x}\|_2^2$$

where α is some regularizing parameter. The result is a solution that fits \mathbf{x} to \mathbf{b} but penalizes solutions with large norms. Minimizing $\|\mathbf{Ax} - \mathbf{b}\|_2^2 + \alpha^2 \|\mathbf{x}\|_2^2$ is

³See <https://github.com/nmfsc/data/UScities.csv>

equivalent to solving the stacked-matrix equation

$$\begin{bmatrix} \mathbf{A} \\ \alpha \mathbf{I} \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix}, \quad (3.1)$$

which we can do using least squares methods such as QR factorization.

Let's also examine the explicit normal equation solution to the regularized minimization problem. As before, solve the minimization problem by finding the \mathbf{x} such that $\nabla\Phi(\mathbf{x}) = \mathbf{0}$. We have $2\mathbf{A}^T\mathbf{A}\mathbf{x} - 2\mathbf{A}^T\mathbf{b} + 2\alpha^2\mathbf{x} = \mathbf{0}$ which we can solve to get

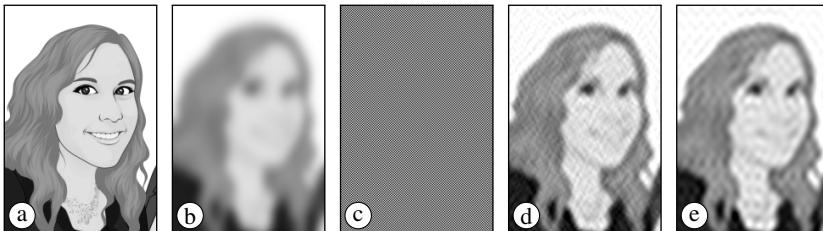
$$\mathbf{x} = (\mathbf{A}^T\mathbf{A} + \alpha^2\mathbf{I})^{-1}\mathbf{A}^T\mathbf{b} = \mathbf{R}_\alpha\mathbf{b}. \quad (3.2)$$

When \mathbf{A} has many more columns than rows, computing the QR factorization of the stacked-matrix equation (3.1) or the regularized normal equation (3.2) can be inefficient because we are working with even larger matrices. However, by noting that $\mathbf{A}^T(\mathbf{A}\mathbf{A}^T + \alpha^2\mathbf{I}) = (\mathbf{A}^T\mathbf{A} + \alpha^2\mathbf{I})\mathbf{A}^T$, we have

$$(\mathbf{A}^T\mathbf{A} + \alpha^2\mathbf{I})^{-1}\mathbf{A}^T = \mathbf{A}^T(\mathbf{A}\mathbf{A}^T + \alpha^2\mathbf{I})^{-1}.$$

So, we can instead compute $\tilde{\mathbf{x}} = \mathbf{A}^T(\mathbf{A}\mathbf{A}^T + \alpha^2\mathbf{I})^{-1}\mathbf{b}$.

Example. Photographic images often contain motion or focal blur and noise that one might want to remove or minimize. Suppose that \mathbf{x} is some original image and $\mathbf{b} = \mathbf{Ax} + \mathbf{n}$ is the observed image where \mathbf{A} is a blurring matrix and \mathbf{n} is a noise vector. The blurring matrix \mathbf{A} does not have a unique inverse, and therefore the problem is ill-posed. Compare the following images:



The first frame (a) shows the original image \mathbf{x} . Blur and noise have been added to this image to get the second image $\mathbf{b} = \mathbf{Ax} + \mathbf{n}$. Frame (c) shows the solution using the least squares method $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$. Because the blurring matrix \mathbf{A} has a nonzero null space, there are multiple (infinitely many) possible solutions for the least squares algorithm. The method clearly fails. Image (d) shows the regularized least squares solution $(\mathbf{A}^T\mathbf{A} + \alpha^2\mathbf{I})^{-1}\mathbf{b}$ for a well-chosen α . I think that this one is the winner. The solution in image (e) is computed using the pseudoinverse $\mathbf{A}^+\mathbf{b}$. This one is also good, although perhaps not as good as the regularized least squares solution. ▲

► Multiobjective least squares

A regularized solution to an underdetermined system is a special case of a simultaneous solution to two or more systems of equations— $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{Cx} = \mathbf{d}$. We can solve such a problem by stacking the two systems along with a positive weight α , so that we have the residual

$$\mathbf{r} = \begin{bmatrix} \mathbf{r}_1 \\ \alpha \mathbf{r}_2 \end{bmatrix} = \begin{bmatrix} (\mathbf{Ax} - \mathbf{b}) \\ \alpha(\mathbf{Cx} - \mathbf{d}) \end{bmatrix}.$$

The weight α^2 determines the relative importance of each objective $\|\mathbf{r}_1\|_2^2$ and $\|\mathbf{r}_2\|_2^2$. We then minimize $\|\mathbf{r}\|_2^2 = \|\mathbf{r}_1\|_2^2 + \alpha^2 \|\mathbf{r}_2\|_2^2$ by solving

$$\begin{bmatrix} \mathbf{A} \\ \alpha \mathbf{C} \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{b} \\ \alpha \mathbf{d} \end{bmatrix}$$

using QR decomposition or the normal equations. In the case of $\mathbf{C} = \mathbf{I}$ and $\mathbf{d} = \mathbf{0}$, we simply have the Tikhonov regularized least squares. And, of course, we can extend the problem to any number of objective values.

► Constrained least squares

A constrained least squares problem consists of two systems of equations—one $\mathbf{Ax} = \mathbf{b}$ that we are trying to fit as closely as possible and one $\mathbf{Cx} = \mathbf{d}$ that must absolutely fit. For example, $\mathbf{Ax} = \mathbf{b}$ could be a Vandermonde system for points of a piecewise polynomial curve and $\mathbf{Cx} = \mathbf{d}$ could be the matching conditions at the knots. The number of constraints $\mathbf{Cx} = \mathbf{d}$ must be fewer than the number of variables.

We can solve this problem by using the Lagrangian function

$$\mathcal{L}(\mathbf{x}, \lambda) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \lambda^\top (\mathbf{Cx} - \mathbf{d})$$

where the λ is a vector of Lagrange multipliers. The pair $(\tilde{\mathbf{x}}, \lambda)$ is a solution to the constrained minimization problem when $\nabla \mathcal{L}(\mathbf{x}, \lambda) = \mathbf{0}$. This gives us the systems of equation $\mathbf{A}^\top \mathbf{A} \tilde{\mathbf{x}} - \mathbf{A}^\top \mathbf{b} + \mathbf{C}^\top \lambda = \mathbf{0}$. We can rewrite this system along with the system of constraints $\mathbf{C} \tilde{\mathbf{x}} = \mathbf{d}$ as the block matrix

$$\begin{bmatrix} \mathbf{A}^\top \mathbf{A} & \mathbf{C}^\top \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}} \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{A}^\top \mathbf{b} \\ \mathbf{d} \end{bmatrix}.$$

Using Julia we have

```
function constrained_lstsq(A,b,C,d)
    x = [A'*A C'; C zeros(size(C,1),size(C,1))]\[A'*b;d]
    x[1:size(A,2)]
end
```

where A is $m \times n$, b is $m \times 1$, c is $p \times n$, and d is $p \times 1$ with $p \leq m$.

The method of Lagrange multipliers is generalized by the Karush–Kuhn–Tucker (KKT) conditions. Suppose we want to optimize an objective function $f(\mathbf{x})$ subject to constraints $\mathbf{g}(\mathbf{x}) \geq 0$ and $\mathbf{h}(\mathbf{x}) = 0$, where \mathbf{x} is the optimization variables in a convex subset of \mathbb{R}^n . For the corresponding Lagrangian function $\mathcal{L}(\mathbf{x}, \mu, \lambda) = f(\mathbf{x}) + \mu^\top \mathbf{g}(\mathbf{x}) + \lambda^\top \mathbf{h}(\mathbf{x})$, the value \mathbf{x}^* is a local minimum if the following KKT conditions hold:

- $\nabla \mathcal{L}(\mathbf{x}^*, \mu, \lambda) = \nabla f(\mathbf{x}^*) - \mu^\top \nabla \mathbf{g}(\mathbf{x}^*) - \lambda^\top \mathbf{h}(\mathbf{x}^*) = 0$ (stationarity)
- $\mathbf{g}(\mathbf{x}^*) \geq 0$ and $\nabla \mathbf{h}(\mathbf{x}^*) = 0$ (primal feasibility)
- $\mu \geq 0$ (dual feasibility)
- $\mu \circ \mathbf{g}(\mathbf{x}^*) = 0$ (complementary slackness)

where \circ is component-wise (Hadamard) multiplication.

► Sparse least squares

Applying QR decomposition or computing the Gram matrix of the normal equations on a large, sparse matrix will fill-in a sparse matrix. Instead, the normal equation $\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b}$ can be rewritten in terms of the residual as a expanded block system. To see this, recall that the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ is in the left null space of \mathbf{A} , so $\mathbf{A}^\top \mathbf{r} = \mathbf{0}$. From these two equations, we have

$$\begin{bmatrix} \mathbf{0} & \mathbf{A}^\top \\ \mathbf{A} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{r} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{b} \end{bmatrix}.$$

This new matrix maintains the sparsity and can be solved using an iterative method such as the symmetric conjugate-gradient method discussed in Chapter 5. Similarly, for a sparse constrained least squares problem:

$$\begin{bmatrix} \mathbf{0} & \mathbf{A}^\top & \mathbf{C}^\top \\ \mathbf{A} & \mathbf{I} & \mathbf{0} \\ \mathbf{C} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{r} \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{b} \\ \mathbf{d} \end{bmatrix}.$$

3.3 Singular value decomposition

Another way to solve the least squares problem is by singular value decomposition. Recall that for the inconsistent and underdetermined problem

$$\mathbf{A}(\mathbf{x}_{\text{row}} + \mathbf{x}_{\text{null}}) = \mathbf{b}_{\text{column}} + \mathbf{b}_{\text{left null}},$$

we minimize $\|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2$ by forcing $\mathbf{b}_{\text{left null}} = \mathbf{0}$ and ensure a unique solution $\|\mathbf{x}\|_2$ by forcing $\mathbf{x}_{\text{null}} = \mathbf{0}$. In the first case, we are zeroing out the left null space; in the second case, we are zeroing out the null space. We can use singular value decomposition to do precisely this. A matrix's singular value decomposition is

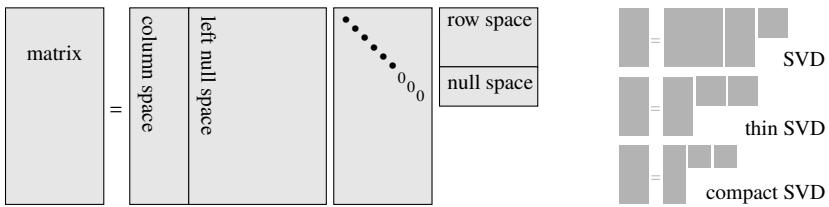


Figure 3.3: The SVD breaks a matrix into the orthonormal bases of its four fundamental subspaces—its column space and left null space along with its row space and null space—coupled through a diagonal matrix of singular values.

$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ where \mathbf{U} and \mathbf{V} are orthogonal matrices and Σ is a diagonal matrix of singular values. By convention the singular values σ_i are always in descending order $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0$. This ordering is also a natural result of the SVD algorithm, which we study in the next chapter after we develop the tools for finding eigenvalues. In this section, we look at applications of singular value decomposition.

The SVD of an $m \times n$ matrix \mathbf{A} produces an $m \times m$ matrix \mathbf{U} , an $m \times n$ diagonal matrix Σ , and an $n \times n$ matrix \mathbf{V}^T . The columns of \mathbf{U} are called left singular vectors and the columns of \mathbf{V} are called the right singular vectors. When \mathbf{A} is a rectangular matrix with $m > n$, only the first n left singular vectors are needed to reconstruct \mathbf{A} . In this case, it's more efficient to use the *thin* or *economy* SVD—an $m \times n$ matrix \mathbf{U} , an $n \times n$ diagonal matrix Σ , and an $n \times n$ matrix \mathbf{V}^T . See Figure 3.3 above.

If \mathbf{A} has rank r , then the first r columns of \mathbf{U} span the column space of \mathbf{A} , the last $m - r$ columns of \mathbf{U} span the left null space of \mathbf{A} , the first r columns of \mathbf{V} span the row space of \mathbf{A} , and the last $n - r$ columns of \mathbf{V} span the null space of \mathbf{A} . This means that \mathbf{A} can be reconstructed using an $m \times r$ matrix \mathbf{U} , an $r \times r$ diagonal matrix Σ , and an $r \times n$ matrix \mathbf{V}^T . This type of reduced SVD is called *compact SVD*.

Often for large matrices, the small singular values do not contribute much to \mathbf{A} . By keeping the k largest singular values and the corresponding singular vectors, we get the closest rank k approximation to \mathbf{A} in the Frobenius norm. The *truncated SVD* of \mathbf{A} is given by an $m \times k$ matrix \mathbf{U}_k , an $k \times k$ diagonal matrix Σ_k , and an $k \times n$ matrix \mathbf{V}_k^T . A truncated SVD is useful for a number of different reasons, such as reducing storage and memory requirements of \mathbf{A} , making computations faster, simplifying a problem by projecting the data to a lower-dimensional subspace, and regularizing a problem by reducing the condition number to σ_1/σ_k .

• The `LinearAlgebra.jl` function `svd(A)` returns the SVD of matrix A as an object. Singular values are in descending order. When the option `full=false` (default), the function returns an economy SVD. The function `svds(A, k)` returns an object containing the first k (default is 6) singular values and associated singular vectors.

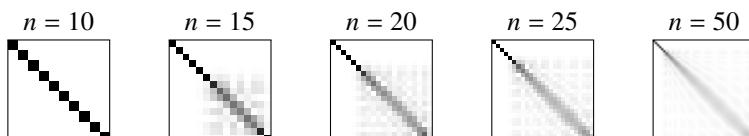
► Pseudoinverse

Suppose that \mathbf{A} is an arbitrary $m \times n$ matrix with the singular value decomposition $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top$. If \mathbf{A} were nonsingular, then $\mathbf{A}^{-1}\mathbf{b}$ would be $\mathbf{V}\Sigma^{-1}\mathbf{U}^\top$, where Σ^{-1} is simply the diagonal matrix of the reciprocals of the singular values. The *Moore–Penrose pseudoinverse* \mathbf{A}^+ of \mathbf{A} is an $n \times m$ matrix defined as $\mathbf{A}^+ = \mathbf{V}\Sigma^+\mathbf{U}^\top$ where Σ^+ is a diagonal matrix of the reciprocals of the nonzero elements of Σ , leaving the zero elements unchanged. This has the effect of zeroing out both the null space and the left null space of \mathbf{A} . Recall that the singular vectors corresponding to the zero singular values are in the null space and left null space of \mathbf{A} . By zeroing these out, not only does the pseudoinverse ensure a solution, it also provides a unique solution. Namely, for an $m \times n$ matrix \mathbf{A} with rank r :

dimensions	$\mathbf{A}^+\mathbf{b}$
$m = n = r$	$\mathbf{A}^{-1}\mathbf{b}$
$m > n = r$	$\arg \min_{\mathbf{x}} \ \mathbf{b} - \mathbf{Ax}\ _2$ and $\mathbf{A}^+ = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top$
$n > m = r$	rowspace projection of \mathbf{b} and $\mathbf{A}^+ = \mathbf{A}(\mathbf{A}\mathbf{A}^\top)^{-1}$
$n \geq m > r$	$\arg \min_{\mathbf{x}} \ \mathbf{b} - \mathbf{Ax}\ _2$ using the rowspace projection of \mathbf{b}

We also can use the pseudoinverse to numerically regularize ill-conditioned problems by removing the small singular values by using a truncated SVD to zero out elements below some tolerance α . By doing so, we lower the condition number to σ_1/α and solve a problem in a lower-dimensional subspace.

Example. Consider the n th-order Hilbert matrix, whose ij -element equals $(i + j - 1)^{-1}$, discussed in the example on page 23. Hilbert matrices are poorly conditioned even for relatively small n . The density plots of the matrices `pinv(hilbert(n))*hilbert(n)` for $n = 10, 15, 20, 25$, and 50 are below. Zero values are white, absolute values of one or greater are black, and intermediate values are shades of gray.



While the pseudoinverse solution has noticeable error when $n > 10$, failure is much more graceful using the pseudoinverse than using Gaussian elimination. The pseudoinverse truncates all singular values less than 10^{-15} which happens near σ_8 , effectively reducing the problem to a using a rank-8 matrix. ▶

• The `LinearAlgebra.jl` function `pinv` computes the Moore–Penrose pseudoinverse by computing the SVD using a default tolerance of `minimum(size(A))*norm(A)*eps()`.

► Tikhonov regularization

Tikhonov regularization introduced the previous section and the pseudoinverse discussed above each provide ways to solve underdetermined problems. The two procedures are actually related. Recall Tikhonov regularization $\hat{\mathbf{x}} = \mathbf{R}_\alpha \mathbf{b}$ where $\mathbf{R}_\alpha = (\mathbf{A}^\top \mathbf{A} + \alpha^2 \mathbf{I})^{-1} \mathbf{A}^\top$. If $\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^\top$, then

$$\begin{aligned}\mathbf{R}_\alpha &= (\mathbf{V} \Sigma \mathbf{U}^\top \mathbf{U} \Sigma \mathbf{V}^\top + \alpha^2 \mathbf{V} \Sigma \mathbf{I} \mathbf{V}^\top)^{-1} \mathbf{V} \Sigma \mathbf{U}^\top \\ &= (\mathbf{V} \Sigma^2 \mathbf{V}^\top + \alpha^2 \mathbf{V} \Sigma \mathbf{I} \mathbf{V}^\top)^{-1} \mathbf{V} \Sigma \mathbf{U}^\top \\ &= \mathbf{V} (\Sigma^2 + \alpha^2 \mathbf{I})^{-1} \Sigma \mathbf{U}^\top \\ &= \mathbf{V} \Sigma_\alpha^{-1} \mathbf{U}^\top\end{aligned}$$

where Σ_α^{-1} is a diagonal matrix with diagonal components

$$\frac{\sigma_i}{\sigma_i^2 + \alpha^2}.$$

It is useful to rewrite these components as

$$\frac{\sigma_i^2}{\sigma_i^2 + \alpha^2} \frac{1}{\sigma_i} = w_\alpha(\sigma_i) \frac{1}{\sigma_i}$$

where $w_\alpha(\sigma_i)$ is the *Wiener weight*. From this we see that Tikhonov regularization is simply a smooth filter applied to the singular values. In comparison, the truncated SVD filter

$$w_\alpha^{\text{TSVD}}(\sigma) = \begin{cases} 0, & \sigma \leq \alpha \\ 1, & \sigma > \alpha \end{cases}$$

exhibits a sharp cut-off at α . See Figure 3.4 on the following page. Because of this, Tikhonov regularization may outperform the pseudoinverse.

► Principal component analysis

Principal component analysis or PCA is a statistical method that uses singular value decomposition to generate a low-rank approximation of a matrix. An $m \times n$

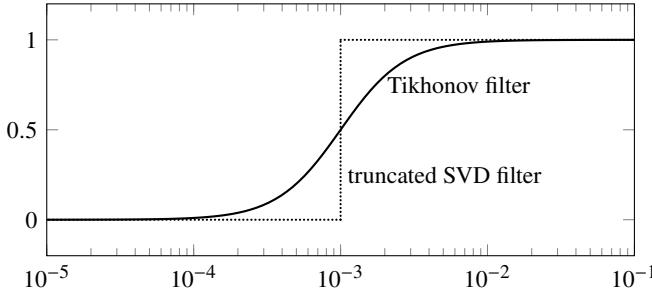


Figure 3.4: Comparison of the Tikhonov filter for $\alpha = 10^{-3}$ and the truncated SVD filter with singular values zeroed out below 10^{-3} .

matrix \mathbf{A} can be written as the sum of rank-one, mutually orthogonal *principal components* $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top = \mathbf{E}_1 + \mathbf{E}_2 + \dots + \mathbf{E}_p$ where $\mathbf{E}_i = \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$ and p is the rank of \mathbf{A} . We can get a lower rank- k approximation \mathbf{A}_k to \mathbf{A} by truncating the sum $\mathbf{A}_k = \mathbf{E}_1 + \mathbf{E}_2 + \dots + \mathbf{E}_k$. In fact, the matrix \mathbf{A}_k is the best rank- k approximation to \mathbf{A} .

Theorem 9 (Eckart–Young–Mirsky theorem). *Let $\mathbf{A}_k = \sum_{i=1}^k \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$ be a rank- k approximation to $\mathbf{A} = \sum_{i=1}^p \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$. If \mathbf{B} is any rank- k matrix, then $\|\mathbf{A} - \mathbf{A}_k\| \leq \|\mathbf{A} - \mathbf{B}\|$ in both the spectral and Frobenius norms.*

Proof. Let's start with the spectral norm. A rank- k matrix \mathbf{B} can be expressed as the product \mathbf{XY}^\top where \mathbf{X} and \mathbf{Y} have k columns. Then because \mathbf{Y} has only rank k , there is a unit vector $\mathbf{w} = \gamma_1 \mathbf{v}_1 + \dots + \gamma_{k+1} \mathbf{v}_{k+1}$ such that $\mathbf{Y}^\top \mathbf{w} = 0$. Furthermore, $\mathbf{Aw} = \sum_{i=1}^{k+1} \gamma_i \sigma_i \mathbf{u}_i$ from which $\|\mathbf{Aw}\|_2^2 = \sum_{i=1}^{k+1} |\gamma_i \sigma_i|^2$. Then

$$\|\mathbf{A} - \mathbf{B}\|_2^2 \geq \|(\mathbf{A} - \mathbf{B})\mathbf{w}\|_2^2 = \|\mathbf{Aw}\|_2^2 \geq \sigma_{k+1}^2 = \|\mathbf{A} - \mathbf{A}_k\|_2^2.$$

Now, consider the Frobenius norm. Let a subscript i denote an approximation using the first i principal components.

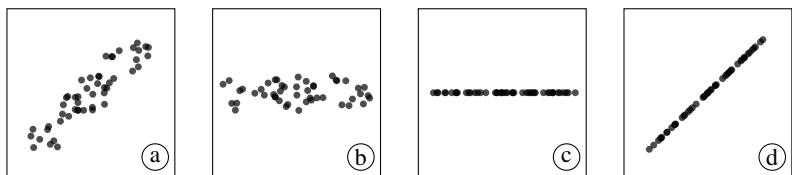
$$\sigma_i(\mathbf{A} - \mathbf{B}) = \sigma_1((\mathbf{A} - \mathbf{B}) - (\mathbf{A} - \mathbf{B})_{i-1}) \geq \sigma_1(\mathbf{A} - \mathbf{A}_{k+i-1}) = \sigma_{k+i}(\mathbf{A}),$$

where the inequality follows from the results about the spectral norm above because $\text{rank}(\mathbf{B} + (\mathbf{A} - \mathbf{B})_{i-1}) \leq \text{rank } \mathbf{A}_{k+i-1}$. So,

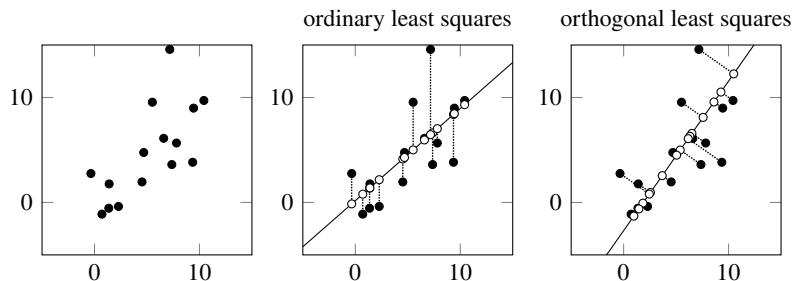
$$\|\mathbf{A} - \mathbf{B}\|_F^2 = \sum_{i=1}^p \sigma_i(\mathbf{A} - \mathbf{B})^2 \geq \sum_{i=k+1}^p \sigma_i(\mathbf{A})^2 = \|\mathbf{A} - \mathbf{A}_k\|_F^2. \quad \square$$

Suppose that we measure the heights \mathbf{h} and weights \mathbf{w} of 50 people. Let \mathbf{A} be the 50×2 standardized data matrix, whose first column is the mean-centered

height $\mathbf{h} - \bar{\mathbf{h}}$ and whose second column is the mean-centered weight $\mathbf{w} - \bar{\mathbf{w}}$. The matrix $\mathbf{A}^\top \mathbf{A}$ is the covariance matrix. If the height is linearly correlated to weight, there is a variable that incorporates both a person's height and weight in the direction of the first right singular value \mathbf{v}_1 with measure σ_1 . There is a second variable in the \mathbf{v}_2 direction with measure σ_2 that accounts for the variation in the data. A rank one approximation $\mathbf{A}_1 = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^\top$ eliminates the \mathbf{v}_2 . Consider the following figures: (a) The data is correlated and lies for the most part along a direction \mathbf{u}_1 . (b) The matrix \mathbf{AV} acts as a rotation of the heights and weights onto the x- and y-axes. (c) We keep only the first principal component—now along the x-axes—and discard the y offsets. (d) After rotating back to our original basis, all the data lies along the \mathbf{v}_1 direction.



Note that unlike least squares fit, in which the data is projected “vertically” onto the height-weight line, principal component analysis projects the data orthogonally onto the height-weight line. Take the scatter plots below. Original data \bullet is shown in the scatter plot on the left. The least squares approach shown in the center figure is an orthogonal projection of the y-components into the column space of the 15×2 Vandermonde matrix. Principal component analysis is an orthogonal projection into the first left singular vector. Note the difference in the solutions using ordinary least squares fit and orthogonal least squares fit. Neither the points \circ nor the line are the same in either figure.



► Total least squares

Let's dig a bit deeper into orthogonal least squares, also known as total least squares. We solved the ordinary least squares problem $\mathbf{Ax} = \mathbf{b}$ by determining the solution to a similar problem: choose the solution to $\mathbf{Ax} = \mathbf{b} + \delta\mathbf{b}$ that has the

smallest residual $\|\delta\mathbf{b}\|_2$. We can generalize the problem of finding the solution to $\mathbf{AX} = \mathbf{B}$ by minimizing not only with respect to the dependent terms \mathbf{B} but also with respect to the independent terms \mathbf{A} . That is, find the smallest $\delta\mathbf{A}$ and $\delta\mathbf{B}$ that satisfies $(\mathbf{A} + \delta\mathbf{A})\mathbf{X} = \mathbf{B} + \delta\mathbf{B}$. Such a problem is called the total least squares problem. We'll take \mathbf{A} to be an $m \times n$ matrix, \mathbf{X} to be an $n \times p$ matrix, and \mathbf{B} to be an $m \times p$ matrix. In two-dimensions, total least squares regression is referred to as Deming regression or orthogonal regression.

The total least squares problem $(\mathbf{A} + \delta\mathbf{A})\mathbf{X} = \mathbf{B} + \delta\mathbf{B}$ can be written as the block matrix

$$[\mathbf{A} + \delta\mathbf{A} \quad \mathbf{B} + \delta\mathbf{B}] \begin{bmatrix} \mathbf{X} \\ -\mathbf{I} \end{bmatrix} = \mathbf{0}. \quad (3.3)$$

We want to find

$$\arg \min_{\delta\mathbf{A}, \delta\mathbf{B}} \|[\delta\mathbf{A} \quad \delta\mathbf{B}]\|_{\text{F}}.$$

Take the singular value decomposition $\mathbf{U}\Sigma\mathbf{V}^T = [\mathbf{A} \quad \mathbf{B}]$:

$$[\mathbf{A} \quad \mathbf{B}] = [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \Sigma_1 & \mathbf{0} \\ \mathbf{0} & \Sigma_2 \end{bmatrix} \begin{bmatrix} \mathbf{V}_{11} & \mathbf{V}_{12} \\ \mathbf{V}_{21} & \mathbf{V}_{22} \end{bmatrix}^T$$

where \mathbf{U}_1 , Σ_1 , \mathbf{V}_{11} , and \mathbf{V}_{12} have n columns like \mathbf{A} and \mathbf{U}_2 , Σ_2 , \mathbf{V}_{21} , and \mathbf{V}_{22} have p columns like \mathbf{B} . By the Eckart–Young–Mirsky theorem, the toerems $\delta\mathbf{A}$ and $\delta\mathbf{B}$ are those for which

$$[\mathbf{A} + \delta\mathbf{A} \quad \mathbf{B} + \delta\mathbf{B}] = [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \Sigma_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{V}_{11} & \mathbf{V}_{12} \\ \mathbf{V}_{21} & \mathbf{V}_{22} \end{bmatrix}^T.$$

By linearity

$$\begin{aligned} [\delta\mathbf{A} \quad \delta\mathbf{B}] &= -[\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \Sigma_2 \end{bmatrix} \begin{bmatrix} \mathbf{V}_{11} & \mathbf{V}_{12} \\ \mathbf{V}_{21} & \mathbf{V}_{22} \end{bmatrix}^T \\ &= -\mathbf{U}_2 \Sigma_2 \begin{bmatrix} \mathbf{V}_{12} \\ \mathbf{V}_{22} \end{bmatrix}^T \\ &= -[\mathbf{A} \quad \mathbf{B}] \begin{bmatrix} \mathbf{V}_{12} \\ \mathbf{V}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{V}_{12} \\ \mathbf{V}_{22} \end{bmatrix}^T \end{aligned}$$

Equivalently,

$$[\delta\mathbf{A} \quad \delta\mathbf{B}] \begin{bmatrix} \mathbf{V}_{12} \\ \mathbf{V}_{22} \end{bmatrix} = -[\mathbf{A} \quad \mathbf{B}] \begin{bmatrix} \mathbf{V}_{12} \\ \mathbf{V}_{22} \end{bmatrix}$$

or

$$[\mathbf{A} + \delta\mathbf{A} \quad \mathbf{B} + \delta\mathbf{B}] \begin{bmatrix} \mathbf{V}_{12} \\ \mathbf{V}_{22} \end{bmatrix} = \mathbf{0}.$$

If \mathbf{V}_{22} is nonsingular, then

$$[\mathbf{A} + \delta\mathbf{A} \quad \mathbf{B} + \delta\mathbf{B}] \begin{bmatrix} \mathbf{V}_{12}\mathbf{V}_{22}^{-1} \\ \mathbf{I} \end{bmatrix} = \mathbf{0}.$$

So from (3.3), we have

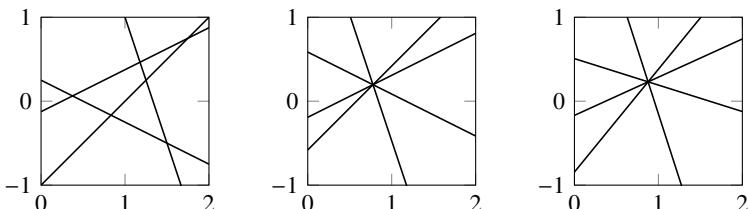
$$\mathbf{X} = -\mathbf{V}_{12}\mathbf{V}_{22}^{-1}.$$

We can implement the total least squares solution in Julia with

```
function tls(A,B)
    n = size(A,2)
    V = svd([A B]).V
    -V[1:n,n+1:end]/V[n+1:end,n+1:end]
end
```

Example. Let's find the intersections of four lines $2x + 4y = 1$, $x - y = 1$, $3x + y = 4$, and $4x - 8y = 1$. While there is no solution, we can find a best solution. This problem has the matrix form $\mathbf{Ax} = \mathbf{b}$, where the elements of \mathbf{A} are the coefficients of x and y , \mathbf{x} is the coordinates (x, y) , and the vector \mathbf{b} is the set of constants.

```
A = [2 4; 1 -1; 3 1; 4 -8]; b = [1; 1; 4; 1];
xols = A\b; xt1s = tls(A,b)
```



The figure on the left above shows the original lines. The middle one shows the ordinary least squares $\mathbf{Ax} = \mathbf{b} + \delta\mathbf{b}$ solution of around $(0.777, 0.196)$. The solution slides the lines around the plane to find the intersection but does not change their slopes. The one on the right shows the total least squares solution $(\mathbf{A} + \delta\mathbf{A})\mathbf{x} = \mathbf{b} + \delta\mathbf{b}$ solution of around $(0.877, 0.230)$. Now, lines can slide around the plane and change their slopes. ◀

► Rank reduction and image compression

Singular value decomposition is sometimes used to demonstrate image compression. In practice, other common approaches are much faster and more effective. For example, JPEG compression uses discrete cosine transforms and JPEG 2000 uses discrete wavelet transforms to achieve a significant reduction in file size without a significant reduction in image quality, particularly on images with smooth gradients like photographs. Lossless methods such as PNG and GIF reduce file size without any loss in quality by finding patterns in the data.⁴ On the other hand, singular value decomposition is relatively slow and even moderate compression can result in noticeable artifacts. Furthermore, singular value decomposition produces matrices containing both positive and negative floating-point values that must be then quantized. Storing these matrices may, in fact, require more space than the original raw image. Nonetheless, examining SVD image compression is worthwhile because it can help us better understand how an SVD works on other structured data that we might want to reduce in rank using singular value decomposition.

A typical image type such as a JPEG or PNG will use one byte to quantize each pixel for each color. One byte (8 bits) is sufficient for 256 shades of gray, and three bytes (24 bits) are enough for three red-green-blue (RGB) or hue-saturation-value (HSV) channels. Three bytes produces 2^{24} (or roughly 16 million) color variations.⁵ We can think of a raw m by n grayscale bitmap image as an $m \times n$ matrix and then compute an economy SVD of that matrix, storing the resulting right and left singular matrices along with the corresponding singular values. A color image consists of three channels, and the corresponding matrix is $m \times 3n$. Let's consider the SVD of a grayscale image $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top$ with $m > n$ (so that the rank is at most n):

```
using Images
A = Gray.(load(download(bucket*"laura.png")))
U, σ, V = svd(A);
```

Then a rank- k approximation to \mathbf{A} is $\mathbf{A}_k = \mathbf{U}_k \Sigma_k \mathbf{V}_k^\top$.

```
A_k = Gray.( U[:,1:k] * Diagonal(σ[1:k]) * V[:,1:k]' )
```

See the figure on the facing page and the QR link at the bottom of this page. The Frobenius norm, which gives the root mean squared error of two images by

⁴Deflate compression used in PNGs combines Huffman coding that generates a dictionary to replace frequently occurring sequences of bits with short ones and Lempel-Ziv-Storer-Szymanski (LZSS) compression that replaces repeated sequences of bits with pointers to earlier occurrences.

⁵A GIF only supports an 8-bit color palette for each image, which results in posterization and makes it less suitable for color photographs. On the other hand, a PNG will support up to 16 bits per channel.



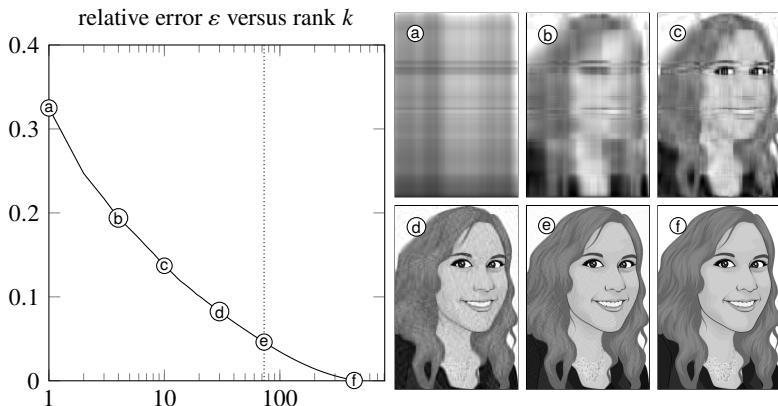


Figure 3.5: The relative Frobenius error of an SVD compressed image as a function of rank. The dotted line shows equivalent lossless PNG compression.

comparing them pixel-wise, is the sum of the singular values

$$\|\mathbf{A} - \mathbf{A}_k\|_F^2 = \sum_{i=1}^n \sigma_i^2 - \sum_{i=1}^k \sigma_i^2 = \sum_{i=k+1}^n \sigma_i^2.$$

We can see this by checking that `norm(A-A_k)≈norm(σ[k+1:end])` returns the value true. Let's plot the error as a function of rank, shown in the figure above:

```
ε² = 1 .- cumsum(σ)/sum(σ); scatter(ε²,xaxis=:log)
```

A rank k image will require roughly $k(n+m)$ bytes compared to nm bytes for an uncompressed image. So, unless k is significantly smaller than n and m , the resulting storage will exceed the storage of the original image. Furthermore, an image compressed as a (lossless) PNG or lossless JPEG will typically be much smaller than a compressed image without any loss in quality.

► Image recognition

Image recognition is an important area of research. One method, the eigenface method developed for facial recognition, applies singular value decomposition to a set of training images to produce a smaller set of orthogonal images called eigenfaces (Sirovich and Kirby [1987]).

Take a set of n $p \times q$ -pixel grayscale training images, and reshape each image into a $pq \times 1$ array. We'll denote each as \mathbf{d}_j for $j = 1, 2, \dots, n$. It is common practice to mean center and standardize these vectors by subtracting the element-wise means of $\{\mathbf{d}_j\}$ and dividing by the element-wise non-zero standard

deviations of $\{\mathbf{d}_j\}$ to reduce the condition number. Let $\mathbf{D} = [\mathbf{d}_1 \ \mathbf{d}_2 \ \cdots \ \mathbf{d}_n]$ with the singular value decomposition $\mathbf{D} = \mathbf{U}\Sigma\mathbf{V}^\top$. By keeping the largest k singular values and discarding the rest, we can create a lower rank approximation $\mathbf{U}_k\Sigma_k\mathbf{V}_k^\top$.

The projection of \mathbf{D} onto the column space of \mathbf{U}_k is

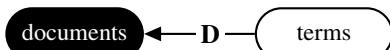
$$\mathbf{U}_k\mathbf{U}_k^\top\mathbf{D} = \mathbf{U}_k\mathbf{W}_k$$

where $\mathbf{W}_k = \mathbf{U}_k^\top\mathbf{D} = \Sigma_k\mathbf{V}_k^\top$. We only need to store is the $k \times n$ signature matrix \mathbf{W}_k and $pq \times k$ eigenface matrix \mathbf{U}_k . One of the benefits of the eigenface method is that although we use n training images, we can use substantially lower-dimensional subspace by discarding the singular vectors corresponding to the smallest singular values. See the figure on the facing page.

Latent (from the Latin for “hidden”) refers to variables that are not directly observed yet help explain a model. Latent variables often arise out of dimensionality reduction and better tie together underlying concepts. The lower-dimensional space of these variables is called the latent space. Figure 3.7 on page 82 shows the two-dimensional latent space for the set of letter images. Contributions from the first right singular vector are along the horizontal axis and from the second right singular vector are along the vertical axis. Even in a two-dimensional latent space, we can see quite a bit of clustering. Notice how images (the letters) with similar appearances are quite close together in the latent space. The O is quite near the Q—which is simply an O with a tail. The T, I, and J—the only letters with center stems—all live in the same lower-left corner of the latent space. The letters with that have left stems F, P, E, L, R, K, N, D, and U all reside in the upper half-plane. Letters with left curved strokes C, G, O, and Q are all in the lower-right quadrant. The letters M and W are close together yet away from all the others. Is it perhaps because of their broad stretch?

► Latent semantic analysis

Suppose that you want to find similar books in a library or a journal article that best matches a given query. One way to do this is by making a list of all possible terms and counting the frequency of each term in the documents as components of a vector (sometimes called a “bag of words”). The j th element of a vector \mathbf{d}_i tells us the number of times that term j appears in document i . The $m \times n$ document-term matrix $\mathbf{D} = [\mathbf{d}_1 \ \mathbf{d}_2 \ \cdots \ \mathbf{d}_n]$ maps words to the documents containing those words:



Given a query \mathbf{q} , we can find the most relevant documents to the query by finding the closest vector \mathbf{d}_i to \mathbf{q} . By using *cosine similarity*, we identify the document i that maximizes $\cos \theta_i = \mathbf{q}^\top \mathbf{d}_i / \|\mathbf{q}\| \|\mathbf{d}_i\|$. Values close to one represent a good

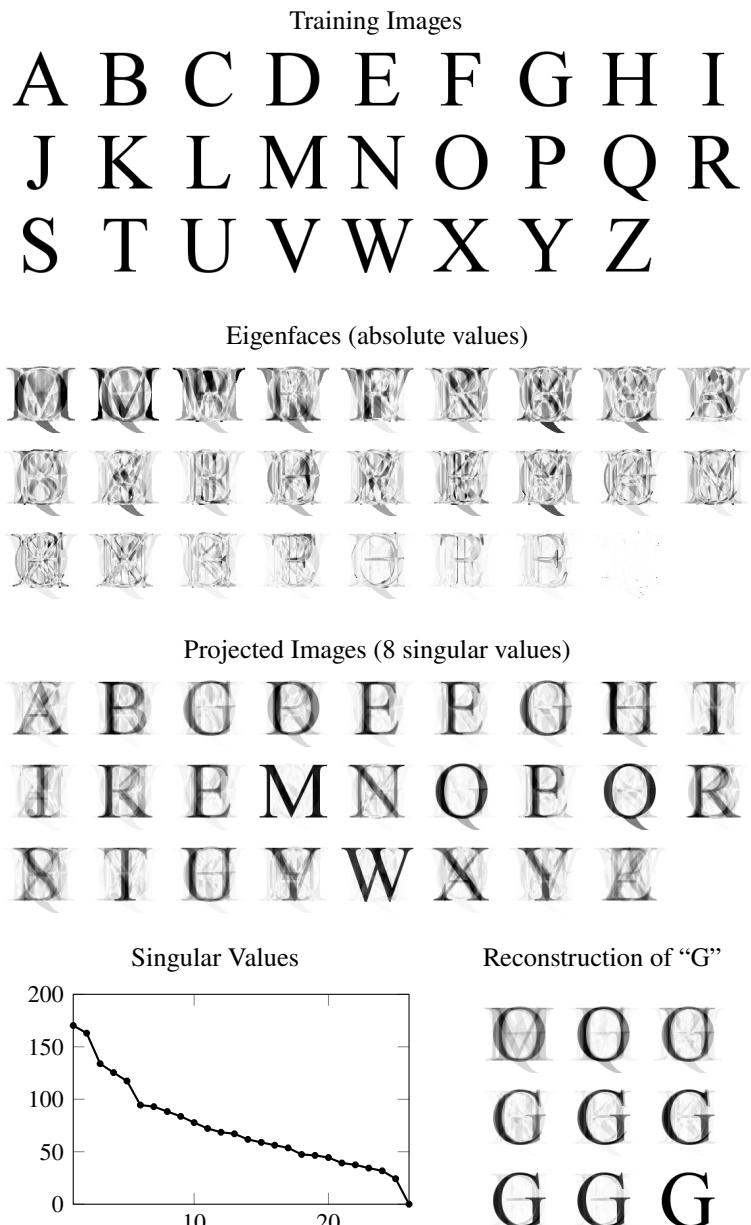


Figure 3.6: These 120×120 pixel images span a 26-dimensional subspace of \mathbb{R}^{14400} . Eigenfaces (ordered by their associated singular values) form an orthogonal basis for the subspace spanned by the training images. Also, see the QR link below.



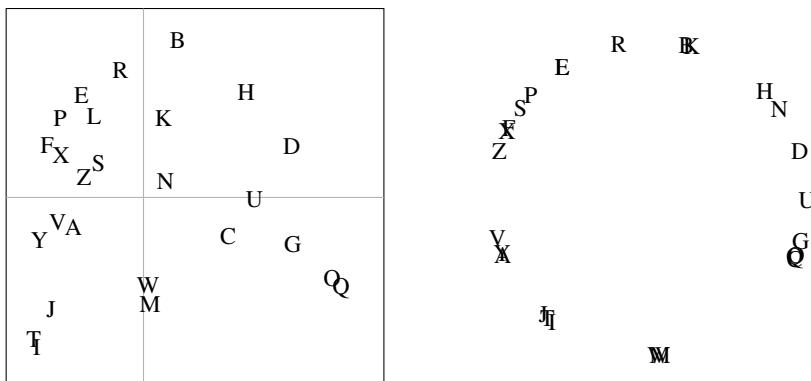
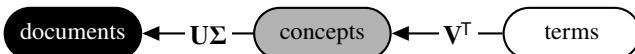


Figure 3.7: Left: The two-dimensional latent space of the letters with the v_1 and v_2 -axes of the right-singular vectors shown. Notice how letters with similar appearances are quite close together in the latent space such as the “O” and “Q” and the “T,” “I,” and “J.” Right: The directions of the same letters. Also, see the three-dimensional latent space of letters at the QR link at the bottom of this page.

match, and values close to zero represent a poor match. This approach is called *explicit semantic analysis*.

Searching for a document using only terms is limiting because you need to explicitly match terms. For example, when searching for books or articles about Barcelona, terms like Spain, Gaudi, Picasso, Catalonia, city, and soccer are all conceptually relevant. In this case, rather than clustering documents by explicit variables like terms, it would be better to cluster documents using latent variables like concepts. This is the idea behind *latent semantic analysis*. We can think of concepts as intermediaries between text and documents. Take the singular value decomposition of the document-term matrix $\mathbf{D} = \mathbf{U}\Sigma\mathbf{V}^T$. The transform \mathbf{V}^T maps from terms to latent concept space and the mapping $\mathbf{U}\Sigma$ completes the mapping to documents:



We don't need or want the full rank of \mathbf{D} , so instead let's take a low rank k approximation \mathbf{D}_k and its singular value decomposition $\mathbf{U}_k\Sigma_k\mathbf{V}_k^T$. Then the latent space vectors $\hat{\mathbf{d}}_i$ are the columns of \mathbf{V}_k^T . In other words, $\hat{\mathbf{d}}_i = \Sigma_k^{-1}\mathbf{U}_k^T\mathbf{d}_i$. We can similarly map a query vector \mathbf{q} to its latent space representation by taking the projection $\hat{\mathbf{q}} = \Sigma_k^{-1}\mathbf{U}_k^T\mathbf{q}$. In this case, we look for the column i that maximizes $\cos \theta_i = \hat{\mathbf{q}}^T\hat{\mathbf{d}}_i / \|\hat{\mathbf{q}}\| \|\hat{\mathbf{d}}_i\|$.



3.4 Non-negative matrix factorization

Singular value decomposition breaks a matrix into orthogonal factors that we can use for rank reduction such as principal component analysis or feature extraction such as latent semantic indexing. One issue with singular value decomposition is that the factors contain negative components. For some applications, negative components may not be meaningful. Another issue with singular value decomposition is that the factors typically do not preserve sparsity, a possibly undesirable trait. For these applications, it is useful to have a method of factorization that preserves nonnegative elements. Take an $m \times n$ matrix \mathbf{X} whose elements are all nonnegative. Nonnegative matrix factorization (NMF) finds a rank p approximate matrix factorization $\mathbf{W}\mathbf{H}$ to \mathbf{X} where \mathbf{W} and \mathbf{H} are nonnegative $m \times p$ and $p \times n$ matrices.

What are the interpretations of the factors \mathbf{W} and \mathbf{H} ? Suppose that we are using NMF for text analysis. In this case, each column of \mathbf{X} may correspond to one of m words and each row may correspond to one of n documents. The matrix \mathbf{W} relates p concepts to the n documents and each element of \mathbf{H} tells us the importance of a specific topic to a specific word.

Nonnegative matrix factorization can also be used for hyperspectral image analysis. A hyperspectral image is an image cube consisting of possibly hundreds of layers of images, each one capturing a wavelength band of a broad spectrum. Hyperspectral remote sensing can be used to detect diseased plants in cropland, mineral ores and oil, or concealed and camouflaged targets. Hyperspectral unmixing tries to identify the constitutive materials such as grass, roads, or metal surfaces (called endmembers) within a hyperspectral image and to classify which pixels contain which endmembers and in what proportion. An $m_1 \times m_2 \times n$ image cube can be reshaped into a $m_1 m_2 \times n$ matrix \mathbf{X} with a spectral signature for each pixel. In the nonnegative matrix factorization, \mathbf{W} is the spectral signature of an endmember and \mathbf{H} is the abundance of the endmember in each pixel.

Nonnegative matrix factorization is a constrained optimization problem. We want to find the matrices $\mathbf{W} \geq 0$ and $\mathbf{H} \geq 0$ that minimize the $\|\mathbf{X} - \mathbf{WH}\|_F^2$. We use the Frobenius norm because we want an element-by-element or pixel-by-pixel comparison. A common approach to constrained optimization problems is using the Karush–Kuhn–Tucker (KKT) conditions introduced on page 70. For an objective function $F(\mathbf{W}, \mathbf{H}) = \frac{1}{2} \|\mathbf{X} - \mathbf{WH}\|_F^2$ and a constraint $g(\mathbf{W}, \mathbf{H}) = (\mathbf{W}, \mathbf{H}) \geq 0$, the KKT conditions can be expressed as:

$$\begin{array}{lll} \textcircled{1} \quad \mathbf{W} \geq 0, & \textcircled{2} \quad \nabla_{\mathbf{W}} F = (\mathbf{WH} - \mathbf{X})\mathbf{H}^T \geq 0, & \textcircled{3} \quad \mathbf{W} \circ \nabla_{\mathbf{W}} F = 0; \text{ and} \\ \textcircled{4} \quad \mathbf{H} \geq 0, & \textcircled{5} \quad \nabla_{\mathbf{H}} F = \mathbf{W}^T(\mathbf{WH} - \mathbf{X}) \geq 0, & \textcircled{6} \quad \mathbf{H} \circ \nabla_{\mathbf{H}} F = 0. \end{array}$$

where \circ is component-wise (Hadamard) multiplication. By substituting $\textcircled{2}$ into $\textcircled{3}$ we have $\mathbf{W} \circ \mathbf{WHH}^T - \mathbf{W} \circ \mathbf{XH}^T = 0$. From this equality it follows that $\mathbf{W} = \mathbf{W} \circ \mathbf{XH}^T \oslash \mathbf{WHH}^T$ where \oslash is component-wise (Hadamard) division.

Similarly by substituting ⑤ into ⑥ we have $\mathbf{H} = \mathbf{H} \circ \mathbf{W}^\top \mathbf{X} \oslash \mathbf{W}^\top \mathbf{W} \mathbf{H}$. These two equations can be implemented as iterative multiplicative updates:

$$\mathbf{W} \leftarrow \mathbf{W} \circ \mathbf{X} \mathbf{H}^\top \oslash \mathbf{W} \mathbf{H} \mathbf{H}^\top$$

$$\mathbf{H} \leftarrow \mathbf{H} \circ \mathbf{W}^\top \mathbf{X} \oslash \mathbf{W}^\top \mathbf{W} \mathbf{H}.$$

The method is relatively slow to converge and not guaranteed to converge, but implementation is simple. Also, we may need to take care to avoid 0/0 if \mathbf{W} has a zero row or \mathbf{H} has a zero column.

For a given sparse, nonnegative matrix \mathbf{X} , we start with two random, non-negative matrices \mathbf{W} and \mathbf{H} . A naïve Julia implementation of nonnegative matrix factorization using multiplicative updates is

```
function nmf(X,p=6)
    W = rand(Float64, (size(X,1), p))
    H = rand(Float64, (p,size(X,2)))
    for i in 1:50
        W = W.*((X*H') ./ (W*(H*H') .+ (W.≈0)))
        H = H.*((W'*X) ./ ((W'*W)*H .+ (H.≈0)))
    end
    (W,H)
end
```

To set a stopping criterion we can look for convergence of the residual $\|\mathbf{X} - \mathbf{WH}\|_F^2$, stopping if the change in the residual from one iteration to the next falls below some threshold. Alternatively, we can stop if $\|\mathbf{W} \circ \nabla_{\mathbf{W}} F\|_F + \|\mathbf{H} \circ \nabla_{\mathbf{H}} F\|_F$ falls below a threshold or if \mathbf{W} or \mathbf{H} becomes stationary. To go deeper into non-negative matrix factorization see Nicolas Gillis' article "The Why and How of Nonnegative Matrix Factorization.'

 The JuliaStats community's NMF.jl package does nonnegative matrix factorization.

3.5 Exercises

3.1. Consider the overdetermined system $x = 1$ and $x = 2$. Find the “best” solution in three ways by minimizing the 1-, 2- and ∞ -norms of the residual. Which approaches are well-posed?

3.2. The continuous L^2 -norm is

$$(f, g) = \int_0^1 f(x)g(x) \, dx.$$

Use the definition of the angle subtended by vectors

$$\theta = \cos^{-1} \frac{(\mathbf{u}, \mathbf{v})}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

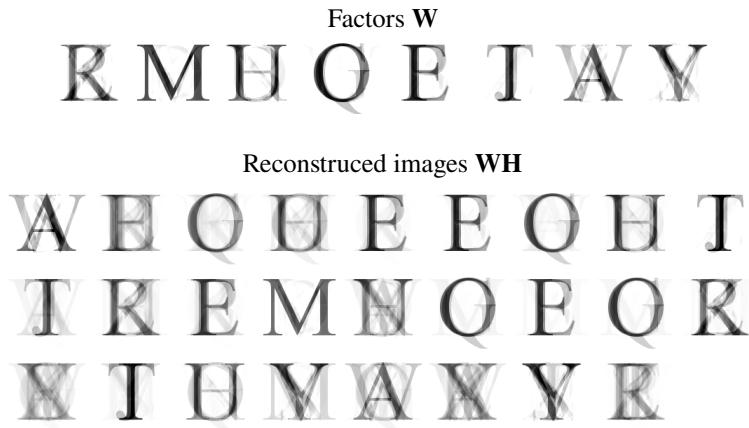


Figure 3.8: Nonnegative matrix factorization with rank $p = 8$ was applied to the 26 training images in Figure 3.6 on page 81. The reconstructed images are combinations of the eight factors \mathbf{W} using positive weights \mathbf{H} .

to show that the space of monomials $\{1, x, x^2, \dots, x^n\}$ is far from orthogonal when n is large. Note that the components $h_{ij} = (x^i, x^j)$ form the Hilbert matrix.

3.3. Prove that the Moore–Penrose pseudoinverse satisfies the properties:

$$\mathbf{A}\mathbf{A}^+\mathbf{A} = \mathbf{A}, \quad \mathbf{A}^+\mathbf{A}\mathbf{A}^+ = \mathbf{A}^+, \quad (\mathbf{A}\mathbf{A}^+)^T = \mathbf{A}\mathbf{A}^+, \quad \text{and} \quad (\mathbf{A}^+\mathbf{A})^T = \mathbf{A}^+\mathbf{A}.$$

- 4 3.4. The Filippelli problem was contrived to benchmark statistical software packages. The National Institute of Standards and Technology (NIST) “Filip” Statistical Reference Dataset⁶ consists of 82 (y, x) data points along with a certified degree-10 polynomial

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_{10} x^{10}.$$

Find the parameters $\{\beta_0, \beta_1, \beta_2, \dots, \beta_{10}\}$ by first constructing a Vandermonde matrix \mathbf{V} with $V_{ij} = x_i^{p-j}$ and then using the normal equation, QR decomposition, and the pseudoinverse to solve the system. What happens if you construct the Vandermonde matrix in increasing versus decreasing order or powers? What happens if you choose different cutoff values for the pseudo-inverse? What

⁶<http://www.itl.nist.gov/div898/strd/lls/data/Filip.shtml>

happens if you standardize the x and y data sets by their respective means and dividing them by their respective standard deviations? The Filip dataset and certified parameters are available “filip.csv” and “filip_coeffs.csv” at <https://github.com/nmfsc/data/>.

• The `SpecialMatrices.jl` function `Vandermonde` generates a Vandermonde matrix for input (x_0, x_1, \dots, x_n) with rows given by $[1 \quad x_i \quad \dots \quad x_i^{P-1} \quad x_i^P]$.

- ✍ 3.5. The daily temperature of a city over the span of several years can be modeled as a sinusoidal function. Use linear least squares to develop such a model historical data recorded in Washington, DC between 1967 to 1971, available as as “`dailytemps.csv`” at <https://github.com/nmfsc/data/>. Or choose your own data at <https://www.ncdc.noaa.gov>
- ✍ 3.6. Principal component analysis is often used for dimensionality reduction and as the basis of the eigenface method of image classification. Suppose that we want to identify handwritten digits for automatic mail sorting. Variations in the writing styles would require us to collect a wide range of handwriting samples to serve as a training set. The MNIST (Modified National Institute of Standards and Technology) dataset is a mixture of handwriting samples from American Census Bureau employees and high school students. (LeCun et al. [1998]) Each image is scaled to a 28×28 -pixel grayscale image. For reference, the following 9s are sampled from the dataset:



Use the `MLDatasets.jl` package to load the MNIST dataset.⁷ The training set has sixty-thousand images with labels and the test set has ten-thousand images with labels. Sort and reshape the training data to form ten $28^2 \times n_i$ matrices \mathbf{D}_i using labels $i = \{0, 1, \dots, 9\}$ where n_i is the number of images for each label. Now, compute a low-rank SVD of \mathbf{D}_i to get \mathbf{V}_i . The columns of \mathbf{V}_i form the basis for the low-rank subspace. To classify a new test image \mathbf{d} , we find the subspace to which \mathbf{d} is closest by comparing \mathbf{d} with its projection $\mathbf{V}_i \mathbf{V}_i^\top \mathbf{d}$

$$\arg \min_{i \in \{0, 1, \dots, 9\}} \|\mathbf{V}_i \mathbf{V}_i^\top \mathbf{d} - \mathbf{d}\|_2.$$

Check how well your method properly identifies the test images.

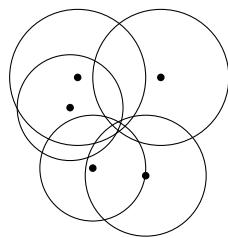
- 3.7. Apply Zipf’s law to the frequency of surnames in the United States. You can find a 2010 census data set “`surnames.csv`” at <https://github.com/nmfsc/data/>.

⁷Alternatively, you can download the training and test data as the MAT-file `emnist-mnist.mat` from <https://www.nist.gov/itl/products-and-services/emnist-dataset> or <https://github.com/nmfsc/data>.

3.8. A model is a simplification of the real world. Take Hollywood actors, for example. We can characterize actors by roles that they often play. Russell Crowe is largely thought of as an action film actor, Ben Stiller as a comedic actor, and Kate Winslet as dramatic actor. Others have crossed over between comedic roles and serious ones. Build an actor similarity model using film genres to determine which actors are most similar to a given actor. Use the data sets from exercise 2.8 along with the adjacency matrix “movie-genre.csv” that relates movies to genres listed in the file “genres.txt.” Use cosine similarity to determine which actors are most similar to a chosen actor. How well does this approach work in clustering actors? What are the limitations of this model? How does the dimension of the latent space change the model?

3.9. Multilateration determines an object’s position by comparing the times of arrival of a signal at different monitoring stations. Multilateration can locate a gunshot in an urban neighborhood using a network of microphones or an earthquake’s epicenter using seismographs scattered around the world. Consider a two-dimensional problem with n stations each located at (x_i, y_i) and a time of arrival t_i . The source (x, y) and time of transmission t are computed from the intersection of the n circles

$$(x - x_i)^2 + (y - y_i)^2 = c^2(t_i - t)^2$$



for $i = 1, 2, \dots, n$ where c is the signal speed. This system can be solved by first reducing it to a system $n - 1$ linear equations. Because of variations in signal speed, the diffraction of waves around objects, measurement errors, etc., the system may be inconsistent. In this case, we can use least squares to find a best solution. Take $c = 1$ and determine (x, y, t) for the data

$$(x_i, y_i, t_i) = \{(3, 3, 12), (1, 15, 14), (10, 2, 13), (12, 15, 14), (0, 11, 12)\}.$$

Chapter 4

Computing Eigenvalues



Using pencil and paper, one often finds the eigenvalues of a matrix by determining the zeros of the characteristic polynomial $\det(\mathbf{A} - \lambda\mathbf{I})$. This approach works for small matrices, but we cannot hope to apply it to a general matrix larger than 4×4 . The Abel–Ruffini theorem states that there are no general solutions to polynomial equations of degree five or higher using a finite number of algebraic operations (adding, subtracting, multiplying, dividing, and taking an integer or fractional power). As a consequence, there can be no direct method for determining eigenvalues of a general matrix of rank five or higher. We must find the eigenvalues using iterative methods instead.

Of course, it is possible to compute the characteristic polynomial and then use a numerical method such as the Newton–Horner method to approximate its roots. However, in general, such an approach is unstable. In fact, a common technique to find the roots of a polynomial $p(x)$ is to generate the companion matrix of $p(x)$, i.e., the matrix whose characteristic polynomial is $p(x)$, and then determine the eigenvalues of the companion matrix using the QR method introduced in this chapter.

The `Polynomials.jl` function `roots` finds the roots of a polynomial $p(x)$ by computing the eigenvalues for the companion matrix of $p(x)$.

4.1 Eigenvalue sensitivity and estimation

Before we discuss methods for determining eigenvalues, let's look at the estimation and stability of eigenvalues. Suppose that \mathbf{A} is semisimple with the diagonalization $\mathbf{A} = \mathbf{S}^{-1}\mathbf{AS}$, and suppose that $\delta\mathbf{A}$ is some change in \mathbf{A} . Then the

perturbation in the eigenvalues is

$$\Lambda + \delta\Lambda = S^{-1}(A + \delta A)S$$

and hence

$$\delta\Lambda = S^{-1}\delta AS.$$

Taking the matrix norm

$$\|\delta\Lambda\| \leq \|S^{-1}\| \|\delta A\| \|S\| = \kappa(S) \|\delta A\|.$$

So, $\delta\Lambda$ is magnified by the condition number of the matrix of eigenvectors.

Let's look at stability again. The *left eigenvector* of a matrix A is a row vector y^H that satisfies

$$y^H A = \lambda y^H.$$

Suppose that A , x and λ vary with some perturbation parameter and let δA , δx and $\delta\lambda$ denote their derivatives with respect to this parameter. Differentiating $Ax = \lambda x$ gives us

$$\delta Ax + A\delta x = \delta\lambda x + \lambda\delta x.$$

Multiply by the left eigenvector

$$y^H \delta Ax + y^H A \delta x = y^H \delta\lambda x + y^H \lambda \delta x.$$

Since $y^H A \delta x = y^H \lambda \delta x$, we have $y^H \delta Ax = y^H \delta\lambda x$, and so

$$\delta\lambda = \frac{y^H \delta Ax}{y^H x}.$$

Therefore,

$$\|\delta\lambda\| \leq \frac{\|y\| \|x\|}{y^H x} \|\delta A\|.$$

We define the *eigenvalue condition number* to be

$$\kappa(\lambda, A) = \frac{\|y\| \|x\|}{y^H x},$$

that is, one over the cosine of the angle between the right and left eigenvectors. We can compute the eigenvalue condition number by using the spectral decomposition $A = S\Lambda S^{-1}$:

```
function condeig(A)
    S_r = eigvecs(A)
    S_l = inv(S_r')
    S_l ./= sqrt.(sum(abs.(S_l.^2), dims=1))
    1 ./ abs.(sum(S_r.*S_l, dims=1))
end
```

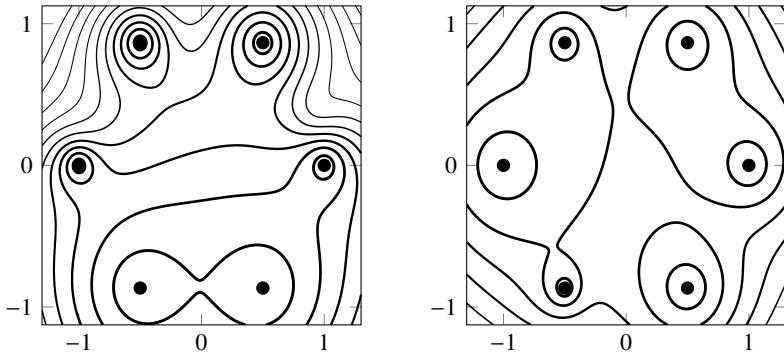


Figure 4.1: Contours of ε -pseudospectra of two similar matrices. Each contour represents a change of $\varepsilon = 0.0005$. For each matrix the 2-condition number is on the order of 10^6 and the condition number of the matrix of eigenvectors is on the order of 10^3 .

Finding the spectrum of a matrix \mathbf{A} is the same as asking which values λ make $\mathbf{A} - \lambda\mathbf{I}$ singular, i.e.,

$$\lambda(\mathbf{A}) = \{z \in \mathbb{C} \mid \mathbf{A} - z\mathbf{I} \text{ is singular}\}.$$

Numerically, such a question may be ill posed. Instead, we should ask when $\mathbf{A} - \lambda\mathbf{I}$ is close to singular, that is, when $\|(\mathbf{A} - \lambda\mathbf{I})^{-1}\|$ is large. In this case, it is useful to talk about the pseudospectrum of the matrix \mathbf{A} .

We define the ε -pseudospectrum of \mathbf{A} as the set

$$\lambda_\varepsilon(\mathbf{A}) = \{z \in \mathbb{C} \mid \|(\mathbf{A} - z\mathbf{I})^{-1}\| \geq \frac{1}{\varepsilon}\}.$$

In other words, z is an eigenvalue of $\mathbf{A} + \mathbf{E}$ with $\|\mathbf{E}\| < \varepsilon$. For the Euclidean norm,

$$\lambda_\varepsilon(\mathbf{A}) = \{z \in \mathbb{C} \mid \sigma_{\min}(\mathbf{A} - z\mathbf{I}) \leq \varepsilon\}.$$

See Figure 4.1 above.

Theorem 10 (Gershgorin circle theorem). *Let $\mathbf{A} \in \mathbb{C}^{n \times n}$. Then*

$$\lambda(\mathbf{A}) \subseteq \mathcal{S}_R = \bigcup_{i=1}^n \mathcal{R}_i \quad \text{where} \quad \mathcal{R}_i = \{z \in \mathbb{C} : |z - a_{ii}| \leq \sum_{j=1, j \neq i}^n |a_{ij}|\}.$$

The sets \mathcal{R}_i are called Gershgorin circles.

Proof. Any eigenvalue along the diagonal of \mathbf{A} is clearly in a Gershgorin circle, so we only need to consider eigenvalues that are not diagonal elements of \mathbf{A} . For each $\lambda \in \lambda(\mathbf{A})$, we introduce the matrix

$$\mathbf{B}_\lambda = \mathbf{A} - \lambda \mathbf{I} = (\mathbf{D} - \lambda \mathbf{I}) + \mathbf{M},$$

where \mathbf{D} is composed of the diagonal of \mathbf{A} and \mathbf{M} is composed of the off-diagonal elements. There is a nonzero vector \mathbf{x} (an eigenvector of \mathbf{A}) in the null space of \mathbf{B}_λ , so

$$\mathbf{B}_\lambda \mathbf{x} = ((\mathbf{D} - \lambda \mathbf{I}) + \mathbf{M})\mathbf{x} = \mathbf{0}.$$

Equivalently, $\mathbf{x} = (\mathbf{D} - \lambda \mathbf{I})^{-1} \mathbf{M} \mathbf{x}$. In the ∞ -norm,

$$\|\mathbf{x}\|_\infty = \|(\mathbf{D} - \lambda \mathbf{I})^{-1} \mathbf{M} \mathbf{x}\|_\infty \leq \|(\mathbf{D} - \lambda \mathbf{I})^{-1} \mathbf{M}\|_\infty \|\mathbf{x}\|_\infty$$

Therefore,

$$1 \leq \|(\mathbf{D} - \lambda \mathbf{I})^{-1} \mathbf{M}\|_\infty = \sup_{1 \leq i \leq n} \sum_{\substack{j=1 \\ j \neq i}}^n \frac{|a_{ij}|}{|a_{ii} - \lambda|}.$$

It follows that

$$|a_{ii} - \lambda| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|.$$

Hence, $\lambda \in \mathcal{R}_i$. □

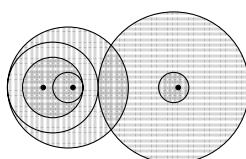
Corollary 11. Let $\mathbf{A} \in \mathbb{C}^{n \times n}$. Then $\lambda(\mathbf{A}) \subseteq \mathcal{S}_{\mathcal{R}} \cap \mathcal{S}_C$ where

$$\mathcal{S}_C = \bigcup_{i=1}^n C_i \quad \text{where} \quad C_i = \{z \in \mathbb{C} : |z - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ji}|\}.$$

Proof. Substitute the 1-norm for ∞ -norm in the Gershgorin circle theorem to give column circles instead of row circles. □

Example. The Gershgorin circles are plotted for the following matrix:

$$\mathbf{A} = \begin{bmatrix} 10 & 2 & 3 \\ 1 & 2 & 1 \\ 0 & 1 & 3 \end{bmatrix}$$



Row circles are shaded , column circles are shaded , and the eigenvalues lie in their intersections. ◀

Example. A stochastic or Markov matrix is a square, nonnegative matrix whose rows or columns each sum to one. When the rows each sum to one, it's called a right stochastic matrix; and when the columns each sum to one, it's called a left stochastic matrix. A stochastic or probability vector is a nonnegative vector whose elements sum to one. Stochastic matrices describe the transition in a Markov chain, a sequence of events where the probability of an event occurring depends only on the state of the previous event. It's clear that the ones vector $\mathbf{1} = (1, 1, \dots, 1)$ is an eigenvector with a corresponding eigenvalue 1 of any right stochastic matrix. Furthermore, from the Gershgorin circle theorem, we know that the spectral radius of a stochastic matrix is at most 1. So, it follows that 1 is the largest eigenvalue of a stochastic matrix. \blacktriangleleft

Theorem 12 (Perron–Frobenius theorem). *A positive square matrix (a matrix whose elements are all positive) has a unique largest real eigenvalue. The corresponding eigenvector has strictly positive components.*

Proof. There are several approaches to proving the Perron–Frobenius theorem. We'll take a geometric approach. Let \mathbf{A} be an $n \times n$ matrix with positive elements. Let X be the set of points (x_1, x_2, \dots, x_n) on the unit sphere in the nonnegative orthant—the points with $x_1^2 + x_2^2 + \dots + x_n^2 = 1$ and $x_i \geq 0$. Define the mapping $T : X \rightarrow X$ as $T\mathbf{x} = \mathbf{Ax}/\|\mathbf{Ax}\|_2$. Then, T is a contraction mapping over X because the elements of \mathbf{A} are positive. Therefore, by the Banach fixed-point theorem (page 195) the mapping T has a unique fixed point $\mathbf{v} \in X$. That means \mathbf{A} has a *unique* eigenvector \mathbf{v} with strictly positive components. Let the corresponding eigenvalue be λ . We can also consider a similar contraction mapping that generates a unique left eigenvector \mathbf{u} with strictly positive components and eigenvalue λ' . Furthermore, $\lambda' = \lambda$ because $\lambda'\mathbf{u}^\top \mathbf{v} = \mathbf{u}^\top \mathbf{Av} = \lambda \mathbf{v}^\top \mathbf{u}$ and $\mathbf{u}^\top \mathbf{v} > 0$. Now, let μ be any other real eigenvalue of \mathbf{A} and let \mathbf{w} be the corresponding eigenvector. Then at least one component of \mathbf{w} is negative. Let $|\mathbf{w}|$ be the vector whose components are the absolute values of components of \mathbf{w} . It follows that

$$|\mu| \mathbf{u}^\top |\mathbf{w}| = \mathbf{u}^\top |\mathbf{Aw}| < \mathbf{u}^\top \mathbf{A} |\mathbf{w}| = \lambda \mathbf{u}^\top |\mathbf{w}|.$$

Because \mathbf{u} and $|\mathbf{w}|$ are positive, $\mathbf{u}^\top |\mathbf{w}|$ is positive. So $|\mu| < \lambda$.

As an extension, the Perron–Frobenius theorem also holds for primitive matrices. A *primitive matrix* is a square non-negative matrix some power of which is a positive matrix. \square

4.2 The power method

Let $\mathbf{A} \in \mathbb{C}^{n \times n}$ be a diagonalizable matrix with eigenvalues ordered by magnitude $|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|$. We say that λ_1 is the *dominant eigenvalue* of \mathbf{A} . An eigenvector associated with λ_1 is a dominant eigenvector. Now, consider the

iterative method $\mathbf{x}^{(k)} = \mathbf{A}\mathbf{x}^{(k-1)}$ for an initial guess $\mathbf{x}^{(0)}$. As long as $\mathbf{x}^{(0)}$ is not orthogonal to a dominant eigenvector, the sequence $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots$ will converge to a dominant eigenvector.

Because \mathbf{A} is diagonalizable, its eigenvectors $\{\mathbf{x}_i\}$ form a basis of \mathbb{C}^n , and every vector $\mathbf{x}^{(0)}$ can be represented as a linear combination of the eigenvectors

$$\mathbf{x}^{(0)} = \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + \cdots + \alpha_n \mathbf{x}_n.$$

By applying \mathbf{A} to this initial vector, we have

$$\begin{aligned}\mathbf{x}^{(1)} &= \mathbf{A}\mathbf{x}^{(0)} = \alpha_1 \mathbf{A}\mathbf{x}_1 + \alpha_2 \mathbf{A}\mathbf{x}_2 + \cdots + \alpha_n \mathbf{A}\mathbf{x}_n \\ &= \alpha_1 \lambda_1 \mathbf{x}_1 + \alpha_2 \lambda_2 \mathbf{x}_2 + \cdots + \alpha_n \lambda_n \mathbf{x}_n\end{aligned}$$

And after k iterations

$$\begin{aligned}\mathbf{x}^{(k)} &= \mathbf{A}^k \mathbf{x}^{(0)} = \alpha_1 \lambda_1^k \mathbf{x}_1 + \alpha_2 \lambda_2^k \mathbf{x}_2 + \cdots + \alpha_n \lambda_n^k \mathbf{x}_n \\ &= \alpha_1 \lambda_1^k \left(\mathbf{x}_1 + \frac{\alpha_2}{\alpha_1} \left(\frac{\lambda_2}{\lambda_1} \right)^k \mathbf{x}_2 + \cdots + \frac{\alpha_n}{\alpha_1} \left(\frac{\lambda_n}{\lambda_1} \right)^k \mathbf{x}_n \right) \\ &= \lambda_1^k \alpha_1 \mathbf{x}_1 + O(|\lambda_2/\lambda_1|^k) \text{ terms.}\end{aligned}$$

The power method converges linearly as $O(|\lambda_2/\lambda_1|^k)$. Let $\mathbf{x}^{(k)}$ equal the normalized k th iterate and \mathbf{x}_i a unit eigenvalue. Then the error at the k th iterate is

$$\|\mathbf{x}^{(k)} - \mathbf{x}_1\| = \left\| \sum_{i=2}^n \frac{\alpha_i}{\alpha_1} \left(\frac{\lambda_i}{\lambda_1} \right)^k \mathbf{x}_i \right\| \leq \left| \frac{\lambda_2}{\lambda_1} \right|^k \left\| \sum_{i=2}^n \frac{\alpha_i}{\alpha_1} \mathbf{x}_i \right\| \leq C \left| \frac{\lambda_2}{\lambda_1} \right|^k.$$

For the Euclidean norm $C = \alpha_1^{-1} \sqrt{\alpha_2^2 + \alpha_3^2 + \cdots + \alpha_n^2}$.

To prevent over or underflow, we normalize $\mathbf{x}^{(k)}$ at each iteration. Starting with an initial guess $\mathbf{x}^{(0)}$,

$$\begin{array}{ll}\text{Multiply:} & \mathbf{x}^{(k+1)} = \mathbf{A}\mathbf{x}^{(k)} \\ \text{Normalize:} & \mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k+1)} / \|\mathbf{x}^{(k+1)}\|.\end{array}$$

At each step, the approximate dominant eigenvalue is $\mathbf{x}^{(k)\text{H}} \mathbf{A} \mathbf{x}^{(k)}$. Stop when the difference in iterates $\|\mathbf{x}^{(k+1)} - \text{sign}(\mathbf{x}_1^{(k+1)} \mathbf{x}_1^{(k)}) \mathbf{x}^{(k)}\|$ or the angle between iterates $(1 - \mathbf{x}^{(k+1)\text{H}} \mathbf{x}^{(k)})$ is within a prescribed tolerance.

Example. The results of round-robin chess tournaments are often recorded on square matrix with 1 for a victory, $\frac{1}{2}$ for a draw, and 0 for a defeat. A player's tournament score is given by the sum of the scores along the rows.

Player	Sza	Tai	Gli	Kor	Cas	Ros	Cas	Kle	Pts.	S.B.
Szabó	—	1	1/2	1/2	1/2	1	1	1	5 1/2	16 1/2
Taimanov	0	—	1	1	1	1/2	1	1	5 1/2	16
Gligoric	1/2	0	—	1/2	1	1	1	1	5	12 1/2
Kortchnoi	1/2	0	1/2	—	1/2	1	1	1	4 1/2	11 1/2
Casas	1/2	0	0	1/2	—	1	1/2	0	2 1/2	7 3/4
Rossetto	0	1/2	0	0	0	—	1/2	1	2	5
Casabella	0	0	0	0	1/2	1/2	—	1/2	1 1/2	3
Klein	0	0	0	0	1	0	1/2	—	1 1/2	3 1/4

Figure 4.2: Crosstable of the 1960 Santa Fe chess tournament with the combined scores (*Pts.*) and the tie breaking Sonneborn–Berger score (*S.B.*)

Mathematically, for a matrix \mathbf{M} the combined scores are $\mathbf{r} = \mathbf{M}\mathbf{1}$, where $\mathbf{1}$ is a vector of ones. These scores determine ranking and prize money. But, what happens when there is a tie? In 1873, Austrian chess master Oscar Gelbfuhs proposed using a weighted row sum, multiplying each game score by the opponent's cumulative score. In this way, a player earns more points in defeating a strong opponent than in defeating a weak opponent. Take the 1960 Santa Fe tournament (the table above), in which László Szabó and Mark Taimanov both had combined raw scores of $5\frac{1}{2}$. With Gelbfuhs' improved scoring system (now called the Sonneborn–Berger system), Szabó would have

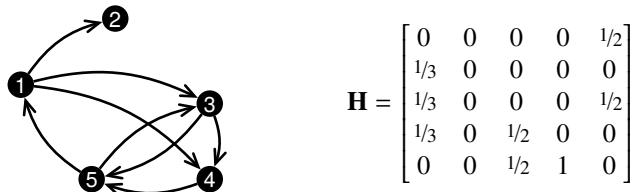
$$1 \cdot 5\frac{1}{2} + \frac{1}{2} \cdot 5 + \frac{1}{2} \cdot 4\frac{1}{2} + \frac{1}{2} \cdot 2\frac{1}{2} + 1 \cdot 2 + 1 \cdot 1\frac{1}{2} + 1 \cdot 1\frac{1}{2} = 16\frac{1}{2}$$

beating Taimanov, who had a score of 16. Mathematically speaking, the Sonneborn–Berger scores are the row sums of \mathbf{M}^2 , i.e., $\mathbf{r} = \mathbf{M}^2\mathbf{1}$. If we consider these scores as an iterative refinement on the raw scores, we don't need to stop after one go. We might consider $\mathbf{r} = \mathbf{M}^k\mathbf{1}$ for some higher power k . But, what k ? In 1895, mathematician Edmund Landau (then seventeen years old) examined the problem and suggested computing $\mathbf{Mr} = \lambda\mathbf{r}$, stating that a player's ranking ought to be proportional sum of the rankings of those players defeated by that player. Of course, this is simply an eigenvector problem.¹ At the time, it still wasn't known if a meaningful eigenvector would always even exist. Ten years later, Perron and Frobenius independently derived their namesake theorem proving the unique existence of a solution.

Centrality is a measure of influence in a network.² It could measure the

¹The March 1902 issue of *British Chess Magazine* praised Landau's formulation as "clear-sighted and precise" but warned "doubtless nevertheless, an accurate estimate of the relative value of games would give extraordinary trouble to a chess player without mathematical training; and such an estimate, even if competently made, would meet with universal distrust."

²The term *prestige* is sometimes used for directed networks, with *measure of influence* for outgoing edges and *measure of support* for incoming edges.

Figure 4.3: A graph and its normalized hyperlink matrix \mathbf{H} .

strength of chess masters or ranking of football teams; it could measure peer status among kindergarteners or trust among colleagues; and it could measure popularity of websites. Eigenvector centrality measures the influence of a node in a network by computing the principal eigenvector of the network's adjacency matrix. It is the basis of Landau's chess ranking system, and it is the basis for PageRank, the first and best-known algorithm used by Google to sort website pages.³

The concept behind the PageRank algorithm arises analogously out of the power method. Imagine that millions of bots (or people) are browsing the internet, each one clicking links at random over and over again. No matter what pages they may have started on, they will eventually limit the probability distribution of all pages they could visit. Pages with more visitors at any given time are arguably more important than those with fewer visitors. This probability is the PageRank.

Suppose that there are n web pages. Let o_j be the total number of outgoing links from a page j . Define the hyperlink matrix \mathbf{H} as the weighted incidence matrix where the ij -element equals $1/o_j$ if there is a hyperlink from page j to page i and zero if there isn't. See the figure above. Pages with no outgoing links are problematic because if one of our bots finds its way to such a page, it will get trapped with no way out. Eventually, over time, all of the bots would become trapped. Instead, we'll make the rule that on web pages with no outgoing links—so-called dangling nodes—we'll choose a random page from any of the n possible webpages. We do this by setting all elements in column j to $1/n$. Now, we have the stochastic matrix $\mathbf{S} = \mathbf{H} + \mathbf{e}\mathbf{v}^\top$ where \mathbf{e} is an $n \times 1$ vector whose components all equal $1/n$ and \mathbf{v} is the $n \times 1$ vector where element $v_j = 1$ if page j is a dangling node and $v_j = 0$ otherwise. Occasionally, someone may decide not to follow any link on the webpage that he or she is currently viewing and instead opens an arbitrary webpage. To model such a behavior, we define a new matrix $\mathbf{G} = \alpha\mathbf{S} + (1 - \alpha)\mathbf{E}$, where the α is the likelihood of following any link on the page and \mathbf{E} is an $n \times n$ matrix whose elements all equal $1/n$. A typical value for the damping factor α is 0.85. The matrix \mathbf{G} is often called the Google matrix.

³Katz centrality is another variant of eigenvector centrality that adds a damping factor, because you might not trust a “friend of a friend” as much as you trust a friend.

A non-negative vector \mathbf{x} is a *probability vector* if its column sum equals one. The i th element of \mathbf{x} is the probability of being on page i , and i th element of \mathbf{Gx} is the probability of being on page i a short time later. Since \mathbf{G} is a stochastic matrix, $\lim_{k \rightarrow \infty} \mathbf{G}^k$ will converge to a steady-state operator. By the Perron–Frobenius theorem, the eigenvector corresponding to dominant eigenvalue $\lambda = 1$ gives the steady-state probability of being on a given webpage. This eigenvector is the PageRank.

Because we only need to find the dominant eigenvector, we can use the power method $\mathbf{x}^{(k+1)} \leftarrow \mathbf{Gx}^{(k)}$. The matrix \mathbf{G} is dense and requires $O(n^2)$ operations for every matrix–vector multiplication, but the matrix \mathbf{H} is quite sparse and only requires $O(n)$ operations. Let's reformulate the problem in terms of \mathbf{H} . Because $\|\mathbf{x}^{(k)}\|_1 = 1$, it follows that $\mathbf{Ex}^{(k)} = \mathbf{e}$. Now we have

$$\mathbf{x}^{(k+1)} \leftarrow \alpha \mathbf{Hx}^{(k)} + \alpha \mathbf{v}^T \mathbf{x}^{(k)} \mathbf{e} + (1 - \alpha) \mathbf{e}. \quad (4.1)$$

The following Julia code computes the PageRank of the graph in Figure 4.3.

```
H = [0 0 0 0 1; 1 0 0 0 0; 1 0 0 0 1; 1 0 1 0 0; 0 0 1 1 0]
v = all(H.==0,dims=1)
H = H ./ (sum(H,dims=1)+v)
n = size(H,1)
d = 0.85
x = ones(n,1)/n
for i in 1:9
    x = d*(H*x) .+ d/n*(v*x) .+ (1-d)/n
end
```

Full-matrix form is fine for smallish matrices like this one, but for larger matrices we should use sparse form. After a few iterations, the solution \mathbf{x} converges to $(0.176, 0.097, 0.227, 0.193, 0.307)$, which we can use to order the nodes as $\{5, 3, 4, 1, 2\}$. If you want to go even deeper, see Vigna [2016] and Langville and Meyer [2004]. ▀

► Inverse and shifted power method

The power method itself only gets us the dominant eigenvector. Getting just this one may be enough for some problems, but often we may need more of them or all of them. We can extend the power method to get other eigenvectors.

Suppose that the spectrum of \mathbf{A} is

$$\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \dots, \lambda_{n-1}, \lambda_n\}$$

where $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_{n-1}| > |\lambda_n|$. Then, the spectrum of \mathbf{A}^{-1} is

$$\lambda(\mathbf{A}^{-1}) = \left\{ \frac{1}{\lambda_1}, \frac{1}{\lambda_2}, \dots, \frac{1}{\lambda_{n-1}}, \frac{1}{\lambda_n} \right\}$$

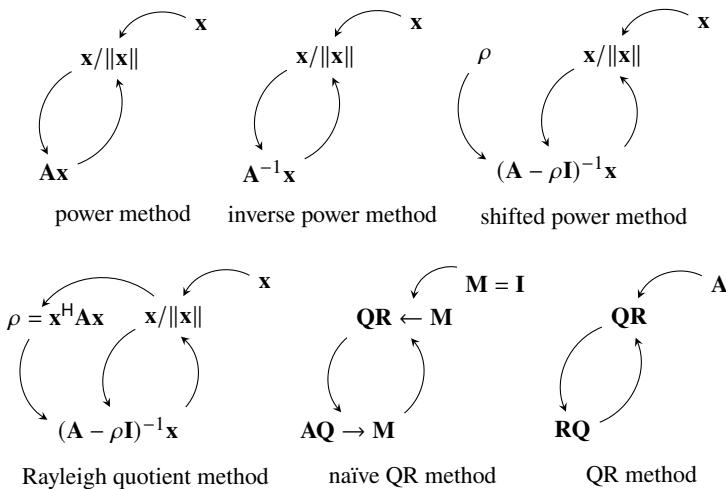


Figure 4.4: Power method and some of its variations.

with the same associated eigenspace. We can get the eigenvector associated with the eigenvalue of \mathbf{A} closest to 0 by applying the power method to \mathbf{A}^{-1} . The convergence ratio is $O(|\lambda_n/\lambda_{n-1}|)$.

So, now we have the largest and the smallest eigenvalues. What about the rest of them? Recall that if $\rho \in \mathbb{C}$, the spectrum of $\mathbf{A} - \rho \mathbf{I}$ is

$$\lambda(\mathbf{A}) = \{\lambda_1 - \rho, \lambda_2 - \rho, \dots, \lambda_{n-1} - \rho, \lambda_n - \rho\}$$

with the same associated eigenspace.

This reveals a clever approach to get the other eigenvectors called *shift-and-invert*. We make a good guess of the value of one of the eigenvalues, say $\rho \approx \lambda_i$ for some i . Now apply power iteration to $(\mathbf{A} - \rho \mathbf{I})^{-1}$. We will recover the eigenvector associated with the eigenvalue λ_i . The ratio of convergence is $|\lambda_i - \rho|/|\lambda_k - \rho|$ where λ_k is the next closest eigenvalue to ρ . In practice, for each iteration, we solve

$$(\mathbf{A} - \rho \mathbf{I}) \mathbf{x}^{(k+1)} = \mathbf{q}^{(k)} \quad \text{with} \quad \mathbf{q}^{(k)} = \frac{\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|}.$$

Computation requires $\frac{2}{3}n^3$ flops for LU-decomposition and then only $2n^2$ flops for each subsequent iteration.

► Rayleigh quotient iteration

The rate of convergence for the shift-and-invert method depends on having a good initial guess for the shift ρ . It would seem that we could improve the method

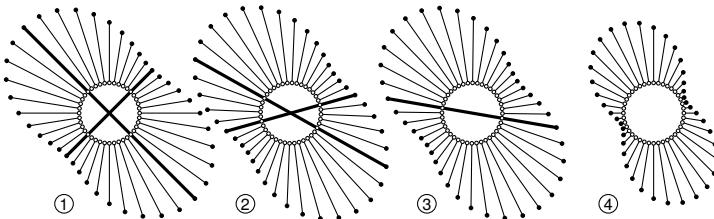


Figure 4.5: Rayleigh quotient $\rho_A(\mathbf{x})\mathbf{x}$ for the matrices in Figure 1.1. Eigenvectors are depicted with thick line segments.

by refining the shift ρ at each iteration. One way we can do this is by using the eigenvalue equation $\mathbf{Ax} = \rho\mathbf{x}$. Unless (ρ, \mathbf{x}) is already an eigenvalue-eigenvector pair of \mathbf{A} , this equation is inconsistent. Instead, we will find the “best” ρ . That is, let’s determine ρ that minimizes the 2-norm of the residual

$$\|\mathbf{r}\|_2 = \|\mathbf{Ax} - \rho\mathbf{x}\|_2.$$

At the minimum $\frac{d}{d\rho} \|\mathbf{r}\|_2^2 = 0$:

$$0 = \frac{d}{d\rho} \|\mathbf{Ax} - \rho\mathbf{x}\|_2^2 = -2\mathbf{x}^H \mathbf{Ax} + 2\rho\mathbf{x}^H \mathbf{x}$$

Therefore, the “best” choice for ρ is

$$\rho_A(\mathbf{x}) = \frac{\mathbf{x}^H \mathbf{Ax}}{\mathbf{x}^H \mathbf{x}}$$

This number, called the *Rayleigh quotient*, can be thought of as an eigenvalue approximation. The Courant–Fischer theorem, also known as the min-max theorem, formalizes this thought.

Theorem 13 (Courant–Fischer theorem). *Let \mathbf{A} be a Hermitian matrix with eigenvalues $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$ and corresponding eigenvectors $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$. Then $\lambda_1 = \max_{\mathbf{x}} \rho_A(\mathbf{x})$, $\lambda_n = \min_{\mathbf{x}} \rho_A(\mathbf{x})$, and in general $\lambda_k = \min_{\mathbf{x} \in S_k} \rho_A(\mathbf{x})$ where $S_k = \text{span}\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$.*

Proof. Take the spectral decomposition $\mathbf{A} = \mathbf{Q}^H \Lambda \mathbf{Q}$ where \mathbf{Q} is an orthogonal matrix. Because $\mathbf{x}^H \mathbf{Ax} = (\mathbf{Qx})^H \Lambda (\mathbf{Qx})$ and $\|\mathbf{Qx}\|_2 = \|\mathbf{x}\|_2$, it is sufficient to consider the case of $\mathbf{A} = \Lambda$, i.e., \mathbf{A} as a diagonal matrix. In this case, $\mathbf{x}^H \mathbf{Ax} = \sum_{i=1}^n \lambda_i x_i^2$ and $S_k = \text{span}\{\xi_1, \xi_2, \dots, \xi_k\}$, the first k standard basis vectors of \mathbb{R}^n . If $\mathbf{x} \in S_k$, then

$$\mathbf{x}^H \mathbf{Ax} = \sum_{i=1}^n \lambda_i x_i^2 = \sum_{i=1}^k \lambda_i x_i^2 \geq \lambda_k \sum_{i=1}^k x_i^2 = \lambda_k \|\mathbf{x}\|_2^2.$$

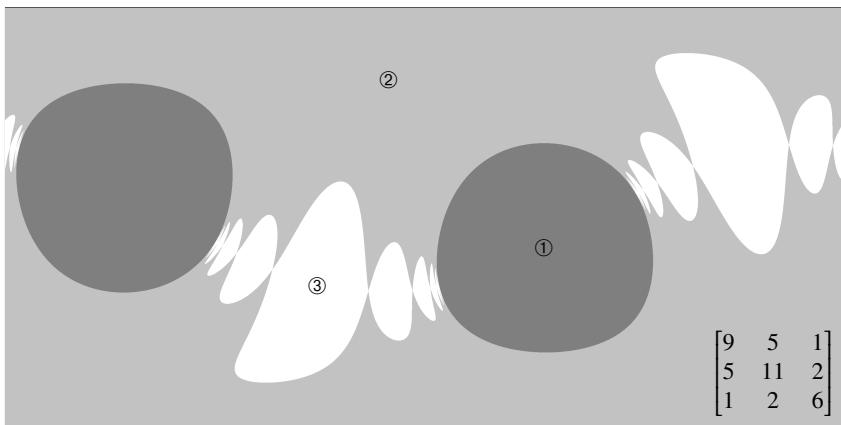


Figure 4.6: Basins of attraction for Rayleigh iteration in the (φ, θ) -plane. Initial guesses in ■ regions converge to the eigenvector at ①, guesses in ▨ regions converge to the eigenvector at ②, and guesses in □ regions converge to the eigenvector at ③. See the front cover and the QR link at the bottom of this page.

And, if \mathbf{x} happens to be the eigenvector ξ_k , then $\mathbf{x}^H \mathbf{A} \mathbf{x} = \lambda_k$. It follows that $\lambda_k = \min_{\mathbf{x} \in S_k} \rho_A(\mathbf{x})$. Finally consider λ_1 .

$$\mathbf{x}^H \mathbf{A} \mathbf{x} = \sum_{i=1}^n \lambda_i x_i^2 \leq \lambda_1 \sum_{i=1}^n x_i^2 = \lambda_1 \|\mathbf{x}\|_2^2$$

and $\rho_A(\mathbf{x}) = \lambda_1$ for $\mathbf{x} = \xi_1$. So, $\lambda_1 = \max_{\mathbf{x}} \rho_A(\mathbf{x})$. □

Note that by the Courant–Fischer theorem $\lambda_{\min} \leq \rho \leq \lambda_{\max}$ for a symmetric matrix. Figure 4.5 on the previous page depicts the mapping $\mathbf{x} \mapsto \rho_A(\mathbf{x})\mathbf{x}$ for the matrices from Figure 1.1 on page 13. The Rayleigh quotient is a best approximation to the eigenvalue and equals eigenvalues along the eigenvectors. While the Rayleigh quotient is bounded by the largest and smallest eigenvalues in symmetric matrices like ①, in general, it may be larger or smaller than either.

By combining Rayleigh quotient approximation with the shift-and-invert method, we get the following method:

Calculate:	$\rho^{(k)} = \frac{\mathbf{x}^{(k)H} \mathbf{A} \mathbf{x}^{(k)}}{\mathbf{x}^{(k)H} \mathbf{x}^{(k)}}$
Solve:	$(\mathbf{A} - \rho^{(k)} \mathbf{I}) \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$
Normalize:	$\mathbf{x}^{(k+1)} \leftarrow \frac{\mathbf{x}^{(k+1)}}{\ \mathbf{x}^{(k+1)}\ _2}$



Rayleigh quotient iteration is locally cubically convergent if \mathbf{A} is Hermitian and quadratically convergent otherwise. See Demmel [1997]. Like many nonlinear iterative methods (such as Newton's method), Rayleigh quotient iteration exhibits basins of attraction and sensitivity to initial conditions. See Figure 4.6 on the preceding page.

4.3 The QR method

The QR method was invented independently in 1960 by Russian mathematician Vera Kublanovskaya and English computer scientist John Francis. It has since then become “firmly established as *the* most important algorithm for eigenvalue problems” and one of the top ten algorithms of the 20th century.⁴ (Watkins [2011]) The QR method is an extension of the power method to get all the eigenvectors all at once rather than one at a time. Recall the idea of the power method. Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ with eigenvalues $|\lambda_1| \geq |\lambda_2| \geq \dots \geq |\lambda_n|$ and associated eigenvectors $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$. Choose an initial vector $\mathbf{x}^{(0)}$. At each iteration,

1. Multiply $\mathbf{x}^{(k)}$ by \mathbf{A} : $\mathbf{x}^{(k+1)} = \mathbf{A}\mathbf{x}^{(k)}$.
2. Normalize to avoid over- or underflow: $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k+1)} / \|\mathbf{x}^{(k+1)}\|$.

Repeat until convergence. This method finds the dominant eigenvector \mathbf{v}_1

Now, suppose we start with n initial vectors and want to get all n eigenvectors. Let's develop a method so that the first vector sequence converges to \mathbf{v}_1 , the second vector sequence converges to \mathbf{v}_2 , and so forth. Of course, we can work with all the vectors at once by treating them as columns of a matrix. It shouldn't matter what set we start with as long it spans \mathbb{R}^n . So, let's take the standard unit vectors $\{\mathbf{x}_1^{(0)}, \mathbf{x}_2^{(0)}, \dots, \mathbf{x}_n^{(0)}\} = \mathbf{I} = \{\xi_1, \xi_2, \dots, \xi_n\}$ as our initial set of vectors.

It should be clear that if we apply the power method directly to \mathbf{I} , all the columns will converge to the dominant eigenvector unless one happens to already be an eigenvector. Let's fix this problem and get all the eigenvectors. We won't do anything to the first column $\mathbf{x}_1^{(k+1)}$ except normalize it to $\mathbf{q}_1^{(k+1)}$. It converges to the dominant eigenvector \mathbf{v}_1 , and within several iterations it should give us a good approximation to \mathbf{v}_1 . Let $\mathbf{q}_2^{(k+1)}$ be the normalized, orthogonal projection of $\mathbf{A}\mathbf{q}_2^{(k)}$ into $\text{span}\{\mathbf{q}_1^{(k+1)}\}^\perp$. Since $\mathbf{q}_1^{(k+1)}$ is close to \mathbf{v}_1 , this projection will kill off much of the component in the \mathbf{v}_1 direction. This means that the change from $\mathbf{q}_2^{(k)}$ to $\mathbf{q}_2^{(k+1)}$ is now dominated by \mathbf{v}_2 . Let $\mathbf{q}_3^{(k+1)}$ be the orthogonal projection of $\mathbf{q}_3^{(k)}$ into $\text{span}\{\mathbf{q}_1^{(k+1)}, \mathbf{q}_2^{(k+1)}\}^\perp$. Since $\mathbf{q}_1^{(k+1)}$ is close to \mathbf{v}_1 and $\mathbf{q}_2^{(k+1)}$ is close

⁴Remarkably, for 45 years, John Francis was entirely unaware of the impact of his contribution. Shortly after publishing his results, he left the field and never looked back. It wasn't until 2007 that Francis was tracked down and learned of his achievements. By then, he was retired and living in a seaside resort on the south east coast of England, sailing and working on a degree in mathematics. While Francis had briefly attended the University of Cambridge in the 1950s, he never completed a degree. He subsequently received an honorary doctorate from the University of Sussex in 2015.

to \mathbf{v}_2 , this projection will ensure that the change from $\mathbf{q}_3^{(k)}$ to $\mathbf{q}_3^{(k+1)}$ is now dominated by \mathbf{v}_3 . And so forth.

Our method is now as follows. Take $\mathbf{Q}_0 = \mathbf{I}$. For each iteration

1. Multiply by \mathbf{A} : $\mathbf{M}_{k+1} = \mathbf{AQ}_k$
2. Take the QR-decomposition of the resulting matrix: $\mathbf{Q}_{k+1}\mathbf{R}_{k+1} = \mathbf{M}_{k+1}$

We can write the two-part iteration as

$$\mathbf{Q}_{k+1}\mathbf{R}_{k+1} = \mathbf{M}_{k+1} = \mathbf{AQ}_k. \quad (4.2)$$

It's not yet clear what this equation means. Let's change the basis to \mathbf{Q}_k . To do so, we simply need to "rotate" by $\mathbf{Q}_k^{-1} = \mathbf{Q}_k^\top$. Multiplying (4.2) by \mathbf{Q}_k^\top gives us

$$\mathbf{Q}_k^\top \mathbf{Q}_{k+1} \mathbf{R}_{k+1} = \mathbf{Q}_k^\top \mathbf{M}_{k+1} = \mathbf{Q}_k^\top \mathbf{AQ}_k. \quad (4.3)$$

This says that $\mathbf{Q}_k^\top \mathbf{Q}_{k+1} \mathbf{R}_{k+1}$ is unitarily similar to \mathbf{A} . So, $\mathbf{Q}_k^\top \mathbf{Q}_{k+1} \mathbf{R}_{k+1}$ has the same eigenvalues as \mathbf{A} . If the iteration converges, then $\mathbf{Q}_k \rightarrow \mathbf{Q}_{k+1}$ and $\mathbf{Q}_k^\top \mathbf{Q}_{k+1} \rightarrow \mathbf{I}$, and we are left with \mathbf{R}_{k+1} on the left-hand side. The matrix \mathbf{R}_{k+1} is an upper triangular matrix with the eigenvalues along the diagonal, giving us the Schur decomposition of \mathbf{A} .

We are not done yet. Define $\hat{\mathbf{Q}}_1$ to be the change from \mathbf{Q}_k to \mathbf{Q}_{k+1} . That is, let $\mathbf{Q}_{k+1} = \hat{\mathbf{Q}}_{k+1} \mathbf{Q}_k$. Then

$$\mathbf{Q}_{k+1} = \hat{\mathbf{Q}}_{k+1} \hat{\mathbf{Q}}_k \cdots \hat{\mathbf{Q}}_2 \hat{\mathbf{Q}}_1$$

and $\hat{\mathbf{Q}}_1 = \mathbf{Q}_0 = \mathbf{I}$. And (4.3) is equivalent to

$$\hat{\mathbf{Q}}_{k+1} \mathbf{R}_{k+1} = \hat{\mathbf{M}}_{k+1} = \mathbf{Q}_k^\top \mathbf{AQ}_k$$

where $\hat{\mathbf{M}}_{k+1} = \mathbf{Q}_k^\top \mathbf{M}_{k+1}$. From (4.3) we have $\mathbf{R}_{k+1} \mathbf{Q}_k^\top = \mathbf{Q}_{k+1}^\top \mathbf{A}$, and hence $\mathbf{R}_k \mathbf{Q}_{k-1}^\top = \mathbf{Q}_k^\top \mathbf{A}$. So,

$$\hat{\mathbf{Q}}_{k+1} \mathbf{R}_{k+1} = \hat{\mathbf{M}}_{k+1} = \mathbf{R}_k \mathbf{Q}_{k-1}^\top \mathbf{Q}_k = \mathbf{R}_k \hat{\mathbf{Q}}_k$$

We are left with the simple implementation. Starting with with $\mathbf{T}_0 = \mathbf{A}$, for each iteration,

$$\begin{cases} \text{factor} & \mathbf{T}_k \rightarrow \mathbf{Q}_k \mathbf{R}_k \\ \text{form} & \mathbf{T}_{k+1} \leftarrow \mathbf{R}_k \mathbf{Q}_k \end{cases}$$

The QR algorithm provides us with the Schur decomposition of a matrix

$$\mathbf{R} = \mathbf{U}^H \mathbf{AU}$$

where \mathbf{U} is unitary. At each iteration

$$\mathbf{T}_{k+1} = \mathbf{R}_k \mathbf{Q}_k = \mathbf{Q}_k^H \mathbf{Q}_k \mathbf{R}_k \mathbf{Q}_k = \mathbf{Q}_k^H \mathbf{T}_k \mathbf{Q}_k.$$

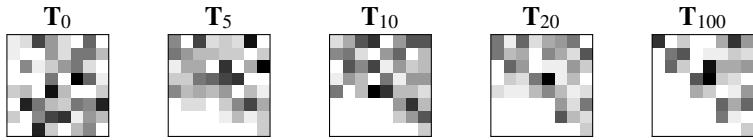
If we continue and unravel the terms all the of the terms,

$$\mathbf{T}_{k+1} = (\mathbf{Q}_0 \mathbf{Q}_1 \cdots \mathbf{Q}_k)^H \mathbf{T}_0 (\mathbf{Q}_0 \mathbf{Q}_1 \cdots \mathbf{Q}_k).$$

Because $(\mathbf{Q}_0 \mathbf{Q}_1 \cdots \mathbf{Q}_k)$ is unitary,

$$\lambda(\mathbf{T}_{k+1}) = \lambda(\mathbf{T}_k) = \lambda(\mathbf{T}_{k-1}) = \cdots = \lambda(\mathbf{T}_0) = \lambda(\mathbf{A}).$$

\mathbf{T}_k converges to an upper triangular matrix, and the diagonal of \mathbf{T}_k gives the eigenvalues of \mathbf{A} . The eigenvalues don't change, the eigenvectors just rotate. The following figure shows the QR method applied to a matrix at iterations \mathbf{T}_0 , \mathbf{T}_5 , \mathbf{T}_{10} , \mathbf{T}_{20} , and \mathbf{T}_{100} :



► Convergence of the QR method

Because the QR method uses the power method, convergence is generally slow. The diagonal elements of \mathbf{R}_k converge to the eigenvalues linearly. Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ have eigenvalues such that $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$. Then

$$\lim_{k \rightarrow \infty} \mathbf{R}_k = \begin{bmatrix} \lambda_1 & r_{12} & \cdots & r_{1n} \\ 0 & \lambda_2 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

with convergence

$$|r_{i,i-1}^{(k)}| = O\left(\left|\frac{\lambda_i}{\lambda_{i-1}}\right|^k\right).$$

Each QR decomposition takes $\frac{4}{3}n^3$ operations and each matrix multiplication takes $\frac{1}{2}n^3$ operations. We need to repeat it $O(n)$ times to get the n eigenvalues. Hence, the net cost of the QR method is $O(n^4)$. We can improve the QR algorithm by doing simple things: reducing the matrix to upper Hessenberg form, shifting the matrix in each iteration to speed up convergence, and deflating.

► The QR algorithm with upper Hessenberg matrices

Because of Abel's theorem, it is impossible in general to determine the Schur decomposition of a matrix \mathbf{A} , i.e., an upper triangular matrix which is unitarily

similar to a matrix \mathbf{A} . But we can get an upper Hessenberg matrix—close to being an upper triangular—that is unitarily similar to \mathbf{A} . Why would we want to do this?

For nonsingular matrices, upper Hessenberg form is preserved by the QR algorithm. Note that if $\mathbf{Q}_n \mathbf{R}_n = \mathbf{H}_n$ then $\mathbf{Q}_n = \mathbf{H}_n \mathbf{R}_n^{-1}$. The inverse of an upper triangular matrix is an upper triangular matrix, and the product of an upper Hessenberg and a triangular matrix is upper Hessenberg. So, \mathbf{Q}_n is upper Hessenberg. Furthermore, $\mathbf{H}_{n+1} = \mathbf{R}_n \mathbf{Q}_n$ is upper Hessenberg. Reducing a matrix to upper Hessenberg form takes $O(n^3)$ operations. But we only need to do this once.

After putting the matrix in upper Hessenberg form, we will need to do $n - 1$ Givens rotations to zero out the subdiagonal elements to get the QR decomposition. So a QR step applied to an upper Hessenberg matrix requires at most $O(n^2)$ operations. For a Hermitian matrix, each QR step takes $O(n)$ operations because we are working on a tridiagonal matrix.

Let's outline how to get an upper Hessenberg matrix. We can do this using Householder reflectors in a manner similar to QR decomposition. We first construct a Householder reflector \mathbf{Q}_1 to zero out the elements below the first subdiagonal element. To do this,

$$\text{partition } \mathbf{A} \text{ as } \begin{bmatrix} a_{11} & \mathbf{c}^\top \\ \mathbf{b} & \hat{\mathbf{A}} \end{bmatrix} \text{ and let } \mathbf{Q}_i \text{ equal } \begin{bmatrix} \mathbf{I} & \mathbf{0}^\top \\ \mathbf{0} & \hat{\mathbf{Q}}_i \end{bmatrix}$$

where $\hat{\mathbf{Q}}_i$ is the reflector that maps a vector \mathbf{b} to $[-\|\mathbf{b}\|_2 \ 0 \ \cdots \ 0]^\top$. Because $\hat{\mathbf{Q}}_i$ is a Householder reflector, $\hat{\mathbf{Q}}_i = \hat{\mathbf{Q}}_i^{-1} = \hat{\mathbf{Q}}_i^\top$. Note that \mathbf{Q}_i is a block matrix with the identity matrix in the upper-left block. So, left-multiplying a matrix by \mathbf{Q}_1 does not change the first row and right-multiplying a matrix by \mathbf{Q}_1 does not change the first column. Pictorially, $\mathbf{A}_1 = \mathbf{Q}_1 \mathbf{A} \mathbf{Q}_1^\top$ is

$$\begin{bmatrix} 1 & & \\ & \ddots & \\ & & \hat{\mathbf{Q}}_1 \end{bmatrix} \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} 1 & & \\ & \ddots & \\ & & \hat{\mathbf{Q}}_1 \end{bmatrix} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} 1 & & \\ & \ddots & \\ & & \hat{\mathbf{Q}}_1 \end{bmatrix} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}.$$

We construct a similar Householder reflector \mathbf{Q}_2 to zero out the elements below the second subdiagonal. Pictorially, $\mathbf{A}_2 = \mathbf{Q}_2 \mathbf{A}_1 \mathbf{Q}_2^\top$ is

$$\begin{bmatrix} 1 & 1 & & \\ & \ddots & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \hat{\mathbf{Q}}_2 \end{bmatrix} \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ & \ddots & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \hat{\mathbf{Q}}_2 \end{bmatrix} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ & \ddots & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \hat{\mathbf{Q}}_2 \end{bmatrix} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}.$$

Continuing in this manner, we will form an upper Hessenberg matrix that is unitarily similar to \mathbf{A} . Since it is similar, it has the same eigenvalues. If \mathbf{A} is a Hermitian matrix, then the upper Hessenberg matrix is a tridiagonal matrix.

• The `LinearAlgebra.jl` function `hessenberg(A)` computes the unitarily similar upper Hessenberg form of a matrix A and returns a Hessenberg object consisting of a unitary matrix and a Hessenberg matrix. Either can be converted to a regular matrix object using the `Matrix` method.

► Shifted QR iteration

Recall that if the eigenvalues of A are $\{\lambda_1, \lambda_2, \dots, \lambda_n\}$, then the eigenvalues of $A - \rho I$ are $\{\lambda_1 - \rho, \lambda_2 - \rho, \dots, \lambda_n - \rho\}$. The diagonal elements of A_k converge to the eigenvalues linearly with the values of the subdiagonals given by

$$|a_{i,i-1}^{(k)}| = O\left(\left|\frac{\lambda_i - \rho}{\lambda_{i-1} - \rho}\right|^k\right).$$

We can speed up convergence by applying QR iteration to $A - \rho I$ and choosing ρ to be close to an eigenvalue. This is the same idea that we used in picking the shift in the Rayleigh iteration. Since the diagonal elements of A_k converge to the eigenvalues of A , they are the natural choices for ρ . Once the corresponding diagonal element has converged to an eigenvalue, we move to another one.

Start with the lower right corner $i = n$. Recall that this diagonal element will converge to the smallest eigenvalue. At each iteration:

1. Set $\rho = a_{ii}^{(k)}$.
2. Factor $(A_k - \rho I) \rightarrow Q_{k+1} R_{k+1}$
3. Restore $A_{k+1} \leftarrow (R_{k+1} Q_{k+1} + \rho I)$

Once $a_{ii}^{(k)}$ converges to an eigenvalue, move to the previous diagonal element ($i \rightarrow i - 1$) and repeat the process. Note that

$$A_{k+1} = R_{k+1} Q_{k+1} + \rho I = Q_{k+1}^T (A_k - \rho I) Q_{k+1} + \rho I = Q_{k+1}^T A_k Q_{k+1}.$$

So, the new A_{k+1} has the same eigenvalues as the old A_k .

The convergence with each shift is $O(|\lambda_i - \rho|/|\lambda_{i-1} - \rho|)$ where $\rho \approx \lambda_i$. Overall, we get quadratic convergence. For Hermitian matrices, where the eigenvectors are orthogonal, we get cubic convergence. This type of shifting is called *Rayleigh quotient shifting*. See the figure on the following page or the QR link below, which shows the convergence of the QR method and convergence of the shifted QR method to eigenvalues of a 6×6 complex matrix. The regular QR method takes 344 iterations to converge to an error of 10^{-3} while the shifted QR method requires only 16 iterations. Notice that convergence is slower for eigenvalues whose magnitudes are close.

Occasionally, Rayleigh quotient shifting will fail to converge. For example, using $\rho = 2$ on the matrix

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix},$$



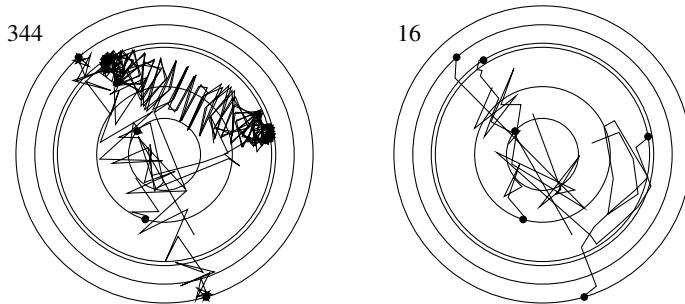


Figure 4.7: Convergence of the QR method (left) and the shifted QR method (right) to eigenvalues of a 6×6 complex matrix. The eigenvectors are depicted by \bullet and line segments show the paths of convergence.

which has eigenvalues $\lambda = 1$ and 3 , results in an orthogonal matrix

$$\mathbf{A} - \rho \mathbf{I} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

which has eigenvalues ± 1 , each of equal magnitude. Each eigenvector pulls equally, and the algorithm cannot decide to which eigenvalue to go to. One fix to this problem is to use the *Wilkinson shift*. The Wilkinson shift defines ρ to be the eigenvalue of the submatrix

$$\begin{bmatrix} a_{i-1,i-1}^{(k-1)} & a_{i-1,i}^{(k-1)} \\ a_{i,i-1}^{(k-1)} & a_{i,i}^{(k-1)} \end{bmatrix}$$

that is closest to $a_{i,i}^{(k-1)}$. The eigenvalues of a 2×2 matrix are easily computed by using the quadratic formula to solve the characteristic equation.

► Deflation

QR decomposition requires $O(n^3)$ operations and computing the product \mathbf{RQ} takes another $O(n)$ operations. We can speed up the method by using *deflation*. Start with the shift $\rho = a_{i,i}$. When the magnitude subdiagonal element $a_{i,i-1}$ is small, the diagonal element $a_{i,i}$ is a close approximation of the eigenvalue λ_i . At this point set the shift ρ to equal $a_{i-1,i-1}$ and recover λ_i . To reduce the number of computations at the next step, we consider the $(i-1) \times (i-1)$ principal submatrix obtained by removing the last row and last column. We continue QR decomposition on this smaller matrix. And continue like this successively with smaller and smaller matrices. Complex eigenvalues of real matrices are extracted in pairs.

4.4 Implicit QR

At each step of the shifted QR method, we compute

$$(\mathbf{A} - \rho\mathbf{I}) = \mathbf{QR}, \quad \hat{\mathbf{A}} = \mathbf{RQ} + \rho\mathbf{I} \quad (4.4)$$

where \mathbf{A} is in proper upper Hessenberg form. This gives us the similarity transform

$$\hat{\mathbf{A}} = \mathbf{Q}^T \mathbf{A} \mathbf{Q}. \quad (4.5)$$

In practice, we don't need (or necessarily want) to compute the QR decomposition (4.4) explicitly. When the shift ρ is close to an eigenvalue (which is what we want it to be), the QR method is sensitive to round-off and hence may be unstable. The solution is to never explicitly subtract off $\rho\mathbf{I}$ and consequently to never compute \mathbf{R} . Instead, we can do this implicitly. But the trick now is determining the sequence of Givens rotations or Householder reflections $\mathbf{Q}_1 \mathbf{Q}_2 \cdots \mathbf{Q}_n$ that gives us \mathbf{Q} . As it turns out, such an implicit method is no more difficult than the explicit method.

Consider the upper Hessenberg matrix $\mathbf{A} - \rho\mathbf{I}$. We left-multiply by a Givens rotation \mathbf{Q}_1^T to zero out element a_{21} ,

$$\begin{bmatrix} a_{11}-\rho & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22}-\rho & a_{23} & a_{24} & a_{25} \\ a_{32} & a_{33}-\rho & a_{34} & a_{35} & \\ a_{43} & a_{44}-\rho & a_{45} & & \\ a_{44} & a_{55}-\rho & & & \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * & * \\ 0 & x & * & * & * \\ a_{32} & a_{33}-\rho & a_{34} & a_{35} & \\ a_{43} & a_{44}-\rho & a_{45} & & \\ a_{44} & a_{55}-\rho & & & \end{bmatrix}$$

The values in the first and second rows are changed. Notably, element (2, 1) is changed from a_{21} to 0 and element (2, 2) is changed from $a_{22} - \rho$ to some value x . Suppose that we were to apply the same Givens rotation \mathbf{Q}_1^T to our original matrix \mathbf{A} .

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{32} & a_{33} & a_{34} & a_{35} & \\ a_{43} & a_{44} & a_{45} & & \\ a_{44} & a_{55} & & & \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * & * \\ \rho s & x + \rho c & * & * & * \\ a_{32} & a_{33} & a_{34} & a_{35} & \\ a_{43} & a_{44} & a_{45} & & \\ a_{44} & a_{55} & & & \end{bmatrix}$$

This time element (2, 1) is changed from a_{21} to ρs and element (2, 2) is changed from a_{22} to $x + \rho c$, where s and c denote $\sin \theta$ and $\cos \theta$ of the Givens rotations. Now, right-multiply by \mathbf{Q}_1 to complete the similarity transform

$$\rightarrow \begin{bmatrix} * & * & * & * & * \\ sx & cx & * & * & * \\ sa_{32} & ca_{32} & a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} & & \\ a_{44} & a_{55} & & & \end{bmatrix}$$

In the next step we need a Givens rotation \mathbf{Q}_2^T to zero out a_{32} . Exactly the same \mathbf{Q}_2^T works for both \mathbf{A} and $\mathbf{A} - \rho\mathbf{I}$. The same is true for \mathbf{Q}_3^T and so on.

So, all we need to do is start the implicit QR step using the same column \mathbf{q} as we would have used to start the explicit QR method and then continue by reducing the matrix into upper Hessenberg form. The resulting upper Hessenberg matrix using the implicit method will equal the upper Hessenberg matrix using the explicit method. This idea is closely related to the Implicit Q theorem.

Theorem 14 (Implicit Q theorem). *Let $\mathbf{H} = \mathbf{Q}^T \mathbf{A} \mathbf{Q}$ be the reduction of a matrix \mathbf{A} to Hessenberg form and the elements in the lower diagonal of \mathbf{H} are non-zero. Then \mathbf{Q} and \mathbf{H} are uniquely determined by the first column of \mathbf{Q} up to a sign.*

Proof. Consider the following pictorial representation of nonzero elements:

$$\mathbf{A} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}, \quad \text{and} \quad \mathbf{Q} = \begin{bmatrix} \circ & \circ & \circ & \circ \\ \bullet & \circ & \circ & \circ \\ \bullet & \bullet & \circ & \circ \\ \bullet & \bullet & \bullet & \circ \end{bmatrix}.$$

If \mathbf{A} is an $n \times n$ matrix, then \mathbf{A} has n^2 degrees of freedom \bullet . Additionally, \mathbf{H} has $(n-2)(n-1)/2$ explicitly zero elements, so it has $n^2 - (n-2)(n-1)/2$ degrees of freedom. A unitary matrix \mathbf{Q} has $n(n-1)/2$ degrees of freedom. To see this, imagine that we build \mathbf{Q} , starting with the leftmost column. The first column of \mathbf{Q} can be any vector, as long as it has a length of one. So, it has $n-1$ degrees of freedom. The second column can be any vector, as long as it has a length of one and is orthogonal to the first one. So, it has $n-2$ degrees of freedom. The third column gives another $n-3$ degrees of freedom. And so forth, giving $n(n-1)/2$ degrees of freedom. Between \mathbf{H} and \mathbf{Q} , we have $n^2 + n - 1$ degrees of freedom, which is $n-1$ too many. The missing $n-1$ degrees of freedom are used to determine the first column of \mathbf{Q} . \square

The algorithm for *one* implicit QR iteration can be summarized as:

1. Let $\tilde{\mathbf{H}} = \mathbf{H} - \rho\mathbf{I}$.
2. Determine a Householder or Givens transformation \mathbf{Q}^T that zeros out the $(2, 1)$ element of $\tilde{\mathbf{H}}$.
3. Compute $\tilde{\mathbf{H}} \leftarrow \mathbf{Q}^T \tilde{\mathbf{H}} \mathbf{Q}$.
4. Now, continue with use Householder or Givens transformations to reduce the new $\tilde{\mathbf{H}}$ into upper Hessenberg form by “chasing the bulge”:
 - a) For $i = 1, 2, \dots, n-2$ determine the Householder or Givens transformation \mathbf{Q}_i that zeros out element $(i+2, i)$ using element $(i+1, i)$.
 - b) Compute $\mathbf{A} \leftarrow \mathbf{Q}^T \mathbf{H} \mathbf{Q}$.

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{H} \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{Q}_1^T \mathbf{H} \mathbf{Q}_1 \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{Q}_2^T \cdots \mathbf{Q}_2 \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{Q}_3^T \cdots \mathbf{Q}_3 \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{Q}_4^T \cdots \mathbf{Q}_4 \end{bmatrix}$$

By this point, the QR method bears little resemblance to the underlying power method. But just as we could speed up convergence of the power method by taking multiple steps ($\mathbf{A}^2\mathbf{x}$ versus \mathbf{Ax} for example), we can speed up the QR method using multiple steps. Namely, we can take $(\mathbf{H} - \rho_1\mathbf{I})(\mathbf{H} - \rho_2\mathbf{I}) \cdots (\mathbf{H} - \rho_k\mathbf{I})$ in one QR step. In practice, k is typically 1, 2, 4, or 6. Note that the bulge that we need to chase is size k .

Taking multiple steps is especially appealing when working with real matrices that have complex eigenvalues because we can avoid complex arithmetic (cutting computation by half). We can formulate a double-step QR method by taking $\rho_1 = \rho$ and $\rho_2 = \bar{\rho}$. In this case, $\tilde{\mathbf{A}} = \mathbf{H}^2 - 2\operatorname{Re}(\rho)\mathbf{H} + |\rho|^2\mathbf{I}$.

• The `LinearAlgebra.jl` function `eigen` returns an `Eigen` object containing eigenvalues and eigenvectors. The functions `eigvecs` returns a matrix of eigenvectors and `eigvals` returns an array of eigenvalues.

4.5 Getting the eigenvectors

We still have not explicitly computed the eigenvectors using the QR method. While we could keep track of the rotator or reflector matrices \mathbf{Q}_i as they accumulate, such an approach is inefficient. Instead, computing the eigenvectors after we have found the eigenvalues is typically a simple task. In this section, we will look at two methods: using a shifted power method and using the Schur form.

In the shifted power method, we start again from the upper Hessenberg matrix form of the matrix $\mathbf{H} = \mathbf{Q}^T \mathbf{A} \mathbf{Q}$. Let ρ be our approximation to an eigenvalue. Then using the shifted power method

$$(\mathbf{H} - \rho\mathbf{I})\mathbf{z}^{(k+1)} = \mathbf{q}^{(k)} \quad \text{with} \quad \mathbf{q}^{(k)} = \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|}$$

will return an approximation to the associated eigenvector \mathbf{q} of \mathbf{H} . When ρ is a good approximation, only one iteration is typically needed. The eigenvector to \mathbf{A} is then $\mathbf{Q}^T \mathbf{q}$.

To get the eigenvectors using the Schur form $\mathbf{Q}^H \mathbf{A} \mathbf{Q} = \mathbf{T}$ without accumulating the matrix \mathbf{Q} , we can use an *ultimate shift* strategy. First, we use the QR method to find only the eigenvalues. Once we have the approximate eigenvalues, we rerun the QR method using these eigenvalues as shifts and accumulate the matrix \mathbf{Q} , using two steps per shift to ensure convergence. If the matrix is symmetric, then \mathbf{T} is diagonal and \mathbf{Q} gives us the eigenvectors.

Let's look at the nonsymmetric case. Suppose that λ is the k th eigenvalue of the upper triangular matrix

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_{11} & \mathbf{v} & \mathbf{T}_{13} \\ \mathbf{0} & \lambda & \mathbf{w}^T \\ \mathbf{0} & \mathbf{0} & \mathbf{T}_{33} \end{bmatrix},$$

where \mathbf{T}_{11} is a $(k-1) \times (k-1)$ upper triangular matrix and \mathbf{T}_{33} is an $(n-k) \times (n-k)$ upper triangular matrix. Furthermore, suppose that λ is a simple eigenvalue so that $\lambda \notin \lambda(\mathbf{T}_{11}) \cup \lambda(\mathbf{T}_{33})$. Then eigenvector problem $(\mathbf{T} - \lambda \mathbf{I})\mathbf{y} = \mathbf{0}$ can be written as

$$\begin{aligned} (\mathbf{T}_{11} - \lambda \mathbf{I}_{k-1})\mathbf{y}_{k-1} + \mathbf{v}\mathbf{y} + \mathbf{T}_{13}\mathbf{y}_{n-k} &= \mathbf{0} \\ \mathbf{w}^T \mathbf{y}_{n-k} &= 0, \\ (\mathbf{T}_{33} - \lambda \mathbf{I}_{n-k})\mathbf{y}_{n-k} &= \mathbf{0} \end{aligned}$$

where $\mathbf{y} = [\mathbf{y}_{k-1} \quad \mathbf{y} \quad \mathbf{y}_{n-k}]^T$. Because λ is simple, $\mathbf{T}_{11} - \lambda \mathbf{I}_{k-1}$ and $\mathbf{T}_{33} - \lambda \mathbf{I}_{n-k}$ are both nonsingular. So, $\mathbf{y}_{n-k} = \mathbf{0}$ and $\mathbf{y}_{k-1} = \mathbf{y}(\mathbf{T}_{11} - \lambda \mathbf{I}_{k-1})^{-1}\mathbf{v}$. Since \mathbf{y} is arbitrary, we will set it to one. Then

$$\mathbf{y} = \begin{bmatrix} (\mathbf{T}_{11} - \lambda \mathbf{I}_{k-1})^{-1}\mathbf{v} \\ 1 \\ \mathbf{0} \end{bmatrix}$$

which can be evaluated using backward substitution. The eigenvector for \mathbf{A} is then $\mathbf{x} = \mathbf{Q}\mathbf{y}$.

4.6 Arnoldi method

What if we only want a few of the largest eigenvalues of a large, sparse $n \times n$ matrix? The QR method is inefficient when n is large—much larger than say 1000—especially if we only need a few eigenvalues, because it operates on all n dimensions and it fills in the sparse structure. We can do better by restricting ourselves to a low-dimensional subspace. By using a subspace that is one-tenth the size of the original subspace, we'll get an algorithm that is almost one thousand times faster. Of course, we would need to find a subspace that includes the eigenvectors we want. If we are happy with approximate eigenvalues, we can use the Rayleigh–Ritz method.

The *Rayleigh–Ritz method* computes approximations $(\tilde{\lambda}_i, \tilde{\mathbf{x}}_i)$ of the eigenvalue–eigenvector pair $(\lambda_i, \mathbf{x}_i)$ of \mathbf{A} , called Ritz pairs, by solving the eigenvalue problem in a space that approximates an eigenspace of \mathbf{A} . Let \mathbf{A} be an $n \times n$ matrix and let \mathbf{V} be an $n \times m$ matrix whose columns are an orthonormal basis that approximates an m -dimensional eigensubspace of \mathbf{A} . The Rayleigh–Ritz method goes as follows: take the projection of \mathbf{A} into the column space of \mathbf{V} , solve

the eigenvalue problem in this space, and finally express the solutions in the canonical basis. That is, take $\mathbf{R} = \mathbf{V}^T \mathbf{A} \mathbf{V}$, solve $\mathbf{R}\mathbf{v}_i = \tilde{\lambda}_i \mathbf{v}_i$, finally calculate $\tilde{\mathbf{x}}_i = \mathbf{V}\mathbf{v}_i$. We still need a way to find a subspace V that is close to an eigenspace of \mathbf{A} . An effective way to do this is by constructing a Krylov subspace. A Rayleigh–Ritz method that uses such a subspace is called an Arnoldi method. Arnoldi method combines the power method and Gram–Schmidt to get the first m eigenvectors approximations.

The power method in section 4.2 preserves the sparsity of the matrix, but we only get one eigenvalue at a time because we throw away information about the other eigenvectors by keeping only the most recent vector. What do we get if we were to keep a history of the vectors? Take an initial guess \mathbf{q} , which for a semisimple matrix can be expressed in terms of an eigenvector basis $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$:

$$\begin{aligned}\mathbf{q} &= c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \cdots + c_n \mathbf{v}_n \\ \mathbf{A}\mathbf{q} &= c_1 \lambda_1 \mathbf{v}_1 + c_2 \lambda_2 \mathbf{v}_2 + \cdots + c_n \lambda_n \mathbf{v}_n \\ \mathbf{A}^2 \mathbf{q} &= c_1 \lambda_1^2 \mathbf{v}_1 + c_2 \lambda_2^2 \mathbf{v}_2 + \cdots + c_n \lambda_n^2 \mathbf{v}_n \\ &\vdots \\ \mathbf{A}^{k-1} \mathbf{q} &= c_1 \lambda_1^{k-1} \mathbf{v}_1 + c_2 \lambda_2^{k-1} \mathbf{v}_2 + \cdots + c_n \lambda_n^{k-1} \mathbf{v}_n\end{aligned}$$

For $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$, we see that (or at least hope that) the subspace

$$\text{span}\{\mathbf{q}, \mathbf{A}\mathbf{q}, \mathbf{A}^2\mathbf{q}, \dots, \mathbf{A}^{k-1}\mathbf{q}\}$$

is close to the subspace spanned by the first m eigenvectors

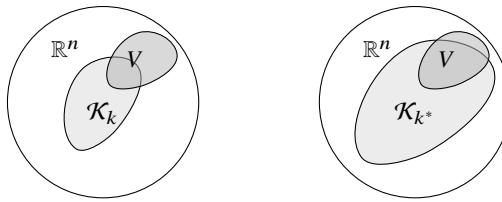
$$\text{span}\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \dots, \mathbf{v}_k\}.$$

The *Krylov subspace* $\mathcal{K}_k(\mathbf{A}, \mathbf{q})$ is the subspace spanned by

$$\mathbf{q}, \mathbf{A}\mathbf{q}, \mathbf{A}^2\mathbf{q}, \dots, \mathbf{A}^{k-1}\mathbf{q}.$$

These vectors are the columns of the Krylov matrix \mathbf{K}_k . The Krylov subspace we are using is much smaller than the original n -dimensional subspace, so it is unlikely that it will contain our desired eigenvectors. Instead of finding the true eigenvector-eigenvalue pairs in all of \mathbb{R}^n , we will instead find approximate eigenvector-eigenvalue pairs in the Krylov subspace $\mathcal{K}_k(\mathbf{A}, \mathbf{q})$.

The m -dimensional Krylov subspace $\mathcal{K}_k \subset \mathbb{R}^n$ does not contain the eigenspace V —only its projection into \mathcal{K}_k . If \mathbf{q} happens to be already an eigenvector of \mathbf{A} , then the Krylov subspace will only consist of the space spanned by \mathbf{q} . This is typically not an issue, because if \mathbf{q} is chosen at random, it will likely contain components of all eigenvectors of \mathbf{A} . By taking a larger Krylov subspace \mathcal{K}_{k^*} with $k^* > k$, we can get a better approximation at the cost of more computing time:



The Krylov subspace approximation introduces another problem. Because $\mathbf{A}^k \mathbf{q}$ converges in direction to the dominant eigenvector, the last several columns of \mathbf{K}_k point in nearly the same direction. So, \mathbf{K}_k is ill-conditioned. To fix the ill-conditioning of \mathbf{K}_k , we need to replace the vectors $\mathbf{q}, \mathbf{A}\mathbf{q}, \dots, \mathbf{A}^{m-1}\mathbf{q}$ with the orthonormal set $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_k$. The process of generating the orthonormal basis for $\mathcal{K}_k(\mathbf{A}, \mathbf{q})$, called the *Arnoldi method*. The Arnoldi method is similar to the Gram–Schmidt process, except we don't need to start with a set of m basis vectors already in hand. Instead, we only need \mathbf{q} .

First, normalize the first vector $\mathbf{q}_1 \leftarrow \mathbf{q}_1 / \|\mathbf{q}_1\|_2$. On the next step, take

$$\mathbf{q}_2 = \mathbf{A}\mathbf{q}_1 - (\mathbf{A}\mathbf{q}_1, \mathbf{q}_1) \mathbf{q}_1 = \mathbf{A}\mathbf{q}_1 - h_{11}\mathbf{q}_1$$

and then normalize $\tilde{\mathbf{q}}_2 \leftarrow \mathbf{q}_2 / \|\mathbf{q}_2\|$. And, on the subsequent steps, take

$$\mathbf{q}_{j+1} = \mathbf{A}\mathbf{q}_j - \sum_{i=1}^j \mathbf{q}_i h_{ij}$$

where h_{ij} is the Gram–Schmidt coefficient $h_{ij} = (\mathbf{A}\mathbf{q}_j, \mathbf{q}_i)$. In practice, we implement this step using the modified Gram–Schmidt with reorthogonalization. Complete the step by normalizing $\mathbf{q}_{j+1} \rightarrow \mathbf{q}_{j+1} / h_{j+1,j}$ where $h_{j+1,j} = \|\mathbf{q}_{j+1}\|_2$.

The Arnoldi method can be written as the following algorithm

Guess an initial $\mathbf{q}^{(1)}$ with $\|\mathbf{q}^{(1)}\|_2 = 1$

for $j = 1, \dots, k$

$$\begin{cases} \mathbf{q}_{j+1} \leftarrow \mathbf{A}\mathbf{q}_j \\ \mathbf{q}_{j+1} \leftarrow \mathbf{q}_{j+1} / \|\mathbf{q}_{j+1}\| \\ \text{for } i = 1, \dots, j \\ \quad \begin{cases} h_{i,j} \leftarrow (\mathbf{q}_i, \mathbf{q}_{j+1}) \\ \mathbf{q}_{j+1} \leftarrow \mathbf{q}_{j+1} - h_{i,j}\mathbf{q}_i \\ h_{j+1,j} \leftarrow \|\mathbf{q}_{j+1}\| \\ \mathbf{q}_{j+1} \leftarrow \mathbf{q}_{j+1} / h_{j+1,j} \end{cases} \end{cases}$$

In matrix form, $\mathbf{A}\mathbf{Q}_k = \mathbf{Q}_{k+1}\mathbf{H}_{k+1,k}$ where $\mathbf{Q}_k = [\mathbf{q}_1 \quad \mathbf{q}_2 \quad \cdots \quad \mathbf{q}_k]$ and

$$\mathbf{H}_{k+1,k} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

is a non-square upper Hessenberg matrix. Note we can complete the $\mathbf{H}_{k+1,k}$ by adding a column ξ_1 to get an upper triangular matrix

$$\mathbf{R} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \ddots & \cdot & \cdot \\ \cdot & \cdot & \ddots & \cdot \\ \cdot & \cdot & \cdot & \ddots \end{bmatrix}$$

in which case $\mathbf{K}_k = \mathbf{Q}_k \mathbf{R}$ is the QR decomposition of the Krylov matrix \mathbf{K}_k . We can also rewrite the Arnoldi decomposition by separating out the bottom row of $\mathbf{H}_{k+1,k}$ to get

$$\underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{Q}_k} = \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{Q}_k} \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{H}_k} + \underbrace{\begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \\ 0 & \cdots & 0 & h_{k+1,k} \end{bmatrix}}_{\mathbf{q}_{k+1}}$$
 (4.6)

Because \mathbf{Q}_k^\top is orthogonal to \mathbf{q}_{k+1} , multiplying (4.6) by \mathbf{Q}_k^\top zeros out the last term and leaves us with $\mathbf{Q}_k^\top \mathbf{A} \mathbf{Q}_k = \mathbf{H}_k$. This means that in the projection, the upper Hessenberg matrix \mathbf{H}_k has the same eigenvalues as \mathbf{A} , i.e., the Ritz values. More precisely, if (\mathbf{x}, μ) is an eigenpair of \mathbf{H}_k , then $\mathbf{v} = \mathbf{Q}_k \mathbf{x}$ satisfies

$$\begin{aligned} \|\mathbf{Av} - \mu\mathbf{v}\|_2 &= \|\mathbf{AQ}_k \mathbf{x} - \mu \mathbf{Q}_k \mathbf{x}\|_2 \\ &= \|(\mathbf{AQ}_k - \mathbf{Q}_k \mathbf{H}_k)\mathbf{x}\|_2 \\ &= h_{k+1,k} \|\xi_{k+1}^\top \mathbf{x}\|_2 \\ &= h_{k+1,k} |x_{k+1}| \end{aligned}$$

The residual norm $h_{k+1,k} |x_{k+1}|$ is called the *Ritz estimate*. If (μ, \mathbf{v}) is a good approximation to an eigenpair of \mathbf{A} , then the Ritz estimate is small. See the figure on the next page and the QR link at the bottom of this page.

It is mathematically intuitive that the success of Arnoldi iteration depends on choosing a good starting vector \mathbf{q} . When \mathbf{q} is chosen at random, it likely has significant components in all eigenvector directions, not just the first k eigenvectors. So, the Krylov subspace is not a great match. How do we get a good starting vector \mathbf{q} for the Krylov subspace in practice? One solution is to do the Arnoldi method itself to get a good starting vector and then restart using this better guess. This method is called the *implicitly restarted Arnoldi method*. Suppose that we want to get the m eigenvectors corresponding to the largest eigenvalues. Let's take the Arnoldi subspace to have $k = j + m \approx 2m$ dimensions. First, run the Arnoldi method to get k Arnoldi vectors \mathbf{Q}_k and \mathbf{H}_k . Now, we will suppress the components of the eigenvectors of $\lambda_{m+1}, \lambda_{m+2}, \dots, \lambda_k$ in our k -dimensional Arnoldi subspace. To do this we run j steps of the shifted QR method, one step for each of the j eigenvectors we are trying to filter out.



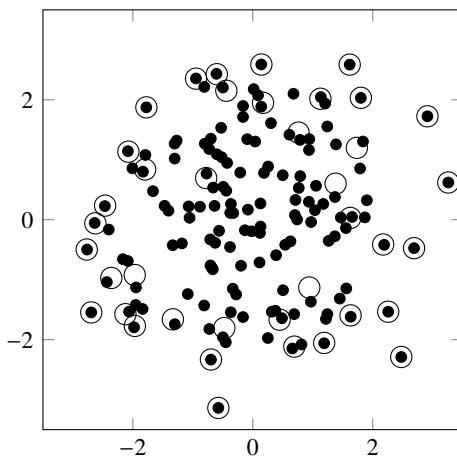


Figure 4.8: Location of the Ritz values \circ for \mathbf{K}_{40} and the eigenvalues \bullet in the complex plane for a 144×144 complex tridiagonal matrix starting with a random initial vector. Notice the Ritz values are closer to eigenvalues that have large magnitudes.

For each shift, we use the approximate Ritz value $\{\mu_{k+1}, \dots, \mu_k\}$. We take the accumulated unitary matrix \mathbf{V}_k from the j QR steps to change the basis to our Arnoldi subspace. Namely,

$$\begin{aligned}\mathbf{H} &\leftarrow \mathbf{V}_k^\top \mathbf{H}_k \mathbf{V}_k \\ \mathbf{Q} &\leftarrow \mathbf{Q}_k \mathbf{V}_k\end{aligned}$$

The first k columns of \mathbf{Q} are our new first k Arnoldi vectors. We zero out the last j Arnoldi vectors (and the last j rows and columns of \mathbf{H}) and find new ones by running the Arnoldi method starting with \mathbf{q}_{k+1} . We continue like this until convergence.

- The Arpack.jl (a wrapper of ARPACK) function `eigs(A, n)` computes n eigenvalues of A using the implicitly restarted Arnoldi method.

The Arnoldi method applied to a symmetric, sparse matrix is called the *Lanczos method*. Symmetric matrices have two clear advantages over nonsymmetric matrices—the eigenvectors are orthogonal and the upper Hessenberg matrix is tridiagonal, so the complexity is only $O(n)$.

4.7 Singular value decomposition

Recall that the singular value decomposition of an $m \times n$ matrix \mathbf{A} is $\mathbf{U}\Sigma\mathbf{V}^\top$ where \mathbf{U} is an $m \times m$ unitary matrix of eigenvectors of $\mathbf{A}\mathbf{A}^\top$, Σ is an $m \times n$ matrix of singular values $\sigma(\mathbf{A}) = \sqrt{\mathbf{A}^\top \mathbf{A}}$, and \mathbf{V} is an $n \times n$ unitary matrix of eigenvectors of $\mathbf{A}^\top \mathbf{A}$. In this section we'll look at the Golub–Kahan–Reinisch algorithm for the SVD—there are several others. The algorithm consists of two steps: transformation to upper bidiagonal form and application of the QR algorithm.

Step 1. Compute

$$\tilde{\mathbf{U}}^\top \mathbf{A} \tilde{\mathbf{V}} = \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where \mathbf{B} is an upper bidiagonal matrix. To get such a matrix, we use a series of Householder reflections (or Givens rotations):

$$\begin{array}{c} \mathbf{A} \\ \left[\begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right] \xrightarrow{\tilde{\mathbf{U}}_1^\top} \left[\begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ \vdots & \vdots & \vdots & \vdots \end{array} \right] \xrightarrow{\cdots \tilde{\mathbf{V}}_1} \left[\begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ \vdots & \vdots & \vdots & \vdots \end{array} \right] \\ \xrightarrow{\tilde{\mathbf{U}}_2^\top} \left[\begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ \vdots & \vdots & \vdots & \vdots \end{array} \right] \xrightarrow{\cdots \tilde{\mathbf{V}}_2} \left[\begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ \vdots & \vdots & \vdots & \vdots \end{array} \right] \\ \xrightarrow{\tilde{\mathbf{U}}_3^\top} \left[\begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ \vdots & \vdots & \vdots & \vdots \end{array} \right] \xrightarrow{\cdots \tilde{\mathbf{V}}_3} \left[\begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ \vdots & \vdots & \vdots & \vdots \end{array} \right] \end{array}$$

After $m - 1$ steps, $\tilde{\mathbf{U}} = \tilde{\mathbf{U}}_1 \tilde{\mathbf{U}}_2 \cdots \tilde{\mathbf{U}}_{m-1}$ and $\tilde{\mathbf{V}} = \tilde{\mathbf{V}}_1 \tilde{\mathbf{V}}_2 \cdots \tilde{\mathbf{V}}_{m-1}$.

Step 2. Now, we use the QR method on \mathbf{B} to get the singular values Σ . One cycle of implicit QR on $\mathbf{B}^\top \mathbf{B}$ is

- Take $\rho = b_{n,n}^2 + b_{n-1,n}^2$ (Rayleigh shifting)

- Compute $\mathbf{A}\mathbf{Q}^\top$ for Givens rotation

$$\mathbf{Q} : [b_{11}^2 - \rho & b_{11}b_{12}] \rightarrow [* & 0]$$

- “Chase the bulge” using left and right Givens rotations

$$\mathbf{U} : \begin{bmatrix} b_{i,i} \\ b_{i+1,i} \end{bmatrix} \rightarrow \begin{bmatrix} * \\ 0 \end{bmatrix} \quad \mathbf{V} : [b_{i,i+1} & b_{i,i+2}] \rightarrow [* & 0].$$

- Deflate when $|b_{n-1,n}| < \varepsilon$ by using the upper left $n - 1 \times n - 1$ matrix.

To find the approximate singular values and singular vectors of a large, sparse matrix, we can use the Lanczos method of finding eigenvalues. If \mathbf{u} is a left singular vector of \mathbf{A} and \mathbf{v} is a right singular vector of \mathbf{A} , then combined they are eigenvectors of the symmetric block matrix

$$\underbrace{\begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{A}^\top & \mathbf{0} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \lambda \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix},$$

where the eigenvalue λ is an eigenvalue of $\mathbf{A}^\top \mathbf{A}$. To see this, note that

$$\left. \begin{array}{l} \mathbf{Av} = \lambda \mathbf{u} \\ \mathbf{A}^\top \mathbf{u} = \lambda \mathbf{v} \end{array} \right\} \text{implies} \quad \left. \begin{array}{l} \mathbf{A}^\top \mathbf{Av} = \lambda \mathbf{A}^\top \mathbf{u} = \lambda^2 \mathbf{v} \\ \mathbf{AA}^\top \mathbf{u} = \lambda \mathbf{Av} = \lambda^2 \mathbf{u} \end{array} \right..$$

\mathbf{M} is a Hermitian matrix, so we can apply the Lanczos method to get the largest eigenvalues.

The *randomized SVD algorithm*, which uses the general Rayleigh–Ritz method, is another technique for computing an approximate, low-rank SVD of a large matrix. We start by choosing a low rank k approximation to the column space of \mathbf{A} with an orthonormal basis: $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\top \mathbf{A}$. The k columns of \mathbf{Q} are chosen at random and orthogonalized using Gram–Schmidt projections, Householder reflections, or Givens rotations. Then we compute the SVD of $\mathbf{Q}^\top \mathbf{A}$ as usual. So,

$$\mathbf{A} \approx \mathbf{Q} (\mathbf{Q}^\top \mathbf{A}) = \mathbf{Q} (\mathbf{W} \Sigma \mathbf{V}^\top) = \mathbf{U} \Sigma \mathbf{V}^\top$$

where $\mathbf{U} = \mathbf{Q}\mathbf{W}$. In practice, instead of computing the randomized SVD of \mathbf{A} we compute the randomized SVD of $(\mathbf{A}\mathbf{A}^\top)^r \mathbf{A}$, where r is a small integer power like 1, 2, or 3 to reduce the noise of the principal components that we are removing. Altogether the randomized SVD algorithm for an $n \times n$ matrix \mathbf{A} is

Generate a random $n \times k$ matrix Ω

$\mathbf{QR} \leftarrow \mathbf{A}\Omega$

for $i = 1, \dots, r$

$$\left[\begin{array}{l} \tilde{\mathbf{Q}}\tilde{\mathbf{R}} \leftarrow \mathbf{A}^\top \mathbf{Q} \\ \mathbf{QR} \leftarrow \mathbf{A}\tilde{\mathbf{Q}} \end{array} \right]$$

Compute the SVD: $\mathbf{W}\Sigma\mathbf{V}^\top \leftarrow \mathbf{Q}^\top \mathbf{A}$

$$\mathbf{U} \leftarrow \mathbf{Q}\mathbf{W}$$

To dig deeper into probabilistic algorithms for constructing approximate matrix decompositions, see Halko et al. [2011].

4.8 Exercises

- 4.1. Consider the $n \times n$ matrix \mathbf{A} where the elements are normally distributed random numbers with variance 1. Conjecture about the distribution of eigenvalues $\lambda(A)$ for arbitrary n .
- 4.2. A square matrix is diagonally dominant if the magnitude of the diagonal element of each row is greater than the sum of the magnitude of all other elements in the that row. Use the Gershgorin circle theorem to prove that a diagonally dominant matrix is always invertible.
- 4.3. Use the Gershgorin circle theorem to estimate the eigenvalues of

$$\begin{bmatrix} 9 & 0 & 0 & 1 \\ 2 & 5 & 0 & 0 \\ 1 & 2 & 0 & 1 \\ 3 & 0 & -2 & 0 \end{bmatrix}.$$

Then compute the actual values numerically.

- 4.4. Use Rayleigh iteration to find the eigenvalues of

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 6 & 4 \\ 3 & 0 & 3 & 1 \\ 6 & 3 & 8 & 8 \\ 4 & 1 & 8 & 2 \end{bmatrix}.$$

Confirm that you are getting a cubic rate of convergence for a symmetric matrix.

- 4.5. Write a program that implements the implicit QR method using single-step Rayleigh-shifting with deflation for matrices with real eigenvalues. Test your code on the matrix on $\mathbf{R}\Lambda\mathbf{R}^{-1}$ where Λ is a diagonal matrix of integers $1, 2, \dots, 20$ and \mathbf{R} is a matrix of normally distributed random numbers. Use a tolerance of $\varepsilon = 10^{-12}$. Compare your approximate eigenvalues with the exact eigenvalues. Are there matrices on which your method fails?
- 4.6. Exercise 2.8 examined the paths connecting one Hollywood actor with one another. This exercise determines which actors are at the center of Hollywood. Compute the eigenvector centrality of the actor-actor adjacency matrix. Explain in plain language what centrality means in context of the six degrees game.
- 4.7. Implement the randomized SVD algorithm. Test your code on an array that corresponds to a sufficiently large grayscale image of your choosing. How does the randomized SVD compare with the full SVD in terms of accuracy and performance for low rank?

- 4.8. The infinite matrix \mathbf{A} with entries $a_{11} = 1$, $a_{12} = 1/2$, $a_{21} = 1/3$, $a_{13} = 1/4$, $a_{22} = 1/5$, $a_{31} = 1/6$, and so on, is a bounded operator on ℓ^2 . What is $\|\mathbf{A}\|_2$? (This problem was proposed by Nick Trefethen as part of his “hundred-dollar, hundred-digit challenge.”)
- 4.9. Draw a diagram showing the connections between concepts in this chapter. You might wish to include things such as the power method, inverse power method, Rayleigh iteration, QR method, PageRank, Perron–Frobenius theorem, implicit QR method, Arnoldi method, and singular value decomposition.

Chapter 5

Iterative Methods for Linear Systems



For small and medium-sized matrices, Gaussian elimination is fast. Even for a 1000×1000 matrix, Gaussian elimination takes less than a second on my exceptionally ordinary laptop. Now, suppose that we want to solve a partial differential equation in three dimensions. We make a discrete approximation to the solution using 100 grid points in each dimension at each time step. Such a problem requires that we solve a system of a million equations at each time step. Now, using Gaussian elimination on my laptop would take over thirty years. Fortunately, such a system is sparse—only 0.001 percent of the elements in the matrix are nonzero. If the matrix has a narrow bandwidth or can be reordered to have narrow bandwidth, Gaussian elimination is still fast. But for a system with no redeeming features other than its sparsity, a better approach is using an iterative method. Gaussian elimination takes $O(n^3)$ operations, but matrix-vector multiplication takes only $O(n^2)$ operations and as little as $O(n)$ operations if the system is sparse. An iterative method will beat a direct method as long as the number of iterations needed for convergence is much smaller than n .

There's another reason why iterative solvers are often used. While direct solvers must always solve each problem from scratch with zero knowledge, an iterative solver may be started with a good initial guess. Suppose that we are solving a time-dependent partial differential equation. The solution at each new time step does not change much from the solution computed at the previous time step. If we are using a direct solver, hopefully, we've been wise and have the LU or Cholesky decomposition, so we only need $O(n^2)$ operations to compute the solution. Although, now we need to contend with fill-in. On the other hand, iterative methods can use the solution from the previous step as a good initial guess for the new time step. And iterative method methods maintain sparsity.

Also, sometimes we may not need as completely accurate a solution as the one provided a direct solver. For example, a finite difference approximation to a

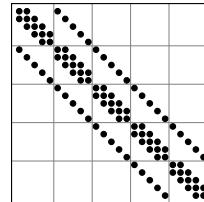
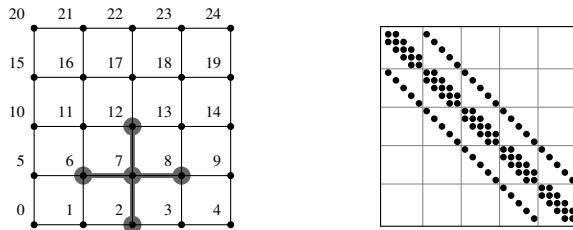


Figure 5.1: The finite difference stencil for the discrete two-dimensional Laplacian on a 5×5 mesh. By labeling the points in lexicon fashion, we can construct a 25×25 block tridiagonal matrix.

partial differential equation may already have a substantial amount of truncation error. In this case, it may be good enough to use a solver that simply gets us within an error threshold. Or, when solving an optimization problem, finding a solution that's close might be as good as finding a solution to machine precision, especially if other error may have been introduced into the model.

One more reason why iterative methods are helpful is that such methods reduce the error in the solution with each iteration. Using an iterative method as a corrector to a direct method can help us solve even ill-conditioned matrices accurately.

Example. Consider the two-dimensional Poisson equation

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x, y)$$

with Dirichlet boundary conditions $f(x, y) = 0$ for $x = 0, x = 1, y = 0$ and $y = 1$. The Poisson equation can be used to model several physical problems, such as the steady-state heat distribution or electrical charge distribution $u(x, y)$ given a source $f(x, y)$ the shape of an elastic membrane $z = u(x, y)$ given a load $f(x, y)$.

A simple yet effective numerical method for solving the Poisson equation is a finite difference method. Consider partitioning a square domain into n intervals each of length h . For the unit square, we would take $h = (n - 1)^{-1}$. Using Taylor series approximation, the discrete Laplacian operator can be represented by

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} \approx \frac{4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}}{h^2}$$

at each grid point (x_i, y_j) . Hence we have a linear system of n^2 equations

$$\frac{4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}}{h^2} = f_{ij}$$

for $1 \leq i, j \leq n$ with $u_{ij} \equiv u(x_i, y_j)$ and $f_{ij} \equiv u(x_i, y_j)$. In matrix-vector notation, $\mathbf{Ax} = \mathbf{b}$ where the column vectors

$$\mathbf{x} = [u_{11} \quad u_{21} \quad \cdots \quad u_{n1} \quad u_{12} \quad u_{22} \quad \cdots \quad u_{nn}]^\top$$

$$\mathbf{b} = [f_{11} \quad f_{21} \quad \cdots \quad f_{n1} \quad f_{12} \quad f_{22} \quad \cdots \quad f_{nn}]^\top.$$

The matrix \mathbf{A} is a sparse, block-tridiagonal matrix with only $5n^2$ nonzero entries out of n^4 elements. See Figure 5.1 on the facing page. \blacktriangleleft

5.1 Jacobi and Gauss–Seidel methods

Suppose that we want to solve $\mathbf{Ax} = \mathbf{b}$. In index format,

$$\sum_{j=1}^n a_{ij}x_j = b_i.$$

Let's start by isolating each element x_i :

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^n a_{ij}x_j \right).$$

The elements of the unknown \mathbf{x} appear on both sides of the equation. One way to solve this problem is by updating x_i iteratively as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right). \quad (5.1)$$

In this method, called the *Jacobi method*, each element of \mathbf{x} is updated independently, which makes vectorization easy.

Another idea is using the newest and best approximations available, overwriting the elements of \mathbf{x} . That is, at each iteration, we compute

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right). \quad (5.2)$$

This is the *Gauss–Seidel method*. Unlike the Jacobi method, which needs to store two copies of \mathbf{x} , the Gauss–Seidel method allows us to use the same array for each iteration, overwriting each element sequentially. Each iteration of the Gauss–Seidel algorithm looks like this

$$\text{for } i = 1, 2, \dots, n$$

$$\left[\begin{array}{l} x_i \leftarrow \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^n a_{ij}x_j \right) / a_{ii} \end{array} \right]$$

At this point, we should ask two questions. When do the Jacobi and Gauss–Seidel methods converge? And how quickly does each converge? To answer these questions, let's look at the Jacobi and Gauss–Seidel methods with a bit more generality. The Jacobi and Gauss–Seidel methods are examples of linear iterative methods. Consider using the splitting $\mathbf{A} = \mathbf{P} - \mathbf{N}$ for some \mathbf{P} and \mathbf{N} . In this case, the linear system $\mathbf{Ax} = \mathbf{b}$ is the same as

$$\mathbf{Px} = \mathbf{Nx} + \mathbf{b}. \quad (5.3)$$

Given $\mathbf{x}^{(0)}$, we can compute $\mathbf{x}^{(k)}$ by solving the system

$$\mathbf{Px}^{(k+1)} = \mathbf{Nx}^{(k)} + \mathbf{b}. \quad (5.4)$$

We call \mathbf{P} the *preconditioner*. To understand why it's called this, suppose that we want to solve the system $\mathbf{Ax} = \mathbf{b}$. Applying \mathbf{P}^{-1} to $\mathbf{Ax} = \mathbf{b}$ gives us $\mathbf{P}^{-1}\mathbf{Ax} = \mathbf{P}^{-1}\mathbf{b}$. The condition number of this new system is $\kappa(\mathbf{P}^{-1}\mathbf{A})$. If we can choose $\mathbf{P} \approx \mathbf{A}$, then we can get a well-conditioned system. For a general linear iterative method $\mathbf{Px}^{(k+1)} = \mathbf{Nx}^{(k)} + \mathbf{b}$ to be a good method

1. the preconditioner \mathbf{P} should be easy to invert; and
2. the method should converge quickly.

One type of preconditioner often used for a general matrix is incomplete LU factorization. Incomplete LU factorization takes $\mathbf{P} = \mathbf{LU}$ where $\mathbf{LU} \approx \mathbf{A}$ and the LU decomposition to get \mathbf{L} and \mathbf{U} is fast to compute. For example, we might choose the preconditioner \mathbf{P} that matches \mathbf{A} in certain elements and is otherwise zero.

Let's return to Jacobi and Gauss–Seidel methods. The Jacobi method splits the matrix into diagonal and off-diagonal matrices:

$$\mathbf{A} = \mathbf{P} - \mathbf{N} = \mathbf{D} + \mathbf{M} : \quad \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} = \begin{bmatrix} \bullet & & & \\ & \bullet & & \\ & & \bullet & \\ & & & \bullet \end{bmatrix} + \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}$$

For an initial guess $\mathbf{x}^{(0)}$, we solve $\mathbf{Dx}^{(k+1)} = -\mathbf{Mx}^{(k)} + \mathbf{b}$. The Gauss–Seidel method splits the matrix into lower triangular and strictly upper triangular matrices:

$$\mathbf{A} = \mathbf{P} - \mathbf{N} = \mathbf{L} + \mathbf{D} + \mathbf{U} : \quad \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} = \begin{bmatrix} \bullet & & & \\ \bullet & \bullet & & \\ \bullet & \bullet & \bullet & \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} + \begin{bmatrix} \bullet & & & \\ & \bullet & & \\ & & \bullet & \\ & & & \bullet \end{bmatrix} + \begin{bmatrix} & & & \\ & \bullet & & \\ & & \bullet & \\ & & & \bullet \end{bmatrix}$$

For an initial guess $\mathbf{x}^{(0)}$, we solve $(\mathbf{L} + \mathbf{D})\mathbf{x}^{(k+1)} = -\mathbf{Ux}^{(k)} + \mathbf{b}$.

The error at the k th iteration is $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$. Subtracting (5.4) from (5.3), gives us $\mathbf{Pe}^{(k+1)} = \mathbf{Ne}^{(k)}$. Equivalently,

$$\mathbf{e}^{(k+1)} = \mathbf{P}^{-1}\mathbf{Ne}^{(k)}.$$

An iterative method creates a *contraction mapping* such that $\mathbf{e}^{(k)} \rightarrow \mathbf{0}$ as $k \rightarrow \infty$. In terms of the initial error $\mathbf{e}^{(0)}$,

$$\mathbf{e}^{(k)} = (\mathbf{P}^{-1}\mathbf{N})^k \mathbf{e}^{(0)}, \quad (5.5)$$

and by submultiplicativity it follows that

$$\|\mathbf{e}^{(k)}\| \leq \|(\mathbf{P}^{-1}\mathbf{N})\|^k \|\mathbf{e}^{(0)}\|.$$

This is true for *every* induced norm. Recall from theorem 5 that all induced matrix norms are bounded below by the spectral radius. Hence,

$$\|\mathbf{e}^{(k)}\| \leq \rho(\mathbf{P}^{-1}\mathbf{N})^k \|\mathbf{e}^{(0)}\|.$$

So, if $\rho(\mathbf{P}^{-1}\mathbf{N}) < 1$ then the method converges. On the other hand, suppose that $\mathbf{P}^{-1}\mathbf{N}$ has an eigenvalue λ with $|\lambda| \geq 1$. Then if $\mathbf{e}^{(0)}$ happens to be an eigenvector associated with λ , the error $\|\mathbf{e}^{(k+1)}\| = \lambda^k \|\mathbf{e}^{(0)}\|$ and the method never converges. We can summarize the discussion above in the following theorem.

Theorem 15. *The iteration $\mathbf{P}\mathbf{x}^{(k+1)} = \mathbf{N}\mathbf{x}^{(k)} + \mathbf{b}$ converges if and only if the spectral radius $\rho(\mathbf{P}^{-1}\mathbf{N}) < 1$.*

A matrix is *diagonally dominant* if the magnitudes of the diagonal elements of each row is greater than the sum of the magnitude of all other elements in that row, i.e., $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$.

Theorem 16. *If a matrix is diagonally dominant, then the Jacobi and Gauss–Seidel methods converge.*

Proof. For the Jacobi method,

$$\rho(\mathbf{D}^{-1}\mathbf{M}) \leq \|\mathbf{D}^{-1}\mathbf{M}\|_\infty = \max_i \sum_{j \neq i} \left| \frac{a_{ij}}{a_{ii}} \right| < 1.$$

So, the Jacobi method converges. Put another way, a method $\mathbf{P}\mathbf{e}^{(k+1)} = \mathbf{N}\mathbf{e}^{(k)}$ converges if it is a contraction mapping on the error. For the Jacobi method $a_{ii}e_i^{(k+1)} = \sum_{j \neq i} a_{ij}e_j^{(k)}$ from which

$$\|\mathbf{e}^{(k+1)}\|_\infty \leq \frac{\sum_{j \neq i} |a_{ij}|}{|a_{ii}|} \|\mathbf{e}^{(k)}\|_\infty.$$

Intuitively, the Gauss–Seidel ought to converge even faster, because it has even more terms in the denominator. But, because some of the terms might be negative, it's not completely obvious. So, let's be explicit:

$$\|\mathbf{e}^{(k+1)}\|_\infty \leq \frac{\sum_{j>i} |a_{ij}|}{|a_{ii}| - \sum_{j<i} |a_{ij}|} \|\mathbf{e}^{(k)}\|_\infty < \frac{|a_{ii}| - \sum_{j<i} |a_{ij}|}{|a_{ii}| - \sum_{j \neq i} |a_{ij}|} \|\mathbf{e}^{(k)}\|_\infty = \|\mathbf{e}^{(k)}\|_\infty$$

because the matrix is diagonally dominant. \square

The *convergence factor* r_k of an iterative method is given by the ratio of errors at each iteration: $r_k = \|\mathbf{e}^{(k)}\|/\|\mathbf{e}^{(k-1)}\|$. The average convergence factor is over m iterations is the geometric mean $(\prod_{k=1}^m r_k)^{1/m}$. From this we can compute the average convergence rate: $-\frac{1}{m} \sum_{k=1}^m \log r_k$. It is often easier (and just as meaningful) to frame the performance of a method in terms of the long-term convergence rate. We define the *asymptotic convergence rate* as $\lim_{k \rightarrow \infty} -\log r_k$.

Example. Let's examine the convergence using the $n \times n$ discrete Laplacian

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}. \quad (5.6)$$

From exercise 1.5, we can compute the spectral radius of $\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ as $\cos \pi h \approx 1 - \frac{1}{2}\pi^2 h^2$, where $h = (n-1)^{-1}$ is the grid size over a unit interval. This gives us the convergence factor of the Jacobi method. The asymptotic convergence rate of the Jacobi method when n is large is $-\log \rho(\mathbf{P}^{-1}\mathbf{N}) \approx \frac{1}{2}\pi^2 h^2$, where we used the approximation $\log(1-z) \approx z + \frac{1}{2}z^2 + \dots$. We can determine the convergence rate of the Gauss–Seidel method by computing the spectral radius of $(\mathbf{L} + \mathbf{D})^{-1}\mathbf{U}$: $\cos^2 \pi h \approx 1 - \pi^2 h^2$. The asymptotic convergence rate of the Gauss–Seidel method is $\pi^2 h^2$, twice that of the Jacobi method.

If \mathbf{A} is a rather small 20×20 matrix, then the convergence factor for the Jacobi method is 0.99, and for the Gauss–Seidel, it is 0.98. The asymptotic convergence rates are 0.01 and 0.02, respectively. Hence, it may conceivably take as many as 450 iterations and 225 iterations, respectively, to get the error to one-hundredth of the initial error.¹ Pretty lousy, especially when we could simply use naïve Gaussian elimination and get an exact solution in less than one-tenth of the time. And for a larger matrix, it's much worse. If we double the number of grid points, we'll need to quadruple the number of iterations to achieve the same accuracy. What can we do to speed up convergence? ◀

5.2 Successive over relaxation

The preconditioners in both the Jacobi and Gauss–Seidel methods are easy to invert. Now, our goal is to speed up convergence by somehow choosing a splitting $\mathbf{A} = \mathbf{P} - \mathbf{N}$ to quickly shrink the error $\mathbf{e}^{(k+1)} = (\mathbf{P}^{-1}\mathbf{N})^k \mathbf{e}^{(0)}$. The initial error $\mathbf{e}^{(0)}$ is a linear combination of the eigenvectors of $\mathbf{P}^{-1}\mathbf{N}$. Each of its components is scaled by their respective eigenvalues at every step of the iteration. A component

¹The missing arithmetic is $\log 0.01/\log 0.99 = 450$ and $\log 0.01/\log 0.98 = 225$.

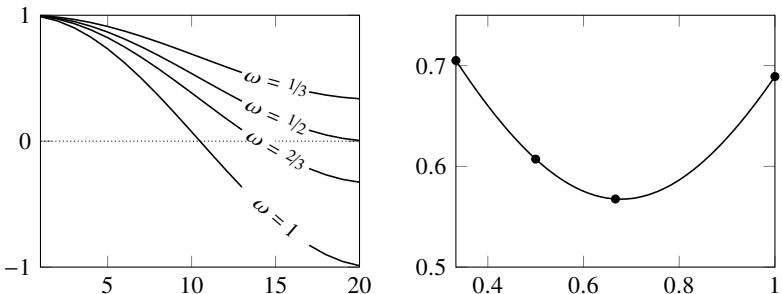


Figure 5.2: Left: eigenvalues of the weighted Jacobi method for the 20×20 discrete Laplacian (5.6) for different weights ω . Right: root mean squared values of the eigenvalues as a function of ω .

with a small eigenvalue will swiftly fade away. One that has a magnitude near one will linger like a drunken, boorish guest after a party. If we happen to split the matrix \mathbf{A} in a way that leaves the spectral radius of $\mathbf{P}^{-1}\mathbf{N}$ greater than one, then we'll have an unstable method. While the best splitting will be problem-specific, we can think of a good splitting as one that minimizes all eigenvalues. In the next section, we'll look at a method that systematically targets groups of the eigenvalues at a time.

Let's start by modifying the Jacobi method. Perhaps we can speed things up by weighting the preconditioner. If we take $\mathbf{P}^{-1} = \omega\mathbf{D}^{-1}$, we'll reduce the condition number $\kappa(\mathbf{P}^{-1}\mathbf{A})$. Such an approach is called a *weighted Jacobi method* with the splitting

$$\mathbf{A} = \mathbf{P} - \mathbf{N} = (\omega^{-1}\mathbf{D}) + (\mathbf{A} - \omega^{-1}\mathbf{D}).$$

For the discrete Laplacian (5.6) in the previous example, the eigenvalues of $\mathbf{P}^{-1}\mathbf{N}$ are now $\lambda_k = 1 + \omega(\cos k\pi h - 1)$. See the figure above. The original, unweighted Jacobi takes $\omega = 1$. The closer an eigenvalue is to zero, the faster its associated component will fade away with each iteration. A good choice for ω might be one that minimizes the root mean squared values of the eigenvalues λ_k , and for the discrete Laplacian (5.6), that choice for ω happens to be about $\frac{2}{3}$. While we are reducing the magnitude of some eigenvalues by tuning ω , we are increasing the magnitude of other eigenvalues, including the dominant one.

Before modifying the Gauss–Seidel method, let's think a bit more about the meaning of ω . The residual at the k th iteration is

$$\mathbf{r}^{(k)} = \mathbf{A}\mathbf{e}^{(k)} = \mathbf{b} - \mathbf{Ax}^{(k)}.$$

Using this definition and $\mathbf{N} = \mathbf{A} - \mathbf{P}$, we can rewrite $\mathbf{x} = \mathbf{P}^{-1}(\mathbf{Nx} + \mathbf{b})$ as

$$\mathbf{x}^{(k+1)} = \mathbf{P}^{-1}(\mathbf{Nx}^{(k)} + \mathbf{b}) = \mathbf{x}^{(k)} + \mathbf{P}^{-1}\mathbf{r}^{(k)}.$$

We call $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{\delta}$ for some vector $\boldsymbol{\delta}$ the *relaxation* of \mathbf{x} . Think of “relaxation” as $\mathbf{x}^{(k)}$ returning to a steady state from a perturbed state. An iterative method can be thought of as an analog to finding the steady-state solution to a time-dependent problem. For example, consider the one-dimensional Poisson equation and its finite difference approximation

$$-\frac{d^2u}{dx^2} = f(x) \quad \text{and} \quad \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} = f_i.$$

The resulting system of equations $\mathbf{Au} = \mathbf{b}$ can be solved iteratively as

$$\mathbf{Pu}^{(k+1)} + (\mathbf{A} - \mathbf{P})\mathbf{u}^{(k)} = \mathbf{b},$$

which can be rewritten as the equivalent

$$\mathbf{u}^{(k+1)} - \mathbf{u}^{(k)} = \mathbf{P}^{-1}(\mathbf{b} - \mathbf{Au}^{(k)}).$$

Note the similarity between this system and the forward Euler method

$$\mathbf{u}^{(k+1)} - \mathbf{u}^{(k)} = \Delta t(\mathbf{b} - \mathbf{Au}^{(k)})$$

used to solve the continuous time-dependent heat equation with source term

$$\frac{\partial u(t, x)}{\partial t} = f(x) + \frac{\partial^2 u(t, x)}{\partial x^2}.$$

The speed and stability of the forward Euler method is tuned by modifying the step size Δt . Analogously, the speed and stability of an iterative method are tuned by modifying the preconditioner \mathbf{P}^{-1} . The Jacobi method simultaneously relaxes $\mathbf{x}^{(k)}$, and the weighted Jacobi method simultaneously under relaxes $\mathbf{x}^{(k)}$ by choosing a weight $0 < \omega \leq 1$ to increase stability.

The Gauss–Seidel method successively relaxes $\mathbf{x}^{(k)}$. The goal behind the *successive over-relaxation* (SOR) method is to speed up convergence by choosing $\omega > 1$ to go beyond the Gauss–Seidel correction. The naïve algorithm for the SOR method is

```
for  $i = 1, 2, \dots, n$ 
   $x^* \leftarrow \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^n a_{ij}x_j \right) / a_{ii}$ 
   $\delta \leftarrow x^* - x_i$ 
   $x_i \leftarrow x_i + \omega\delta$ 
```

The constant ω is called the relaxation factor. The SOR algorithm simply computes the step size δ determined by the Gauss–Seidel algorithm and scales that step size by ω . When $\omega = 1$, the SOR method is simply the Gauss–Seidel method. In general, the SOR method has the splitting

$$\mathbf{A} = \mathbf{P} - \mathbf{N} = \left(\mathbf{L} + \omega^{-1}\mathbf{D} \right) + \left((1 - \omega^{-1})\mathbf{D} + \mathbf{U} \right).$$

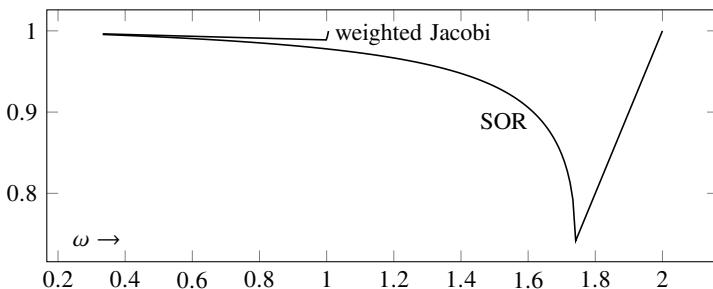


Figure 5.3: The spectral radius $\rho(\mathbf{P}^{-1}\mathbf{N})$ of the weighted Jacobi and SOR methods for the 20×20 discrete Laplacian as functions of the relaxation factor ω .

For an initial guess $\mathbf{x}^{(0)}$, we iterate on

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

The trick is to choose an ω so that the SOR method converges quickly, i.e., choose the ω that minimizes $\rho(\mathbf{P}^{-1}\mathbf{N})$. The choice is problem-specific and not trivial. For the discrete Laplacian (5.6), the optimal relaxation factor can be determined to be $\omega = 2/(1 + \sin \pi h)$, for which $\rho(\mathbf{P}^{-1}\mathbf{N}) = 1 - 2\pi h + O(h^2)$. See Watkins [2014].

The convergence factor for a 20×20 discrete Laplacian is computed in the figure above for several values of ω . When $\omega \approx 1.72$, the convergence factor for the SOR method reaches a minimum of about 0.74. By using the SOR method on the example on page 124, we can reduce the number of iterations from 225 down to about 15—a significant improvement.² Still, a 20×20 matrix is quite small, and we want a method that works well on massive matrices.

5.3 Multigrid method

To design a method that works well on big matrices, we need to contend better with the slowly decaying, low-frequency eigenvector components. The solution to the Poisson equation takes so many iterations to converge because the finite difference stencil for the discrete Laplacian only allows mesh points to communicate with neighboring mesh points. Using a three-point stencil in each direction like the one in Figure 5.1 on page 120, information can be passed two mesh points with each iteration. If the discretization has n mesh points in each dimension, it will take $n/2$ iterations just to pass information from one side of the domain to the other. The result is slow convergence. Of course, we could use a larger stencil. Information would travel more quickly across the domain, but then we are adding

²The missing arithmetic is $\log 0.01/\log 0.74 \approx 15$.

more nonzero elements to our sparse matrix. If we let the stencil stretch across the entire domain, we have the fast Poisson solver, which is tremendously fast on regular domains. We'll come back to fast Poisson solvers in the next chapter, but for now, we want a solver that can work on all types of domains.

Another idea is to use a coarser grid. For an $n \times n$ matrix, an optimal SOR method needs $O(n)$ iterations for convergence. The suboptimal Gauss-Seidel and Jacobi methods need $O(n^2)$. So, reducing the size of our system will have a tremendous impact on reducing the number of iterations. But, with a coarser grid, the solution would not be as accurate.

The multigrid method attempts to get both speed and accuracy by iterating on different grid refinements, using a coarse grid to speed up convergence and a fine grid to get higher accuracy. The multigrid method embodies the following idea:

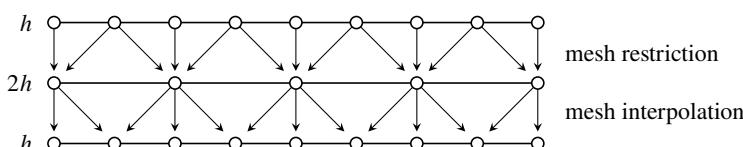
- solve the problem on a coarse mesh to kill off the residual from the low frequencies components;
- solve the problem on a fine mesh for high accuracy; and
- use the solution on the coarse mesh as a starting guess for the solution on the fine mesh (and vice versa).

Consider the simple one-dimensional Poisson equation discussed in the previous sections. Let \mathbf{A}_h have a mesh size h and let \mathbf{A}_{2h} be \mathbf{A}_h restricted to a mesh size $2h$. One single cycle (with mesh h) can be implemented as follows:

1. Iterate a few times on $\mathbf{A}_h \mathbf{u}_h = \mathbf{b}_h$ using weighted Jacobi, Gauss-Seidel, or SOR. This will shrink the residual $\mathbf{r}_h = \mathbf{b}_h - \mathbf{A}_h \mathbf{u}_h$ from the higher frequencies eigenvector components and smooth out the solution.
2. Restrict the residual \mathbf{r}_h to a coarse grid by taking $\mathbf{r}_{2h} = \mathbf{R}_h^{2h} \mathbf{r}_h$ where the restriction matrix is defined as

$$\mathbf{R}_h^{2h} = \frac{1}{4} \begin{bmatrix} 3 & 1 & & & & \\ & 1 & 2 & 1 & & \\ & & & 1 & 2 & 1 \\ & & & & \ddots & \\ & & & & & 1 & 3 \end{bmatrix}$$

mapping a $(2n + 1)$ -dimensional vector \mathbf{r}_h to an $(n + 1)$ -vector \mathbf{r}_{2h} .

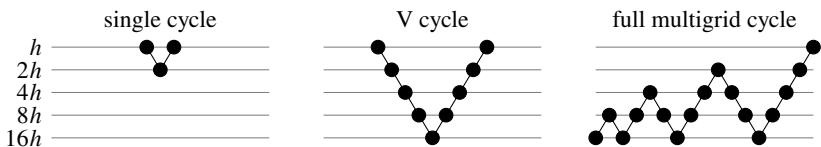


3. Solve (iterate a few times) $\mathbf{A}_{2h}\mathbf{e}_{2h} = \mathbf{r}_{2h}$ where $\mathbf{e}_{2h} = \mathbf{u}_h - \mathbf{u}_{2h}$. Not surprisingly, $\mathbf{A}_{2h} = \mathbf{R}_h^{2h}\mathbf{A}_h$. Because the size of the matrix has changed, we may also need to adjust the relaxation factor ω accordingly.
4. Interpolate the error \mathbf{e}_{2h} back to the fine grid by taking $\mathbf{e}_h = \mathbf{I}_{2h}^h \mathbf{e}_{2h}$ where the interpolation is defined as

$$\mathbf{I}_{2h}^h = \frac{1}{2} \begin{bmatrix} 2 & 1 & & & & & \\ & 1 & 2 & 1 & & & \\ & & & 1 & 2 & 1 & \\ & & & & & & \ddots \\ & & & & & & \\ & & & & & & 1 & 2 \end{bmatrix}^\top.$$

5. Add \mathbf{e}_h to \mathbf{u}_h .
6. Iterate $\mathbf{A}_h\mathbf{u}_h = \mathbf{b}_h$.

Steps 1–6 make up a single cycle multigrid. In practice, multiple cycles are typically used when restricting to coarser and coarser meshes before interpolating back. Or, multigrid may be started with a coarse grid. Here are different cycles used in the multigrid method to restrict and interpolate between the original mesh h and a coarser mesh $16h$:



Example. Let's compare the Jacobi, Gauss–Seidel, SOR, and multigrid methods on a model problem. Take the Poisson equation $u'' = f(x)$ with zero boundary conditions, where the source function is a combination of derivatives of Dirac delta distribution $f(x) = \delta'(x - \frac{1}{2}) - \delta'(x + \frac{1}{2})$. The solution is a rectangle function $u(x) = 1$ for $x \in (-\frac{1}{2}, +\frac{1}{2})$ and $u(x) = 0$ otherwise. Take $n = 129$ grid points and choose the relaxation factor ω to be optimal for the SOR method and multigrid method. For the multigrid method, use a V-cycle with two restrictions and two interpolations and compute one SOR iteration at each step. Finally, take an initial guess of $u(x) = \frac{1}{2}$. The solution is plotted in the figure on the following page. Notice that while the SOR and multigrid methods converge relatively quickly, the Jacobi and Gauss–Seidel methods still have significant error after many iterations and convergence is slowing down. ▶

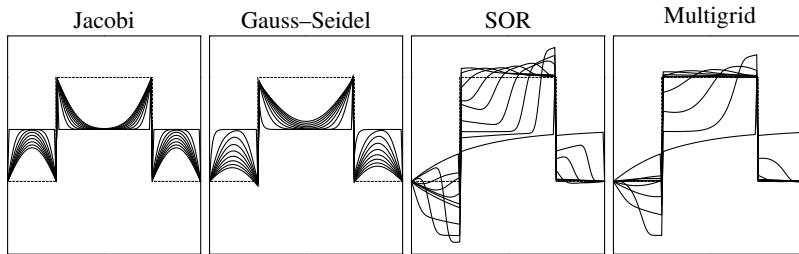


Figure 5.4: Iterative solutions to the Poisson equation converging to the exact solution, the rectangle function. Snapshots are taken every twentieth iteration for a total of 160 iterations. Also, see the QR link at the bottom of this page.

5.4 Solving the minimization problem

In Chapter 3 we solved an overdetermined system by solving an equivalent minimization problem. And in the last chapter, we approximated eigenvalues of sparse matrices by finding the solution that minimized the error in the Krylov subspace. The Poisson equation $-\Delta u = f$ can also be re-expressed as the problem of finding the solution u that minimizes the energy functional $\int_{\Omega} \frac{1}{2} |\nabla u|^2 - uf \, dV$. In this section, we examine how to formulate an iterative method to solve the minimization problem.

► Gradient descent method

One of the difficulties of the SOR method is that you need to know the optimal relaxation parameter ω . If \mathbf{A} is a symmetric, positive-definite matrix, a better approach is recasting the problem as a minimization problem.

Theorem 17. *If \mathbf{A} is a symmetric, positive-definite matrix, then the solution to the equation $\mathbf{Ax} = \mathbf{b}$ is the unique minimizer of the quadratic form*

$$\Phi(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{b} = \frac{1}{2} \|\mathbf{x}\|_{\mathbf{A}}^2 - (\mathbf{x}, \mathbf{b}).$$

Proof. Take $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ and consider any other vector \mathbf{y} . Then

$$\begin{aligned} \Phi(\mathbf{y}) - \Phi(\mathbf{x}) &= \frac{1}{2} \mathbf{y}^T \mathbf{A} \mathbf{y} - \mathbf{y}^T \mathbf{b} - \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{x}^T \mathbf{b} \\ &= \frac{1}{2} \mathbf{y}^T \mathbf{A} \mathbf{y} - \mathbf{y}^T \mathbf{A} \mathbf{x} - \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{x}^T \mathbf{A} \mathbf{x} \\ &= \frac{1}{2} \mathbf{y}^T \mathbf{A} \mathbf{y} - \frac{1}{2} \mathbf{y}^T \mathbf{A} \mathbf{x} - \frac{1}{2} \mathbf{y}^T \mathbf{A} \mathbf{x} + \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} \\ &= \frac{1}{2} \mathbf{y}^T \mathbf{A} \mathbf{y} - \frac{1}{2} \mathbf{y}^T \mathbf{A} \mathbf{x} - \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{y} + \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} \\ &= \frac{1}{2} (\mathbf{y} - \mathbf{x})^T \mathbf{A} (\mathbf{y} - \mathbf{x}) > 0 \end{aligned}$$

because \mathbf{A} is symmetric, positive definite. So, $\Phi(\mathbf{y}) > \Phi(\mathbf{x})$. □



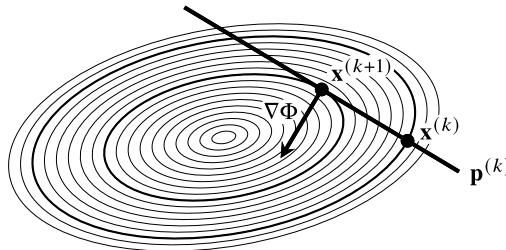


Figure 5.5: The optimal vector $\mathbf{x}^{(k+1)}$ along direction $\mathbf{p}^{(k)}$.

We can think of the solution \mathbf{x} as being at the bottom of an n -dimensional “oblong bowl.” Let $\mathbf{x}^{(0)}$ be an initial guess for \mathbf{x} and let $\mathbf{x}^{(k)}$ be the k th subsequent estimate

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \quad (5.7)$$

for some direction $\mathbf{p}^{(k)}$. The error at the k th step is $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$. If we want to get to the bottom of the “bowl” only knowing the local topography at $\mathbf{x}^{(k)}$, then the best path would be to take the steepest descent, i.e., the direction of the negative gradient. The gradient of Φ at $\mathbf{x}^{(k)}$ is

$$\nabla\Phi(\mathbf{x}^{(k)}) = \mathbf{A}\mathbf{x}^{(k)} - \mathbf{b} = -\mathbf{r}^{(k)}.$$

That is, the negative gradient is simply the residual. This means that the k th subsequent estimate (5.7) can be rewritten as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}. \quad (5.8)$$

For now, consider the more general iterative method (5.7) for an arbitrary direction vector $\mathbf{p}^{(k)}$, keeping in mind that $\mathbf{p}^{(k)} = \mathbf{r}^{(k)}$ for the method of gradient descent. Let’s find α_k . Since the evaluation of $\mathbf{A}\mathbf{x}^{(k)}$ to get the residual can require a lot of computer processing, we try to reduce the number of times we compute an update to the direction. We will only change direction when we are closest to the solution \mathbf{x} along our current direction. A vector \mathbf{x} is said to be *optimal* with respect to a non-zero direction \mathbf{p} if $\Phi(\mathbf{x}) \leq \Phi(\mathbf{x} + \gamma\mathbf{p})$ for all $\gamma \in \mathbb{R}$. That is, \mathbf{x} is optimal with respect to \mathbf{p} if Φ increases as we move away from \mathbf{x} along the direction \mathbf{p} . Put another way, \mathbf{x} is optimal with respect to a direction \mathbf{p} if the directional derivative of Φ at \mathbf{x} in the direction \mathbf{p} is zero, i.e., $\nabla\Phi(\mathbf{x}) \cdot \mathbf{p} = 0$. If \mathbf{x} is optimal with respect to every direction in a vector space V , we say the \mathbf{x} is optimal with respect to V . Looking at (5.8), we want to know when $\mathbf{x}^{(k+1)}$ is optimal with respect to a direction $\mathbf{r}^{(k)}$.

Because $\nabla\Phi(\mathbf{x}^{(k+1)}) = -\mathbf{r}^{(k+1)}$, it follows that $\mathbf{x}^{(k+1)}$ is optimal with respect to $\mathbf{p}^{(k)}$ —in the direction of $\mathbf{r}^{(k)}$ —if and only if $\mathbf{r}^{(k+1)} \cdot \mathbf{p}^{(k)} = 0$, i.e., if and only

if $\mathbf{r}^{(k+1)}$ is orthogonal to $\mathbf{p}^{(k)}$.

$$\begin{aligned}\mathbf{r}^{(k+1)} &= \mathbf{b} - \mathbf{Ax}^{(k+1)} \\ &= \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}) \\ &= \mathbf{b} - \mathbf{Ax}^{(k)} - \alpha_k \mathbf{Ap}^{(k)} \\ &= \mathbf{r}^{(k)} - \alpha_k \mathbf{Ap}^{(k)}.\end{aligned}\tag{5.9}$$

When $\mathbf{r}^{(k+1)} \perp \mathbf{p}^{(k)}$:

$$0 = \mathbf{p}^{(k)\top} \mathbf{r}^{(k+1)} = \mathbf{p}^{(k)\top} \mathbf{r}^{(k)} - \alpha_k \mathbf{p}^{(k)\top} \mathbf{Ap}^{(k)}.$$

Therefore, $\mathbf{x}^{(k+1)}$ is optimal with respect to $\mathbf{r}^{(k)}$ when

$$\alpha_k = \frac{\mathbf{p}^{(k)\top} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)\top} \mathbf{Ap}^{(k)}}.$$

By taking $\mathbf{p}^{(k)} = \mathbf{r}^{(k)}$ (for the gradient descent method), we have

$$\alpha_k = \frac{\mathbf{r}^{(k)\top} \mathbf{r}^{(k)}}{\mathbf{r}^{(k)\top} \mathbf{Ar}^{(k)}}.$$

At this new vector $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}$, we change directions, again taking the steepest descent in the direction of the negative gradient $-\nabla\Phi(\mathbf{x}^{(k+1)})$. We continue like this until either we find the minimum \mathbf{x} or we are close enough. That's the idea. Let's write it out as an algorithm:

```
make an initial guess  $\mathbf{x}$ 
for  $k = 0, 1, 2, \dots$ 
   $\left[ \begin{array}{l} \mathbf{r} \leftarrow \mathbf{b} - \mathbf{Ax} \\ \text{if } \|\mathbf{r}\| < \text{tolerance, then we're done} \\ \alpha \leftarrow \mathbf{r}^\top \mathbf{r} / \mathbf{r}^\top \mathbf{Ar} \\ \mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{r} \end{array} \right]$ 
```

► Gauss-Seidel and SOR as descent methods

Let's go back and look at the general descent method (5.7)

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)},$$

where $\mathbf{x}^{(k)}$ is optimal with respect to the direction $\mathbf{p}^{(k)}$ when

$$\alpha_k = \frac{\mathbf{p}^{(k)\top} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)\top} \mathbf{Ap}^{(k)}}.$$

Consider taking the direction $\mathbf{p}^{(k)}$ successively along each of the coordinate axes ξ_i . For example, if \mathbf{A} is 3×3 , we take

$$\mathbf{p}^{(1)} = \xi_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{p}^{(2)} = \xi_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{p}^{(3)} = \xi_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{p}^{(4)} = \xi_1, \text{ and so on.}$$

Then for an n -dimensional space

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \frac{\xi_i^\top \mathbf{r}^{(k)}}{\xi_i^\top \mathbf{A} \xi_i} \xi_i \quad \text{with } i = k \pmod{n} + 1.$$

We can rewrite this expression in component form by first noting that

$$\xi_i^\top \mathbf{r}^{(k)} = b_i - \sum_{j=1}^n a_{ij} x_j \quad \text{and} \quad \xi_i^\top \mathbf{A} \xi_i = a_{ii}$$

and then noting that $\mathbf{x}^{(k)} + \alpha_k \xi_i$ only updates the i th component of $\mathbf{x}^{(k)}$. In other words,

$$\begin{aligned} x_i^{(k+1)} &= x_i^{(k)} + \frac{1}{a_{ii}} \left(- \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i}^n a_{ij} x_j^{(k)} + b_i \right) \\ &= \frac{1}{a_{ii}} \left(- \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} + b_i \right), \end{aligned}$$

which is simply the Gauss–Seidel method (5.2). Depending on the shape of our “bowl,” it may be advantageous to take a shorter or longer stepsize than optimal

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \omega \alpha_k \mathbf{p}^{(k)}.$$

By taking our directions successively along the coordinate axes with the scaling factor $0 < \omega < 2$, we have the SOR method. See Figure 5.6 on the next page.

Theorem 18. *If a matrix is symmetric, positive definite, then the Gauss–Seidel method converges for any $\mathbf{x}^{(0)}$. Furthermore, the SOR method converges if and only if $0 < \omega < 2$.*

Proof. By theorem 17, if \mathbf{A} is a symmetric, positive-definite matrix, then Φ has a unique minimizer. As a descent method, the Gauss–Seidel converges to the bottom of the “bowl.” We can consider the “bowl” as a circular paraboloid that is sheared and stretched along a coordinate axis. This follows directly from QR-decomposition.

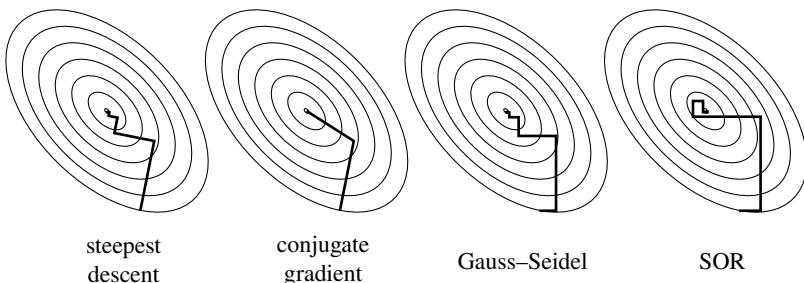
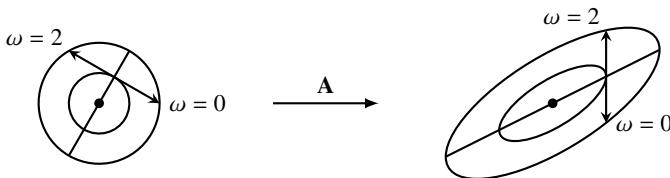


Figure 5.6: Directions taken by different iterative methods to get to the solution.



The value $\omega = 2$ takes us to the point at the same elevation on the opposite side of the bowl along a path parallel to a coordinate axis. Any value $0 < \omega < 2$ positions us lower in the bowl and $\omega = 1$ positions us optimally along the direction. \square

► Conjugate gradient method

While locally gradient descent does give the best choice of directions, globally it doesn't, especially when the "bowl" is very eccentric, i.e., when the condition number of the matrix is large. So, in practice, the gradient descent method doesn't always work particularly well. Let's improve it by taking globally optimal directions that are not necessarily along the residuals.

What went wrong with the gradient method? Each new search direction $\mathbf{p}^{(k)}$ was determined using only information from the most recent search direction $\mathbf{p}^{(k-1)}$. As before, let $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$ for some direction $\mathbf{p}^{(k)}$. Suppose that $\mathbf{x}^{(k+1)}$ is optimal with respect to $\mathbf{p}^{(k)}$ as we did with the gradient method, i.e., $\mathbf{p}^{(k)} \perp \nabla \Phi(\mathbf{x}^{(k+1)})$ or equivalently $\mathbf{p}^{(k)} \perp \mathbf{r}^{(k+1)}$. Let's impose the further condition that $\mathbf{x}^{(k+1)}$ is also optimal with respect to $\mathbf{p}^{(j)}$ for all $j = 0, 1, \dots, k$. That is, $\mathbf{r}^{(k+1)} \perp \mathbf{p}^{(j)}$ for all $j = 0, 1, \dots, k$.

Let's take a moment to think about what this new optimality condition really means. If the position $\mathbf{x}^{(k+1)}$ is optimal with respect to each of the directions $\mathbf{p}^{(k)}, \mathbf{p}^{(k-1)}, \dots, \mathbf{p}^{(0)}$, then $\mathbf{x}^{(k+1)}$ is optimal with respect to the entire span $\{\mathbf{p}^{(k)}, \mathbf{p}^{(k-1)}, \dots, \mathbf{p}^{(0)}\}$ by linearity of the gradient. As we iterate, the subspace spanned by the directions $\mathbf{p}^{(k)}, \mathbf{p}^{(k-1)}, \dots, \mathbf{p}^{(0)}$ will continue to grow

as long as each new direction is linearly independent. That sounds pretty good. Let's find the directions $\mathbf{p}^{(k)}, \mathbf{p}^{(k-1)}, \dots, \mathbf{p}^{(0)}$ and show that they are linearly independent.

From the updated residual, we have

$$\mathbf{r}^{(k+1)} = \mathbf{b} - \mathbf{Ax}^{(k+1)} = \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}) = \mathbf{r}^{(k)} - \alpha_k \mathbf{Ap}^{(k)},$$

and from our optimality condition, we have

$$0 = \mathbf{p}^{(j)\top} \mathbf{r}^{(k+1)} = \mathbf{p}^{(j)\top} \mathbf{r}^{(k)} - \alpha_k \mathbf{p}^{(j)\top} \mathbf{Ap}^{(k)}$$

for all $j \leq k$. For $j = k$, it follows that

$$\alpha_k = \frac{\mathbf{p}^{(k)\top} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)\top} \mathbf{Ap}^{(k)}}.$$

And, because $\mathbf{r}^{(k)} \perp \mathbf{p}^{(j)}$ for all $j < k$, we must have

$$0 = -\alpha_k \mathbf{p}^{(j)\top} \mathbf{Ap}^{(k)}.$$

In other words, to preserve optimality between iterates, $\mathbf{p}^{(k)}$ and $\mathbf{p}^{(j)}$ must be *A-conjugate* for all $j < k$:

$$\left(\mathbf{p}^{(j)}, \mathbf{p}^{(k)} \right)_\mathbf{A} \equiv \mathbf{p}^{(j)\top} \mathbf{Ap}^{(k)} = 0 \quad \text{for } j = 0, 1, \dots, k-1. \quad (5.10)$$

As before, let's take the first direction equal to the gradient at $\mathbf{x}^{(0)}$, i.e., $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$. Then each subsequent direction will be the A-conjugate to this gradient. Hence, the name *conjugate gradient method*.

It still makes sense to take the directions as close to the residuals as possible, yet still mutually conjugate. Take

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)} \quad (5.11)$$

where β_k is chosen so that

$$\left(\mathbf{p}^{(k+1)}, \mathbf{p}^{(j)} \right)_\mathbf{A} = 0, \quad j = 0, 1, \dots, k. \quad (5.12)$$

It may seem almost magical that by choosing $\mathbf{p}^{(k+1)}$ with explicit dependence only on $\mathbf{p}^{(k)}$ we ensure that $\mathbf{p}^{(k+1)}$ is A-conjugate to all $\mathbf{p}^{(j)}$ for $j = 0, 1, 2, \dots, k$. Let's see why it works. This mathematical sleight-of-hand will be made even more clear when we discuss Krylov subspaces in the next section. If we multiply (5.11) by $\mathbf{Ap}^{(k)}$ and enforce (5.12) when $j = k$, then it follows that

$$\beta_k = \frac{\mathbf{r}^{(k+1)\top} \mathbf{Ap}^{(k)}}{\mathbf{p}^{(k)\top} \mathbf{Ap}^{(k)}} = \frac{\left(\mathbf{p}^{(k)}, \mathbf{r}^{(k+1)} \right)_\mathbf{A}}{\left(\mathbf{p}^{(k)}, \mathbf{p}^{(k)} \right)_\mathbf{A}}.$$

Now, let's show that this choice for β_k ensures that (5.12) holds for all $j < k$. Let

$$V_k = \text{span}\{\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \dots, \mathbf{p}^{(k-1)}\}.$$

Since $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$, it follows from (5.11) that

$$V_k = \text{span}\{\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \dots, \mathbf{r}^{(k-1)}\}.$$

By our optimality condition we must have $V_k \perp \mathbf{p}^{(k+1)}$. Also,

$$\begin{aligned} \mathbf{p}^{(k)} &= \mathbf{r}^{(k)} - \beta_{k-1} \mathbf{p}^{(k-1)} \\ &= (\mathbf{b} - \mathbf{Ax}^{(k)}) - \beta_{k-1} \mathbf{p}^{(k-1)} \\ &= \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k-1)} + \alpha_k \mathbf{p}^{(k-1)}) - \beta_{k-1} \mathbf{p}^{(k-1)} \\ &= \mathbf{r}^{(k-1)} - \alpha_k \mathbf{Ap}^{(k-1)} - \beta_{k-1} \mathbf{p}^{(k-1)} \end{aligned}$$

So, $\mathbf{Ap}^{(k-1)} \in V_{k+1}$ and hence $\mathbf{Ap}^{(j)} \in V_{k+1}$ for all $j = 0, 1, \dots, k-1$. Therefore, $\mathbf{Ap}^{(j)} \perp \mathbf{p}^{(k+1)}$ or equivalently $(\mathbf{p}^{(k+1)}, \mathbf{p}^{(j)})_{\mathbf{A}} = 0$ for $j = 0, 1, \dots, k-1$. Note that we are performing Arnoldi iteration to generate an A-conjugate set of directions.

Let's summarize the conjugate gradient method by explicitly writing out its algorithm. Compare the following algorithm with the one for the gradient method on page 132.

```

make an initial guess x
p ← r ← b - Ax
for k = 0, 1, 2, ...
  α ← r^T p / p^T Ap
  x ← x + α p
  r ← r - α Ap
  if ||r|| < tolerance, then we're done
  β ← r^T Ap / p^T Ap
  p ← r - β p

```

The conjugate gradient method is actually a direct method. If \mathbf{A} is an $n \times n$ symmetric, positive-definite matrix, then the conjugate gradient method yields an exact solution after at most n steps. But when n is large, it is usually unnecessary to run the method for the full n iterations. Instead, we typically consider the conjugate gradient an iterative method and stop after significantly fewer than n iterations. In particular, the conjugate gradient method converges very quickly if the condition number is close to one. We can also use a symmetric, positive-definite preconditioner \mathbf{P} to reduce the condition number $\kappa(\mathbf{P}^{-1} \mathbf{A}) < \kappa(\mathbf{A})$. For a simple Jacobi preconditioner (diagonal scaling), we take $\mathbf{P} = \mathbf{D}$. The algorithm for the preconditioned conjugate gradient method replaces

$$\begin{aligned}\beta &\leftarrow \mathbf{r}^T \mathbf{A} \mathbf{p} / \mathbf{p}^T \mathbf{A} \mathbf{p} & \text{with} \\ \mathbf{p} &\leftarrow \mathbf{r} - \beta \mathbf{p} & \text{solve } \mathbf{Pz} = \mathbf{r} \\ && \beta \leftarrow \mathbf{z}^T \mathbf{A} \mathbf{p} / \mathbf{p}^T \mathbf{A} \mathbf{p} \\ && \mathbf{p} \leftarrow \mathbf{z} - \beta \mathbf{p}\end{aligned}$$

in the conjugate gradient method above.

• The IterativeSolvers.jl function `cg` implements the conjugate gradient method—preconditioned conjugate gradient method if a preconditioner is also provided.

5.5 Krylov methods

The iterative methods discussed in this chapter can be viewed as Krylov methods. Consider a classical iterative method such as Jacobi or Gauss–Seidel with

$$\mathbf{Px}^{(k+1)} = \mathbf{Nx}^{(k)} + \mathbf{b}.$$

The error

$$\mathbf{e}^{(k)} = \mathbf{P}^{-1} \mathbf{N} \mathbf{e}^{(k-1)} = (\mathbf{I} - \mathbf{P}^{-1} \mathbf{A}) \mathbf{e}^{(k-1)},$$

and so

$$\mathbf{e}^{(k)} = (\mathbf{I} - \mathbf{P}^{-1} \mathbf{A})^k \mathbf{e}^{(0)}.$$

Then

$$\begin{aligned}\mathbf{x}^{(k)} - \mathbf{x}^{(0)} &= \mathbf{e}^{(k)} - \mathbf{e}^{(0)} = [(\mathbf{I} - \mathbf{P}^{-1} \mathbf{A})^k - \mathbf{I}] \mathbf{e}^{(0)} \\ &= \left[\sum_{j=1}^k \binom{k}{j} (-1)^j (\mathbf{P}^{-1} \mathbf{A})^j \right] \mathbf{e}^{(0)}.\end{aligned}$$

This says that $\mathbf{x}^{(k)} - \mathbf{x}^{(0)}$ lies in the Krylov space

$$\mathcal{K}_k(\mathbf{P}^{-1} \mathbf{A}, \mathbf{z}) = \text{span} \{ \mathbf{z}, \mathbf{P}^{-1} \mathbf{A} \mathbf{z}, \dots, (\mathbf{P}^{-1} \mathbf{A})^k \mathbf{z} \}$$

with $\mathbf{z} = \mathbf{P}^{-1} \mathbf{A} \mathbf{e}^{(0)} = \mathbf{P}^{-1} \mathbf{r}^{(0)}$. The gradient descent method is also a Krylov method. Equation (5.9) says that the residuals are elements of the Krylov subspace

$$\mathcal{K}_k(\mathbf{A}, \mathbf{r}^{(0)}) = \text{span} \{ \mathbf{r}^{(0)}, \mathbf{A} \mathbf{r}^{(0)}, \dots, \mathbf{A}^k \mathbf{r}^{(0)} \}.$$

In the previous sections, we considered a general iterative method

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} = \mathbf{x}^{(0)} + \sum_{j=1}^k \alpha_j \mathbf{p}^{(j)}$$

for some directions $\{\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \dots, \mathbf{p}^{(k)}\}$. For the gradient descent method, we took the directions along residuals $\mathbf{p}^{(j)} = \mathbf{r}^{(j)}$. For the conjugate gradient method, we showed that the directions formed a subspace

$$V_{k+1} = \text{span} \{ \mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \dots, \mathbf{p}^{(k)} \} = \text{span} \{ \mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \dots, \mathbf{r}^{(k)} \}.$$

Furthermore, we noted that the V_{k+1} is actually a Krylov subspace

$$V_{k+1} = \text{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}, \dots, \mathbf{A}^k\mathbf{r}^{(0)}\}.$$

In this section, we'll expand on the idea of using a Krylov subspace to generate an iterative method $\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \boldsymbol{\delta}^{(k)}$, where $\boldsymbol{\delta}^{(k)} \in \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{r}^{(0)})$ with an initial guess $\mathbf{x}^{(0)}$. Let's consider two methods

1. Choose $\mathbf{x}^{(k+1)}$ that minimizes the error $\|\mathbf{e}^{(k+1)}\|_{\mathbf{A}}$.
2. Choose $\mathbf{x}^{(k+1)}$ that minimizes the residual $\|\mathbf{r}^{(k+1)}\|_2$.

The first approach is often called the Arnoldi method or the Full Orthogonalization Method (FOM). When \mathbf{A} is a symmetric, positive-definite matrix, FOM is equivalent to the conjugate gradient method. The second approach is known as the general minimal residual method (GMRES). It is called simply the minimal residual method (MINRES) when \mathbf{A} is symmetric.

Before looking at either of the two methods, let's recall the Arnoldi process from the previous chapter. The Arnoldi process is a variation of the Gram–Schmidt method that generates an orthogonal basis for a Krylov subspace. By choosing an orthogonal basis, we ensure that the system is well-conditioned. Starting with the residual $\mathbf{r}^{(0)}$, take

$$\mathbf{q}^{(1)} \leftarrow \mathbf{r}^{(0)} / \|\mathbf{r}^{(0)}\|_2$$

and set $h_{11} = \|\mathbf{r}^{(0)}\|_2$. At the k th iteration take

$$\begin{aligned} h_{ik} &= (\mathbf{q}^{(i)}, \mathbf{A}\mathbf{q}^{(k)}) \quad \text{for } i = 1, 2, \dots, k, \\ \mathbf{q}^{(k+1)} &= \mathbf{A}\mathbf{q}^{(k)} - \sum_{i=1}^k h_{ik} \mathbf{q}^{(k)}, \quad \text{and set} \\ \mathbf{q}^{(k+1)} &\leftarrow \mathbf{q}^{(k+1)} / h_{k+1,k} \quad \text{where } h_{k+1,k} = \|\mathbf{q}^{(k+1)}\|. \end{aligned}$$

We can summarize the method pictorially as

$$\underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{Q}_k} = \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{Q}_{k+1}} \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{H}_{k+1,k}} \quad (5.13)$$

By separating the bottom row of $\mathbf{H}_{k+1,k}$ and the last column of \mathbf{Q}_{k+1} , we have

$$\mathbf{AQ}_k = \mathbf{Q}_k \mathbf{H}_k + h_{k+1,k} \mathbf{q}^{(k+1)} \boldsymbol{\xi}_k^T \quad (5.14)$$

where $\boldsymbol{\xi}_k = (0, \dots, 0, 1)$. Note that if \mathbf{A} is Hermitian or real symmetric, then \mathbf{H}_k is tridiagonal. In this case, the Arnoldi method simplifies and is called the Lanczos method.

► Full orthogonalization method (FOM)

Let's examine the first Krylov method, in which we choose the $\mathbf{x}^{(k+1)}$ that minimizes the error $\|\mathbf{e}^{(k+1)}\|_{\mathbf{A}}$. At the k th iteration, we choose the approximation $\mathbf{x}^{(k+1)}$ with $\mathbf{x}^{(k+1)} - \mathbf{x}^{(0)} \in \mathcal{K}_k(\mathbf{A}, \mathbf{r}^{(0)})$ to minimize the error in the energy norm

$$\|\mathbf{e}^{(k+1)}\|_{\mathbf{A}}^2 = \mathbf{e}^{(k+1)\top} \mathbf{A} \mathbf{e}^{(k+1)}.$$

Any vector in the Krylov subspace can be expressed as a linear combination of the basis elements $\mathbf{Q}_k \mathbf{z}^{(k)}$ for some $\mathbf{z}^{(k)}$. Take

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \mathbf{Q}_k \mathbf{z}^{(k)}$$

with $\mathbf{z}^{(k)}$ to be determined. Then because $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$, we have

$$\mathbf{e}^{(k+1)} = \mathbf{e}^{(0)} - \mathbf{Q}_k \mathbf{z}^{(k)}.$$

So, we need to find the vector $\mathbf{z}^{(k)}$ that minimizes

$$\|\mathbf{e}^{(0)} - \mathbf{Q}_k \mathbf{z}^{(k)}\|_{\mathbf{A}}^2 = (\mathbf{e}^{(0)} - \mathbf{Q}_k \mathbf{z}^{(k)})^\top \mathbf{A} (\mathbf{e}^{(0)} - \mathbf{Q}_k \mathbf{z}^{(k)}),$$

which happens when the gradient is zero:

$$2(\mathbf{Q}_k^\top \mathbf{A} \mathbf{Q}_k \mathbf{z}^{(k)} - \mathbf{Q}_k^\top \mathbf{A} \mathbf{e}^{(0)}) = \mathbf{0}.$$

Hence,

$$\mathbf{Q}_k^\top \mathbf{A} \mathbf{Q}_k \mathbf{z}^{(k)} = \mathbf{Q}_k^\top \mathbf{A} \mathbf{e}^{(0)} = \mathbf{Q}_k^\top \mathbf{r}^{(0)}.$$

From (5.14) it follows that $\mathbf{Q}_k^\top \mathbf{A} \mathbf{Q}_k = \mathbf{H}_k$ because $\mathbf{Q}_k \perp \mathbf{q}^{(k+1)}$. Also, because $\mathbf{r}^{(0)} = \|\mathbf{r}^{(0)}\|_2 \mathbf{q}^{(1)}$ and $\mathbf{Q}_k^\top \mathbf{q}^{(1)} = \xi_1$, it follows that $\mathbf{z}^{(k)}$ must solve the upper Hessenberg system

$$\mathbf{H}_k \mathbf{z}^{(k)} = \|\mathbf{r}^{(0)}\|_2 \xi_1$$

where $\xi_1 = (1, 0, \dots, 0)$. Note that the norm of the residual

$$\|\mathbf{r}^{(k)}\|_2 = \|\mathbf{b} - \mathbf{A} \mathbf{x}^{(k)}\|_2 = |h_{k+1,k}| \cdot |\xi_k^\top \mathbf{z}^{(k)}| = |h_{k+1,k}| \cdot |\zeta_k^{(k)}|,$$

which we can use as a stopping criterion when the value is less than $\varepsilon \|\mathbf{r}^{(0)}\|_2$ for some tolerance ε .

It may not be obvious, but FOM outlined above is just the conjugate gradient method when \mathbf{A} is a symmetric, positive-definite matrix. Start with

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \mathbf{Q}_k \mathbf{z}^{(k)} = \mathbf{x}^{(0)} + \mathbf{Q}_k \mathbf{H}_k^{-1} \|\mathbf{r}^{(0)}\|_2 \xi_k.$$

Taking the LU decomposition of \mathbf{H}_k , we have

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \mathbf{Q}_k \mathbf{U}_k^{-1} \mathbf{L}_k^{-1} \|\mathbf{r}^{(0)}\|_2 \xi_k = \mathbf{x}^{(0)} + \mathbf{P}_k \mathbf{L}_k^{-1} \|\mathbf{r}^{(0)}\|_2 \xi_k$$

where the columns of \mathbf{P}_k are the directions $\mathbf{p}^{(k)}$. For a symmetric matrix \mathbf{A} , we have

$$\underbrace{\mathbf{P}_k^T \mathbf{A} \mathbf{P}_k}_{\text{symmetric}} = \underbrace{\mathbf{U}_k^{-T} \mathbf{Q}_k^T \mathbf{A} \mathbf{Q}_k \mathbf{U}_k^{-1}}_{\text{symmetric}} = \mathbf{U}_k^T \mathbf{H}_k \mathbf{U}_k^{-1} = \underbrace{\mathbf{U}_k^{-T} \mathbf{L}_k}_{\text{lower triangular}}.$$

Hence, $\mathbf{P}_k^T \mathbf{A} \mathbf{P}_k$ is a diagonal matrix, and it follows that

$$(\mathbf{p}^{(i)}, \mathbf{A} \mathbf{p}^{(j)})_2 = (\mathbf{p}^{(i)}, \mathbf{p}^{(j)})_{\mathbf{A}} = 0 \quad \text{for } i \neq j,$$

which says that the directions are A-conjugate.

► General minimized residual (GMRES)

This time we will choose $\mathbf{x}^{(k+1)}$ that minimizes the residual $\|\mathbf{r}^{(k+1)}\|_2$. As before, we take

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \mathbf{Q}_k \mathbf{z}^{(k)}$$

for some vector $\mathbf{z}^{(k)}$ to be determined. From $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$, we have

$$\begin{aligned} \mathbf{r}^{(k+1)} &= \mathbf{r}^{(0)} - \mathbf{A}\mathbf{Q}_k \mathbf{z}^{(k)} \\ &= \mathbf{Q}_{k+1} \mathbf{Q}_{k+1}^T \mathbf{r}^{(0)} - \mathbf{A}\mathbf{Q}_k \mathbf{z}^{(k)} \\ &= \mathbf{Q}_{k+1} \left(\|\mathbf{r}^{(0)}\|_2 \boldsymbol{\xi}_1 - \mathbf{H}_{k+1,k} \mathbf{z}^{(k)} \right) \end{aligned}$$

where $\boldsymbol{\xi}_1 = (1, 0, \dots, 0)$. We now have the problem of finding $\mathbf{z}^{(k)}$ to minimize

$$\|\mathbf{r}^{(k+1)}\|_2 = \left\| \|\mathbf{r}^{(0)}\|_2 \boldsymbol{\xi}_1 - \mathbf{H}_{k+1,k} \mathbf{z}^{(k)} \right\|_2.$$

The solution to this problem is the solution to the overdetermined system

$$\mathbf{H}_{k+1,k} \mathbf{z}^{(k)} = \|\mathbf{r}^{(0)}\|_2 \boldsymbol{\xi}_1,$$

which we can solve using QR decomposition. Because $\mathbf{H}_{k+1,k}$ is upper Hessenberg, we can use a series of Givens rotations to zero out the subdiagonal elements, giving us

$$\hat{\mathbf{R}} \mathbf{z}^{(k)} = \hat{\mathbf{Q}}^T \|\mathbf{r}^{(0)}\|_2 \boldsymbol{\xi}_1,$$

where $\hat{\mathbf{Q}}$ is an orthogonal matrix and $\hat{\mathbf{R}}$ is an upper triangular matrix. The residual $\|\mathbf{r}^{(k+1)}\|_2$, given by the bottom element of this expression, is

$$\|\mathbf{r}^{(0)}\|_2 (\boldsymbol{\xi}_1^T \hat{\mathbf{Q}} \boldsymbol{\xi}_{k+1}) = \|\mathbf{r}^{(0)}\|_2 \hat{q}_{1,k+1}.$$

We stop when $\|\mathbf{r}^{(k)}\|_2 < \varepsilon \|\mathbf{r}^{(0)}\|_2$ for some tolerance ε we stop. Otherwise, we continue by finding the next basis vector $\mathbf{q}^{(k+2)}$ of the Krylov subspace using the Arnoldi process.

● The `IterativeSolvers.jl` function `gmres` implements the generalized minimum residual method and `minres` implements the minimum residual method.

5.6 Exercises

5.1. Consider the 2-by-2 matrix

$$\mathbf{A} = \begin{bmatrix} 1 & \sigma \\ -\sigma & 1 \end{bmatrix}.$$

Under what conditions will Gauss–Seidel converge? For what range ω will the SOR method converge? What is the optimal choice of ω ?

5.2. Show that the conjugate gradient method requires no more than $k + 1$ steps to converge for a symmetric, positive-definite matrix with k distinct eigenvalues.

5.3. Consider the three-dimensional Poisson equation $-\Delta u(x, y, z) = f(x, y, z)$ with Dirichlet boundary conditions $u(x, y, z) = 0$ over the unit cube. Just as in the two-dimensional case, we can use a finite difference approximation for the minus Laplacian $-\partial^2 u / \partial x^2 - \partial^2 u / \partial y^2 - \partial^2 u / \partial z^2$. But instead of a five-point stencil, now we'll use a seven-point stencil

$$\frac{6u_{ijk} - u_{i-1,j,k} - u_{i+1,j,k} - u_{i,j-1,k} - u_{i,j+1,k} - u_{i,j,k-1} - u_{i,j,k+1}}{h^2}.$$

Suppose that we discretize space using 50 points in each dimension. The finite difference matrix \mathbf{A} to the problem $\mathbf{Ax} = \mathbf{b}$ is a 125000×125000 sparse matrix. Even though \mathbf{A} has over 15 billion elements, only 0.005 percent of them are nonzero—still, it has almost a million nonzero elements.

Consider the source term

$$f(x, y, z) = (x - x^2)(y - y^2) + (x - x^2)(z - z^2) + (y - y^2)(z - z^2).$$

In this case, the finite difference method will produce an exact solution

$$u(x, y, z) = \frac{1}{2}(x - x^2)(y - y^2)(z - z^2).$$

Implement the finite difference scheme using Jacobi, Gauss–Seidel, SOR, and conjugate gradient methods starting with an initial solution $u(x, y, z) \equiv 0$. Take $\omega = 1.9$ for the SOR method. Plot the error of the methods for the first 500 iterations and comment on the convergence. Hint: An easy way to build the finite difference matrix is by using a Kronecker tensor product. The Kronecker tensor product is defined as

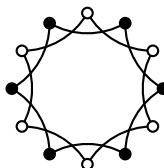
$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}.$$

The seven-point finite difference operator is $\mathbf{D} \otimes \mathbf{I} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{D} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{I} \otimes \mathbf{D}$ where \mathbf{D} is a 50×50 discrete Laplacian (5.6) and \mathbf{I} is a 50×50 identity matrix.

- 5.4. Let \mathbf{A} be the 20000×20000 matrix whose entries are zero everywhere except for the primes $2, 3, 5, 7, \dots, 224737$ along the main diagonal and the number 1 in all the positions a_{ij} with $|i - j| = 1, 2, 4, 8, \dots, 16384$. What is the $(1, 1)$ -entry of \mathbf{A}^{-1} ? (This problem was proposed by Nick Trefethen as part of his “hundred-dollar, hundred-digit challenge.”)

Chapter 6

The Fast Fourier Transform



It's hard to overstate the impact that the fast Fourier transform (FFT) has had in science and technology. It has been called "the most important numerical algorithm of our lifetime" by prominent mathematician Gilbert Strang, and it is invariably included in top-ten lists of algorithms. The FFT is an essential tool for signal processing, data compression, and partial differential equations. It is used in technologies as varied as digital media, medical imaging, and stock market analysis. At its most basic level, the FFT is a recursive implementation of the discrete Fourier transform. This implementation that puts the "fast" in fast Fourier transform takes what would normally be an $O(n^2)$ -operation method and makes it an $O(n \log_2 n)$ -operation one. In this chapter, we'll examine the algorithm and a few of its applications.

6.1 Discrete Fourier transform

Suppose that we want to find a polynomial $y = c_0 + c_1z + c_2z^2 + \cdots + c_{n-1}z^{n-1}$ that passes through the points $(z_j, y_j) \in \mathbb{C}^2$ for $j = 0, 1, \dots, n - 1$. In this case, we simply solve the system $\mathbf{V}\mathbf{c} = \mathbf{y}$ where \mathbf{V} is the Vandermonde matrix to get the coefficients c_j of the polynomial:

$$\begin{bmatrix} 1 & z_0 & z_0^2 & \cdots & z_0^{n-1} \\ 1 & z_1 & z_1^2 & \cdots & z_1^{n-1} \\ 1 & z_2 & z_2^2 & \cdots & z_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & z_{n-1} & z_{n-1}^2 & \cdots & z_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}.$$

Now, suppose that we restrict the $\{z_j\}$ to equally spaced points on the unit circle. That is, take

$$z_j = e^{-i2\pi j/n} = \omega_n^j$$

by choosing nodes clockwise around the unit circle starting with $z_0 = 1$. The value $\omega_n = \exp(-i2\pi/n)$ is called an *n*th root of unity because it solves the equation $z^n = 1$. (In fact, ω_n^k are all *n*th roots of unity because they all are solutions to $z^n = 1$.) In this case, the Vandermonde matrix is

$$\mathbf{F}_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}. \quad (6.1)$$

The system $\mathbf{y} = \mathbf{F}_n \mathbf{c}$ expressed in index notation is

$$y_k = \sum_{j=0}^{n-1} c_j e^{-i2\pi j k / n} = \sum_{j=0}^{n-1} c_j \omega_n^{jk}.$$

The matrix \mathbf{F}_n , called the *discrete Fourier transform* (DFT), is a scaled unitary matrix. That is, $\mathbf{F}_n^\text{H} \mathbf{F}_n = n \mathbf{I}$. This fact follows from a simple application of the geometric series. Recall that if $z \neq 1$, then

$$\sum_{j=0}^{n-1} z^j = \frac{z^n - 1}{z - 1}.$$

Computing $\mathbf{F}_n^\text{H} \mathbf{F}_n$,

$$\sum_{j=0}^{n-1} \omega_n^{jk} \bar{\omega}_n^{jl} = \sum_{j=0}^{n-1} \omega_n^{j(k-l)} = \begin{cases} \frac{\omega_n^{n(k-l)} - 1}{\omega_n^{k-l} - 1} = \frac{1 - 1}{\omega_n^{k-l} - 1} = 0, & \text{when } k \neq l \\ \sum_{j=0}^{n-1} 1 = n, & \text{when } k = l \end{cases}$$

where $\bar{\omega}$ is the complex conjugate of ω .

It also follows that the *inverse discrete Fourier transform* (IDFT) is simply $\mathbf{F}_n^{-1} = \mathbf{F}_n^\text{H} / n = \bar{\mathbf{F}}_n / n$. The DFT is symmetric but not Hermitian. Unlike the usual Vandermonde matrix which is often ill-conditioned for large n , the DFT is perfectly conditioned because it is a scaled unitary matrix.

The k th column (and row) of \mathbf{F}_n forms a subgroup generated by ω_n^k . That is, $\{kj \mid j = 0, 1, 2, \dots, n-1\}$ forms a group under addition modulo n . For example, taking $n = 12$ using 2 as a generator, we have

$$\{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24\} \pmod{12} \equiv \{2, 4, 6, 8, 10, 0\}.$$

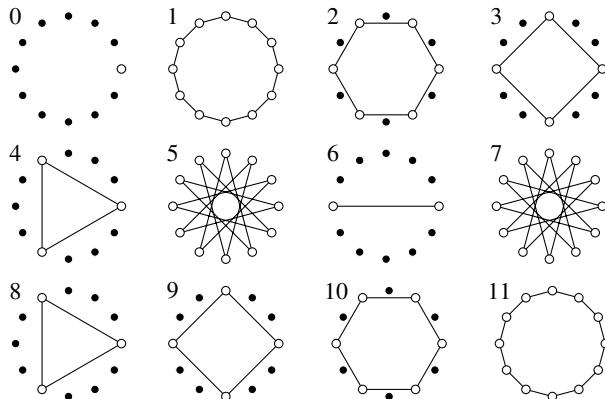


Figure 6.1: Subgroups using the generator ω_{12}^j for $j = 0, 1, \dots, 11$.

See Figure 6.1 above. Using 10 as a generator gives us the same group, because $10 \equiv -2 \pmod{12}$. In particular, note that this subgroup corresponds to the six 6th roots of unity. That is, the subgroup generated by ω_{12}^2 equals the group generated by ω_6^1 .

If the generator k is relatively prime with respect to n , i.e., if k and n are coprime, then the number of elements of the group is n . We say the group has order n . For example, both 5 and 7 are coprime to 12, and they generate subgroups of order 12. Otherwise, k generates a subgroup of order $n/\gcd(k, n)$.

Associated with the subgroup $\{0, 2, 4, 6, 8, 10\}$ is a *coset* that is formed by taking each element of the group and adding one. So,

$$\{0, 2, 4, 6, 8, 10\} \text{ has the coset } \{1, 3, 5, 7, 9, 11\}.$$

Equivalently, for $\omega \equiv \omega_{12}$

$$\{1, \omega^2, \omega^4, \omega^6, \omega^8, \omega^{10}\} \text{ times } \omega \text{ equals } \{\omega, \omega^3, \omega^5, \omega^7, \omega^9, \omega^{11}\}.$$

For $k = 3$, the group $\{0, 3, 6, 9\}$ has three cosets: itself, $\{1, 4, 7, 10\}$, and $\{2, 5, 8, 11\}$. For $k = 5$, there is only one coset. The number of cosets of the subgroup generated by k under addition modulo n equals $\gcd(k, n)$. The number of cosets determines the radix which is the basis for Cooley–Tukey algorithm.

6.2 Cooley–Tukey algorithm

There are several FFT algorithms. Each works by taking advantage of the group structure of the DFT matrix. The prototypical one, the Cooley–Tukey algorithm, was developed by James Cooley and John Tukey in 1965—actually a rediscovery

of an unpublished discovery by Gauss in 1805. The development of their FFT was prompted by physicist Richard Garwin, designer of the first hydrogen bomb and researcher at IBM Watson laboratory, who at the time was interested in verifying the Soviet Union's compliance to the Nuclear Test Ban Treaty. Because the Soviet Union would not agree to inspections within their borders, Garwin turned to remote seismic monitoring of nuclear explosions. But, time series analysis of the off-shore seismological sensors would require a fast algorithm for computing the DFT, which normally took $O(n^2)$ operations. Garwin prompted Cooley to develop such an algorithm along with Tukey. And with a few months of effort, they had developed a recursive algorithm that took only $O(n \log_2 n)$ operations. When the sensors were finally deployed, they were able to locate nuclear explosions to within 15 kilometers.

In this section, we look at the Cooley–Tukey radix-2 (base-2) algorithm. Consider a DFT whose size n is divisible by two:

$$y_j = \sum_{k=0}^{n-1} \omega_n^{kj} c_k = \underbrace{\sum_{k=0}^{n/2-1} \omega_n^{2kj} c_{2k}}_{\text{even index}} + \underbrace{\sum_{k=0}^{n/2-1} \omega_n^{(2k+1)j} c_{2k+1}}_{\text{odd index}}.$$

Let $m = n/2$. Since $\omega_n^2 = \omega_m$, we have

$$y_j = \sum_{k=0}^{m-1} \omega_m^{kj} c'_k + \omega_n^j \sum_{k=0}^{m-1} \omega_m^{kj} c''_k$$

where $c'_k = c_{2k}$ and $c''_k = c_{2k+1}$. So, the output can be computed by using the sum of two smaller DFTs of size $m = n/2$:

$$y_j = y'_j + \omega_n^j y''_j.$$

We only need to compute $j = 0, 1, \dots, m-1$ because $\omega_m^{k(m+j)} = \omega_m^{km} \omega_m^{kj} = \omega_m^{kj}$ and $\omega_n^{m+j} = \omega_n^m \omega_n^j = -\omega_n^j$. So

$$y_{m+j} = y'_j - \omega_n^j y''_j.$$

Therefore,

$$\begin{cases} y_j = y'_j + \omega_n^j y''_j & \text{for } j = 0, 1, \dots, m-1. \\ y_{m+j} = y'_j - \omega_n^j y''_j \end{cases} \quad (6.2)$$

To get a better picture of the math behind the algorithm, let's look at the matrix formulation of the radix-2 FFT. The $n = 4$ DFT is

$$\mathbf{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & -i \end{bmatrix}.$$

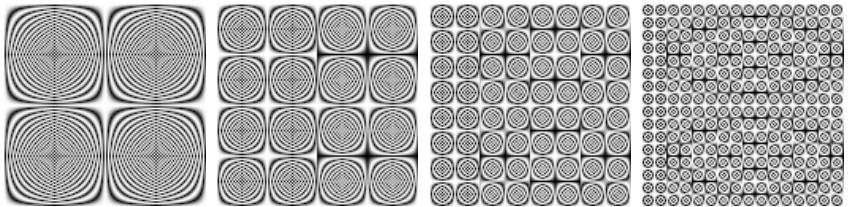


Figure 6.2: \mathbf{F}_{128} (left), after one permutation, after a second permutation, and after a third permutation (right) of the columns.

Consider the even-odd permutation matrix (a reverse perfect shuffle¹)

$$\mathbf{P}_4^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Then

$$\mathbf{F}_4 \mathbf{P}_4^{-1} = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -i & i \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & i & -i \end{array} \right] = \left[\begin{array}{cc} \mathbf{F}_2 & \mathbf{\Omega}_2 \mathbf{F}_2 \\ \mathbf{F}_2 & -\mathbf{\Omega}_2 \mathbf{F}_2 \end{array} \right] = \left[\begin{array}{cc} \mathbf{I}_2 & \mathbf{\Omega}_2 \\ \mathbf{I}_2 & -\mathbf{\Omega}_2 \end{array} \right] \left[\begin{array}{cc} \mathbf{F}_2 & \\ & \mathbf{F}_2 \end{array} \right],$$

where

$$\mathbf{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{\Omega}_2 = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \text{diag}(1, \omega_4)$$

The matrix

$$\mathbf{B}_4 = \begin{bmatrix} \mathbf{I}_2 & \mathbf{\Omega}_2 \\ \mathbf{I}_2 & -\mathbf{\Omega}_2 \end{bmatrix}$$

is called the *butterfly matrix*. The DFT is then

$$\mathbf{F}_4 = \mathbf{B}_4 \begin{bmatrix} \mathbf{F}_2 & \\ & \mathbf{F}_2 \end{bmatrix} \mathbf{P}_4 = \mathbf{B}_4 \begin{bmatrix} \mathbf{F}_2 \mathbf{P}_2^{-1} & \\ & \mathbf{F}_2 \mathbf{P}_2^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{P}_2 & \\ & \mathbf{P}_2 \end{bmatrix} \mathbf{P}_4 = \mathbf{B}_4 \begin{bmatrix} \mathbf{B}_2 & \\ & \mathbf{B}_2 \end{bmatrix} \mathbf{P}$$

where \mathbf{P} is the permutation matrix of successive even/odd rearrangements. The pattern reveals itself for larger matrices as well. See the figure above, which

¹To dig deeper into shuffling, see the article “The mathematics of perfect shuffles” by mathematician Persi Diaconis and others. When he was fourteen, Diaconis dropped out of school and ran off to follow a legendary sleight-of-hand magician. After learning that probability could improve his card skills and that mathematical analysis was necessary to understand probability, Diaconis took up mathematics and later went on to earn a PhD in mathematical statistics from Harvard.

shows the first three permutations acting on the DFT matrix \mathbf{F}_{128} . Note the emergence of \mathbf{F}_{64} , \mathbf{F}_{32} and \mathbf{F}_{16} with each permutation.

The *Kronecker product* of a $p \times q$ matrix \mathbf{A} and an $r \times s$ matrix \mathbf{B} is a $pr \times qs$ matrix equal to

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1q}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_p\mathbf{B} & \cdots & a_{pq}\mathbf{B} \end{bmatrix}.$$

Then for $n = 4$,

$$\mathbf{F}_4 = (\mathbf{I}_1 \otimes \mathbf{B}_4)(\mathbf{I}_2 \otimes \mathbf{B}_2)\mathbf{P}.$$

And in general,

$$\mathbf{F}_n = (\mathbf{I}_1 \otimes \mathbf{B}_n)(\mathbf{I}_2 \otimes \mathbf{B}_{n/2}) \cdots (\mathbf{I}_{n/2} \otimes \mathbf{B}_2)\mathbf{P} \quad (6.3)$$

where

$$\mathbf{B}_{2n} = \begin{bmatrix} \mathbf{I}_n & \mathbf{\Omega}_n \\ \mathbf{I}_n & -\mathbf{\Omega}_n \end{bmatrix},$$

the diagonal matrix $\mathbf{\Omega} = \text{diag}(1, \bar{\omega}_{2n}, \bar{\omega}_{2n}^2, \dots, \bar{\omega}_{2n}^{n-1})$, and the permutation matrix

$$\mathbf{P} = (\mathbf{I}_{n/4} \otimes \mathbf{P}_4) \cdots (\mathbf{I}_2 \otimes \mathbf{P}_{n/2})(\mathbf{I}_1 \otimes \mathbf{P}_n).$$

 The function `kron(A, B)` returns the Kronecker tensor product $\mathbf{A} \otimes \mathbf{B}$.

Directly computing the DFT of an n dimensional vector requires $2n^2$ multiplications and additions. We can instead compute it with (6.2) using two DFTs of length $m = n/2$ and an additional $n/2$ multiplications and n additions. That is, a total of $\frac{1}{2}n^2 + \frac{3}{2}n$ operations. But we're not finished yet. If n is a power of two, then we can continue by applying (6.2) this idea recursively, continuing subdividing the nodes in half. For $n = 2^p$, the number of multiplications and additions is

$$\begin{aligned} M_p &= 2M_{p-1} + 2^{p-1} \\ A_p &= 2A_{p-1} + 2^p. \end{aligned}$$

The solution to this difference equation is

$$\begin{aligned} M_p &= p2^{p-1} = \frac{1}{2}n \log_2 n \\ A_p &= p2^p = n \log_2 n. \end{aligned}$$

So the number of operations is $O(\frac{3}{2}n \log_2 n)$. This means that a thousand-point FFT is roughly a hundred times faster than a direct computation $O(2n^2)$. Similar, algorithms work for n power of three, five, etc. (radix-3, radix-5, etc.).

Using (6.1) and (6.2), the radix-2 algorithm can be written as a recursive function in Julia for a column vector \mathbf{c} :

```

function fftx2(c)
n = length(c)
ω = exp(-2im*π/n)
if mod(n,2) == 0
    k = collect(0:n/2-1)
    u = fftx2(c[1:2:n-1])
    v = (ω.^k).*fftx2(c[2:2:n])
    return([u+v; u-v])
else
    k = collect(0:n-1)
    F = ω.^(k*k')
    return(F*c)
end
end

```

The IFFT can be adapted from the FFT using the identity $\mathbf{F}_n^{-1} = \bar{\mathbf{F}}_n/n$:

```

ifftx2(y) = conj(fftx2(conj(y)))/length(y);

```

The radix-2 FFT works by recursively breaking the DFT matrix into smaller and smaller pieces. If n is a power of two or a composite of small primes such as $7200 = 2^5 \times 3^2 \times 5^2$, the recursive algorithm is efficient. But if n is a large prime number or a composite with a large prime number, such as $7060 = 2^2 \times 5 \times 353$, the recursion breaks down. In this case, other algorithms must be used. We'll examine two such algorithms in section 6.4.

6.3 Toeplitz and circulant matrices

A *Toeplitz matrix* is a square matrix that is constant along each of the descending diagonals, e.g..

$$\mathbf{T} = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & a_{-2} & \cdots & a_{-(n-2)} \\ a_2 & a_1 & a_0 & a_{-1} & \ddots & a_{-(n-3)} \\ \vdots & a_2 & a_1 & a_0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & a_{-1} \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_1 & a_0 \end{bmatrix}$$

for some values $\{a_{-(n-1)}, a_{-(n-2)}, \dots, a_0, \dots, a_{(n-1)}\}$.

 The `ToeplitzMatrices.jl` function `Toeplitz` constructs a Toeplitz matrix object.

A Toeplitz matrix is *circulant* if it has the form

$$\mathbf{C} = \begin{bmatrix} c_0 & c_{n-1} & c_{n-2} & \cdots & c_1 \\ c_1 & c_0 & c_{n-1} & c_{n-2} & \cdots & c_2 \\ c_2 & c_1 & c_0 & c_{n-1} & \ddots & c_3 \\ \vdots & c_2 & c_1 & c_0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ c_{n-1} & c_{n-2} & c_{n-3} & \cdots & c_1 & c_0 \end{bmatrix}$$

for some values $\{c_0, c_1, \dots, c_{n-1}\}$. Note that \mathbf{C} is a Krylov matrix formed by the downshift permutation matrix

$$\mathbf{R} = [\xi_2 \ \xi_2 \ \cdots \ \xi_n \ \xi_1] = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

That is,

$$\mathbf{C} = [\mathbf{c} \ \mathbf{R}\mathbf{c} \ \mathbf{R}^2\mathbf{c} \ \cdots \ \mathbf{R}^{n-1}\mathbf{c}] \text{ with } \mathbf{c} = [c_0 \ c_1 \ \cdots \ c_{n-1}]^\top.$$

The circulant matrix \mathbf{C} can be written as $\mathbf{C} = c_0\mathbf{I} + c_1\mathbf{R} + c_2\mathbf{R}^2 + \cdots + c_{n-1}\mathbf{R}^{n-1}$, and \mathbf{R} is itself a circulant matrix.

 The `ToeplitzMatrices.jl` function `Circulant` constructs a circulant matrix object.

Theorem 19. A circulant matrix \mathbf{C} , whose first column is \mathbf{c} , has the diagonalization $\mathbf{C} = \mathbf{F}^{-1} \operatorname{diag}(\mathbf{Fc}) \mathbf{F}$, where \mathbf{F} is the discrete Fourier transform.

Proof. Let \mathbf{F} be an $n \times n$ discrete Fourier transform matrix and \mathbf{R} an equally sized downshift permutation matrix. Let's denote \mathbf{F} using its elements as $[\omega^{ij}]$ where $\omega = e^{-2\pi i/n}$. Define the diagonal matrix $\mathbf{W} = \operatorname{diag}(1, \omega, \omega^2, \dots, \omega^{n-1})$. Then, $\mathbf{FR} = [\omega^{i(j+1)}] = [\omega^{ij}\omega^i]$ and $\mathbf{WF} = [\omega^i\omega^{ij}]$. So, $\mathbf{R} = \mathbf{F}^{-1}\mathbf{WF}$. For $\mathbf{c} = (c_0, c_1, \dots, c_{n-1})$,

$$\mathbf{C} = c_0\mathbf{I} + c_1\mathbf{R} + c_2\mathbf{R}^2 + \cdots + c_{n-1}\mathbf{R}^{n-1}$$

and it follows that

$$\begin{aligned} \mathbf{C} &= \mathbf{F}^{-1} \left(c_0\mathbf{I} + c_1\mathbf{W} + c_2\mathbf{W}^2 + \cdots + c_{n-1}\mathbf{W}^{n-1} \right) \mathbf{F} \\ &= \mathbf{F}^{-1} \operatorname{diag} \left(\sum_{j=0}^{n-1} c_j, \sum_{j=0}^{n-1} c_j\omega^j, \dots, \sum_{j=0}^{n-1} c_j\omega^{j(n-1)} \right) \mathbf{F} \end{aligned}$$

Hence, $\mathbf{C} = \mathbf{F}^{-1} \operatorname{diag}(\mathbf{Fc}) \mathbf{F}$. □

► Fast convolution

The convolution of two functions or arrays can be described as their shifted inner products. For continuous functions $f(t)$ and $g(t)$, we have the convolution

$$(f * g)(t) = \int_{-\infty}^{\infty} f(s)g(t-s) ds.$$

When $\int_{-\infty}^{\infty} f(s) ds = 1$, you can think of $f * g$ as a moving average of g using f as a window function. For arrays \mathbf{u} and \mathbf{v} , the k th element of the circular convolution $\mathbf{u} * \mathbf{v}$ is simply

$$(\mathbf{u} * \mathbf{v})_k = \sum_{j=0}^{n-1} u_j v_{k-j \pmod n}.$$

Example. While the notion of a convolution may at first seem foreign to many, we are actually taught convolutions in grade school. Consider the product of 123 and 241. We compute the product using a convolution

$$\begin{array}{r} & 1 & 2 & 3 \\ \times & 2 & 4 & 1 \\ \hline & 1 & 2 & 3 \\ & 4 & 8 & 12 \\ 2 & 4 & 6 \\ \hline 2 & 8 & 15 & 14 & 3 \\ \hline 2 & 9 & 6 & 4 & 3 \end{array} \quad (\text{and carrying...})$$

So, $123 \times 241 = 29643$. ◀

The circular convolution of \mathbf{u} and \mathbf{v} is expressed in matrix notation as $\mathbf{u} * \mathbf{v} = \mathbf{C}_u \mathbf{v}$ where \mathbf{C}_u is the circulant matrix constructed from \mathbf{u} :

$$\begin{bmatrix} u_0 & u_4 & u_3 & u_2 & u_1 \\ u_1 & u_0 & u_4 & u_3 & u_2 \\ u_2 & u_1 & u_0 & u_4 & u_3 \\ u_3 & u_2 & u_1 & u_0 & u_4 \\ u_4 & u_3 & u_2 & u_3 & u_0 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}.$$

From theorem 19, $\mathbf{u} * \mathbf{v} = \mathbf{C}_u \mathbf{v} = \mathbf{F}^{-1} \text{diag}(\mathbf{F}\mathbf{u}) \mathbf{F}\mathbf{v}$. In practice, we would compute $\mathbf{u} * \mathbf{v}$ as $\mathbf{F}^{-1}(\mathbf{F}\mathbf{u} \circ \mathbf{F}\mathbf{v})$, where \circ denotes the Hadamard product, i.e., the element-wise product. This means that we can compute the convolution simply by computing the IDFT of the component-wise product of the DFTs of \mathbf{u} and \mathbf{v} . This evaluation requires computation of three FFTs. Instead of the $2n^2$ operations required to perform the convolution directly, it only takes $2n \log_2 n$ operations..

• The DSP.jl function `conv` returns the convolution of two n -dimensional arrays using either direct computation or FFTs depending on the size of the arrays.

► Fast Toeplitz multiplication

In the previous section, we saw that circulant matrix multiplication can be evaluated quickly using FFTs. We can do the same thing for a general Toeplitz matrix by padding it out to become a circulant matrix. For example,

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} a & d & e \\ b & a & d \\ c & b & a \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ * \\ * \end{bmatrix} = \begin{bmatrix} a & d & e & c & b \\ b & a & d & e & c \\ c & b & a & d & e \\ e & c & b & a & d \\ d & e & c & b & a \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ 0 \\ 0 \end{bmatrix}$$

where we discard the $*$ terms in the answer. In general, we need to pad out an $n \times n$ Toeplitz matrix by at least $n - 1$ rows and columns to create an $(2n - 1) \times (2n - 1)$ circulant matrix. It may often be more efficient to overpad the matrix to be a power of two or some other product of small primes. For example,

$$\mathbf{T} = \begin{bmatrix} a & d & e \\ b & a & d \\ c & b & a \end{bmatrix} \quad \rightarrow \quad \mathbf{C} = \begin{bmatrix} a & d & e & 0 & 0 & 0 & c & b \\ b & a & d & e & 0 & 0 & 0 & c \\ c & b & a & d & e & 0 & 0 & 0 \\ 0 & c & b & a & d & e & 0 & 0 \\ 0 & 0 & c & b & a & d & e & 0 \\ 0 & 0 & 0 & c & b & a & d & e \\ e & 0 & 0 & 0 & c & b & a & d \\ d & e & 0 & 0 & 0 & c & b & a \end{bmatrix}.$$

We never need to explicitly construct the full matrix \mathbf{C} —instead, we perform a fast convolution using the first column of \mathbf{C} .

Example. Consider the $n \times n$ Toeplitz matrix whose first column is given by an n -long array \mathbf{c} and whose first row is given by an n -long array \mathbf{r} . Suppose that we also pad the matrix with zeros to get a $2^p \times 2^p$ circulant matrix for some p , so we can efficiently use our radix-2 function `fftx2`. The naïve Julia code is

```
function fasttoeplitz(c,r,x)
    n = length(x)
    Δ = nextpow(2,n) - n
    x₁ = [c; zeros(Δ); r[end:-1:2]]
    x₂ = [x; zeros(Δ+n-1)]
    ifftx2(fftx2(x₁).*fftx2(x₂))[1:n]
end
```

The ToeplitzMatrices.jl package overloads matrix multiplication with fast Toeplitz multiplication when using a Toeplitz object, so it is quite fast and much faster than the code above. ◀

6.4 Bluestein and Rader factorization

A k -smooth number (or simply a smooth number) is an integer whose prime factors are less than or equal to k . For example, $720 = 2^4 \times 3^2 \times 5$ is a 5-smooth number, also known as a regular number. A k -rough number is an integer whose prime factors are all greater than or equal to k . The number $721 = 7 \times 103$ is 5-rough. The Cooley–Tukey algorithm is designed to recursively break down composite numbers into their prime factors, so it works best on smooth numbers. And it absolutely does not work on arrays whose lengths are prime numbers. Luckily, there are methods such as Bluestein and Rader factorization when n is a prime number. While not as efficient as the original Cooley–Tukey algorithm, these methods can work in conjunction with the Cooley–Tukey algorithm by smashing through rough numbers. To see this, let's plot several run times of Julia's FFT for smooth numbers and prime numbers:

```
using FFTW, Primes, Plots
N = 10000
smooth(n,N) = (1:N)[all.(x->x<=n, factor.(Set,1:N))]
t1 = [(x = randn(n); (n,@elapsed(fft(x)))) for n∈primes(N)]
t2 = [(x = randn(n); (n,@elapsed(fft(x)))) for n∈smooth(5,N)]
plot(t1,label="prime"); plot!(t2,label="5-smooth")
```

The figure on the following page shows the plots with several outliers removed. Note that while the FFT is significantly slower on rough numbers than on smooth numbers, it is still orders of magnitude faster than directly computing the DFT.

► Bluestein factorization

Let's start with the DFT

$$y_j = \sum_{k=0}^{n-1} c_k \omega_n^{jk}.$$

By noting that $(j - k)^2 = j^2 - 2jk + k^2$, we can replace jk in the DFT by $(j^2 - (j - k)^2 + k^2)/2$:

$$y_j = \sum_{k=0}^{n-1} c_k \omega_n^{jk} = \omega_n^{j^2/2} \sum_{k=0}^{n-1} c_k \omega_n^{-(j-k)^2/2} \omega_n^{k^2/2}.$$

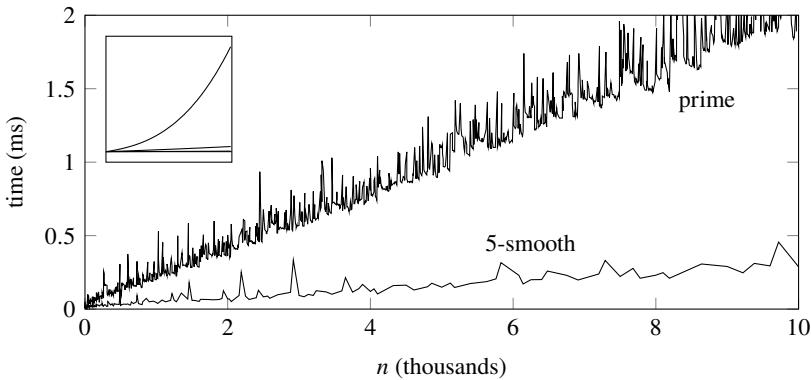


Figure 6.3: FFT run times on vectors of length n . The lower curve shows 5-smooth numbers and the upper curve shows prime numbers. The inset shows both curves relative to the run time of directly computing a DFT.

Furthermore, since $\sqrt{\omega_n} = \omega_{2n}$, it follows that

$$y_j = \omega_{2n}^{j^2} \sum_{k=0}^{n-1} c_k \omega_{2n}^{-(j-k)^2} \omega_{2n}^{k^2}.$$

This says that the DFT matrix \mathbf{F} equals \mathbf{WTW} , where \mathbf{W} is a diagonal matrix whose diagonal elements are given by

$$\mathbf{w} = \begin{bmatrix} 1 & \omega_{2n} & \omega_{2n}^4 & \omega_{2n}^9 & \dots & \omega_{2n}^{(n-1)^2} \end{bmatrix}$$

and \mathbf{T} is a symmetric Toeplitz matrix whose the first row is given by $\bar{\mathbf{w}}$. For example, for \mathbf{F}_5

$$\mathbf{w} = [1 \ \omega \ \omega^4 \ \omega^9 \ \omega^6] \quad \text{and} \quad \bar{\mathbf{w}} = [1 \ \omega^9 \ \omega^6 \ \omega^1 \ \omega^4]$$

where $\omega \equiv \omega_{10}$. Therefore, \mathbf{F}_5 equals

$$\begin{bmatrix} 1 & & & & \\ \omega & \omega^4 & & & \\ & \omega^9 & \omega^6 & & \\ & & \omega^6 & \omega^9 & \\ & & & \omega^4 & \omega^6 \end{bmatrix} \begin{bmatrix} 1 & \omega^9 & \omega^6 & \omega^1 & \omega^4 \\ \omega^9 & 1 & \omega^9 & \omega^6 & \omega^1 \\ \omega^6 & \omega^9 & 1 & \omega^9 & \omega^6 \\ \omega^1 & \omega^6 & \omega^9 & 1 & \omega^9 \\ \omega^4 & \omega^1 & \omega^6 & \omega^9 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & \\ \omega & \omega^4 & & & \\ & \omega^9 & \omega^6 & & \\ & & \omega^6 & \omega^9 & \\ & & & \omega^4 & \omega^6 \end{bmatrix}.$$

So, $\mathbf{y} = \mathbf{Fc}$ equals $\mathbf{w} \circ (\bar{\mathbf{w}} * (\mathbf{w} \circ \mathbf{c}))$ where \circ denotes the Hadamard product.

When computing a Toeplitz product, we can pad the vectors with any number of zeros to get a length for which the Cooley–Tukey algorithm is fast. We need

to compute three DFTs over a vector of at least $2n - 1$ elements. The following Julia code implements the Bluestein algorithm using the `fasttoeplitz` function, defined on page 152:

```
function bluestein_fft(x)
    n = length(x)
    ω = exp.((1im*π/n)*(0:n-1).^2)
    ω.*fasttoeplitz(conj(ω), conj(ω), ω.*x)
end
```

► Rader factorization

The Cooley–Tukey algorithm’s superpower is recursively breaking a large matrix into many smaller matrices that can be efficiently multiplied. The algorithm can effortlessly handle an array of 576 elements, because 576 is a product of small factors $576 = 2^6 \times 3^2$. But, a prime number like 587 is cryptonite to the Cooley–Tukey algorithm. Luckily, there is an algorithm called Rader factorization whose superpower is prime numbers—in fact, this is only when it works. Rader factorization permutes the rows and the columns of an $n \times n$ DFT matrix to get a matrix

$$\begin{bmatrix} 1 & \mathbf{1}^\top \\ \mathbf{1} & \mathbf{C}_{n-1} \end{bmatrix}$$

where \mathbf{C}_{n-1} is an $(n - 1) \times (n - 1)$ circulant matrix and $\mathbf{1}$ is a vector of ones. Multiplication by the circulant matrix \mathbf{C}_{n-1} can be done using three DFTs of size $n - 1$. If n is prime, then $n - 1$ is at least divisible by two, allowing us to jump-start the Cooley–Tukey algorithm. Say that we start with 587 elements. Rader factorization converts the problem into three 586-element DFTs, each of which can be broken into two 293-element DFTs using the Cooley–Tukey algorithm. Because 293 is a prime, we’ll need to use Rader factorization again. We can factor 292 into $2^2 \times 73$, meaning two iterations of Cooley–Tukey followed by Rader again. Finally, 72 factors into 2^3 and 3^2 using the Cooley–Tukey algorithm.

Rader factorization relies on the existence of a permutation. That there is such a permutation is a result of number theory. We take the following theorem as given and simply demonstrate it with an example.

Theorem 20. *If n is prime, then there is an integer $r \in \{2, 3, \dots, n - 1\}$ such that $\{1, 2, 3, \dots, n - 1\} = \{r, r^2 \pmod{n}, r^3 \pmod{n}, \dots, r^{n-1} \pmod{n}\}$. The integer r is called a primitive root (or generator) modulo n . There exists an integer s such that $sr \pmod{n} = 1$ for prime n . We call s the inverse of r and denote it by r^{-1} . If r is a primitive root, then so is r^{-1} .*

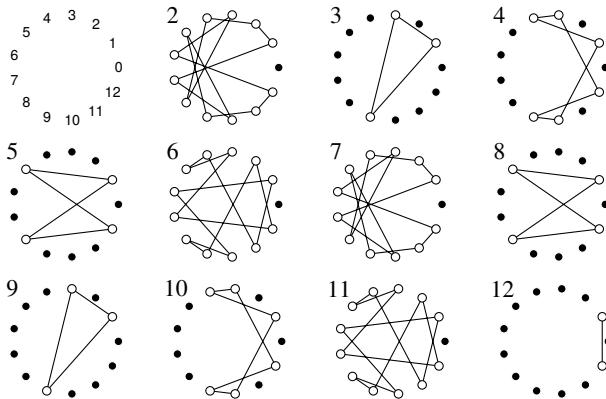


Figure 6.4: Finite cyclic groups of order 13.

Example. The finite cyclic groups of order $n = 13$ are plotted in the figure above. We can see that $r = 2$ is a primitive root modulo 13 because

$$2^j \pmod{13} = \{2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1\}$$

is a permutation of $j = \{1, 2, \dots, 12\}$. The integer 7 is the inverse of 2 because $2 \cdot 7 = 14 = 1 \pmod{13}$. Notice that the group generated by 7 shares the same cycle graph as the group generated by 2 but in the reverse direction:

$$7^j \pmod{13} = \{7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1\}.$$

Integers 6 and 11 are also primitive roots modulo 13. The order of the multiplicative group of integers modulo n is given by Euler's totient $\varphi(n)$, which equals $n - 1$ when n is prime. The number of elements in the groups must be a factor of $\varphi(13) = 12$, i.e. $\{2, 3, 4, 6, 12\}$. We see that 12 generates a group of order 2; 3 and 9 generate groups of order 3; 5 and 8 generate groups of order 4; 4 and 10 generate groups of order 6; and the primitive roots 2, 6, 7, and 11 generate groups of order 12. ◀

Suppose that we have the discrete Fourier transform

$$y_k = \sum_{j=0}^{n-1} c_j \omega_n^{jk}$$

where n is prime. If r is a primitive root modulo n , then by theorem 20 we can take the permutation $j \rightarrow r^j$ and $k \rightarrow r^{-k}$ for $j, k = 1, 2, \dots, n - 1$ giving us

$$y_0 = \sum_{j=0}^{n-1} c_j \quad \text{and} \quad y_{(r^{-k})} = c_0 + \sum_{j=1}^{n-1} c_{(r^j)} \omega_n^{(r^{j-k})}.$$

Note that the sum in the expression can be written as $\mathbf{P}_s^{-1} \mathbf{C}_{n-1} \mathbf{P}_r \mathbf{c}$, where \mathbf{C}_{n-1} is the circulant matrix whose first row is given

$$\begin{bmatrix} \omega_n^r & \omega_n^{r^2} & \cdots & \omega_n^{r^{n-1}} \end{bmatrix},$$

\mathbf{P}_r is the permutation matrix generated by r

$$\{r, r^2 \pmod{n}, r^3 \pmod{n}, \dots, r^{n-1} \pmod{n}\},$$

and \mathbf{P}_s is the permutation matrix generated by r^{-1}

$$\{r^{-1}, r^{-2} \pmod{n}, r^{-3} \pmod{n}, \dots, r^{-n+1} \pmod{n}\}.$$

Finding a primitive root modulo n is not trivial, but it is straight-forward. We'll just check each integer r until we find one that works. Let p be the factors of $\varphi(n) = n - 1$. If $r^{p_i} = 1$ for any $p_i \in p$, then we know that r cannot be a primitive root.

```
using Primes
function primitiveroot(n)
    φ = n - 1
    p = factor(Set,φ)
    for r = 2:n-1
        all( [powermod(r, φ÷pi, n) for pi∈p] .!= 1 ) && return(r)
    end
end
```

Now, we can implement the Rader factorization:

```
function rader_fft(x)
    n = length(x)
    r = primitiveroot(n)
    P_+ = powermod.(r, 0:n-2, n)
    P_- = circshift(reverse(P_+),1)
    ω = exp.((2im*π/n)*P_-)
    c = x[1] .+ ifft(fft(ω).*fft(x[2:end][P_+]))
    [sum(x); c[reverse(invperm(P_-))]]
end
```

While $n - 1$ is even and hence can be factored, it may still be advantageous to pad the matrix \mathbf{C}_{n-1} with zeros to make the evaluation of the three DFTs more efficient. We would need at least another $n - 1$ zeros to use a fast Toeplitz routine plus additional zeros to make the number of elements power smooth.

6.5 Applications

A typical use of the DFT is constructin the frequency components of a time-varying or space-varying function such as a sound wave. There are several other

important uses for the DFT that we examine in this section. Before doing so, it's helpful to understand a little more about how the FFT is implemented in practice.

► The fast Fourier transform in practice

The Fastest Fourier Transform in the West (FFTW) is a C library consisting of several FFT algorithms chosen using heuristics and trial. The first time that the library is called it may try several different FFT algorithms for the same problem and thereafter use the fastest. The FFTW library decomposes the problem using the composite Cooley–Tukey algorithm recursively until the problem can be solved using fixed-size codelets, which use split-radix, Cooley–Tukey, and a prime factor algorithm. If n is a prime number, the FFTW library decomposes the problem using Rader decomposition and then uses the Cooley–Tukey algorithm to compute the three $(n - 1)$ -point DFTs. FFTW saves the fastest approach for a given array size as the so-called *wisdom* to be used when an FFT is needed for another array of the same size. If a problem requires several repeated FFTs, it may be advantageous for FFTW to try all of the possible algorithms initially to see which is fastest. On the other hand, if the FFT is only needed once or twice, it may be better to simply let FFTW estimate the fastest approach.

- The FFTW.jl package provides several Julia bindings for FFTW. These include `fft` and `ifft` for complex-input transforms and `rfft` and `irfft` for real-input transforms.

The `fft` and `ifft` functions in Julia are all multi-dimensional transforms unless an additional parameter is provided. For example, `fft(x)` performs an FFT over all dimensions of `x`, whereas `fft(x, 2)` only performs an FFT along the second dimension of `x`. If you expect to use the same transform repeatedly on an array with the shape of `A`, you can define `F = plan_fft(A)` and then use `F*A` or `F\A`—the multiplication and inverse operators are overloaded.

While FFTs of every (or almost every) scientific computing language typically puts the zero frequency in the first position output array, often it is more mathematically meaningful or convenient to have the zero frequency in the middle position of the output array. This can be done by swapping the left and right halves of the array.

- The FFTW.jl functions `fftshift` and `ifftshift` shift the zero frequency component to the center of the array.

Discrete cosine transforms (DCT) and discrete sine transforms (DST) can be constructed using the DFT by extending even functions and odd functions as periodic functions. Because there are different ways to define this even/odd extension, there are different variants for each the DST and the DCT—eight of them (each called type-1 through type-8), although only the first four of them are commonly used in practice. Each of these four types is available through

the FFTW.jl real-input/real-output transform `FFTW.r2r(A, kind [, dims])`. The parameter `kind` specifies the type: `FFTW.REDFTxy` where `xy`, which equals `00`, `01`, `10`, or `11`, selects type-1 to type-4 DCT. Similarly, `FFTW.RODFTxy` selects a type-1 to type-4 DST. Because the DCT and DST are scaled orthogonal matrices, the same functions can be reused for the inverse DCT and inverse DST by dividing by a normalization constant.

 The FFTW.jl functions `dct` and `idct` compute the type-2 DCT and inverse DCT

► Fast Poisson solver

Recall the three-dimensional Poisson equation $-\Delta u(x, y, z) = f(x, y, z)$ with Dirichlet boundary conditions $u(x, y, z) = 0$ over the unit cube from Problem 5.3. This problem $\mathbf{A}\mathbf{u} = \mathbf{f}$ was solved using a nine-point stencil to approximate the Laplacian operator as $\mathbf{A} = \mathbf{D} \otimes \mathbf{I} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{D} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{I} \otimes \mathbf{D}$ where \mathbf{D} is the tridiagonal matrix $\text{tridiag}(1, -2, 1)/(\Delta x)^2$ and \mathbf{I} is the corresponding identity matrix. Because the matrix was large and sparse, we used an iterative method to find the solution. A faster method is to use a fast Poisson solver. From exercise 1.5, the eigenvalues of \mathbf{D} are

$$\lambda_k = \left(2 - 2 \cos \frac{k\pi}{n+1} \right) / (\Delta x)^2$$

where $k = 1, 2, \dots, n$ and the eigenvectors are given by

$$\mathbf{x}_k = \begin{bmatrix} \sin\left(\frac{k\pi}{n+1}\right) & \sin\left(\frac{2k\pi}{n+1}\right) & \sin\left(\frac{3k\pi}{n+1}\right) & \cdots & \sin\left(\frac{nk\pi}{n+1}\right) \end{bmatrix}^\top.$$

Take two arbitrary $n \times n$ matrices \mathbf{M} and \mathbf{N} and an $n \times n$ identity matrix \mathbf{I} . If $\mathbf{M}\mathbf{x}_i = \lambda_i \mathbf{x}_i$ and $\mathbf{M}\mathbf{y}_i = \mu_i \mathbf{y}_i$, then

$$(\mathbf{M} \otimes \mathbf{N})(\mathbf{x}_i \otimes \mathbf{y}_j) = \lambda_i \mu_j (\mathbf{x}_i \otimes \mathbf{y}_j).$$

Furthermore, the matrix $(\mathbf{I} \otimes \mathbf{M}) + (\mathbf{N} \otimes \mathbf{I})$ has eigenvalues $\{\lambda_i + \lambda_j\}$ with $i, j \in \{1, 2, \dots, n\}$. From this it follows that the eigenvalues of \mathbf{A} are

$$\lambda_{ijk} = \left(6 - 2 \cos \frac{i\pi}{n+1} - 2 \cos \frac{j\pi}{n+1} - 2 \cos \frac{k\pi}{n+1} \right) / (\Delta x)^2$$

and the i, j, k -component of eigenvalues $\mathbf{x}_{\zeta\xi\eta}$ is

$$\sin\left(\frac{i\zeta\pi}{n+1}\right) \sin\left(\frac{j\xi\pi}{n+1}\right) \sin\left(\frac{k\eta\pi}{n+1}\right)$$

with integers $\zeta, \xi, \eta \in \{1, 2, \dots, n\}$. The diagonalization of equation $\mathbf{A}\mathbf{u} = \mathbf{f}$ is $\mathbf{S}\mathbf{A}\mathbf{S}^{-1}\mathbf{u} = \mathbf{f}$. And the diagonalization of its solution $\mathbf{u} = \mathbf{A}^{-1}\mathbf{f}$ is $\mathbf{u} = \mathbf{S}\Lambda^{-1}\mathbf{S}^{-1}\mathbf{f}$. The operator \mathbf{S} is the discrete sine transform (DST), which in component form is

$$\hat{f}_{lmn} = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n f_{ijk} \sin\left(\frac{i\zeta\pi}{n+1}\right) \sin\left(\frac{j\xi\pi}{n+1}\right) \sin\left(\frac{k\eta\pi}{n+1}\right). \quad (6.4)$$

We can use an FFT to compute the discrete sine transform by first noting

$$\sin \frac{k\pi}{n} = \frac{e^{i\pi/n} - e^{-ik\pi/n}}{2i}$$

and extending the input function (which is zero at the endpoints) to make a periodic function. The indices of the DFT run from 0 to $n - 1$ while the indices of the DST run from 1 to n , so we'll need to add a zero to the input vector \mathbf{f} to account for the constant (zero frequency) term. So, the discrete sine transform of (f_1, f_2, \dots, f_n) can be computed by taking components 2 through $N + 1$ of the discrete Fourier transform of $(0, f_1, f_2, \dots, f_n, 0, -f_n, \dots, -f_2, -f_1)$. The DST \mathbf{S} is a real symmetric, scaled orthogonal matrix, so its inverse $\mathbf{S}^{-1} = 2/(n+1)\mathbf{S}$. The following Julia code solves exercise 5.3 using a fast Poisson solver. We'll first define a type-1 DST and inverse DST.

```
using FFTW
dst(x) = FFTW.r2r(x, FFTW.RODFT00)
idst(x) = dst(x)/(2^ndims(x)*prod(size(x).+1))
```

Now, we can solve the problem.

```
n = 50; x = (1:n)/(n+1); Δx = 1/(n+1)
v = 2 .- 2cos.(x*π)
λ = [v[i]+v[j]+v[k] for i∈1:n, j∈1:n, k∈1:n]./Δx^2
f = [(x-x^2)*(y-y^2) + (x-x^2)*(z-z^2)+(y-y^2)*(z-z^2)
      for x∈x,y∈x,z∈x]
u = idst(dst(f)./λ);
```

Comparing the numerical solution with the analytic solution,

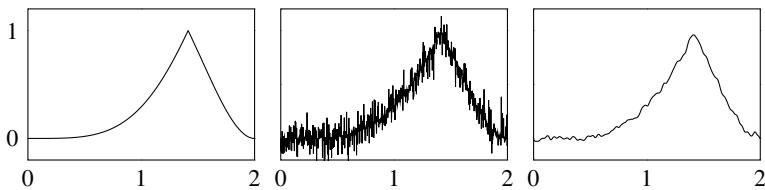
```
norm(u - [(x-x^2)*(y-y^2)*(z-z^2)/2 for x∈x,y∈x,z∈x])
```

we find an error of 1.5×10^{-14} . It is often much more convenient to operate on the basis functions directly to solve a partial differential equation instead of first discretizing in space. This approach is discussed in Chapter 16.

► Data filtering

Suppose that we have noisy data. High frequencies can be filtered and data can be regularized by using a convolution with a smooth kernel function such as

a Gaussian distribution. A kernel with a support of m points applied over a domain with n points requires $O(mn)$ operations applied directly. It requires $O(n)$ operations to compute the equivalent product in the Fourier domain, with an additional two $O(n \log n)$ operations to put it in the Fourier domain and bring it back. The extra overhead in computing Fourier transforms may be small when n and m are both large. In the example below, noise was added to the data on the left to get the center figure. A Gaussian distribution was used as a kernel of the convolution to filter the noise and get the figure on the right.



► Image and data compression

On page 78 we examined SVD rank reduction as a novel way to compress an image. In practice, photographs are typically saved using JPEG compression, which divides an image into 8×8 pixel blocks and then takes discrete cosine transforms (DCTs) of each of these blocks. Similarly, audio is often compressed as an MP3, which also uses DCT compression of blocks. Using Fourier series as a way of deconstructing the natural world should be, well, natural. Joseph Fourier introduced his namesake series as a solution to the heat equation, and a hundred years later Louis de Broglie suggested that all matter is waves.

The Fourier transform of a smooth function decays rapidly in the frequency domain. So, for sufficiently smooth data, high-frequency information can be discarded without introducing substantial error. Since the DFT is a scaled unitary matrix, both the 2-norm and the Frobenius norm are preserved by the DFT: $\|\mathbf{e}\|_2 = \|\mathbf{F}\mathbf{e}\|_2$ and $\|\mathbf{e}\|_F = \|\mathbf{F}\mathbf{e}\|_F$. This tells us that the pixelwise root mean squared error of an image equals the pixelwise root mean squared error of the DCT of an image. By clipping or zeroing out the high-frequency components in the Fourier transformed data, we can keep relevant information with only a marginal increase in the error. However, if the data is not smooth, significant information resides in the higher frequency components, and discarding it can result in substantial error. The so-called Gibbs phenomenon associated lossy compression of data with discontinuities can result in quite noticeable JPEG ringing artifacts at sharp edges.

Let's consider the following code that compresses a grayscale image A to a factor of p from its original size. For a color image, we could consider each RGB or HSV channel independently. The array B_p is a truncated DCT of A , and the function returns a reconstruction of compressed image.

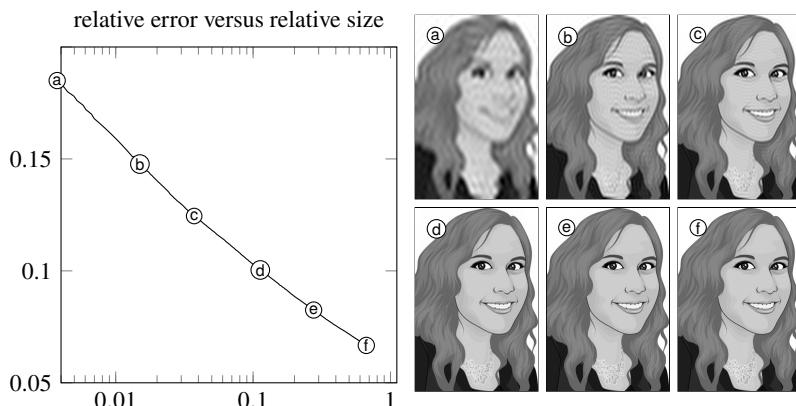


Figure 6.5: The relative Frobenius error of a DCT compressed image as a function of relative rank.

```
using Images, FFTW
function dctcompress(A,p)
    m,n = size(A)
    m_p,n_p = Int.(floor.((m,n).*sqrt(p)))
    B_p = dct(Float64.(A))[1:m_p,1:n_p]
    A_p = idct([B_p zeros(m_p,n-n_p); zeros(m-m_p,n)])
    Gray.(clamp01!(A_p))
end
```

Let's compare an original image and 95-percent compressed image.

```
using Images
A = Gray.(load(download(bucket*"laura.png")))
[A dctcompress(A,0.05)]
```

To be fair, B_p is a 64-bit floating-point array and would still need to be quantized in some nonlinear manner to 256 discrete values. By computing the Frobenius errors from several values of p , we can determine the relative efficiency of DCT compression. See the figure above and the QR link below.

6.6 Exercises

- 4 6.1. Modify the function `fftx2` on page 148 to implement an FFT algorithm suitable for powers of 3. Verify your radix-3 FFT using a vector of size $3^4 = 81$.



 6.2. *Fast multiplication of large integers.* Consider two polynomials

$$p(x) = p_n x^n + \cdots + p_1 x + p_0 \quad \text{and} \quad q(x) = q_n x^n + \cdots + q_1 x + q_0.$$

By taking $x = 10$, the polynomials $p(x)$ and $q(x)$ return the decimal numbers $p(10)$ and $q(10)$, respectively. By choosing the basis $\{x^{2n}, \dots, x, 1\}$, the polynomials can be represented by coefficient vectors

$$\mathbf{p} = (p_0, p_1, \dots, p_n, 0, \dots, 0,) \quad \text{and} \quad \mathbf{q} = (q_0, q_1, \dots, q_n, 0, \dots, 0,)$$

where \mathbf{p} and \mathbf{q} are padded with n zeros. The product $p(x)q(x)$ is represented by

$$(p_0 q_0, p_0 q_1 + p_1 q_0, \dots, p_{n-1} q_n + p_n q_{n-1}, p_0 q_0) \tag{6.5}$$

where the j th element is given by $\sum_{i=0}^j p_i q_{j-i}$. For example, when $n = 2$, the product of \mathbf{p} and \mathbf{q} is computed using

$$\begin{bmatrix} q_0 \\ q_1 & q_0 \\ q_2 & q_1 & q_0 \\ & q_2 & q_1 \\ & & q_2 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix}.$$

This product is equivalent to the circular convolution

$$\begin{bmatrix} q_0 & 0 & 0 & q_2 & q_1 \\ q_1 & q_0 & 0 & 0 & q_2 \\ q_2 & q_1 & q_0 & 0 & 0 \\ 0 & q_2 & q_1 & q_0 & 0 \\ 0 & 0 & q_2 & q_1 & q_0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ 0 \\ 0 \end{bmatrix}$$

In this case, the j th term of the product vector is $\sum_{i=0}^{2n} p_i q_{j-i} (\bmod 2n)$. Recall that the DFT of a circular convolution is a componentwise product. Hence, we can compute the product of two large numbers by

1. Treating each of the n digits as an element of an array.
2. Padding each array with n zeros.
3. Taking the discrete Fourier transform of both arrays, multiplying them componentwise and then taking the inverse discrete Fourier transform to transform them back.
4. Carrying the digits so that each element only contains a single digit. For example, multiplying 35 and 19 with the array representations $(5, 3, 0, 0, 0)$ and $(9, 1, 0, 0, 0)$ results in $(45, 32, 3, 0, 0)$, which must be corrected to give $(5, 6, 6, 0, 0)$ or 665.

The benefit of this approach is that we can reduce the number of operations from $O(n^2)$ to $O(n \log n)$. With some modification, this algorithm is called Schönhage–Strassen algorithm.

Write an algorithm that uses FFTs to compute the product of the primes

32769132993266709549961988190834461413177642967992942539798288533

and

3490529510847650949147849619903898133417764638493387843990820577.

The product of these two numbers is RSA-129, the value of which can be easily found online to verify that your algorithm works. RSA is one of the first commonly used public-key cryptologic algorithms.

- 6.3. The fast discrete cosine transform can be computed using an FFT by first doubling the domain and mirroring the signal into the domain extension as we did with the fast sine transform. Alternatively, rather than positioning nodes at the meshpoints $\{1, 2, \dots, n - 1\}$, we can position the nodes between the meshpoints $\{\frac{1}{2}, \frac{3}{2}, \dots, n - \frac{1}{2}\}$, counting over every second node and then reflecting the signal back at the boundary. For example, with eight nodes ①②③④⑤⑥⑦⑦⑥⑤④③②①① would have the reordering ①②④⑥⑦⑤③①. This way we need only n points rather than $2n$ points to compute the DCT. In two and three dimensions, the number of points are quartered and eighthed. This DCT

$$\sum_{j=0}^{n-1} f_k \cos \left[\frac{\pi}{n} \left(j + \frac{1}{2} \right) k \right] \quad \text{for } k = 0, 1, \dots, n - 1$$

is called the type-2 DCT or simply DCT-2 and is perhaps the most commonly used form of DCT. Write an algorithm that reproduces an n -point DCT-2 and inverse DCT-2 using n -point FFTs. Then, use your DCT to compress an audio signal or an image.

Numerical Methods for Analysis

Chapter 7

Preliminaries



This chapter provides a short introduction to several key ideas that we will use over the next four chapters. Specifically, we examine how functions, problems, methods, and computational error impact getting a good numerical solution.

7.1 Well-behaved functions

Throughout this book, unless otherwise stated, we'll assume that all functions are nice functions to allow us to hand wave around formalities. Mathematically, the words *nice* and *well-behaved* are used in an ad-hoc way to allow for a certain laziness of expression. We might reason that an analytic function is better-behaved than a merely continuous function, which itself is better-behaved than a discontinuous one. And, that one is surely better-behaved than a function that blows up. But, what is a well-behaved function? Perhaps the easiest way to introduce well-behaved functions is with examples of ones that are not so well-behaved, some rather naughty functions.

Example. Weierstrass functions are examples mathematical monsters. They are everywhere continuous and almost nowhere differentiable. The following Weierstrass function was proposed by Bernhard Riemann in the 1860s

$$\sum_{n=1}^{\infty} \frac{\sin(\pi n^2 x)}{\pi n^2}.$$

The function is differentiable only at the set of points of the form p/q where p and q are odd integers. At each of these points, its derivative is $-\frac{1}{2}$ (Hardy [1916]).

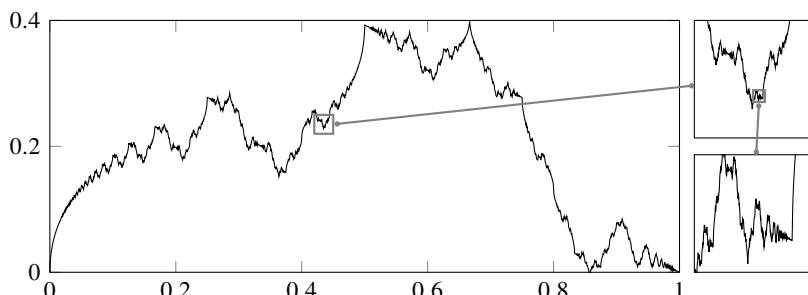


Figure 7.1: The nowhere differentiable Weierstrass function. The callouts depict the fractal nature of this pathological function.

The Weierstrass function is also one of the earliest examples of a fractal, a set with repeating, self-similarity at every scale and having non-integer dimensions. See Figure 7.1 above or the QR link at the bottom of this one. ◀

Example. Karl Thomae's function is $f(x) = 1/q$ if x is the rational number p/q (where p and q are coprime) and $f(x) = 0$ if x is irrational.



Thomae's function is discontinuous at rational numbers, continuous at irrational numbers, and everywhere nondifferentiable. ◀

Example. We can classify arithmetic operations using a hyperoperation sequence. The simplest arithmetic operation is succession, $x + 1$, which gives us the number that comes after x . Next, addition of two numbers $x + y$, which is the succession of x applied y times. Then, multiplication of two numbers xy , which is the addition of x with itself y times. Followed by exponentiation of two numbers x^y , which is multiplication of x with itself y times. The next logical operation is raising x to the x power y times. This operation, called *tetration*, is often denoted as ${}^y x$ or using Knuth's up-arrow notation $x \uparrow\uparrow y$. Tetration can create gargantuan numbers quickly. For example, ${}^1 3 = 3$, ${}^2 3 = 27$, ${}^3 3 = 7625597484987$, and ${}^4 3 = 1258\dots9387$ is a number with over three trillion digits. To put this number in perspective, if ${}^4 3$ were to be written out explicitly in this book, then its spine would be over 38 miles thick. ◀



To dig deeper into pathological functions, see Gelbaum and Olmstead's *Counterexamples in Analysis*. While mathematically almost all functions are pathological, the functions that are important for applications are often quite nice. The typical naughty functions we will encounter might have a cusp, a discontinuity, or a blowup. One feature of nice functions is well-behaved limiting behavior. To help describe the limiting behavior of nice functions we have Landau notation.

► Landau notation

Landau notation, often called big O and little o notation, is used to describe the limiting behavior of a function:

$$\begin{aligned} f(x) \in O(g(x)) &\quad \text{if } \lim_{x \rightarrow \infty} f(x)/g(x) \text{ is bounded.} \\ f(x) \in o(g(x)) &\quad \text{if } \lim_{x \rightarrow \infty} f(x)/g(x) = 0. \end{aligned}$$

One often says that $f(x)$ is on the order of $g(x)$ or $f(x)$ is $O(g(x))$ or simply $f(x) = O(g(x))$ to mean $f(x) \in O(g(x))$. For example, $(n+1)^2 = n^2 + O(n) = O(n^2)$ and $(n+1)^2 = o(n^3)$. Big O notation is often used to describe the complexity of an algorithm. Gaussian elimination, used to solve a system of n linear equations, requires roughly $\frac{2}{3}n^3 + 2n^2$ operations (additions and multiplications). So, Gaussian elimination is $O(n^3)$.

Big O and little o can also be used to simply notation of infinitesimal asymptotics. In this case,

$$\begin{aligned} f(x) \in O(g(x)) &\quad \text{if } \lim_{x \rightarrow 0} f(x)/g(x) \text{ is bounded.} \\ f(x) \in o(g(x)) &\quad \text{if } \lim_{x \rightarrow 0} f(x)/g(x) = 0. \end{aligned}$$

For example, $e^x = 1 + x + O(x^2)$ and $e^x = 1 + x + o(x)$. We can also use big O notation to summarize the truncation error of finite difference approximation to differentiation. For a small offset h , the Taylor series approximation

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \frac{1}{6}h^2f'''(x) + \dots \\ &= f(x) + hf'(x) + O(h^2). \end{aligned}$$

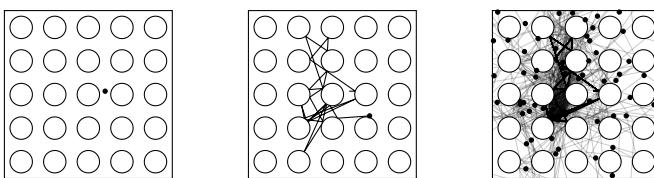
From this, it follows that

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

where we've taken $O(h^2)/h = O(h)$. So, forward differencing provides an $O(h)$ approximation (or first-order approximation) to the derivative.

7.2 Well-posed problems

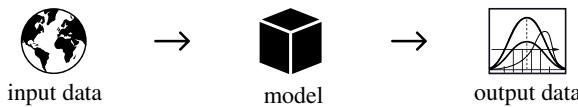
Example. Just as there are ill-behaved functions, there are also ill-behaved problems. Mathematician Nick Trefethen once offered \$100 as a prize for solving ten numerical mathematics problems to ten significant digits. One of them was “A photon moving at speed 1 in the xy -plane starts at $t = 0$ at $(x, y) = (0.5, 0.1)$ heading due east. Around every integer lattice point (i, j) in the plane, a circular mirror of radius $1/3$ has been erected. How far from the origin is the photon at $t = 10$?”. The solution can be found analytically using the geometry of intersections of lines and circles. At each intersection we need to evaluate a square root. And with each numerical evaluation of a square root round-off error is added into the solution, compounding over and over again, losing about a unit of precision with every reflection. While the original problem asked for the solution at $t = 10$ to 100-digits of accuracy, we could easily think about what happens at $t = 20$. The figure on the left below shows the starting position.



The figure in the middle shows the solution for a photon at $t = 20$. Instead of just one photon, let’s solve the problem using 100 photons each with a minuscule initial velocity angle spread of 10^{-16} for each photon (machine floating-point precision). To put it in perspective, if these photons left from a point on the sun, they would all land on earth in an area covered by the period at the end of this sentence. The figure on the right shows the solution for these hundred photons. Also, see the QR link at the bottom of this page. It’s hard to tell which is the right one or whether our original solution in the middle was in fact the correct one to begin with. ◀

So, what makes a problem a “nice” problem? To answer this question, let’s first think about what makes up a problem. Consider the statement “ $F(y, x) = 0$ for some input data x and some output data y .” For example, y could equal the value of polynomial F with coefficients $\{c_n, c_{n-1}, \dots, c_0\}$ of the variable x , e.g., $y = c_n x^n + c_{n-1} x^{n-1} + \dots + c_1 x + c_0$. Or, $F(y, x) = 0$ could be a differential equation where y is some function and x is the initial condition or the boundary value, e.g., $y' = f(t, y)$ with $y(0) = x$. Or, $F(y, x) = 0$ could be an image classifier where x is a set of training images and y is a set of labels. In a simple diagram, our problem consists of input data, a model, and output data.





We can classify the problem in three ways.

1. *Direct problem*: F and x are given and y is unknown. We know the model and the input—determine the output.
2. *Inverse problem*: F and y are given and x is unknown. We know the model and the output—determine the input.
3. *Identification problem*: y and x are given and F is unknown. We know the input and output—determine what's happening inside the black-box model. For example, what are its parameters?

A direct problem is typically easier to solve than an inverse problem, which itself is easier to solve than an identification problem. What makes a problem difficult to solve (either analytically or numerically)? The function $F(y, x)$ may be an implicit function of y and x . The problem may have multiple solutions. The problem may have no solution. Perhaps the problem is so sensitive that we cannot replicate the output if the input changes by even the slightest amount. In general, how do we know whether a problem has a meaningful solution? To answer this question, French mathematician Jacques Hadamard introduced the notion of mathematical well-posedness in 1923. He said that a problem was *well-posed* if

1. a solution for the problem exists (existence);
2. the solution is unique (uniqueness); and
3. this solution depends continuously on the input data (stability).

If any of these conditions fail to hold, the problem is called *ill-posed*.

► Condition number

Existence and uniqueness are straight forward. Let's dig into stability. We say that the output y depends continuously on the input x if small perturbations in the input causes small changes in the output. That is to say, there are no sudden jumps in the solution. Let $x + \delta x$ be a perturbation of the input and $y + \delta y$ be the new output, and let $\|\cdot\|$ be some norm. *Lipschitz continuity* says that there is some κ such that $\|\delta y\| \leq \kappa \|\delta x\|$ for any δx . We call κ the Lipschitz constant or the *condition number*. It is simply an upper bound on the ratio of the change in the output data to the change in the input data. We define the *absolute condition number*

$$\kappa_{\text{abs}}(x) = \sup_{\delta x} \left\{ \frac{\|\delta y\|}{\|\delta x\|} \right\}.$$

If $y \neq 0$ and $x \neq 0$, we define the *relative condition number*

$$\kappa(x) = \sup_{\delta x} \left\{ \frac{\|\delta y\|/\|y\|}{\|\delta x\|/\|x\|} \right\}.$$

A problem is *ill-conditioned* if $\kappa(x)$ is large for any x and *well-conditioned* if it is small for all x .

► Condition number of a direct problem

For now, consider a problem with a unique solution $y = f(x)$. Perturbing the input data x yields $y + \delta y = f(x + \delta x)$. If f is differentiable, then Taylor series expansion of f about x gives us

$$f(x + \delta x) = f(x) + f'(x)\delta x + o(\|\delta x\|),$$

where f' is the Jacobian matrix. Hence, $\delta y = f'(x)\delta x + o(\|\delta x\|)$. So, the absolute condition number is $\kappa_{\text{abs}}(x) = \|f'(x)\|$. Similarly, the relative condition number in the limit as $\delta x \rightarrow 0$ is

$$\kappa(x) = \|f'(x)\| \frac{\|x\|}{\|f(x)\|}.$$

Example. Consider the addition of two real numbers $f(a, b) = a + b$. The gradient of f is

$$f'(a, b) = \left(\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b} \right) = (1, 1).$$

In the ℓ^1 -norm,

$$\kappa(x) = \|f'(x)\|_1 \frac{\|x\|_1}{\|f(x)\|_1} = \|(1, 1)\|_1 \frac{\|(a, b)^T\|}{\|a + b\|} = \frac{|a| + |b|}{|a + b|}.$$

So, if a and b have the same sign then the relative condition number is 2. But, if a and b have opposite signs and are close in magnitude, then the condition number can be quite large. For example, for $a = 1000$ and $b = -999$, but $\kappa(x) = 3998$. Subtraction can potentially lead to cancellation of significant digits.

The usual solution to the quadratic equation $ax^2 + bx + c$ is given by the quadratic formula

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

When $b^2 \gg 4ac$, the quadratic formula can be numerically unstable because of subtractive cancellation. A stable alternative makes use of the *citardauq formula* to avoid subtraction:¹

$$x_{\pm} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

¹As you may have deduced, *citardauq* is simply *quadratic* spelled backwards.

So, take

$$x_- = \frac{-b - \text{sign}(b)\sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_+ = \frac{c}{ax_-}. \quad \blacktriangleleft$$

► Condition of an inverse problem

Consider a differentiable function $y = \varphi(x)$ that is locally invertible. Suppose that we know the output y and want to determine the input $x = \varphi^{-1}(y)$. Then $(\varphi^{-1})'(y) = [\varphi'(x)]^{-1}$. The absolute condition number of the inverse problem $x = \varphi^{-1}(y)$ is

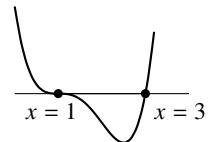
$$\kappa_{\text{abs}}(x) = \|(\varphi^{-1})'(y)\| = \|[\varphi'(x)]^{-1}\|$$

and the relative condition number is

$$\kappa(x) = \|[\varphi'(x)]^{-1}\| \frac{\|\varphi(x)\|}{\|x\|} \quad (7.1)$$

for nonzero x . The inverse problem is ill-posed when $\varphi'(x) = 0$, e.g., when $\varphi(x)$ has a double root. Furthermore, it is ill-conditioned when $\varphi'(x)$ is small and well-conditioned if $\varphi'(x)$ is large. One can derive an analogous definition for the condition number of the problem $F(y, x) = 0$ using the implicit function theorem.

Example. The function $(x-1)^3(x-3)$ has zeros at $x = 1$ and $x = 3$, with respective condition numbers $\kappa(1) = \infty$ and $\kappa(3) = 8$. Numerically finding the zeros of this polynomial is ill-conditioned in the neighborhood of $x = 1$ and well-conditioned in the neighborhood of $x = 3$. \blacktriangleleft



7.3 Well-posed methods

Just as there are pathological functions and pathological problems, there are also pathological solutions, even to well-behaved problems.

Example. The logistic² differential equation is a simple model of population dynamics $s(t)$ with a limited carrying capacity, like rabbits competing for food, water, and nesting space:

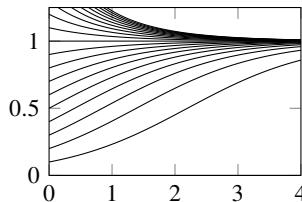
$$\frac{ds}{dt} = s(1 - s). \quad (7.2)$$

When the population s is small, the derivative $ds/dt \approx s$ and growth is proportional to the population. As the population approaches or exceeds the carrying capacity

²The term *logistic* was coined by Belgian mathematician Pierre François Verhulst as *logistique* in his 1845 article on population growth. The name refers the “log-like” similarities of the logistic function $1/(1 + e^{-t})$, with time t as the dependent variable, to the logarithm function near the origin.

$s = 1$, starvation occurs slowing or causing negative population growth. The solution to this differential equation is

$$s(t) = \left[1 - \left(1 - \frac{1}{s(0)} \right) e^{-t} \right]^{-1}$$



The solution is well-behaved and $s(t) \rightarrow 1$ as $t \rightarrow \infty$, reaching an equilibrium between reproduction and starvation.

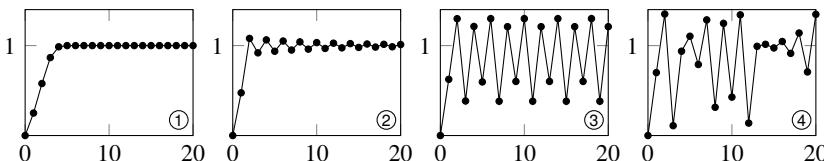
The simplest numerical method of solving the differential equation is to use the forward Euler approximation to the derivative and then iteratively solve the resulting difference equation. In this case, we have

$$\frac{s^{(k+1)} - s^{(k)}}{\Delta t} = s^{(k)}(1 - s^{(k)}) \quad (7.3)$$

where Δt is the time step between subsequent evolutions of the solution $s^{(k)}$. By rescaling $s^{(k)} \mapsto (\Delta t)/(1 + \Delta t)x^{(k)}$ and solving for $x^{(k+1)}$, we get an equivalent equation called the logistic map:

$$x^{(k+1)} = rx^{(k)}(1 - x^{(k)}) \quad \text{where } r = (1 + \Delta t). \quad (7.4)$$

This equation has the equilibrium solution $x = 1 - r^{-1}$. Let's examine the solution to (7.3) for $x(0) = 0.25$ with Δt given by 1, 1.9, 2.5, and 2.8:



When $\Delta t = 1$, the solution to (7.3) closely matches the solution to (7.2). When $\Delta t = 1.9$, the numerical solution exhibits transient oscillations and converges to the equilibrium solution to (7.2). When $\Delta t = 2.5$, the solution converges to a limit cycle with period 4. As Δt increases, so does the periodicity of the limit cycle. When $\Delta t = 2.8$, the numerical solution is completely chaotic. As Δt continues to increase, eventually the solution becomes unstable and blows up. ◀

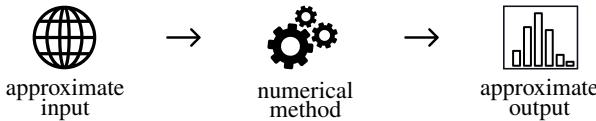
Again, consider the problem statement:

$$F(y, x) = 0 \text{ for some input data } x \text{ and some output data } y.$$

A numerical method for the problem is

$$F_n(y_n, x_n) = 0 \text{ for some input data } x_n \text{ and some output data } y_n. \quad (7.5)$$

We can understand x_n as an approximation to the input data x and y_n as an approximation to the output y .



Analogously to how we defined well-posedness for a problem, we say that the numerical method is well-posed if for any n

1. there is an output y_n corresponding to the input x_n (existence);
2. the computation of y_n is unique for each input x_n (uniqueness);
3. the output y_n depends continuously on the input x_n (stability).

► Consistency, stability, and convergence

We say that the numerical method $F_n(y, x) = 0$ is *consistent* if it approaches the original problem $F(y, x) = 0$ as n increases. In this case, it can be made as close to the original problem as we want, for example, by refining mesh size or iterating longer. Specifically, for every $\varepsilon > 0$ there is an N such that $\|F_n(y, x) - F(y, x)\| < \varepsilon$ for all $n > N$.

Now, suppose we perturb the input data by δx_n and we let δy_n be the corresponding perturbation in the solution determined by the numerical method $F_n(y + \delta y_n, x + \delta x_n) = 0$. We say that the method is *numerically stable* if a small error or perturbation in the approximate input data produces a small perturbation in the output. That is, if $\|\delta y_n\| \leq \kappa \|\delta x_n\|$ for some κ , whenever $\|\delta x_n\| \leq \eta$ for some η . We analogously define the absolute condition number as

$$\kappa_{\text{abs}, n}(x) = \sup_{\delta x_n} \left\{ \frac{\|\delta y_n\|}{\|\delta x_n\|} \right\}.$$

And, if y_n and x_n are nonzero, we define the relative condition number as

$$\kappa_n(x) = \sup_{\delta x_n} \left\{ \frac{\|\delta y_n\| / \|y_n\|}{\|\delta x_n\| / \|x_n\|} \right\}.$$

One would hope that for any good numerical method the approximate solution y_n would become a better approximation to y as n gets larger. We say that a numerical method is *convergent* if the numerical solution can be made as close to the exact solution as we want—for example, by refining the mesh size and

limiting round-off error. That is, for every $\varepsilon > 0$ there is a $\delta > 0$ such that for all sufficiently large n , $\|y(x) - y_n(x + \delta x_n)\| \leq \varepsilon$ whenever $\|\delta x_n\| < \delta$. It should come as no surprise that consistency and stability imply convergence. The following theorem, known as the Lax equivalence theorem or the Lax–Richtmyer theorem, is so important that it is often called the fundamental theorem of numerical analysis.

Theorem 21. *A consistent method is stable if and only if it is convergent.*

Proof. Consider a well-posed problem $F(y(x), x) = 0$ and a numerical method $F_n(y_n(x_n), x_n) = 0$. Assume that the numerical method is differentiable and that its Jacobian is invertible and bounded for all n . By the mean value theorem

$$F_n(y(x), x) = F_n(y_n(x), x) + \left(\frac{\partial F_n}{\partial y} \right) (y(x) - y_n(x))$$

where the Jacobian $(\partial F_n / \partial y)$ is evaluated at some point in the neighborhood of $y(x)$. Therefore,

$$y(x) - y_n(x) = \left(\frac{\partial F_n}{\partial y} \right)^{-1} (F_n(y(x), x) - F_n(y_n(x), x)).$$

We can replace $F_n(y_n(x), x)$ with $F(y(x), x)$ in the expression above because both of them are zero. So,

$$y(x) - y_n(x) = \left(\frac{\partial F_n}{\partial y} \right)^{-1} (F_n(y(x), x) - F(y(x), x)),$$

from which

$$\|y(x) - y_n(x)\| \leq M \|F_n(y(x), x) - F(y(x), x)\|$$

where $M = \|(\partial F_n / \partial y)^{-1}\|$. Because the method is consistent, for every $\varepsilon > 0$ there is an N such that $\|F_n(y(x), x) - F(y(x), x)\| < \varepsilon$ for all $n > N$. So, $\|y(x) - y_n(x)\| \leq M\varepsilon$ for all sufficiently large n .

We will use this result to show that convergence implies stability. If a numerical method is consistent and convergent, then

$$\begin{aligned} \|\delta y_n\| &= \|y_n(x) - y_n(x + \delta x_n)\| \\ &= \|y_n(x) - y(x) + y(x) - y_n(x + \delta x_n)\| \\ &\leq \|y_n(x) - y(x)\| + \|y(x) - y_n(x + \delta x_n)\| \\ &\leq M\varepsilon + \varepsilon' \end{aligned}$$

where the bound $M\varepsilon$ comes from consistency and the bound ε' comes from convergence. Choosing ε and ε' to be less than $\|\delta x_n\|$. Then

$$\|\delta y_n\| \leq (M + 1) \|\delta x_n\|,$$

which says that the method is stable. Now, we'll show stability implies convergence.

$$\begin{aligned}\|y(x) - y_n(x + \delta x_n)\| &= \|y(x) - y_n(x) + y_n(x) - y_n(x + \delta x_n)\| \\ &\leq \|y(x) - y_n(x)\| + \|y_n(x) - y_n(x + \delta x_n)\| \\ &\leq M\epsilon + \kappa\|\delta x_n\|\end{aligned}$$

where the bound $M\epsilon$ comes from consistency and the bound ϵ' comes from stability. By choosing $\|\delta x_n\| \leq \epsilon/\kappa$, we have

$$\|y(x) - y_n(x + \delta x_n)\| \leq (M + 1)\epsilon.$$

So, the method is convergent. \square

We can summarize the concepts for well-posed methods as follows:

problem	$F(y, x) = 0$
method	$F_n(y_n, x_n) = 0$
consistency	$F_n(y, x) \rightarrow F(y, x)$ as $n \rightarrow \infty$
stability	if $\ \delta x_n\ \leq \eta$, then $\ \delta y_n\ \leq \kappa\ \delta x_n\ $ for some κ
convergence	for any ϵ there are η and N such that if $n > N$ and $\ \delta x_n\ \leq \eta$, then $\ y(x) - y_n(x + \delta x_n)\ \leq \epsilon$
equivalence	consistency & stability \leftrightarrow convergence

7.4 Floating-point arithmetic

Programming languages support several numerical data types: often 64-bit and 32-bit floating-point numbers for real and complex data; 64-bit, 32-bit, and 16-bit integers and unsigned integers; 8-bit ASCII and up to 32-bit Unicode characters, and 1-bit Booleans for “true” and “false” information. A complex number is typically stored as an array of two floating-point numbers. Other data types exist including arbitrary-precision arithmetic (also called multiple-precision and bignum), which is often used in cryptography and in computing algebra systems like Mathematica. For example, Python implements arbitrary-precision integers for all integer arithmetic—the only limitation is computer memory. And it does not support 32-bit floating-point numbers, only 64-bit numbers. The Julia data types BigInt and BigFloat implement arbitrary-precision integer and floating-point arithmetic using the GNU MPFR and GNU Multiple Precision Arithmetic Libraries. The function `BigFloat(x, precision=256)` converts the value x to a BigFloat type with precision set by an optional argument. The command

`BigFloat(π , p)` will return the first ($p/\log_2 10$) digits of π . The precision of a variable x is returned by the command `precision(x)`.

It is important to emphasize that computers cannot mix data types. They instead either implicitly or explicitly convert one data type into another. For example, Julia interprets $a=3$ as an integer and $b=1.5$ as a floating-point number. To compute $c=a*b$, a language like Julia automatically converts (or promotes) 3 to the floating-point number 3.0 so that precision is not lost and returns 4.5. Such automatic, implicit promotion lends itself to convenient syntax. Still, some caution should be taken to avoid gotchas. In Julia, the calculation $4/5$ returns 0.8. In C, the calculation $4.0/5.0$ returns 0.8, but $4/5$ returns 0. By explicitly defining $x=\text{uint64}(4)$ as an integer in Matlab, it is cast as a floating-point number to perform the calculation $x/5$ returning an integer value 1. In this manner, $x/5*4$ returns 4 but $4*x/5$ returns 3.

- The function `typeof` returns the data type of a variable.

Floating-point numbers are ubiquitous in scientific computing. So, it is useful to have a basic working understanding of them. The IEEE 754 floating-point representation of real numbers used by most modern programming languages uses 32 bits (4 bytes) with 1 sign bit, 8 exponent bits, and 23 fraction mantissa bits to represent a single-precision number

s	e	m
---	---	---

and 64 bits (8 bytes) with 1 sign bit, 11 exponent bits, and 52 fraction mantissa bits to represent a double-precision number.

s	e	m
---	---	---

The exponents are biased to range between -126 and 127 in single precision and -1022 and 1023 in double precision. This means that single and double precision numbers have about 7 and 16 decimal digits of precision, respectively. The floating-point representation is $(-1)^s \times (1 + \text{mantissa}) \times 2^{(\text{exponent}-b)}$ where the bias $b = 127$ for single precision and $b = 1023$ for double precision. For example, the single-precision numbers

$$\boxed{0} \boxed{10000011} \boxed{000101000000000000000000} \text{ equals} \\ 1.000101_2 \times 2^4 = 10001.01_2 = 2^4 + 2^0 + 2^{-2} = 17.25$$

$$\boxed{1} \boxed{01111110} \boxed{110000000000000000000000} \text{ equals} \\ -1.11_2 \times 2^{-1} = -0.111_2 = -(2^{-1} + 2^{-2} + 2^{-3}) = -0.875$$

The mantissa is normalized between 0 and 1 and the leading “hidden bit” in the mantissa is implicitly a one. Normalizing the numbers in this way gives the floating-point representation an extra bit of precision.

 The `bitstring` function converts a floating-point number to a string of bits.

Note that only combinations of powers of 2 can be expressed exactly. For example, the decimal 0.825 can be expressed exactly as $2^{-1} + 2^{-2} + 2^{-4}$, but decimal 0.1 has no finite binary expression. Indeed, typing `bitstring(0.1)` into Julia returns

Only a finite set of numbers can be represented exactly. All other numbers must be approximated. The *machine epsilon* or *unit roundoff* is the distance between 1 and the next exactly representable number greater than 1.

- The function `eps()` returns the double-precision machine epsilon of 2^{-52} . The function `eps(x)` returns the round-off error at x . For example, `eps(0.0)` returns $5.0e-324$ and `eps(Float32(0))` returns $1.0f-45$.

Julia and Matlab use double-precision floating-point numbers by default. And, Python and R only use double-precision floating-point numbers—they do not use single-precision numbers at all. Python’s NumPy adds a type for single-precision. C refers to single-precision numbers as `float`. Python uses the same term `float` to refer to double-precision. Julia refers to doubles as `Float64` and singles as `Float32`, and similarly NumPy calls them `float64` and `float32`. Matlab uses `double` and `single`. And, the current IEEE 754 standard itself uses the terms `binary64` and `binary32`.

IEEE 754 also has a specification for half-precision floating-point numbers that require only 16 bits of computer memory. Such numbers are typically used in GPUs and deep learning applications where there may be a greater need for speed or memory than precision. Julia refers to half-precision numbers as `Float16`. IEEE 754 also specifies quadruple-precision numbers using 128 bits—one sign bit, 15 exponent bits, and 112 mantissa bits to give about 33 significant digits. IEEE 754 even has a specification for 256-bit octuple-precision floating-point numbers, but it is rarely implemented, in part because of arbitrary-precision software libraries.

Example. One algorithm buried in the source code of the 1999 *Quake III Arena* video game has attracted particular notoriety since it was discovered. The cryptic and seemingly obfuscated C code is translated to Julia code below.³

```
function Q_rsqrt(x::Float32)
```

³The developer's comments that accompanied the source code offered little clarity: "evil floating point bit level hacking" for line ①, "what the fuck?" for the line ②, and "1st iteration" for line ④. The inspiration for the algorithm can be traced through Cleve Moler to mathematicians William Kahan (the principal architect behind the IEEE 754 standard) and K. C. Ng in the mid 1980s.

```

① i = reinterpret(Int32,Float32(x))
② i = Int32(0x5f3759df) - (i>>1)
③ y = reinterpret(Float32,i)
④ y = y * (1.5f0 - (0.5f0 * x * y * y))
end

```

This function, often called the fast inverse square root function, approximates the reciprocal square root of a single-precision floating-point number, i.e., $1/\sqrt{x}$. The reciprocal square root is used when normalizing vectors for shading and reflection in 3D generated scenes. In the late 1990s, $\text{Q_rsqrt}(x)$ was considerably faster than directly computing $1.0/\sqrt{x}$ and its approximation was good enough for a video game. Subsequent advances in computer hardware in the early 2000s have transcended the need such a hack, even for video games.

How does the fast inverse square root algorithm work? Lines ①, ②, and ③ make an initial approximation of $y^{-1/2}$ and line ④ performs one iteration of Newton's method to refine that approximation. We'll discuss Newton's method in the next chapter. For now, let's examine the mathematics behind the initial approximation.

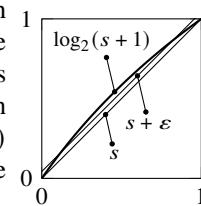
Take a single-precision floating-point number $x = (-1)^s 2^{e-127} (1 + 2^{-23}m)$, where s is a sign bit, e is the exponent, and an m is the fractional mantissa. If $y = x^{-1/2}$, then

$$\log_2 y = \log_2 x^{-1/2} = -\frac{1}{2} \log_2 x. \quad (7.6)$$

For a positive floating-point number

$$\log_2 x = \log_2 \left(2^{e-127} (1 + 2^{-23}m) \right) = (e - 127) + \log_2 (1 + 2^{-23}m). \quad (7.7)$$

The logarithm is a concave function— $\log_2(1 + s) \geq s$ with equality exactly when $s = 0$ or $s = 1$. We can approximate $\log_2(1 + s)$ as $s + \varepsilon$ where $s = 2^{-23}m$ and ε is some offset. Let's choose an ε that minimizes the uniform norm of the error in the approximation. The maximum of $\log_2(1 + s) - (s + \varepsilon)$ occurs at $s = -1 + (\log 2)^{-1}$ at which point the error in the approximation is



$$\delta = 1 - \frac{1 + \log(\log 2)}{\log 2}.$$

Choosing the shift $\varepsilon = \frac{1}{2}\delta \approx 0.0430$ will minimize the uniform error across $s \in [0, 1]$. Now we just need to compute (7.6) using (7.7) by approximating $\log_2(1 + s)$ with $s + \varepsilon$. By letting $[0 \ e_y \ m_y]$ and $[0 \ e_x \ m_x]$ be the single-precision floating-point representations of y and x , respectively, we only need to express e_y in terms of e_x and m_y in terms of m_x .

$$\begin{aligned}
& (e_y - 127) + (2^{-23}m_y + \varepsilon) \\
&= -\frac{1}{2} \left((e_x - 127) + (2^{-23}m_x + \varepsilon) \right) \\
&= \left(\left(-\frac{1}{2}e_x \right) - 127 \right) + \left(2^{-23} \left(-\frac{1}{2}m_x \right) + \varepsilon \right) + \frac{3}{2}(127 - \varepsilon) \\
&= \left(\left(-\frac{1}{2}e_x \right) - 127 \right) + \left(2^{-23} \left(-\frac{1}{2}m_x \right) + \varepsilon \right) + 2^{-23} \left(2^{23} \cdot \frac{3}{2}(127 - \varepsilon) \right).
\end{aligned}$$

Then, the magic number is

$$2^{23} \cdot \frac{3}{2}(271 - \varepsilon) = 2^{23} \cdot \frac{3}{2}(271 - \frac{1}{2}\delta) \approx 1597488310 = 5f37bcb6_{16}.$$

This number is close to but not a match of the magic number used in `Q_rsqrt` function. How the original value for the magic number of the `Q_rsqrt` function was chosen is not known.

The expression $e + 2^{-23}m$ is a binary representation $e.m$, where the binary point separates the integer and fractional parts of the number. For example, if $i = 101.101 = 5\frac{5}{8}$, then $\frac{1}{2}i = 10.1101 = 2\frac{13}{16}$ is obtained by simply shifting all of the bits one place to the right `i>>1`.

To summarize the `Q_rsqrt` algorithm, line ① reinterprets the floating-point number x as the integer $i = e_x + 2^{-23}m_x$; line ② computes $\log y = -\frac{1}{2} \log x$ using as a linear, integer approximation and overwrites i ; line ③ reinterprets floating-point number y from the integer $i = e_y + 2^{-23}m_y$; and line ④ computes one Newton iteration.

How well does the fast inverse square root algorithm work? Let's use the `BenchmarkTools.jl` library. To prevent Julia from optimizing the function output to a constant during compilation, we'll benchmark it using random input data.

```
using BenchmarkTools
@btime Q_rsqrt(Float32(x)) setup=(x=rand()) seconds=3
@btime 1/sqrt(x) setup=(x=rand()) seconds=3
```

We find that both methods take around two nanoseconds and `Q_rsqrt` is significantly less accurate. The average relative error is 2×10^{-2} after the first approximation. It is 1×10^{-3} after the Newton iteration—good enough for a late 1990s shooter game. Were we to add an additional Newton iteration, the relative error would drop to 2×10^{-6} . ◀

► Round-off error

Example. Siegfried Rump, of the Institute for Reliable Computing, posited the following example of catastrophic cancellation. Consider the calculation

$$y = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

where $a = 77617$ and $b = 33096$. Typing

```
a = 77617; b = 33096
333.75*b^6+a^2*(11*a^2*b^2-b^6-121*b^4-2)+5.5*b^8+a/(2*b)
```

into Julia returns approximately 1.641×10^{21} . Matlab and R both return about -1.181×10^{21} , Python returns about $+1.181 \times 10^{21}$, and Fortran computes it to be about 1.173. What's going on? If we rewrite the calculation as

$$y = z + x + \frac{a}{2b} \text{ where } z = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) \text{ and } x = 5.5b^8,$$

then

$$\begin{aligned} z &= -7917111340668961361101134701524942850 \text{ and} \\ x &= +7917111340668961361101134701524942848. \end{aligned}$$

From $z + x = 2$, we have that $y = -\frac{54767}{66192} \approx -0.827396$. The relative condition number for the problem is

$$\kappa = \frac{|z| + |x|}{|z + x|} \approx 10^{36},$$

which is considerably larger than the capacity of the mantissa for double-precision floating-point numbers. ◀

Example. Patriot surface-to-air missiles were used during the Gulf War to intercept Iraqi Scud missiles. In February 1991, a missile battery in Dhahran, Saudi Arabia, failed to track an incoming Scud missile that subsequently hit an army barracks. The Patriot missile system's internal clock was based on a 1970s design that used 24-bit fixed-point numbers. The system measured time in tenths of seconds, and because 0.1 cannot be represented exactly in binary, the round-off error introduced by a 24-bit register (by keeping only 24 bits) was approximately 9.5×10^{-8} . So, every second the clock lost 9.5×10^{-7} seconds. After the Patriot computer had run for 100 hours, the clock was off by about 0.34 seconds. A Scud missile travels more than half of a kilometer in this time. Although the system detected an incoming missile, the Scud missile was outside of the predicted range-gate, because of the round-off error and the initial detection was believed to be spurious (Blair et al. [1992]). ◀

► Underflow, overflow and NaN

Because the number of bits reserved for the exponent of a floating-point number are limited, floating-point arithmetic has an upper bound (closest to infinity) and a lower bound (closest to zero). Any computation outside of this bound results in either an overflow producing \inf or an underflow producing 0 .

• The command `floatmax(Float64)` returns the largest floating-point number—one bit less than 2^{1024} or about 1.8×10^{308} . The command `floatmin(Float64)` returns the smallest normalized floating-point number— 2^{-1022} or about 2.2×10^{-308} .

IEEE 754 uses gradual underflow versus abrupt underflow, by using denormalized numbers. The smallest normalized floating-point number 2^{-1022} gradually underflows by $(-1)^s \times (0 + \text{mantissa}) \times 2^{-b}$ and loses bits in the mantissa. This means that numbers can be as small as `eps(0.0) = floatmin(Float64)*eps(1.0)` before complete underflow.

Example. A spectacular example of overflow happened in 1996 when the European Space Agency (after spending 10 years and \$7 billion) launched the first Ariane 5 rocket. The rocket crashed within 40 seconds. An error occurred when a 64-bit floating-point number was converted into a 16-bit signed integer. The number was larger than 2^{15} resulting in an overflow. The guidance system shut down, ultimately leading to an explosion and destroying the rocket and cargo valued at \$500 million. Three years later another conversion error—this time between metric and English systems—led to the crash of the \$125 million NASA Mars orbiter. ▶

Some expressions are not defined mathematically—for example, $0/0$. IEEE 754 returns `NaN` (not a number) when it computes these numbers. Other operations that produce `NaN` include `NaN*inf`, `inf-inf`, `inf*0`, and `inf/inf`. Note that while mathematically 0^0 is not defined, IEEE 754 defines it as 1.

• The commands `isinf` and `isnan` check for overflows and `NaN`. The value `NaN` can be used to break up a plot, e.g., `plot([1, 2, 2, 2, 3], [1, 2, NaN, 1, 2])`.

7.5 Computational error

Computational error is the sum of rounding error caused by limited-precision data-type representation and truncation error resulting from a finite-dimensional approximation to the original problem. Let's take a look at each in turn.

► Rounding error

Let $\text{fl}(x)$ denote the floating-point approximation of x . The relative error of casting a non-zero number as a floating-point number is $\varepsilon = (x - \text{fl}(x))/x$ where $|\varepsilon| \leq \text{eps}$ (machine epsilon). We equivalently have that $\text{fl}(x) = (1 + \varepsilon)x$. Let's consider the rounding error caused by floating-point arithmetic: $a + b$. The relative error ε in the floating-point computation is given by $(a + b)(1 + \varepsilon)$:

$$(a + b)(1 + \varepsilon) = \text{fl}[\text{fl}(a) + \text{fl}(b)] = [a(1 + \varepsilon_1) + b(1 + \varepsilon_2)](1 + \varepsilon_3)$$

where ε_i denotes the round-off error. By expanding the expression on the right-hand side and only keeping the linear terms, we have

$$(a + b)(1 + \varepsilon) = a + b + a\varepsilon_1 + b\varepsilon_2 + (a + b)\varepsilon_3.$$

Solving for ε will give us the relative error in the computation

$$\varepsilon = \frac{a\varepsilon_1 + b\varepsilon_2 + (a + b)\varepsilon_3}{a + b}$$

By taking $|\varepsilon_i| \leq \text{eps}$ (machine epsilon), we have the bound

$$|\varepsilon| \leq \frac{|a| + |b| + |a + b|}{|a + b|} \text{eps}$$

Note that if a approximately equals but is not equal to $-b$ and both numbers are large, then ε can be substantially larger than machine epsilon leading to catastrophic cancellation.

We can extend our analysis to a sum of arbitrarily many numbers. To add n numbers sequentially $x_1 + x_2 + \dots + x_n$ requires $2n - 1$ rounding operations. This gives us a rough upper bound on the error

$$\varepsilon \leq \frac{(n+1)|x_1| + (n+1)|x_2| + n|x_3| + (n-1)|x_4| + \dots + 2|x_n|}{|x_1 + x_2 + \dots + x_n|} \text{eps},$$

although the actual error is typically much smaller. From this expression we see that to minimize the propagation of rounding error, one should add smaller terms first.

Example. In 1982 the Vancouver Stock Exchange⁴ introduced an index starting with a value of 1000.000. After each trade, the index was recomputed using four decimals and truncated to three decimal places. After 22 months with roughly 2800 transactions per day, the index was at 524.881. The index should have been 1098.892. What went wrong? Dropping the least significant decimal value instead of rounding to the nearest value resulted in a bias with each calculation. We can model the least-significant decimal values as discrete independent, uniformly distributed random variables: $10^{-4} \cdot \{0, 1, \dots, 9\}$. The mean bias is 0.00045 for each trade. Over 1.2 million transactions the total bias should be down around 554 from the actual index at just under 1099. If the algorithm had instead used proper rounding to avoid bias, then we should expect the sum of the errors to be normally distributed with zero mean and standard deviation $\sigma\sqrt{n}$ where the σ is the standard deviation of the distribution from which we are sampling. (The variance of the sum of independent random variables is the sum of the variances

⁴The building of the defunct Vancouver Stock Exchange now has a swanky cocktail bar.

of those random variables.) The standard deviation for the discrete uniform distribution is $\sigma = 0.0003$. After 1.2 million trades we can expect a standard deviation of about 0.3 and three standard deviations—with over 99 percent of the distribution—of about 1.0. See Huckle and Neckel [2019]. \blacktriangleleft

Example. When x is almost zero, the quantity e^x is very close to one, and calculating $e^x - 1$ can be affected by rounding error. We can avoid rounding error when $x = o(\text{eps})$ by instead computing its Taylor series approximation

$$e^x - 1 = (1 + x + \frac{1}{2}x^2 + \frac{1}{3}x^3 + \dots) - 1 = x + \frac{1}{2}x^2 + o(\text{eps}^2). \quad \blacktriangleleft$$

• The functions `expm1` and `log1p` compute $e^x - 1$ and $\log(x + 1)$ more precisely than `exp(x)-1` and `log(x+1)` in a neighborhood of zero.

► Truncation error

The truncation error of a numerical method is the error that results by using a finite or discrete approximation to an infinite series or continuous function. For example, we can compute the approximation to the derivative of a function using a first-order finite difference approximation. Consider the Taylor series expansion

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2}h^2f''(\xi)$$

for $\xi \in (x, x + h)$. Then

$$f'(x) = \frac{f(x + h) - f(x)}{h} + \frac{1}{2}hf''(\xi)$$

and we have the finite difference approximation

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}$$

with truncation error bounded by $\varepsilon_{\text{trunc}}(h) = \frac{1}{2}hm$ where $m = \sup_{\xi} |f''(\xi)|$. The round-off error is bounded by $\varepsilon_{\text{round}}(h) = 2 \text{eps}/h$. The total error given by $\varepsilon(h) = \varepsilon_{\text{trunc}}(h) + \varepsilon_{\text{round}}(h)$ is minimized when its derivative is zero:

$$\varepsilon'(h) = \frac{1}{2}m - 2 \text{eps}/h^2 = 0.$$

So, the error has a minimum at $h = 2\sqrt{\text{eps}/m}$.

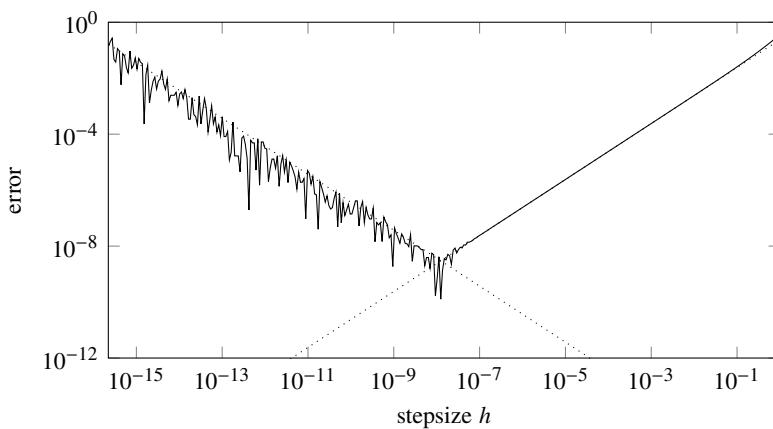


Figure 7.2: The total error for a first-order finite difference approximation. The truncation error decreases as h gets smaller and the round-error increases as h gets smaller. The minimum is near $h = 10^{-8}$.

Example. Consider first-order finite difference approximation of $f'(x)$ for the function $f(x) = \sin x$ at the value $x = 0.5$. We should expect a minimum total error of about 10^{-8} near $h = 10^{-8}$. See Figure 7.2 above. \blacktriangleleft

7.6 Exercises

7.1. By combining the Taylor series expansions

$$\begin{aligned} f(x-h) &= f(x) - hf'(x) + \frac{1}{2}h^2f''(x) - \frac{1}{6}h^3f'''(\xi_1) \\ f(x+h) &= f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \frac{1}{6}h^3f'''(\xi_2) \end{aligned}$$

where $\xi_1 \in (x-h, x)$ and $\xi_2 \in (x, x+h)$, we get the central difference approximation

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

with truncation error bounded by $\epsilon_{\text{trunc}} = \frac{1}{3}h^2m$ where m is the upper bound of $|f'''(\xi)|$ for $(x-h, x+h)$. Determine the value h that minimizes the total error (the sum of the truncation error and the round-off error). Then, plot the total error of the central difference approximation of the derivative to $f(x) = e^x$ at $x = 0$ as a function of the stepsize h to confirm your answer.

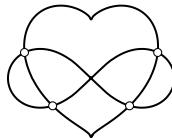
The round-off error of the central difference method is large when h is small. Another way to compute the numerical derivative of a real-valued function and

avoid catastrophic cancellation is by taking the imaginary part of the complex-valued $f(x + ih)/h$. Such an approximation is often called the *complex-step derivative*. Estimate the total error as a function of h using this method and then verify your estimate by plotting the total error as a function of the stepsize h for $f(x) = e^x$ at $x = 0$.

- ✍ 7.2. One way to determine machine epsilon is by computing $7/3 - 4/3 - 1$. Indeed, typing $7/3 - 4/3 - 1 == \text{eps}()$ in Julia returns true. Use binary floating-point representation to demonstrate this identity.
- ✍ 7.3. We can compute $f(x) = (x - 1)^3$ directly as $(x - 1)^3$ or expanded as $x^3 - 3x^2 + 3x - 1$. The second approach is more sensitive to loss of significance when $x \approx 1$. Plot of $(x - 1)^3$ and $x^3 - 3x^2 + 3x - 1$ in the interval $(1 - 10^{-5}, 1 + 10^{-5})$. Then compare the derivative of both expressions of $f(x)$ at $1 + 10^{-5}$ using the central difference approximation $f'(x) \approx (f(x+h) - f(x-h))/2h$ with a log-log plot of the error as a function of h .
- 7.4. In deriving the Q_rsqrt magic number, we chose ε to minimize the L^∞ -norm of $\log(x+1) - (x+\varepsilon)$. Instead, we could have chosen ε to minimize the L^2 -norm, i.e., to minimize the mean error for a number chosen at random. What is the magic number in this case? What is the magic number if we apply the Q_rsqrt algorithm to double-precision floating-point numbers?

Chapter 8

Solutions to Nonlinear Equations



An equation like $x = \cos x$ is transcendental and cannot be solved algebraically to be put into a closed form. Instead, we rely on numerical methods to find a solution. But it isn't just transcendental equations that require numerical solutions. Even simple polynomial equations like $x^5 + 2x^2 - 1 = 0$ do not have closed-form solutions. Abel's impossibility theorem states that there are no algebraic solutions to general polynomial equations of degree five or higher. So, we also need numerical methods to find the roots of most polynomials. In this chapter, we'll examine techniques to solve such problems.

8.1 Bisection method

Consider a real-valued function $f(x)$ which is continuous but not necessarily differentiable. Perhaps the simplest way of finding a zero x^* of $f(x)$ is using the bisection method. Consider two values a and b such that $f(a)$ and $f(b)$ have opposite signs, say $f(a) < 0$ and $f(b) > 0$. We call the interval $[a, b]$ a bracket. By the intermediate value theorem, the function f has a zero in the interval $[a, b]$. Take $c = (a + b)/2$. If $f(c) > 0$, then the zero is in the interval $[a, c]$. Otherwise, the zero is in the interval $[c, b]$. We choose our new bracket—either $[a, c]$ or $[c, b]$ —to be the interval with the zero. We continue iteratively halving each bracket into smaller brackets until the length is within some tolerance. A naïve Julia implementation of the bisection method is

```
function bisection(f,a,b,tolerance)
    while abs(b-a) > tolerance
```

```

c = (a+b)/2
sign(f(c)) == sign(f(a)) ? a = c : b = c
end
(a+b)/2
end

```

The command `bisection(x->x-cos(x),0,2,0.0001)` solves $x = \cos x$. The bisection method is easy enough to implement, but just how good is it? That is to say, when does the method work? And when it works, how quickly should we expect the method to approach a solution? To answer these questions, we'll need to define the rate of convergence.

► Convergence

A sequence $\{x^{(0)}, x^{(1)}, x^{(2)}, \dots\}$ is said to converge to x^* with order p if there is a positive constant C such that the subsequent error $\|x^{(k+1)} - x^*\| \leq C \|x^{(k)} - x^*\|^p$ for all k sufficiently large. A numerical method that generates such a sequence is said to be of *order p*. Let's first consider $p = 1$. When the convergence rate C must be strictly less than 1, we say that the method converges linearly. If the convergence rate C_k varies with each iteration and $\lim_{k \rightarrow \infty} C_k = 0$, the method is superlinear. If $\lim_{k \rightarrow \infty} C_k = 1$, the method is sublinear. If $p = 2$, we say convergence is quadratic. If $p = 3$, convergence is cubic. And so on. A method is *locally convergent* if any $x^{(0)}$ close enough to the zero x^* converges to x^* . A method is *globally convergent* if the method converges to x^* for any initial choice of $x^{(0)}$ in our search interval.

Let's return to the bisection method with an initial bracket $I^{(0)} = [a, b]$. The size of the bracket is $|I^{(0)}| = |b - a|$, and the size of each subsequent bracket is half the size of the previous one. The error at the k th step is $e^{(k)} = x^{(k)} - x^*$ and $e^{(k)} \leq |I^{(k)}| = \frac{1}{2}|I^{(k-1)}| = \frac{1}{4}|I^{(k-2)}| = \dots = 2^{-k}|b - a|$. We see that convergence is linear with a convergence factor $C = \frac{1}{2}$. With each iteration, we add one bit of precision onto the solution. In order to gain a digit of precision in the solution, we will need to take about $\log_2(10) \approx 3.3$ iterations. How many iterations do we need to take to get the error $|e^{(k)}| < \varepsilon$? Since $e^{(k)} \leq 2^{-k}|b - a|$, we will need to take $k = \log_2(|b - a|/\varepsilon)$ iterations.

What's bad about the bisection method is that convergence is relatively slow. What's good about it is that it is globally convergent. This makes the bisection method a useful method for getting close enough to a solution to then employ a faster but only locally convergent method such as Newton's method.

8.2 Newton's method

We can build a better root finding method by incorporating more information about the function $f(x)$. Let x^* to be a zero of the function f . If x is sufficiently

close to x^* , then the Taylor series expansion of f at x^* about x is

$$0 = f(x^*) = f(x) + (x^* - x)f'(x^*) + \frac{1}{2}(x^* - x)^2 f''(\xi)$$

where $\xi \in (x^*, x)$. Solving for x^* , we have

$$x^* = x - \frac{f(x)}{f'(x)} + o(x^* - x).$$

Start with $x^{(0)} = x$ and take the linearized equation

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}. \quad (8.1)$$

This method is called *Newton's method* or the *Newton–Raphson method*.

What's the convergence rate of Newton's method? The solution after k iterations is $x^{(k)} = x^* + e^{(k)}$ where $e^{(k)}$ is the error. Subtracting x^* from both sides of (8.1) and computing the Taylor series expansion of each term yields the following expression for $e^{(k+1)}$:

$$\begin{aligned} e^{(k+1)} &= e^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \\ &= \frac{f'(x^{(k)})e^{(k)} - f(x^{(k)})}{f'(x^{(k)})} \\ &= \frac{f'(x^* + e^{(k)})e^{(k)} - f(x^* + e^{(k)})}{f'(x^* + e^{(k)})} \\ &= \frac{\frac{1}{2}f''(x^*)(e^{(k)})^2 + \frac{1}{3}f^{(3)}(x^*)(e^{(k)})^3 + o((e^{(k)})^3)}{f'(x^*) + f''(x^*)e^{(k)} + o((e^{(k)}))} \\ &= \frac{f''(x^*)}{2f'(x^*)}(e^{(k)})^2 + o((e^{(k)})^2) \end{aligned} \quad (8.2)$$

when $f'(x^*) \neq 0$. Therefore, Newton's method converges quadratically. This means that with every iteration we effectively double the number of correct digits in the solution. with a good starting guess, we should get an answer which is within machine precision in about six iterations.

Example. The Babylonian method

$$x^{(k+1)} = \frac{1}{2} \left(x^{(k)} + \frac{a}{x^{(k)}} \right)$$

is the earliest known algorithm to compute the square root of a number a . The method is a special case of Newton's method applied to $x^2 - a$. Because Newton's method has quadratic convergence, we can expect it to be fast. Let's compute $\sqrt{2}$ with a starting guess of 1:

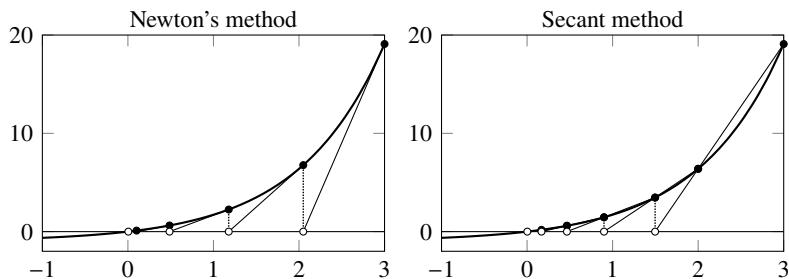


Figure 8.1: Comparison of two methods for finding the zeros of $e^x - 1$.

The correct digits are in boldface. The first iteration gets one correct digit. The second iteration gets three. Then six. Then 12, 25, and finally 48 at the sixth iteration. The number of correct decimals doubles with each iteration, as we should expect for a quadratic method. ◀

Note from (8.2) that if x^* is a double root, then $f'(x^*) = 0$ and the error $e^{(k+1)} \approx \frac{1}{2}e^{(k)}$. In this case, Newton's method only converges linearly at the same rate as the bisection method. Moreover, if x^* is a root with multiplicity n , then Newton's method converges linearly with convergence rate $C = 1 - \frac{1}{n}$. When n is large, convergence can be slow.

► Secant method

Newton's method finds the point of intersection of the line of slope $f'(x^{(k)})$ passing through the point $(x^{(k)}, f(x^{(k)}))$. See Figure 8.1 above. Determining the derivative f' can be difficult analytically and numerically. But we can use other slopes $q^{(k)}$ that are suitable approximations to $f'(x^{(k)})$ to get a method:

$$x^{(k+1)} = x^{(k)} - f(x^{(k)})/q^{(k)}. \quad (8.3)$$

By taking the slope defined by two subsequent iterates

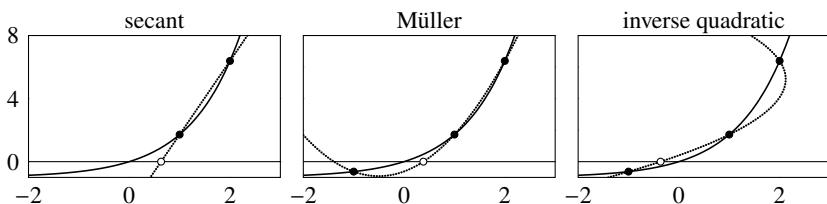
$$q^{(k)} = \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}, \quad (8.4)$$

we have the *secant method*. Note that the method requires two initial values. The secant method converges with order of about 1.63—a bit lower than Newton's method.

► Higher-order methods

We can design methods that converge faster than order two. For example, there are several higher-order extensions of Newton's method called Householder methods. But, because the same net result can be achieved with multiple iterations of a superlinear method, it's often unnecessary to go through the trouble of implementing higher-order methods. For instance, two iterations of the second-order Newton's method is a fourth-order method.

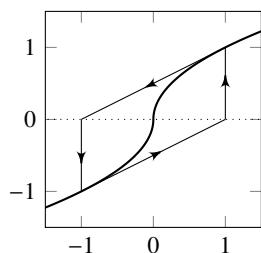
The secant method uses linear interpolation to find $x^{(k+1)}$. One way to improve convergence is by interpolating with a quadratic function instead. Consider the parabola $y = ax^2 + bx + c$ which passes through the points $(x^{(k-2)}, f(x^{(k-2)}))$, $(x^{(k-1)}, f(x^{(k-1)}))$ and $(x^{(k)}, f(x^{(k)}))$. Because the parabola likely crosses the x -axis at two points, we'll need to choose the best one. This method, called *Müller's method*, converges with order of about 1.84. A similar method called *inverse quadratic approximation* uses the parabola $x = ay^2 + by + c$, which crosses the x -axis at $x^{(k+1)} = c$. These methods need three points rather than just two needed for the secant method or one needed for Newton's method.



► Dekker-Brent method

Newton's method and the secant method are not globally convergent. When using these methods, an iterate can converge to a different root, it can go off to infinity, or it can get trapped in cycles.

Example. The function $\text{sign}(x)\sqrt{|x|}$ is continuous and differential everywhere except at the zero $x = 0$. Applying Newton's method gives the iteration



$$x^{(k+1)} = x^{(k)} - 2x^{(k)} = -x^{(k)}.$$

The iteration clearly does not converge, instead oscillating between $x^{(0)}$ and $-x^{(0)}$ for any non-zero starting guess. Newton's method applied to a similar function $\sqrt[3]{x}$ is $x^{(k+1)} = x^{(k)} - 3x^{(k)} = -2x^{(k)}$. Starting at any non-zero guess, $x^{(0)} = 1$ for instance, sends us off to infinity $\{1, -2, 4, -8, \dots\}$. ◀

The bisection method is globally convergent and therefore robust, but it is only linearly convergent. The secant method is superlinearly convergent with a convergence rate approaching 1.63, but it is only locally convergent. The Dekker–Brent method is a hybrid of the bisection method and the secant method, taking advantage of the strengths of both methods. Here's an outline of the method:

0. Start with a bracket.
1. Perform one secant iteration to get $x^{(k)}$. If the secant step goes outside the bracketing interval, use bisection instead to get $x^{(k)}$.
2. Refine the bracketing interval using $x^{(k)}$.
3. Repeat until convergence.

• The `NLsolve.jl` function `nlsolve(f, x0)` uses the trust region method to find the zero of the input function. The `Roots.jl` library has several simple functions such as `fzero(f, x0)` for finding roots.

8.3 Fixed-point iterations

If you type any number into a calculator and then hit the cosine button over and over again eventually the display stops changing and gives you 0.7390851. We call this number a *fixed point*. A fixed point of a function $\phi(x)$ is any point x such that $x = \phi(x)$. The simple geometric interpretation of a fixed point is the intersection of the line $y = x$ and the curve $y = \phi(x)$. Newton's method can be studied as an example of a broader class of numerical methods—fixed-point iterations:

$$\text{Given } x^{(0)}, \quad x^{(k+1)} = \phi(x^{(k)}). \tag{8.5}$$

In fact, for any function f , we can always write the problem $f(x) = 0$ as a fixed-point iteration $x = \phi(x)$ where $\phi(x) = x - f(x)$. But we are not guaranteed that a fixed-point iteration will converge. Suppose we take $\phi(x) = 3 \sin x$. The function has three fixed points at 0, at roughly +2.279, and at roughly -2.279. A fixed-point iteration for any point that does not already start at precisely either

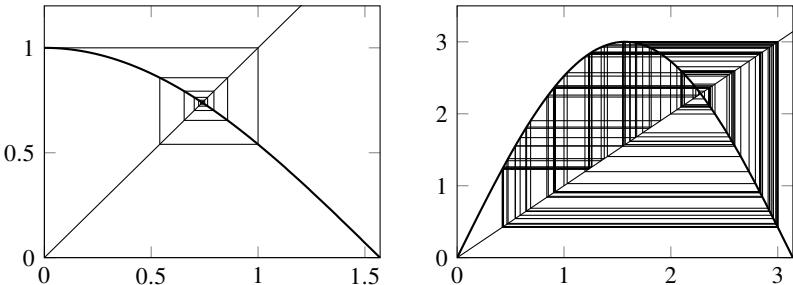


Figure 8.2: The fixed point iterations for $\phi(x) = \cos x$, which converges, and for $\phi(x) = 3 \sin x$, which does not converge.

of these three points will never converge. It will instead bounce around in the neighborhood of either of the two nonzero fixed points. See the figure above. So, let's now identify the conditions that do guarantee convergence.

We will say that $\phi(x)$ is a *contraction mapping* over the interval $[a, b]$ if ϕ is differentiable function with $\phi : [a, b] \rightarrow [a, b]$ and there is a $K < 1$ such that $|\phi'(x)| \leq K$ for all $x \in [a, b]$. The definition of a contraction mapping can be generalized as a nonexpansive map with Lipschitz constant K of a metric space onto itself. In this generalization, the following contraction mapping theorem is known as the Banach fixed-point theorem.

Theorem 22. *If the function ϕ is a contraction map over $[a, b]$, then ϕ has a unique fixed point $x^* \in [a, b]$ and the fixed-point method $x^{(k+1)} = \phi(x^{(k)})$ converges to x^* for any for any $x^{(0)} \in [a, b]$.*

Proof. We'll start by proving uniqueness. The contraction mapping ϕ is a continuous map from $[a, b]$ to $[a, b]$, so ϕ has at least one fixed point. Now, suppose that there are at least two distinct values $x^*, y^* \in [a, b]$ such that $\phi(x^*) = x^*$ and $\phi(y^*) = y^*$. The first-order Taylor approximation of ϕ expanded about x^* and evaluated at y^* is

$$\phi(y^*) = \phi(x^*) + (y^* - x^*)\phi'(ξ)$$

for some $ξ \in [x^*, y^*]$. Therefore,

$$|y^* - x^*| = |\phi(y^*) - \phi(x^*)| = |(y^* - x^*)\phi'(ξ)| \leq K|y^* - x^*|.$$

Since $K < 1$, it must follow that $y^* = x^*$.

Now, we'll prove convergence. The first-order Taylor approximation of ϕ expanded about x^* and evaluated at $x^{(k)}$ is

$$\phi(x^{(k)}) = \phi(x^*) + (x^{(k)} - x^*)\phi'(\xi^{(k)})$$

for some $\xi^{(k)} \in [x^*, x^{(k)}]$. Because $x^{(k+1)} = \phi(x^{(k)})$, it follows that

$$x^{(k+1)} - x^* = \phi(x^{(k)}) - \phi(x^*) = (x^{(k)} - x^*)\phi'(\xi^{(k)}).$$

So, $|x^{(k+1)} - x^*| \leq K|x^{(k)} - x^*| \leq K^{k+1}|x^{(0)} - x^*| \rightarrow 0$ as $k \rightarrow \infty$. \square

The set of points $\{x^{(0)}, \phi(x^{(0)}), \phi^2(x^{(0)}), \phi^3(x^{(0)}), \dots\}$ of the iterated function ϕ is called the forward *orbit* of $x^{(0)}$. A fixed point x^* is an *attractive fixed point* if there is an interval (a, b) such that for any $x^{(k)} \in (a, b)$ the forward orbit of $x^{(k)}$ always remains within (a, b) and converges to $x^{(k)}$. A fixed point x^* is a *repulsive fixed point* if for any $x^{(k)} \in (a, b)$ there is a positive k for which $\phi(x^{(k)}) \notin (a, b)$. Notably, the fixed point x^* is attractive if $|\phi'(x^*)| < 1$ and repulsive if $|\phi'(x^*)| > 1$.

When a fixed-point iteration converges, the error at each step diminishes. We can compute the ratio of the subsequent errors $e^{(k)}$:

$$\lim_{k \rightarrow \infty} \frac{e^{(k+1)}}{e^{(k)}} = \lim_{k \rightarrow \infty} \frac{x^{(k+1)} - x^*}{x^{(k)} - x^*} = \lim_{k \rightarrow \infty} \frac{\phi(x^{(k)}) - \phi(x^*)}{x^{(k)} - x^*} = \lim_{k \rightarrow \infty} \phi'(\xi^{(k)}) = \phi'(x^*).$$

The value $|\phi'(x^*)|$ is called the *asymptotic convergence factor*. In general, we have the following theorem:

Theorem 23. *If $\phi(x)$ is sufficiently smooth in a neighborhood of a fixed point x^* , with $\phi^{(j)}(x^*) = 0$ for $1 \leq j < p$ and $\phi^{(p)}(x^*) \neq 0$, then the fixed-point method converges with order p and has an asymptotic convergence factor $\frac{1}{p!}\phi^{(p)}(x^*)$.*

Proof. The Taylor series expansion of ϕ around x^* is

$$\phi(x^{(k)}) = \phi(x^*) + \frac{1}{p!}\phi^{(p)}(\xi^{(k)})(x^{(k)} - x^*)^p$$

for some $\xi^{(k)} \in [x^{(k)}, x^*]$. By definition $x^{(k+1)} = \phi(x^{(k)})$ and $x^* = \phi(x^*)$, so

$$x^{(k+1)} - x^* = \frac{1}{p!}\phi^{(p)}(\xi^{(k)})(x^{(k)} - x^*)^p.$$

Therefore,

$$\lim_{k \rightarrow \infty} \frac{e^{(k+1)}}{(e^{(k)})^p} = \lim_{k \rightarrow \infty} \frac{x^{(k+1)} - x^*}{(x^{(k)} - x^*)^p} = \lim_{k \rightarrow \infty} \frac{1}{p!}\phi^{(p)}(\xi^{(k)}) = \frac{1}{p!}\phi^{(p)}(x^*). \quad \square$$

► Speeding up the fixed-point method

We can modify a linearly convergent fixed-point method $x^{(k+1)} = \phi(x^{(k)})$ to get faster convergence. To do this, we'll consider Newton's method where $f(x) = x - \phi(x) = 0$:

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - \phi(x^{(k)})}{1 - \phi'(x^{(k)})}. \quad (8.6)$$

Just as we approximated the slope $f'(x)$ in Newton's method to get the secant method, we can approximate $\phi'(x^{(k)})$:

$$\phi'(x^{(k)}) \approx \frac{\phi(x^{(k+1)}) - \phi(x^{(k)})}{x^{(k+1)} - x^{(k)}} = \frac{\phi(\phi(x^{(k)})) - \phi(x^{(k)})}{\phi(x^{(k)}) - x^{(k)}}.$$

Substituting this expression in for $\phi'(x^{(k)})$ gives us

$$x^{(k+1)} = x^{(k)} - \frac{(\phi(x^{(k)}) - x^{(k)})^2}{\phi(\phi(x^{(k)})) - 2\phi(x^{(k)}) + x^{(k)}}. \quad (8.7)$$

This expression is known as *Aitken's extrapolation formula*. Even if a method is not convergent, like $x^{(k+1)} = 3 \sin(x^{(k)})$, Aitken's method converges.

► Stopping criteria

For an iterative method, we can stop once the error is within our desired tolerance. But without explicitly knowing the solution, we can't explicitly compute the error. Instead, we might use the residual or the increment to as a proxy for the error, and stop when one of them falls below a prescribed ε . For a fixed-point method, the residual $\phi(x^{(k)}) - x^{(k)}$ and the increment $\Delta x^{(k+1)} = x^{(k+1)} - x^{(k)}$ are exactly the same. The error of the fixed-point method is

$$e^{(k+1)} = x^* - x^{(k+1)} = \phi(x^*) - \phi(x^{(k)}) \approx \phi'(x^*) e^{(k)}.$$

So,

$$\Delta x^{(k+1)} = e^{(k+1)} - e^{(k)} \approx (1 - \phi'(x^*)) e^{(k)},$$

and hence

$$e^{(k)} \approx \frac{\Delta x^{(k+1)}}{1 - \phi'(x^*)}.$$

For Newton's method $\phi'(x^*) = 1 - f'(x^*)$ at a simple zero, and the error is approximately $\Delta x^{(k+1)}/f'(x^*)$. If $\phi'(x^*)$ is close to 1 and equivalently $f'(x^*)$ is close to 0, the error could be quite large with respect to ε . In this case, using the increment will not be a reliable stopping criteria.

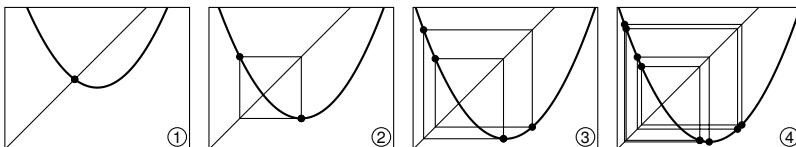
Designing robust stopping criteria is not trivial. For example, consider applying Newton's method to the function $f(x) = e^{-\lambda x}$ for a positive value λ . While there are no solutions, the method will evaluate $x^{(k+1)} = x^{(k)} + \lambda^{-1}$ at each step. In this case, the increment is always λ^{-1} , and using the increment as a stopping criteria will result in an endless loop. Using the residual $e^{-\lambda x}$ as a stopping criteria will eventually terminate the method, but the solution will hardly be a correct one, and if λ is really small, the method may require an unreasonably large number of iterations before it terminates. So, it is advisable to use brackets or some other exception handing when using an iterative method.

8.4 Dynamical systems

Suppose that we want to solve the simple equation $x^2 - x + c = 0$ for a given value c using the fixed-point method. While we could easily compute the solution using the quadratic formula, a toy problem like this will help us better understand the dynamics of the fixed-point method.¹ Start by rewriting the equation as $x = \phi_c(x)$ where $\phi_c(x) = x^2 + c$, and then iterate $x^{(k+1)} = \phi_c(x^{(k)})$. This iterated equation is known as the quadratic map. The quadratic map is equivalent to the logistic map $x^{(k+1)} = rx^{(k)}(1 - x^{(k)})$ introduced on page 174 as a simple model of rabbit population dynamics. If we start with the logistic map and first scale the variables by $-r^{-1}$ and next translate them by $\frac{1}{2}$, we have the quadratic map $x^{(k+1)} = (x^{(k)})^2 + c$ where $c = \frac{1}{4}r(2 - r)$.

By the fixed-point theorem, as long as $|\phi'_c(x)| < 1$ in a neighborhood of a fixed-point x^* , any initial guess in that neighborhood will converge to that fixed point. Because $|\phi'_c(x)| = |2x|$, any fixed point x^* in the interval $(-\frac{1}{2}, \frac{1}{2})$ is attractive. This happens when $-\frac{3}{4} < c < \frac{1}{4}$. When $c > \frac{1}{4}$, the equation $x^2 - x + c = 0$ no longer has a real solution. What about when $c < -\frac{3}{4}$? Let's look at a few cases.

- ① $c = -0.50$: the orbit of $x^{(0)} = 0$ converges to a fixed point.
- ② $c = -1.00$: the orbit of $x^{(0)} = 0$ converges to an orbit with period 2.
- ③ $c = -1.33$: the orbit of $x^{(0)} = 0$ converges to an orbit with period 4.
- ④ $c = -1.38$: the orbit of $x^{(0)} = 0$ converges to an orbit with period 8.

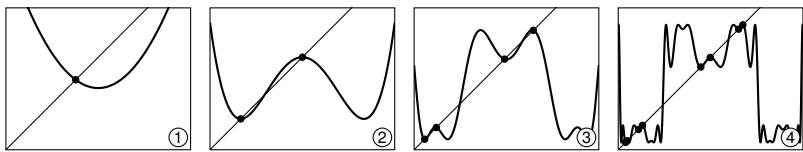


Why does the solution have this periodic limiting behavior? Take the iterated function compositions $\phi_c^2 = \phi_c \circ \phi_c$, $\phi_c^3 = \phi_c \circ \phi_c \circ \phi_c$, and so on. The forward orbit $\{x^{(0)}, \phi_c(x^{(0)}), \phi_c^2(x^{(0)}), \phi_c^3(x^{(0)}), \dots\}$ can be decomposed as

$$\{x^{(0)}, \phi_c^2(x^{(0)}), \phi_c^4(x^{(0)}), \dots\} \cup \{x^{(1)}, \phi_c^2(x^{(1)}), \phi_c^4(x^{(1)}), \dots\}$$

where $x^{(1)} = \phi_c(x^{(0)})$. Plot ② below shows $y = \phi_c^2(x)$ and $y = x$.

¹Robert May's influential 1976 review article "Simple mathematical models with very complicated dynamics" that popularized the logistic map examined how biological feedback loops can lead to chaos.



Notice that the function $\phi_c^2(x)$ has three fixed points, two of which are attractive with $|(\phi_c^2)'(x)| < 1$. Starting with $x^{(0)}$ leads us to one fixed point under the mapping ϕ_c^2 and starting with $x^{(1)}$ leads us to the other fixed point. Similarly, we can further decompose

$$\begin{aligned} \{x^{(0)}, \phi_c^2(x^{(0)}), \phi_c^4(x^{(0)}), \dots\} \cup \{x^{(1)}, \phi_c^2(x^{(1)}), \phi_c^4(x^{(1)}), \dots\} = \\ \{x^{(0)}, \phi_c^4(x^{(0)}), \phi_c^8(x^{(0)}), \dots\} \cup \{x^{(2)}, \phi_c^4(x^{(2)}), \phi_c^8(x^{(2)}), \dots\} \cup \\ \{x^{(1)}, \phi_c^4(x^{(1)}), \phi_c^8(x^{(1)}), \dots\} \cup \{x^{(3)}, \phi_c^4(x^{(3)}), \phi_c^8(x^{(3)}), \dots\}. \end{aligned}$$

Plot ③ above shows $y = \phi_c^4(x)$ and $y = x$. Notice that the function $\phi_c^4(x)$ has four fixed points where $|(\phi_c^4)'(x)| < 1$.

As c is made more and more negative, the period of the limit cycle of the forward orbit of $x^{(0)} = 0$ continues to double at a faster and faster rate. The first doubling happens at $c = -0.75$ and the second doubling at $c = -1.25$. Then at about -1.368 the period doubles again. The rate of period doubling limits a value called the *Feigenbaum constant*, and when $c \approx -1.401155$, the period of the limit cycle approaches infinity. At this point, the map is said to be *chaotic*. But as c is made still more negative, a finite period cycle re-emerges only to again double and re-double. See the QR link at the bottom of this page and the bifurcation diagram on the next page showing the positions of the attractive fixed points as a function of the parameter c . Finally when $c < -2$, the sequence escapes to infinity.

Often examining a more general problem provides greater insight into a specific problem. Generalization is one of the techniques that mathematician George Pólya discusses in his classic book *How to Solve It*. We have examined the quadratic map when the parameter c is real-valued. What happens when the c is complex-valued? Let's take $z^{(k+1)} = \phi_c(z^{(k)}) = (z^{(k)})^2 + c$. For what values c does the fixed-point method work? That is to say, when does $z^{(k)}$ converge to a fixed-point z^* ? By the fixed-point theorem, z is attractive when $|\phi'_c(z)| < 1$. That is, when $|z| < \frac{1}{2}$. So, let's take the boundary $z = \frac{1}{2}e^{i\theta}$ and find out which values c correspond to this boundary. From the original equation $z^2 - z + c = 0$, we have that $c = z - z^2 = \frac{1}{2}e^{i\theta} - \frac{1}{4}e^{2i\theta}$. The plot of this function $c(z)$ is a cardioid \bigcirc , the shape traced by point on a circle rolling around another circle of the same diameter.² For values c inside the cardioid, the orbit $\{0, \phi_c(0), \phi_c^2(0), \dots\}$

²There is an interesting related puzzle: “A coin is rolled around a similar coin without slipping. What is the orientation of the coin when it is halfway around the stationary coin?” The answer is at first counter-intuitive, but it is clear in the equation of the cardioid.



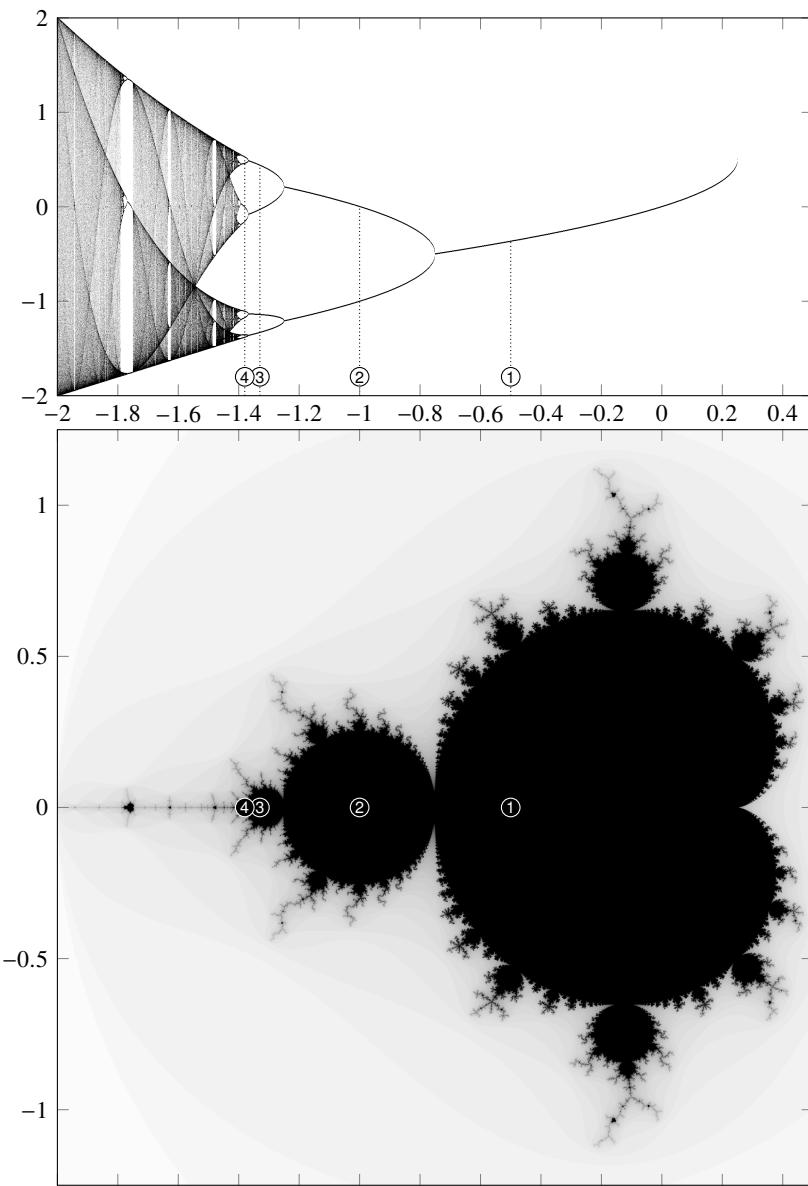


Figure 8.3: Top: The bifurcation diagram showing the limit cycle $\{\phi_c^\infty(0)\}$ of the quadratic map as a function of c . Bottom: The Mandelbrot set is the set of complex values c for which $\phi_c(0)$ has a limit cycle. The cases ①–④ from page 198 are indicated. Pay attention to changes in the period-doubling, bifurcation points, and matching cusps along the real axis of the Mandelbrot set.

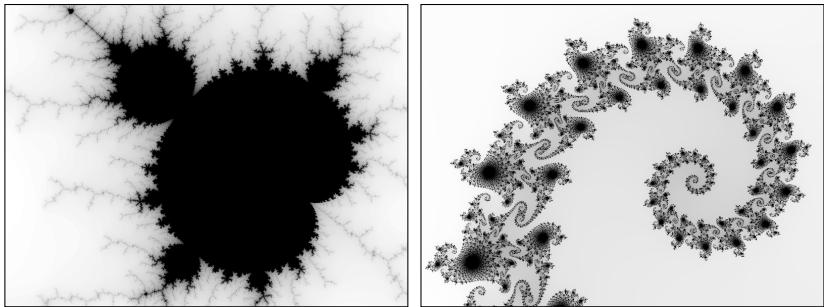


Figure 8.4: Snapshots of the Mandelbrot set showing a self-similar ‘‘minibrot’’ near $c = -0.1602 + 1.0335i$ (left) and a spiral around a Misiurewicz point near $c = -0.7452 + 0.1126i$ (right).

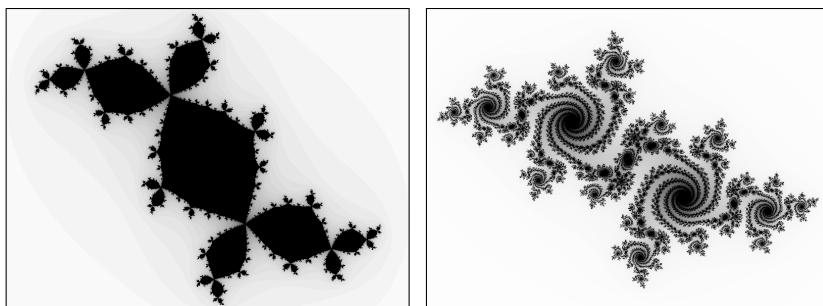
limits a fixed-point.

If we extend c to include orbits $\{0, \phi_c(0), \phi_c^2(0), \dots\}$ whose limit sets are bounded, the set of c is called the *Mandelbrot set*. See the figure on the facing page. At the heart of the Mandelbrot set is the cardioid generated by the fixed points of the quadratic map. There is an infinite set of bulbs for limit sets of period q tangent to the main cardioid at $c = \frac{1}{2}e^{i\theta} - \frac{1}{4}e^{2i\theta}$ with $\theta = 2\pi p/q$ where p and q are relatively prime integers. The Mandelbrot set is one of the best known examples of fractal geometry, exhibiting recursive detail and intricate mathematical beauty. See the figure above.

If the Mandelbrot set is the set of parameters c that results in bounded orbits given $z^{(0)} = 0$, what can we say about the limit sets themselves? We call the forward orbit for a given parameter c a *Julia set*. Whereas the Mandelbrot set lives in the complex c parameter space, the Julia set lives in the complex z -plane. If the orbit of z is bounded, then the Julia set is connected. If the orbit of z is unbounded, then the Julia set is disconnected and called a *Cantor set*. See Figure 8.5 on the following page.

Imaging the Mandelbrot set is straightforward. The following Julia function takes the array bb for the lower-left and upper-right corners of the bounding box; $xpix$ for the number of horizontal pixels; n for the maximum number of iterations; and s for the starting value $z^{(0)}$, which for the Mandelbrot set is 0. The function returns a two-dimensional array M that counts the number of iterations k to escape: $|z^{(k)}| > 2$.

```
function mandelbrot(bb,xpix,n=200,s=0)
    ypix = round(Int,xpix*(bb[4]-bb[2])/(bb[3]-bb[1]))
    M = zeros(Int,ypix,xpix)
    z = float(s)*ones(Complex,ypix,xpix)
    c = LinRange(bb[1],bb[3],xpix)' .+ im*LinRange(bb[4],bb[2],ypix)
```

Figure 8.5: Julia sets for $-0.123 + 0.745i$ (left) and $-0.513 + 0.521i$ (right).

```

for k in 1:n
    mask = (abs.(z) .< 2)
    M[mask] .+= 1
    z[mask] = z[mask].^2 .+ c[mask]
end
return(M)
end

```

The following produces the second image of Figure 8.4:

```

using Images
M = mandelbrot([-0.172, 1.0228, -0.1494, 1.0443], 800)
save("mandelbrot.png", 1 .- M./maximum(M))

```

Imaging the Julia set uses almost identical code. The Mandelbrot set lives in the c -domain with a given value $z^{(0)} = 0$, and the Julia set lives in the z -domain with a given value c . So the code for the Julia set requires only swapping the initializations for variables z and c in the code: $(z, c) = (c, z)$.

Example. Orbits that are chaotic, erratic, and unpredictable sometimes have applications. Such behavior makes elliptic curves especially useful in Diffie–Hellman public key cryptography³ and pseudorandom number generators. An

³Suppose that Alice and Bob want to communicate securely. If they both have the same secret key, they can both encrypt and decrypt messages using this key. But first, they'll need to agree on a key, and they'll need to do so securely. They can use Diffie–Hellman key exchange protocol. Both Alice and Bob use a published elliptic curve with a published P . Alice secretly chooses a private key m and sends Bob her public key mp . Bob secretly chooses his private key n and sends Alice his public key np . When Alice applies her private key to Bob's public key and Bob applies his private key to Alice's public key, they both get the same secret key nmP . With a shared secret key, Alice and Bob can each generate the same cryptographic key to encrypt or decrypt a message. The multiplication is typically modulo some prime.

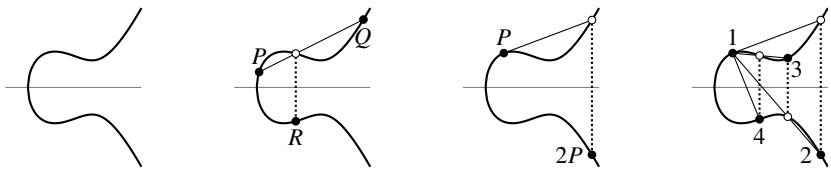


Figure 8.6: An elliptic curve, such as $y^2 = x^3 - 2x + 4$, is symmetric about the x -axis. A secant line through points P and Q passes through another point on the curve, the reflection of which about the x -axis is a point $R = P \oplus Q$. A tangent line through a point P passes through another point on the curve, the reflection of which is $2P = P \oplus P$. Continuing these operations we get $3P, 4P$, and so on.

elliptic curve is any curve of the form

$$y^2 = x^3 + ax + b \quad (8.8)$$

where $x^3 + ax + b$ has distinct roots. Such a curve has two useful geometric properties. First, it is symmetric about the x -axis. And second, a non-vertical straight line through two points on the curve (a secant line) will pass through a third point on the curve. Similarly, a tangent line—the limit of a secant line as two points approach one another—also passes through one other point. Let's label the first two points $P = (x_0, y_0)$ and $Q = (x_1, y_1)$ on the curve, and label the reflection of the third point across the x -axis $R = (x_2, -y_2)$. We can denote $R = P \oplus Q$. It can be easily checked that points generated in this way using \oplus (the reflection of the third colinear point on the curve) form a group. In this group, the point O at infinity is the identity. We define $2P = P \oplus P$, $3P = P \oplus 2P$, and so on. See Figure 8.6 above.

The arithmetic is straightforward. To compute $2P$, start by implicitly differentiating (8.8). We have $2y dy = (3x^2 + a) dx$ from which the slope of the tangent at P is $\lambda = (3x_0^2 + a)/2y_0$. Now, substituting the equation for points along the line $y = \lambda(x - x_0) + y_0$ into (8.8) gives

$$x^3 - (\lambda(x - x_0) + y_0)^2 + ax + b = 0.$$

Notice that the coefficient of the x^2 term of the resulting monic cubic equation is $-\lambda^2$. The coefficient of the second term of a monic cubic polynomial is minus the sum of the roots of the polynomial. Because we have a double root at x_0 , it follows that the third root $x = \lambda^2 - 2x_0$. And after reflecting about the x -axis we have that $y = -\lambda(x - x_0) - y_0$.

We can similarly compute $P \oplus Q$. The slope of a line through two points is $\lambda = (y_1 - y_0)/(x_1 - x_0)$. And from the argument above, the third root is

given by $x = \lambda^2 - x_0 - x_1$. After reflecting about the x -axis, we have that $y = -\lambda(x - x_0) - y_0$. In this way, we can define $2P = P \oplus P$. Then with $P \oplus 2P$ we can define $3P$ and so on. Computing mP using a double-and-add method takes about $\log_2(m)$ doublings and half as many additions.

Finally, if the coefficients a and b and the point P are rational numbers, then all the points mP in the orbit are rational numbers. And, we can define the group as a quotient group $y^2 = x^3 + ax + b \pmod{r}$ for some r . Undoing the modulo operator equates to solving a discrete logarithm problem, which is a computationally hard problem. Pulling all of this together allows one to build a cryptologic system. ◀

8.5 Roots of polynomials

The quadratic formula, for finding the solutions of a quadratic equation, has been known in various forms for the past two millennia. During the Renaissance, mathematicians Scipione del Ferro and Niccolò Fontana Tartaglia independently discovered techniques for finding the solutions of a cubic equation.⁴ At the time a mathematician's fame (and fortune?) was tied to winning competitions with other mathematicians, duels of sort to prove who was the better mathematician. So, while Tartaglia knew a formula for solving a cubic equation, he kept knowledge of it secret and even obfuscated his formula in a poem "*Quando chel cubo con le cose appresso...*" A contemporary, Girolamo Cardano, persuaded Tartaglia to tell him his secret formula with the promise that he would never publish it. Of course, Cardano did exactly that and it is now known as Cardano's formula (although some give a nod to its original inventor and call it the Cardano–Tartaglia formula). Later, Cardano's student Lodovico de Ferrari developed a general technique for solving quartic equations, by reducing them first to cubic equations and then applying Cardano's formula. And that's where it stops. There is no general algebraic formula for quintic polynomials, nor can there be one. In 1824, the 21-year-old mathematician Niels Henrik Abel proved what would later be known as the Abel–Ruffini theorem or Abel's impossibility theorem, by providing a counter example that demonstrated that there are polynomials for which no root-finding formula can exist. Six years later, 18-year-old Évariste Galois generalized Abel's theorem to characterize which polynomials were solvable by radicals, thereby creating Galois theory.⁵ To dig deeper, see Mario Livio's book *The Equation That Couldn't Be Solved*.

⁴As a young boy Niccolò Fontana was maimed when an invading French Army massacred the citizens of Brescia leaving him with a speech impediment, after which people started calling him Tartaglia, meaning “the stutterer.”

⁵Évariste Galois died tragically at age 20 in a duel stemming from a failed love affair. During his short life, Galois' theory was largely rejected as incomprehensible. His work would not published until twelve years after his death. Abel also died tragically young at age of 26 from tuberculosis.

Because we cannot find polynomial roots analytically, we must find them numerically. While any of the methods discussed in the previous section would work, a better design is one that uses the polynomial structure. Namely, because an n th degree polynomial has exactly n complex roots counting multiplicity, once we have found one root, we can factor this root out of the polynomial to simplify the next step. This process is called *deflation*.

► Horner's deflation

Note that the polynomial $p(x) = x^3 - 6x^2 + 11x - 6$ can be written in the Horner form $x(x(x - 6) + 11) - 6$. This new form allows us to evaluate a polynomial more quickly, requiring fewer multiplications while reducing the propagated round-off error. Synthetic multiplication provides us with a convenient means of record-keeping when evaluating a polynomial in Horner form. For a general polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

synthetic multiplication is performed by determining b_0 given the coefficients $\{a_n, a_{n-1}, \dots, a_0\}$ using the accounting device

$$\begin{array}{c|cccc} z & a_n & a_{n-1} & \cdots & a_0 \\ & b_n z & \cdots & & b_1 z \\ \hline b_n & b_{n-1} & \cdots & & b_0 \end{array}$$

The coefficients $b_n = a_n$ and $b_k = a_k + z b_{k+1}$ for $k = 0, 1, \dots, n - 1$, and $p(z) = b_0$. Note that if z is a root of $p(x)$, then $b_0 = 0$. To evaluate $p(4)$

$$\begin{array}{c|cccc} 4 & 1 & -6 & 11 & -6 \\ & 4 & -8 & 12 & . \\ \hline 1 & -2 & 3 & 6 \end{array}$$

So, $p(4) = 6$.

 The function `evalpoly(x, p)` uses Horner's method (or Goertzel's algorithm for complex values) to evaluate a polynomial with coefficients $p = [a_0, a_1, \dots, a_n]$ at x .

By definition $b_k - z b_{k+1} = a_k$ and $b_n = a_n$, so we can rewrite

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

as

$$p(x) = b_n x^n + (b_{n-1} - b_n z) x^{n-1} + \cdots + (b_1 - b_2 z) x + (b_0 - b_1 z).$$

By grouping the coefficients b_k , we have

$$p(x) = (x - z)b_nx^{n-1} + (x - z)b_{n-1}x^{n-2} + \cdots + (x - z)b_1 + b_0.$$

So,

$$p(x) = (x - z)q(x) + b_0 \quad (8.9)$$

where $q(x) = b_nx^{n-1} + b_{n-1}x^{n-2} + \cdots + b_2x + b_1$. This says that if z is the root of $p(x)$ then $b_0 = p(z) = 0$. Therefore, $p(x)$ can be factored as $p(x) = (x - z)q(x)$ for root z . So, $q(x)$ is a polynomial factor. This method of factorization is called synthetic division or Ruffini's rule.

Let's see Ruffini's rule in action by factoring $p(x)$, starting with the root 2:

$$\begin{array}{c|cccc} 2 & 1 & -6 & 11 & -6 \\ & 2 & -8 & 6 \\ \hline & 1 & -4 & 3 & 0 \end{array}$$

The factored polynomial is $x^2 - 4x + 3$. We can continue with another root 3:

$$\begin{array}{c|ccc} 3 & 1 & -4 & 3 \\ & 3 & -3 & . \\ \hline & 1 & -1 & 0 \end{array}$$

We are left with $x - 1$. So, we find that the roots are 3, 2, and 1.

► Newton-Horner method

We can use Newton's method to find a root z and then use Horner's method to deflate the polynomial. Newton's method for polynomials is simple. Using (8.9), we have $p(x) = (x - z)q(x) + b_0$ for some z . So, $p'(x) = q(x) + (x - z)q'(x)$, and therefore the derivative of p evaluated at $x = z$ is simply $p'(z) = q(z)$. Newton's method is then simply

$$x^{(n+1)} = x^{(n)} - \frac{p(x^{(n)})}{p'(x^{(n)})} = x^{(n)} - \frac{p(x^{(n)})}{q(x^{(n)})}$$

Example. Using Newton's method to compute $x^{(1)}$ starting with $x^{(0)} = 4$ for the polynomial $x^3 - 6x^2 + 11x - 6$. Let's determine $p(x^{(0)})$ and $p'(x^{(0)})$:

$$\begin{array}{r} 4 \\ \hline 1 & -6 & 11 & -6 \\ & 4 & -8 & 12 \\ \hline 1 & -2 & 3 & 6 \end{array}$$

$$\begin{array}{r} 4 \\ \hline 1 & -2 & 3 \\ & 4 & 8 \\ \hline 1 & 2 & 11 \end{array}$$

So, $p(4) = 6$ and $p'(4) = 11$. It follows that $x^{(1)} = 4 - \frac{6}{11}$. ◀

Let's summarize the Newton–Horner method for $a_n x^n + \dots + a_1 x + a_0$. Start with the coefficients $(a_n, a_{n-1}, \dots, a_0)$ and an initial guess x for a root. Now, we iterate. First, use Horner's method on $(a_n, a_{n-1}, \dots, a_0)$ with x to compute $(b_n, b_{n-1}, \dots, b_0)$. Then, use Horner's method on $(b_n, b_{n-1}, \dots, b_1)$ with x to compute $(c_n, c_{n-1}, \dots, c_1)$. Finally, compute one Newton step to update the guess $x \leftarrow x - b_0/c_1$. Once b_0 is sufficiently close to zero, we've found a root. Now, we deflate the polynomial by setting $(a_{n-1}, a_{n-2}, \dots, a_0) \leftarrow (b_n, b_{n-1}, \dots, b_1)$ and repeat the method to get the remaining roots. This method only works for real roots—we'd need to add error exceptions for missed complex roots, so we don't get stuck in a neverending loop.

► Companion matrix method

Another way to find the roots of a polynomial $p(x)$ is by finding the eigenvalues of a matrix whose characteristic polynomial is $p(x)$. The *companion matrix* of the polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1} + x^n$$

is the matrix

$$\mathbf{C}_n = \begin{bmatrix} 0 & 0 & \cdots & 0 & -a_0 \\ 1 & 0 & \cdots & 0 & -a_1 \\ 0 & 1 & \cdots & 0 & -a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -a_{n-1} \end{bmatrix}.$$

To show that the polynomial $p(x)$ is indeed the characteristic polynomial of \mathbf{C}_n , we can use Laplace expansion (also known as cofactor expansion) to evaluate $\det(\lambda \mathbf{I} - \mathbf{C}_n)$. For instance, consider the Laplace expansion using cofactors from

the bottom row

$$\begin{vmatrix} \lambda & 0 & 0 & 0 & a_0 \\ -1 & \lambda & 0 & 0 & a_1 \\ 0 & -1 & \lambda & 0 & a_2 \\ 0 & 0 & -1 & \lambda & a_3 \\ 0 & 0 & 0 & -1 & \lambda + a_4 \end{vmatrix} = (\lambda + a_4)\lambda^4 - (-1) \begin{vmatrix} \lambda & 0 & 0 & a_0 \\ -1 & \lambda & 0 & a_1 \\ 0 & -1 & \lambda & a_2 \\ 0 & 0 & -1 & a_3 \end{vmatrix}.$$

Continuing Laplace expansion in this way yields the characteristic polynomial $p(\lambda) = \lambda^5 + a_4\lambda^4 + a_3\lambda^3 + a_2\lambda^2 + a_1\lambda + a_0$.

• The Polynomials.jl, PolynomialRoots.jl, and FastPolynomialRoots.jl libraries all have functions `roots` that find roots from the eigenvalues of the companion matrix.⁶

► Other methods

There are several other popular methods of finding roots of polynomials such as Laguerre's method and the Jenkins–Traub method, which is used in Mathematica's `NSolve` function. These methods typically find a root and then deflate using Horner deflation.

8.6 Systems of nonlinear equations

Suppose that we have a system of equations. Such a system often arises when finding a nonlinear least squares fit or implementing an implicit solver for a partial differential equation. When we found the roots to a nonlinear equation, we had to contend with non-unique (but isolated) solutions. For systems, it is possible to have non-isolated points, so the well-posedness of the problem becomes more complicated.

► Newton's method

Consider a system of n nonlinear equations with n unknowns $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ where $f(\mathbf{x})$ is continuously differentiable. Suppose that we are located at \mathbf{x} near some zero \mathbf{x}^* . Then, we only need to determine the offset vector $\Delta\mathbf{x} = \mathbf{x}^* - \mathbf{x}$ to solve the problem. Taylor series expansion of $f(\mathbf{0})$ about the point \mathbf{x} is

$$0 = f(\mathbf{x}^*) = f(\mathbf{x}) + \mathbf{J}_f(\mathbf{x})\Delta\mathbf{x} + O(\|\Delta\mathbf{x}\|^2)$$

⁶The `FastPolynomialRoots.jl` package uses a Fortran wrapper of the algorithm by Aurentz, Mach, Vandebril and Watkins and can be significantly faster for large polynomials by using an eigenvalue solver that is $O(n^2)$ instead of $O(n^3)$.

where $\mathbf{J}_f(\mathbf{x})$ is the Jacobian matrix with elements given by $J_{ij} = \partial f_i / \partial x_j$. If the Jacobian matrix is nonsingular, we can invert this system of equations to get

$$\Delta \mathbf{x} = -(\mathbf{J}_f(\mathbf{x}))^{-1} f(\mathbf{x}) + O(\|\Delta \mathbf{x}\|^2)$$

and since $\mathbf{x}^* = \mathbf{x} + \Delta \mathbf{x}$

$$\mathbf{x}^* = \mathbf{x} - (\mathbf{J}_f(\mathbf{x}))^{-1} f(\mathbf{x}) + O(\|\Delta \mathbf{x}\|^2).$$

We can use this second-order approximation to develop an iterative method:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - (\mathbf{J}_f(\mathbf{x}^{(k)}))^{-1} f(\mathbf{x}^{(k)}). \quad (8.10)$$

In practice, we do not need to or necessarily want to compute the inverse of the Jacobian matrix explicitly. Rather at each step of the iteration we

$$\text{evaluate } \mathbf{J}_f(\mathbf{x}^{(k)}), \quad (8.11a)$$

$$\text{solve } \mathbf{J}_f(\mathbf{x}^{(k)}) \Delta \mathbf{x}^{(k)} = -f(\mathbf{x}^{(k)}), \quad (8.11b)$$

$$\text{update } \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta \mathbf{x}^{(k)}. \quad (8.11c)$$

If the Jacobian matrix is dense, it will take $O(n^2)$ operations to evaluate $\mathbf{J}_f(\mathbf{x}^{(k)})$ and an additional $O(n^3)$ operations to solve the system. If the system is sparse, the number of operations is $O(n)$ and $O(n^2)$, respectively. There are several things that we can do to speed up operations.

First, we could reuse the Jacobian. We don't need to compute the Jacobian matrix at every step, and instead we can evaluate it every few steps. By using LU decomposition to solve (8.11b), we can save and reuse the upper and lower triangular factors, so that each subsequent iteration take $O(n^2)$ operations.

Alternatively, we could use an iterative method such as Jacobi method, Gauss–Seidel, SOR, or the conjugate gradient methods to solve the linear system (8.11b). Iterative methods work by multiplying instead of factoring and require only $O(n^2)$ operations for dense matrices and only $O(n)$ operations for sparse matrices, but they often require many iterations to converge. But, because we are only computing an approximate solution $\mathbf{x}^{(k+1)}$ with each step, we don't need full convergence for the iterative method to be effective. These methods are discussed in detail in Chapter 5.

Also, we can use a finite difference approximation to evaluate the Jacobian matrix. By definition $J_{ij} = \partial f_i / \partial x_j$, so the j th column of the Jacobian matrix is simply the derivative of $f(\mathbf{x})$ with respect to x_j . The second-order approximation of the j th column of $\mathbf{J}_f(\mathbf{x})$ is

$$[\mathbf{J}_f(\mathbf{x})]_j = \frac{f(\mathbf{x} + h\xi_j) - f(\mathbf{x} - h\xi_j)}{2h} + O(h^2) \quad (8.12)$$

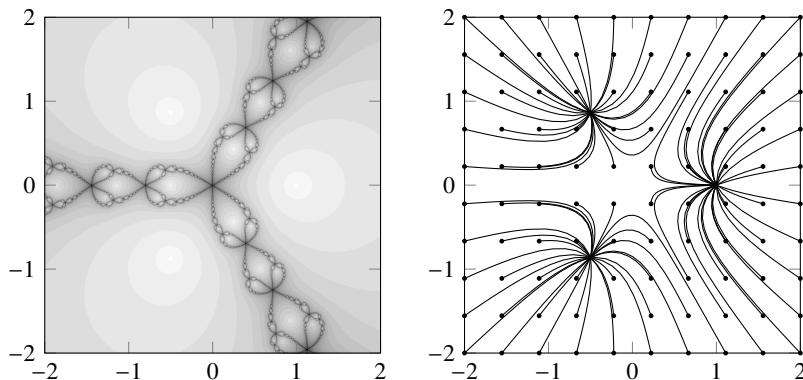


Figure 8.7: Newton fractal (left) for $z^3 - 1$. Darker shades of gray require more iterations to converge. Homotopy continuation (right) for the same problem.

for step size h where ξ_j is the j th vector in the standard basis of \mathbb{R}^n . Using the results of exercise 7.1, for a well-conditioned problem, we can typically take the step size h roughly equal to the cube root of machine epsilon. If $f(\mathbf{x})$ is a real-valued function, then we can also approximate the Jacobian using

$$[\mathbf{J}_f(\mathbf{x})]_j = \frac{\operatorname{Im} f(\mathbf{x} + ih\xi_j)}{h} + O(h^2)$$

taking h roughly equal to the square root of machine epsilon.

Example. Consider roots of the polynomial system

$$\begin{aligned} x^3 - 3xy^2 - 1 &= 0 \\ y^3 - 3x^2y &= 0. \end{aligned}$$

This problem is equivalent to finding the roots of $z^3 - 1$ in the complex plane, and the roots are $(1, 1)$, $(-\sqrt{3}/2, 1/2)$, and $(-\sqrt{3}/2, -1/2)$. If we use Newton's method to find the roots, then the forward orbit of an initial guess either converges to a root or fails to converge. Even, if the orbit does converge, it may not converge to the root closest to the initial guess. The set of starting points for which Newton's method eventually converges to a specific root is called the *basin of attraction*. The union of the boundaries of each basin of attraction is a Julia set. See Figure 8.7 above. \blacktriangleleft

► Secant-like methods

Recall that the secant method used two subsequent iterates $x^{(k)}$ and $x^{(k-1)}$ to compute an approximation of the tangent slope used in Newton's method. Just as

we extended Newton's method to a system of equations, we can also formally extend the secant method to a system of equations by replacing the slope terms used to define the slope used in the secant method (8.4) with their n -dimensional equivalents. We have

$$\mathbf{J}^{(k)} \Delta \mathbf{x}^{(k)} = \Delta \mathbf{f}^{(k)} \quad (8.13)$$

where $\mathbf{J}^{(k)}$ is an $n \times n$ approximation to the Jacobian matrix, $\Delta \mathbf{x}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}$, and $\Delta \mathbf{f}^{(k)} = f(\mathbf{x}^{(k)}) - f(\mathbf{x}^{(k-1)})$. Given an output vector $\Delta \mathbf{f}^{(k)}$ along with input vector $\Delta \mathbf{x}^{(k)}$, we are asked to find the matrix $\mathbf{J}^{(k)}$. The matrix $\mathbf{J}^{(k)}$ has n^2 unknowns and we have only n equations—a bit of a conundrum. How can we determine $\mathbf{J}^{(k)}$? It seems reasonable that $\mathbf{J}^{(k)}$ should be close to the previous $\mathbf{J}^{(k-1)}$, so perhaps we can approximate $\mathbf{J}^{(k)}$ using $\mathbf{J}^{(k-1)}$ with (8.13) as a constraint. In other words, minimize $\|\mathbf{J}^{(k)} - \mathbf{J}^{(k-1)}\|$ subject to the constraint $\mathbf{J}^{(k)} \Delta \mathbf{x}^{(k)} = \Delta \mathbf{f}^{(k)}$. What norm $\|\cdot\|$ do we choose? A smart choice is the Frobenius norm because the Frobenius norm of a matrix is a simple extension of the l^2 -norm for vectors, and minimizing the l^2 -norm results in a linear problem.

We can use the method of Lagrange multipliers. Take $\lambda \in \mathbb{R}^n$ and define

$$\mathcal{L}(\mathbf{J}^{(k)}, \lambda) = \frac{1}{2} \|\mathbf{J}^{(k)} - \mathbf{J}^{(k-1)}\|_{\text{F}}^2 + \lambda^T (\mathbf{J}^{(k)} \Delta \mathbf{x}^{(k)} - \Delta \mathbf{f}^{(k)}).$$

Setting $\nabla \mathcal{L}(\mathbf{J}^{(k)}, \lambda) = 0$ gives us

$$\mathbf{J}^{(k)} - \mathbf{J}^{(k-1)} + \lambda \Delta \mathbf{x}^{(k)\top} = 0 \quad \text{and} \quad \mathbf{J}^{(k)} \Delta \mathbf{x}^{(k)} = \Delta \mathbf{f}^{(k)}.$$

We just need to solve for λ . Right multiply the first equation by $\Delta \mathbf{x}^{(k)}$:

$$\mathbf{J}^{(k)} \Delta \mathbf{x}^{(k)} - \mathbf{J}^{(k-1)} \Delta \mathbf{x}^{(k)} + \lambda \|\Delta \mathbf{x}^{(k)}\|^2 = 0.$$

Now substitute in the second equation:

$$\lambda = -\frac{\Delta \mathbf{f}^{(k)} - \mathbf{J}^{(k-1)} \Delta \mathbf{x}^{(k)}}{\|\Delta \mathbf{x}^{(k)}\|^2}.$$

Therefore, we have

$$\mathbf{J}^{(k)} = \mathbf{J}^{(k-1)} + \frac{\Delta \mathbf{f}^{(k)} - \mathbf{J}^{(k-1)} \Delta \mathbf{x}^{(k)}}{\|\Delta \mathbf{x}^{(k)}\|^2} \Delta \mathbf{x}^{(k)\top}. \quad (8.14)$$

The finished algorithm is called *Broyden's method*:

```

choose an initial guess  $\mathbf{x}$ 
set  $\mathbf{J} \leftarrow \mathbf{J}_f(\mathbf{x})$ 
solve  $\mathbf{J} \Delta \mathbf{x} = -f(\mathbf{x})$ 
update  $\mathbf{x} \leftarrow \mathbf{x} + \Delta \mathbf{x}$ 
while  $\|\Delta \mathbf{x}\| < \text{tolerance}$ 
  [ compute  $\mathbf{J}$  using (8.14)
    solve  $\mathbf{J} \Delta \mathbf{x} = -f(\mathbf{x})$ 
    update  $\mathbf{x} \leftarrow \mathbf{x} + \Delta \mathbf{x}$ 
  ]

```

Broyden's method requires only $O(n)$ evaluations of $f(\mathbf{x}^{(k)})$ at each step to update the slope rather than $O(n^2)$ evaluations that are necessary to recompute the Jacobian matrix. In practice, errors in approximating an updated Jacobian grow with each iteration, and one should periodically restart the method with a fresh Jacobian.

Rather than solving $\mathbf{J}^{(k)} \Delta \mathbf{x}^{(k+1)} = -f(\mathbf{x}^{(k)})$ at each iteration, we can compute $\Delta \mathbf{x}^{(k+1)} = -\mathbf{J}^{(k)} f(\mathbf{x}^{(k)})$ directly with the help of the *Sherman–Morrison formula*

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^T)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^T\mathbf{A}^{-1}}{1 + \mathbf{v}^T\mathbf{A}^{-1}\mathbf{u}}$$

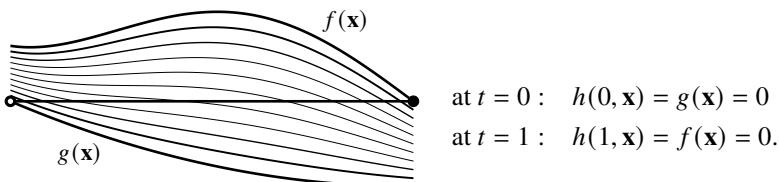
to update $(\mathbf{J}^{(k)})^{-1}$ given $(\mathbf{J}^{(k-1)})^{-1}$:

$$(\mathbf{J}^{(k)})^{-1} = (\mathbf{J}^{(k-1)})^{-1} + \frac{\Delta \mathbf{x}^{(k)} - (\mathbf{J}^{(k-1)})^{-1} \Delta \mathbf{f}^{(k)}}{\|\Delta \mathbf{f}^{(k)}\|^2} \Delta \mathbf{f}^{(k)T}.$$

8.7 Homotopy continuation

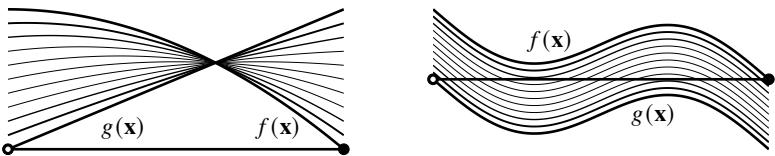
One might wonder if we could solve a simpler problem $g(\mathbf{x}) = 0$ as a means to solve $f(\mathbf{x}) = 0$. Well, we can. One way is to use a homotopy continuation. We say that two functions are *homotopic* if one can be continuously deformed into the other. The deformation is called a homotopy.

Suppose that we have a system of equations $f(\mathbf{x}) = 0$ and a simpler system $g(\mathbf{x}) = 0$ for which the roots are known or can be found easily. One way to define a homotopy is $h(t, \mathbf{x}) = t f(\mathbf{x}) + (1 - t) g(\mathbf{x})$ for $t \in [0, 1]$. Then



Simply put, when the parameter $t = 0$, the solution \mathbf{x} is a zero of g ; and when $t = 1$, the solution \mathbf{x} is a zero of f . We've exchanged the difficult problem of solving $f(\mathbf{x}) = 0$ for the easy problem of solving $g(\mathbf{x}) = 0$. It sounds a little too simple—what's the catch? We still need to track the solutions along the paths $h(t, \mathbf{x})$ for the parameter $t \in [0, 1]$. To do this we may need to solve a nonlinear differential equation.

Take the homotopy $h(t, \mathbf{x}(t)) = 0$ where $\mathbf{x}(t)$ is a zero of the system h for the parameter t . Let's assume that $\mathbf{x}(t)$ exists and is unique for all $t \in [0, 1]$. Otherwise, the method is ill-posed. This is not a trivial assumption—there may be no path connecting a zero of g with a zero of f or the path may become multivalued as the parameter t is changed:



We have $\mathbf{x}(0)$ is a zero of g . We want to find $\mathbf{x}(1)$ which is a zero of f . Differentiating $h(t, \mathbf{x}(t)) = 0$ with respect to t gives us

$$\frac{\partial h}{\partial t} + \mathbf{J}_h(\mathbf{x}) \frac{d\mathbf{x}}{dt} = 0$$

where the Jacobian matrix $\mathbf{J}_h(\mathbf{x}) = \partial h / \partial \mathbf{x}$ and $\partial h / \partial t$ and $d\mathbf{x} / dt$ are both vectors. Therefore,

$$\frac{d\mathbf{x}}{dt} = -\mathbf{J}_h^{-1}(\mathbf{x}) \frac{\partial h}{\partial t}.$$

Solving this differential equation gives us $\mathbf{x}(1)$.

A simple homotopy can be made by choosing $g(\mathbf{x}) = f(\mathbf{x}) - f(\mathbf{x}_0)$ for some \mathbf{x}_0 . Then \mathbf{x}_0 is a known zero of g . In this case

$$\begin{aligned} h(t, \mathbf{x}) &= tf(\mathbf{x}) + (1-t)[f(\mathbf{x}) - f(\mathbf{x}_0)] \\ &= f(\mathbf{x}) - (1-t)f(\mathbf{x}_0). \end{aligned} \tag{8.15}$$

The Jacobian matrix $\mathbf{J}_h(\mathbf{x}) = \mathbf{J}_f(\mathbf{x})$ and $\partial h / \partial t = f(\mathbf{x}_0)$. We'll need to solve the differential equation

$$\frac{d\mathbf{x}}{dt} = -\mathbf{J}_f^{-1}(\mathbf{x})f(\mathbf{x}_0).$$

given the initial conditions \mathbf{x}_0 . Solving such a differential equation is not typically trivial and often requires a numerical solver.

Example. Suppose that we want to solve the following system:

$$\begin{aligned} x^3 - 3xy^2 - 1 &= 0 \\ y^3 - 3x^2y &= 0. \end{aligned}$$

Define the homotopy as in (8.15) where we can take \mathbf{x}_0 to be whatever we want. The Jacobian matrix is

$$\mathbf{J}_f(\mathbf{x}) = \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ -6xy & 3y^2 - 3x^2 \end{bmatrix}.$$

If we take $\mathbf{x}_0 = (1, 1)$, then $f(\mathbf{x}_0) = (-3, -2)$ and we need to solve

$$\frac{d}{dt} \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = - \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ -6xy & 3y^2 - 3x^2 \end{bmatrix}^{-1} \begin{bmatrix} -3 \\ -2 \end{bmatrix}$$

with initial conditions $(x(0), y(0)) = (1, 1)$. We can solve this problem using the following code

```
using DifferentialEquations
f = z -> ((x,y)=tuple(z...); [x^3-3x*y^2-1; y^3-3x^2*y] )
df = (z,p,_) -> ((x,y)=tuple(z...);
    -[3x^2-3y^2 -6x*y; -6x*y 3y^2-3x^2])\p)
z₀ = [1,1]
sol = solve(ODEProblem(df,z₀,(0,1),f(z₀)))
sol.u[end]
```

The numerical solution given by `sol.u[end]` is 0.999, 0.001. To reduce the error, we can finish off the homotopy method by applying a step of Newton's method to $f(\mathbf{x})$ using `sol.u[end]` as the initial guess to get the root $(1, 0)$. See Figure 8.7 on page 210. ◀

The homotopy continuation method looks quite similar in form to Newton's method (8.12). In fact, Newton's method can be derived as a special case of homotopy continuation. Let's define the homotopy

$$h(t, \mathbf{x}) = f(\mathbf{x}) - e^{-t} f(\mathbf{x}_0). \quad (8.16)$$

At $t = 0$, we have $h(0, \mathbf{x}) = f(\mathbf{x}(0)) - f(\mathbf{x}_0) = 0$. And at $t = \infty$, we have $h(\infty, \mathbf{x}) = f(\mathbf{x}(\infty))$. We don't need to find the solution at $t = \infty$ at all. We just need find a solution for t large enough, so that the residual $e^{-t} f(\mathbf{x}_0)$ is small enough. We want to find the path $\mathbf{x}(t)$ along which $h(t, \mathbf{x}(t)) = 0$, so by differentiate (8.16) with respect to t :

$$\frac{d}{dt} h(t, \mathbf{x}) = \mathbf{J}_f(\mathbf{x}) \frac{d\mathbf{x}}{dt} + e^{-t} f(\mathbf{x}_0) = \mathbf{J}_f(\mathbf{x}) \frac{d\mathbf{x}}{dt} + f(\mathbf{x}(t)) = 0.$$

We can rewrite this equation as

$$\frac{d\mathbf{x}}{dt} = -\mathbf{J}_f^{-1}(\mathbf{x}) f(\mathbf{x}).$$

Perhaps the simplest way to numerically solve this differential equation is Euler's method, which uses a forward difference approximation for $d\mathbf{x}/dt$ with a step size Δt :

$$\frac{\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}}{\Delta t} = -\mathbf{J}_f^{-1}(\mathbf{x}^{(k)}) f(\mathbf{x}^{(k)}).$$

By taking $\Delta t = 1$, we simply have Newton's method

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{J}_f^{-1}(\mathbf{x}^{(k)}) f(\mathbf{x}^{(k)}).$$

The Euler method, like Newton's method and other fixed-point methods, has stability restrictions. One way to handle the stability restrictions is by taking

a smaller step size Δt . This, of course, requires more iterations. Homotopy continuation, too, effectively reduces the step size as a means of maintaining stability. In Chapter 12 we examine the forward Euler method and other differential equation solvers, many of which themselves rely on Newton’s method for their stability properties.

• Julia’s `HomotopyContinuation.jl` package can solve systems of polynomial equations.

8.8 Finding a minimum (or maximum)

In elementary calculus, we learned to find a relative minimum (or maximum) of a function by finding where its derivative equaled zero. So it shouldn’t be surprising that numerical methods for finding the extrema of a function are quite similar to numerical methods for finding the zeros of a function.

► Derivative-free methods

The bisection method, introduced in the beginning of this chapter to find the zero of a univariate function, looked for sign changes to successively choose between two brackets. To find a local minimum (or maximum), we’ll look to see where the function is increasing and where it’s decreasing. To do this, we’ll need to consider three brackets.

Suppose that $f(x)$ is unimodal (with a unique minimum) on the interval $[a, b]$. Divide this interval into three brackets $[a, x_1]$, $[x_1, x_2]$, and $[x_2, b]$ for some x_1 and x_2 . If $f(x_1) < f(x_2)$, then the minimum is in the superbracket $[a, x_2]$. We’ll move b to x_2 and select a new x_2 in the new interval $[a, b]$. On the other hand, if $f(x_1) > f(x_2)$, then the minimum is in the superbracket $[x_1, b]$. This time, we’ll move a to x_1 and select a new x_1 in the new interval $[a, b]$. So how should we select x_1 and x_2 ?

Without any additional knowledge about $f(x)$, we can argue by symmetry that size of bracket $[a, x_1]$ should equal the size of bracket $[x_2, b]$ and that the relative sizes of these brackets should be the same regardless of scale of the interval $[a, b]$. To simplify calculations, consider the initial interval $[0, 1]$ with points x_1 and x_2 at x and $1 - x$ for an unknown x . To maintain equal scaling at every iteration, we have that relationship $1 : (1 - x) = (1 - x) : x$. Equivalently, $x^2 - 3x + 1 = 0$ or $x = (3 - \sqrt{5})/2$. The ratio $\varphi = 1 : (1 - x) = (1 + \sqrt{5})/2$ is the golden ratio, also called the golden section. At each iteration, the intervals are scaled in length by $1/\varphi \approx 0.618$, so the golden section search method has linear convergence with a convergence rate of around 0.6.

A common derivative-free search technique for multivariable functions is the Nelder–Mead simplex method. The Nelder–Mead method uses an $n + 1$ polytope in n -dimensional space—a triangle in two-dimensional space, a tetrahedron in three-dimensional space, and so on. The function is evaluated at each of

the polytopes vertices. Through a sequence of reflections, expansions, and contractions, the polytope crawls like an amoeba across the domain seeking a downhill direction that takes it to the bottom.

► Newton's method

A faster way to get the minimum of a function $f(x)$ is by using Newton's method to find where $f'(x) = 0$:

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

Geometrically, each iteration of Newton's method for finding the zero of a function determines a straight line tangent to the function and returns the zero crossing. Each iteration of Newton's method for finding a minimum now determines a parabola tangent to the function and returns its vertex. If the function is convex, then Newton's method has quadratic convergence.

In higher dimensions, we formally replace the Jacobian matrix with the Hessian matrix and vector function with its gradient in (8.10) to get

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - (\mathbf{H}_f(\mathbf{x}^{(k)}))^{-1} \nabla f(\mathbf{x}^{(k)}).$$

The Hessian matrix $\mathbf{H}_f(\mathbf{x})$ has elements $h_{ij} = \partial^2 f / \partial x_i \partial x_j$. When the Hessian matrix is not positive-definite, we can use Levenberg–Marquardt regularization by substituting $(\mathbf{H}_f + \alpha \mathbf{I})$ for \mathbf{H}_f for some positive α .

Computing the Hessian can be costly, particularly when working with a large number of variables. The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm is a quasi-Newton (secant-like) method that approximates the Hessian and updates this approximation with each iteration. Alternatively, we can disregard the Hessian completely by using a general class of gradient descent methods. Gradient descent methods are common in machine learning and neural nets where the number of variables can be quite large and the data is noisy.

► Gradient descent methods

The gradient descent method computes

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla f(\mathbf{x}^{(k)}),$$

where the positive parameter α is called the learning rate controls the relative step size. Imagine descending into a deep valley shrouded in fog. Without any additional knowledge of the terrain, you opt to take the steepest path forward (the negative gradient) some distance. You then reorient yourself and move forward in the direction of the new steepest path. You continue in this manner until hopefully you find your way to the bottom. If you choose α to be too small,

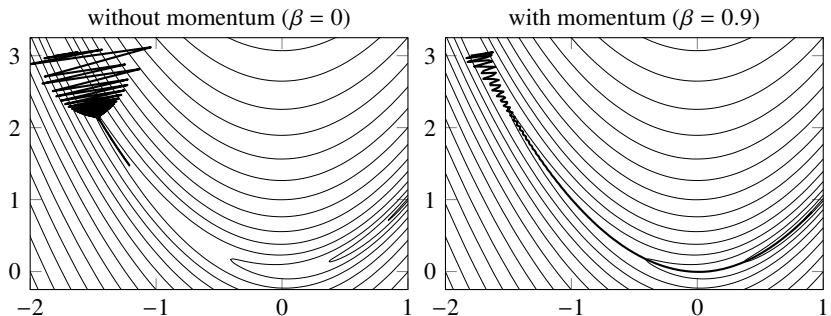


Figure 8.8: The gradient descent method with a learning rate $\alpha = 10^{-3}$ on the Rosenbrock function for 500 iterations without and with momentum.

you'll find yourself frequently checking your gradient and your progress will be slow. If α is too large and the valley is quite narrow, it's highly likely that you'll zigzag back and forth, up and down the valley walls.

In general, we don't need to walk in the direction of the gradient. Instead, we take $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)}$ where the vector $\mathbf{p}^{(k)}$ points us in some direction. One approach introduced by Russian mathematician Boris Polyak in the 1960s, often referred to as momentum or the heavy ball method, adds persistence or inertia into the walk

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \mathbf{p}^{(k)} \quad \text{with} \quad \mathbf{p}^{(k)} = -\nabla f(\mathbf{x}^{(k)}) + \beta \mathbf{p}^{(k-1)},$$

where β is a parameter usually between zero and one. Now, we are less likely to change directions, a bit like a heavy ball rolling down into the valley. The new factor reduces the oscillations and causes acceleration in the same direction resulting in larger step sizes.

Example. The Rosenbrock function, sometimes called Rosenbrock's banana,

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

is a narrow valley with rather steep sides and a relatively flat floor along the parabola $y = x^2$. It has a global minimum at $(x, y) = (1, 1)$. When we descend into the valley using gradient descent, the steep valley sides force us to take tiny steps to maintain stability. Once traveling along the floor of the valley, these tiny steps make for excruciatingly slow progress.

The following code uses the gradient descent method to find the minimum of the Rosenbrock function:

```
 $\nabla f = x \rightarrow [-2(1-x[1])-400x[1]*(x[2]-x[1]^2); 200(x[2]-x[1]^2)]$ 
```

```

x = [-1.8,3.0]; α = 0.001; p = [0,0]; β = 0.9
for i = 1:100
    p = -∇f(x) + β*p
    x += α*p
end

```

The trace is shown in the figure on the preceding page without momentum ($\beta = 0$) and with momentum ($\beta = 0.9$). Notice that the gradient descent without momentum initially bounces back and forth across the valley and never makes significant progress toward the minimum.

In practice, we can use the Optim.jl library to solve the problem:

```

using Optim
f = x -> (1-x[1])^2 + 100(x[2] - x[1]^2)^2
x₀ = [-1.8,3.0]
result = optimize(f, x₀, GradientDescent())
x = Optim.minimizer(result)

```

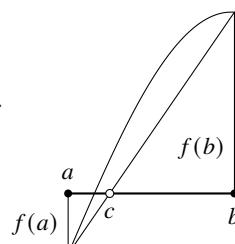
Methods include NelderMead (the default when no derivatives are provided), Newton, BFGS, GradientDescent, and GoldenSection among others. ◀

- The Optim.jl library includes several methods for finding a local minimum.

8.9 Exercises

8.1. The bisection method takes c at the midpoint of the bracket $[a, b]$. If $f(a)$ is closer to zero than $f(b)$ is to zero, then it seems that we might improve the method were we to take c closer to a than to b , and vice versa. One simple way to do this is by linearly interpolating between a and b rather than by averaging a and b :

$$c = \frac{f(a)b - f(b)a}{f(a) - f(b)}$$



This is exactly how the *regula falsi* method subdivides the brackets. Discuss the convergence of the *regula falsi* method. When does it outperform the bisection method? When does it underperform?

8.2. Use theorem 23 to determine the convergence rate of Newton's method.

8.3. Steffensen's method

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{q(x^{(k)})} \quad \text{with} \quad q(x) = \frac{f(x + f(x)) - f(x)}{f(x)}$$

is a variation on Newton's method that uses $q(x)$ as an approximation to the slope $f'(x)$. Show that this method is quadratically convergent under suitable conditions.

- ↳ 8.4. Newton's method converges linearly at a root of multiplicity n . Modify it so that it converges quadratically.
- ↳ 8.5. Newton's method is a special case of a more general class of root-finding algorithms called *Householder methods*:

$$x^{(k+1)} = x^{(k)} + p \frac{(1/f)^{(p-1)}(x^{(k)})}{(1/f)^{(p)}(x^{(k)})}$$

where p is a positive integer and $(1/f)^{(p)}(x)$ denotes the p th derivative of $1/f(x)$. Under suitable conditions, a Householder method has order $p+1$ convergence. Show that when $p=1$, we simply have Newton's method. Use a Householder method to extend the Babylonian method for calculating a square root to get a method with cubic order convergence.

8.6. Edmond Halley, a contemporary of Isaac Newton, invented his own root-finding method:

$$x^{(k+1)} = x^{(k)} - \frac{2f(x^{(k)})f'(x^{(k)})}{2[f'(x^{(k)})]^2 - f(x^{(k)})f''(x^{(k)})}.$$

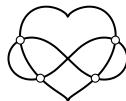
Derive Halley's method and show that it has cubic-order convergence.

- ↳ 8.7. Prove that for sufficiently smooth functions the order of convergence for the secant method is the golden ratio $p = (1 + \sqrt{5})/2$. Hint: when the error is sufficiently small, the order of convergence p can be defined using $|e^{(k+1)}| \approx C|e^{(k)}|^p$ for some positive C .
- 8.8. Use the Newton–Horner method to compute $x^{(1)}$ given $x^{(0)} = 4$ for the polynomial: $p(x) = 3x^5 - 7x^4 - 5x^3 + x^2 - 8x + 2$.
- ↳ 8.9. Suppose that we want to solve $f(x) = 0$. Taking $x = x - \alpha f(x)$ for some α , we get the fixed-point method $x^{(k+1)} = \phi(x^{(k)})$ where $\phi(x) = x - \alpha f(x)$. How does α affect the stability and convergence of the method?
- 8.10. Derive the Newton iteration step of the Q_rsqrt algorithm on page 179.

- 4 8.11. Use Newton's method, Broyden's method, and homotopy continuation to find the intersection of

$$(x^2 + y^2)^2 - 2(x^2 - y^2) = 0$$

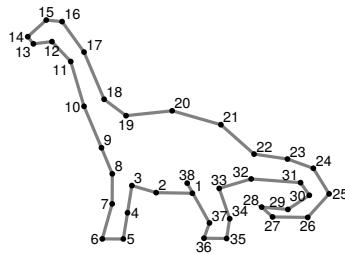
$$(x^2 + y^2 - 1)^3 - x^2 y^3 = 0$$



- 4 8.12. The elliptic curve secp256k1 $y^2 = x^3 + 7 \pmod{r}$ was popularized when Satoshi Nakamoto used it in Bitcoin's public-key digital signature algorithm. Implement the elliptic curve Diffie-Hellman (ECDH) algorithm using secp256k1. The order of the field is the prime number $r = 2^{256} - 2^{32} - 977$ and the published generator point P can be found in the "Standards for Efficient Cryptography 2 (SEC 2)" at <https://www.secg.org/sec2-v2.pdf>.
- 4 8.13. What are the convergence conditions for the gradient method and Polyak's momentum method applied to the quadratic function $f(x) = \frac{1}{2}mx^2$?

Chapter 9

Interpolation



Simple functions such as polynomials or rational functions are often used as surrogates for more complicated ones. These surrogate functions may be easier to integrate, or they might be solved exactly in a differential equation, or they can be plotted as a smooth curve, or they might make computation faster, or perhaps they allow us to fill in missing data points in a consistent manner. Suppose that we have some original function $f(x)$, even if it is not explicitly known. We would like to find another function that happens to be a good enough representation of the original one. For now, let's restrict our surrogate function $p_n(x)$ to the space of degree- n polynomials \mathbb{P}_n . We can largely break our selection of p_n into two categories:

Interpolation Find the polynomial $p_n \in \mathbb{P}_n$ that coincides with the function f at some given points or nodes x_0, x_1, \dots, x_n .

Best approximation Find the polynomial p_n that is the closest element in \mathbb{P}_n to f with respect to some norm: $p_n = \operatorname{arginf}_{q \in \mathbb{P}_n} \|f - q\|$.

In this chapter, we'll look at interpolation. In the next one, we examine best approximation.

9.1 Polynomial interpolation

Consider the space of degree- n polynomials \mathbb{P}_n . By choosing $n+1$ basis elements $\phi_0(x), \phi_1(x), \dots, \phi_n(x)$ in \mathbb{P}_n , we can uniquely express a polynomial $p_n \in \mathbb{P}_n$ as

$$p_n(x) = \sum_{k=0}^n c_k \phi_k(x).$$

Our immediate goal is determining constants c_0, c_1, \dots, c_n such that $p_n(x_i) = y_i$ given x_0, x_1, \dots, x_n and values y_0, y_1, \dots, y_n . In other words, find the coefficients c_k such that

$$\sum_{k=0}^n c_k \phi_k(x_i) = y_i.$$

In matrix form, this says

$$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \phi_2(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_n(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}. \quad (9.1)$$

There are several important polynomial bases we might consider. Common ones include the canonical basis, the Lagrange basis, and the Newton basis, shown in Figure 9.1 on the next page. We can build any degree- n polynomial by using a linear combination of elements from any of these bases. Note in particular that the Newton basis can be defined recursively $v_k(x) = (x - x_k)v_{k-1}(x)$. We'll see that this is a major advantage of the Newton polynomial basis. Before discussing the bases in-depth, let's prove the uniqueness of a polynomial interpolating function. One way to prove the following theorem is to show that the matrix in (9.1) using the canonical basis (called a Vandermonde matrix) is nonsingular. This approach is left as an exercise. Here we will prove it using Newton bases.

Theorem 24. *Given $n+1$ distinct points x_0, x_1, \dots, x_n and arbitrary values y_0, y_1, \dots, y_n , there exists a unique polynomial $p_n \in \mathbb{P}_n$ such that $p_n(x_i) = y_i$ for all $i = 0, 1, \dots, n$.*

Proof. We'll prove existence using induction. For $n = 0$, choose $p_0(x) = y_0$. Suppose that there is some p_{k-1} such that $p_{k-1}(x_i) = y_i$ for all $i = 0, 1, \dots, k-1$. Let

$$p_k(x) = p_{k-1}(x) + c(x - x_0)(x - x_1) \cdots (x - x_{k-1})$$

for some c . Then $p_k \in \mathbb{P}_k$ and $p_k(x_i) = p_{k-1}(x_i) = y_i$ for $i = 0, 1, \dots, k-1$. To determine the value c for which $p_k(x_k) = y_k$, take c that solves

$$y_k = p_{k-1}(x_k) + c(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1}).$$

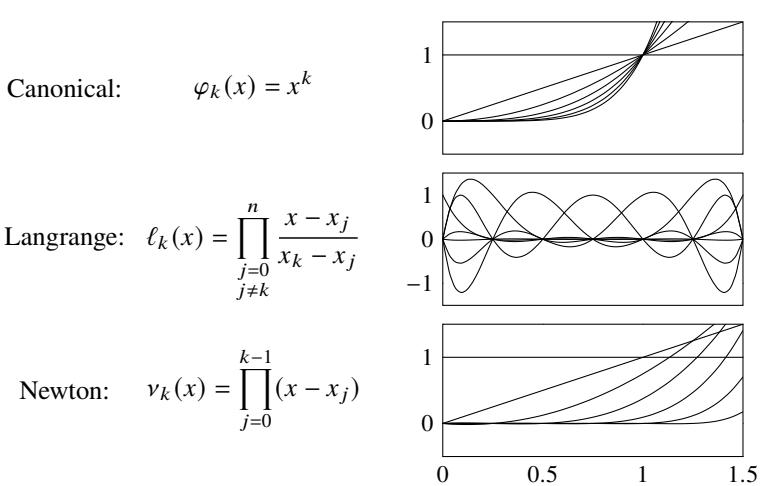


Figure 9.1: Polynomial basis functions for $k = 0, \dots, 6$ with nodes at $x_j = j/4$.

That is,

$$c = \frac{y_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1})}.$$

We now prove uniqueness. Suppose that there were two polynomials p_n and q_n . Then the polynomial $p_n - q_n$ has the property $(p_n - q_n)(x_i) = 0$ for all $i = 0, \dots, n$. So, $p_n - q_n$ has $n + 1$ roots. But because $p_n - q_n \in \mathbb{P}_n$, it can have at most n roots unless it is identically zero. So, p_n must equal q_n . \square

► Canonical basis

Consider the problem of fitting a degree- n polynomial $y = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n$ to a set of data points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ using the canonical or standard basis $\{1, x, \dots, x^n\}$. Given $n + 1$ distinct points, we can determine a n th-degree polynomial by solving the *Vandermonde* system $\mathbf{V}\mathbf{c} = \mathbf{y}$. From (9.1),

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Solving the system takes $O(n^3)$ operations, and numerically the Vandermonde matrix is increasingly ill-conditioned as the degree increases.

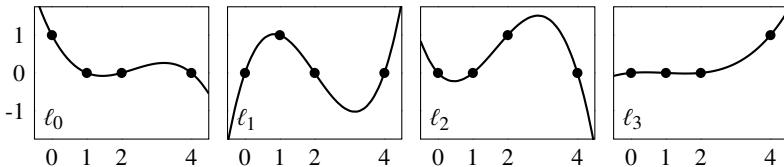
It's easy to construct a Vandermonde matrix using broadcasting: $x.^{(\theta:n)'}.$ For something with more features, try the `SpecialMatrices.jl` function `Vandermonde` that overloads the \ operator with the $O(n^2)$ -time Björck and Pereyra algorithm.

► Lagrange polynomial basis

For a Lagrange polynomial

$$y_j = \sum_{i=0}^n y_i \ell_i(x_j) \quad \text{with} \quad \ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}.$$

That is, $\ell_i(x_j) = \delta_{ij}$. For a quick confirmation of this, note where $\ell_i(x)$ is 0 and 1 in the Lagrange basis for $\{x_0, x_1, x_2, x_3\} = \{0, 1, 2, 4\}$:



Because $\ell_i(x_j) = \delta_{ij}$, (9.1) becomes simply

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Hence, it is trivial to find the approximating polynomial using the Lagrange form. But the Lagrange form does have some drawbacks. Each evaluation of a Lagrange polynomial requires $O(n^2)$ additions and multiplications. They are difficult to differentiate and integrate. If you add new data pair (x_n, y_n) , you must start over from scratch. And the computations can be numerically unstable.

► Newton polynomial basis

The Newton polynomial basis fixes each of these limitations of the Lagrange form by recursively defining the basis elements $v_i(x)$:

$$1, (x - x_0), (x - x_0)(x - x_1), \dots, \prod_{i=0}^{n-1} (x - x_i).$$

Defining $f(x_i) = y_i$ and using the Newton basis, (9.1) becomes

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & (x_1 - x_0) & 0 & \cdots & 0 \\ 1 & (x_2 - x_0) & \prod_{i=0}^1 (x_2 - x_i) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & (x_n - x_0) & \prod_{i=0}^1 (x_n - x_i) & \cdots & \prod_{i=0}^{n-1} (x_n - x_i) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}.$$

The lower triangular system can be solved using forward substitution. Note that the coefficient c_i is only a function of x_0, x_1, \dots, x_i . To denote this dependency we will use the notation $c_i = f[x_0, x_1, \dots, x_i]$. For example, using forward substitution to find $\{c_0, c_1, c_2\}$ for three nodes

$$\begin{aligned} f[x_0] &= c_0 = f(x_0) \\ f[x_0, x_1] &= c_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\ f[x_0, x_1, x_2] &= c_2 = \frac{\frac{f(x_2) - c_0 - c_1(x_2 - x_0)}{(x_2 - x_1)(x_2 - x_0)}}{\frac{f(x_2) - f(x_0)}{x_2 - x_0} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}} \\ &= \frac{x_2 - x_0}{x_2 - x_1} \end{aligned}$$

Note that c_2 is the coefficient for x^2 term of $p_2(x)$. Because $p_2(x)$ is unique by theorem 24, it doesn't matter in which order we take the nodes $\{x_0, x_1, x_2\}$ for $f[x_0, x_1, x_2]$. So, by interchanging x_0 and x_1 we have

$$f[x_0, x_1, x_2] = \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0} = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}.$$

We can compute the rest of the coefficients recursively using *divided differences*, starting with $f[x_i] = f(x_i)$ for $i = 0, 1, \dots, k$ and then constructing

$$f[x_j, \dots, x_k] = \frac{f[x_{j+1}, \dots, x_k] - f[x_j, \dots, x_{k-1}]}{x_k - x_j}.$$

It's convenient to arrange the divided differences as a table to aid in calculation¹

x_0	$f[x_0]$	$ $	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, x_1, x_2, x_3]$
x_1	$f[x_1]$	$ $	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$	
x_2	$f[x_2]$	$ $	$f[x_2, x_3]$		
x_3	$f[x_3]$				

¹Nineteenth-century mathematician and inventor Charles Babbage, sometimes called the grandfather of the modern computer, invented a mechanical device he called the “difference engine.” The steel and brass contraption was intended to automate and make error-free the mathematical drudgery of computing divided difference tables. Ada Lovelace, enamored by Babbage’s even grander designs, went on to propose the world’s first computer program.

Then, the coefficients c_i come from each column of the first row

$$\{c_0, c_1, c_2, c_3\} = \{f[x_0], f[x_0, x_1], f[x_0, x_1, x_2], f[x_0, x_1, x_2, x_3]\},$$

and we can use Horner's method to evaluate the interpolating polynomial

$$p_n(x) = c_0 + (x - x_0) \left(c_1 + (x - x_1) (c_2 + (x - x_2) c_3) \right).$$

Example. Let's find the interpolating polynomial for the function $f(x)$ where

$$\begin{array}{rccccc} x & = & 1 & 3 & 5 & 6 \\ f(x) & = & 0 & 4 & 5 & 7 \end{array}$$

We first build the divided-difference table. We'll use \square to denote unknown values.

$$\begin{array}{cc|ccccc} & & 1 & 0 & \square & \square & \square & \\ \text{From} & & 3 & 4 & \square & \square & & \\ & & 5 & 5 & \square & & & \\ & & 6 & 7 & & & & \end{array} \quad \text{we have} \quad \begin{array}{cc|ccccc} & & 1 & 0 & 2 & -3/8 & 7/40 \\ & & 3 & 4 & 1/2 & & 1/2 \\ & & 5 & 5 & 2 & & \\ & & 6 & 7 & & & \end{array} .$$

To fill in the upper left value, we take $(4 - 0)/(3 - 1)$ to get 2. The value below it is $(5 - 4)/(5 - 3)$ or $1/2$. Now, we can use these new values to compute to unknown value to their right: $(1/2 - 2)/(5 - 1) = -3/8$. The rest of the table is filled-out in a similar manner. From the first row, we have the coefficients of the interpolating polynomial

$$p_n(x) = 0 + 2(x - 1) - 3/8(x - 1)(x - 3) + 7/40(x - 1)(x - 3)(x - 5),$$

which can be evaluated using Horner's method

$$p_n(x) = 0 + (x - 1) \left(2 + (x - 3) \left(-\frac{3}{8} + (x - 5) \cdot \frac{7}{40} \right) \right). \quad \blacktriangleleft$$

► Hermite interpolation

We can extend the previous discussion to find interpolating polynomials that not only match the values of functions at nodes but also match their derivatives at these nodes. That is, let's find a unique polynomial $p_n \in \mathbb{P}_n$ such that $p_n^{(j)}(x_i) = f^{(j)}(x_i)$ for all $i = 0, 1, \dots, k$ and $j = 0, 1, \dots, \alpha_i$. The polynomial p_n is called the *Hermite interpolation* of f at the points x_i .

We are already familiar with Hermite interpolation when we have only one node—it's simply the Taylor polynomial approximation. When we have multiple nodes we can compute the Newton form of the Hermite interpolation by using a table of divided differences. We'll use the mean value theorem for divided differences.

Theorem 25. Given $n + 1$ distinct nodes $\{x_0, x_1, \dots, x_n\}$, there is a point ξ on the interval bounded by the smallest and largest nodes such that the n th divided difference of a function $f(x)$ is $f[x_0, x_1, \dots, x_n] = f^{(n)}(\xi)/n!$.

Proof. Let $p_n(x)$ be the interpolating polynomial of $f(x)$ using the Newton basis. The leading term of $p_n(x)$ is $f[x_0, \dots, x_n](x - x_0) \dots (x - x_{n-1})$. Take $g(x) = f(x) - p_n(x)$, which has $n + 1$ zeros at each of the nodes $\{x_0, x_1, \dots, x_n\}$. By Rolle's theorem, $g'(x)$ has at least n zeros on the interval, $g''(x)$ has at least $n - 1$ zeros, and so on until $g^{(n)}$ has at least one zero on the interval. We'll take ξ to be one of these zeros. So,

$$0 = g^{(n)}(\xi) = f^{(n)}(\xi) - p_n^{(n)}(\xi) = f^{(n)}(\xi) - n!f[x_0, x_1, \dots, x_n]$$

and it follows that $f[x_0, x_1, \dots, x_n] = f^{(n)}(\xi)/n!$. \square

As a consequence of the mean value theorem, for any node x_i of $n + 1$ nodes, the divided difference is

$$f[x_i, x_i, \dots, x_i] = \lim_{x_1, \dots, x_n \rightarrow x_i} f[x_0, x_1, \dots, x_n] = f^{(n)}(x_i)/n!.$$

For example, $f[1] = f(1)$, $f[1, 1] = f'(1)$, $f[1, 1, 1] = f''(1)/2$, and so on.

Example. Compute the polynomial p of minimal degree satisfying

$$p(1) = 2, \quad p'(1) = 3, \quad p(2) = 6, \quad p'(2) = 7, \quad p''(2) = 8.$$

To do this we compute the divided differences. First note that $p[1, 1] = p'(1) = 3$, $p[2, 2] = p'(2) = 7$, and $p[2, 2, 2] = p''(2)/2! = 8/2 = 4$. Then we fill in the divided difference table starting with \square to denote unknown values.

From	1 2	3 \square \square \square		1 2	3 1 2 \square	
	1 2	\square \square \square		1 2	4 3 1	
	2 6	7 4		2 6	7 4	
	2 6	\square		2 6	\square	
	2 6			2 6		

By taking the coefficients from the first row, we conclude that $p(x)$ is

$$2 + 3(x - 1) + 1(x - 1)^2 + 2(x - 1)^2(x - 2) - 1(x - 1)^2(x - 2). \quad \blacktriangleleft$$

9.2 How good is polynomial interpolation?

Theorem 26. Let $p_n(x)$ be a degree- n interpolating polynomial of a sufficiently smooth function $f(x)$ at $n + 1$ nodes x_0, x_1, \dots, x_n on the interval $[a, b]$. Then for each $x \in [a, b]$ there is a $\xi \in [a, b]$ such that the pointwise error

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^n (x - x_i).$$

Proof. If x is a node, then $f(x) - p_n(x) = 0$, and the claim is clearly true. So, let x be any point other than a node $\{x_0, x_1, \dots, x_n\}$. Define

$$g(z) = (f(z) - p_n(z)) - (f(x) - p_n(x)) \frac{\prod_{i=0}^n (z - x_i)}{\prod_{i=0}^n (x - x_i)}.$$

Note that x is now a parameter of $(n+1)$ -differentiable function g and that

$$g(x) = g(x_0) = g(x_1) = \dots = g(x_n) = 0.$$

That is to say, g has at least $n+2$ zeros on the interval $[a, b]$. By Rolle's theorem $g'(x)$ has at least $n+1$ zeros on the interval, $g^{(2)}(x)$ has at least n zeros on the interval, and so on until $g^{(n+1)}$ has at least one zero on the interval. Let's call this zero ξ . We can explicitly compute the $(n+1)$ st derivative of $g(z)$ as

$$g^{(n+1)}(z) = \left(f^{(n+1)}(z) - 0 \right) - \frac{f(x) - p_n(x)}{\prod_{i=0}^n (x - x_i)} (n+1)!,$$

where $p_n^{(n+1)}(z) = 0$ because p_n has degree at most n . At $z = \xi$,

$$0 = g^{(n+1)}(\xi) = f^{(n+1)}(\xi) - \frac{f(x) - p_n(x)}{\prod_{i=0}^n (x - x_i)} (n+1)!.$$

So, it follows that

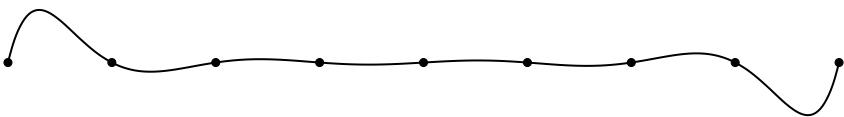
$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^n (x - x_i). \quad \square$$

The error estimate above can be extended to Hermite interpolation as well. The proof of the following theorem is similar to that of theorem 26 and is left as an exercise.

Theorem 27. Let x_0, x_1, \dots, x_n be nodes on the interval $[a, b]$ and let f be a sufficiently smooth function on that interval. If $p(x)$ is an interpolating polynomial of degree at most $2n+1$ with $p(x_i) = f(x_i)$ and $p'(x_i) = f'(x_i)$, then for each $x \in [a, b]$ there is a $\xi \in [a, b]$ such that the pointwise error

$$f(x) - p_n(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{i=0}^n (x - x_i)^2.$$

We can see from the terms of the expression for pointwise error that it is affected by the smoothness of the function $f(x)$ and the number and placement of the nodes. We can't control the smoothness of $f(x)$, but we can control the number and position of the nodes. Take a look at the plot of $\prod_{i=0}^8 (x - x_i)$ for nine equally spaced nodes:



Near either end of the interval the value $|\prod_{i=0}^8 (x - x_i)|$ is quite large, and if we add more uniformly spaced nodes the overshoot becomes larger. This behavior, called *Runge's phenomenon*, explains why simply adding nodes does not necessarily improve accuracy. We can instead choose the position of the nodes to minimize the uniform norm² $\|\prod_{i=0}^8 (x - s_i)\|_\infty$. And this is exactly what we do by choosing nodes s_i that are zeros of the Chebyshev polynomial:



Now the black curve of $\prod_{i=0}^8 (x - s_i)$ with Chebyshev nodes s_i is overlaid on the lighter gray curve of $\prod_{i=0}^8 (x - x_i)$ with uniformly spaced nodes x_i . The Chebyshev nodes decrease the error near the ends of the interval at the cost of increasing the error near the center.

Let's take a quick peek at Chebyshev polynomials and their zeros—we'll come back to them in the next chapter. Chebyshev polynomials can be defined using

$$T_n(x) = \cos(n \arccos x) \quad (9.2)$$

over the interval $[-1, 1]$. While this definition might seem a bit tightly packed, one interpretation of it is provided by Forman Acton in a parenthetical comment of his classic 1970 book *Numerical Methods that Work*: “they are actually cosine curves with a somewhat disturbed horizontal scale, but the vertical scale has not been touched.” See the figure on the following page. Several important properties can be derived directly from this definition. Chebyshev polynomials can be defined using the recursion formula:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x) \text{ where } T_0(x) = 1 \text{ and } T_1(x) = x.$$

²Also called the Chebyshev norm, the infinity norm, and the supremum norm.

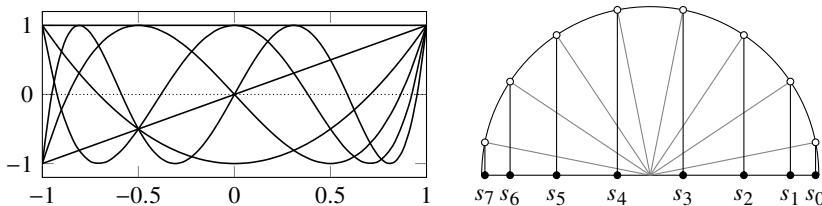


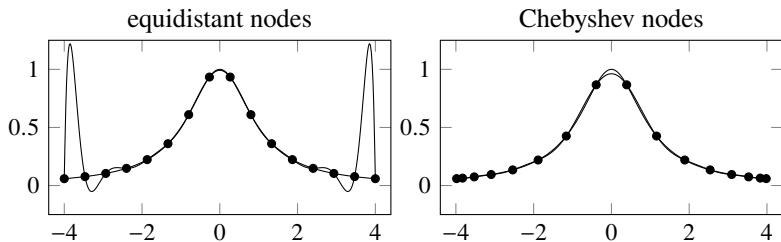
Figure 9.2: Chebyshev polynomials $T_0(x), T_1(x), \dots, T_5(x)$ (left). Chebyshev nodes of $T_8(x)$ are abscissas of eight equispaced nodes on the unit circle (right).

The zeros of the Chebyshev polynomial, called Chebyshev nodes, occur at $s_i = \cos((i - \frac{1}{2})\pi/n)$. These Chebyshev nodes are also the abscissas of equispaced points along the unit semicircle. See the figure above. The Chebyshev polynomial's extrema occur at $\hat{s}_i = \cos(i\pi/n)$ and take the values $T_n(\hat{s}_i) = (-1)^i$. It follows that Chebyshev polynomials are bounded below by -1 and above by $+1$ for $x \in [-1, 1]$. Finally, the leading coefficient of $T_n(x)$ is 2^{n-1} .

Theorem 28. *Every polynomial of degree n with leading coefficient 2^{n-1} has a maximum absolute value of at least 1 in the interval $[-1, 1]$.*

Proof. Let p_n be a polynomial of degree n and leading coefficient 2^{n-1} . Then $T_n - p_n$ is a polynomial of degree $n - 1$ or less. Suppose that $|p_n(x)| < 1$ for all $x \in [-1, 1]$. At the Chebyshev extrema, $T_n(\hat{s}_i) = +1$ for even i and $p_n(\hat{s}_i) < 1$, so it follows that $p_n(\hat{s}_i) - T_n(\hat{s}_i) < 0$. Similarly, $T_n(\hat{s}_i) = -1$ for odd i and $p_n(\hat{s}_i) > -1$, so it follows that $p_n(\hat{s}_i) - T_n(\hat{s}_i) > 0$. This means that the polynomial $p_n - T_n$ is alternating between positive and negative over the $n + 1$ abscissae. So, it has at least n roots in $[-1, 1]$. But $p_n - T_n$ is a polynomial of degree at most $n - 1$, so it can have at most $n - 1$ roots—a contradiction. Therefore, there must be some $x \in [-1, 1]$ such that $|p_n(x)| \geq 1$. \square

Example. In 1885, at the age of seventy, Karl Weierstrass proved his famous theorem that any continuous function can be uniformly approximated by a sequence of polynomials. Specifically, the Weierstrass approximation theorem states that if f is a continuous function on the interval $[a, b]$, then for every $\varepsilon > 0$ there exists a polynomial p_n such that $|f(x) - p_n(x)| < \varepsilon$ for all $x \in [a, b]$. In 1901, Weierstrass' former student Carl Runge, demonstrated that the theorem need not hold for polynomials interpolated through equidistant nodes. Namely, a polynomial interpolant of the function $(x^2 + 1)^{-1}$ using equidistant nodes produces large oscillations near the endpoints that grow unbounded as the polynomial degree increases.



Weierstrass' theorem does hold for a polynomial through Chebyshev nodes. Chebyshev nodes eliminate Runge's phenomenon near the edges but increases the pointwise error around the origin. The function $(x^2 + 1)^{-1}$ is called the Runge function or the witch of Agnesi.³

◀

9.3 Splines

One problem with polynomial interpolation is that polynomials basis elements are global by nature, so they are ill-suited to approximate functions with predominant local features like discontinuities. We've already seen this in Runge's phenomenon. One alternative is to use radial basis functions, discussed in exercise 9.3. Another alternative is to use a piecewise polynomial function called a *spline*. In this section, we will focus on cubic splines, but the ideas will apply to any degree spline.

► Cubic splines

A cubic spline $s(x)$ is a C^2 -continuous, piecewise-cubic polynomial through points (x_j, y_j) known as *knots*. Let x_0, x_1, \dots, x_n be $n + 1$ distinct nodes with $a = x_0 < x_1 < \dots < x_n = b$. And let $\{s_0(x), s_1(x), \dots, s_{j-1}(x)\}$ be a set of cubic polynomials. We want a function $s(x)$ such that $s(x) = s_j(x)$ over the interval $x \in (x_j, x_{j+1})$ for each j . To determine $s(x)$ we must match the cubic polynomials $s_j(x)$ at the boundaries of the subintervals. A cubic polynomial has four degrees of freedom. Hence, we have four unknowns for each of the n subintervals—a total of $4n$ unknowns. Let's look at these constraints. The spline interpolates each of the $n + 1$ knots giving us $n + 1$ constraints. Because $s(x) \in C^2$, it follows that $s_{j-1}(x_j) = s_j(x_j)$, $s'_{j-1}(x_j) = s'_j(x_j)$, and $s''_{j-1}(x_j) = s''_j(x_j)$ for $j = 1, 2, \dots, n - 1$. This gives us another $3(n - 1)$ constraint for a total of $4n - 2$ constraints. This leaves us with two degrees of freedom remaining.

³The witch of Agnesi was named after by the Italian mathematician Maria Agnesi, who described it in her 1748 book *Foundations of Analysis for Italian Youth*. The name comes from John Colson's mistranslation of *aversiera* ("versine curve") as *avversiera* ("witch"). The versine (or versed sine) function is $\text{versin } \theta = 1 - \cos \theta$. It's one of several oddball trigonometric functions that are rarely used in modern mathematics: versine, coversine, haversine, archacovercosine,

We'll need to enforce two additional constraints to remove these remaining degrees of freedom and get a unique solution. Common constraints include the complete or clamped boundary condition where $s'(a)$ and $s'(b)$ are specified; natural boundary condition where $s''(a) = s''(b) = 0$; periodic boundary condition where $s'(a) = s'(b)$ and $s''(a) = s''(b)$; and not-a-knot condition. The not-a-knot condition uses the same cubic polynomial across the first two subintervals by setting $s'''_1(x_1) = s'''_1(x_1)$, effectively making x_1 "not a knot." It does the same across the last two subintervals by setting $s'''_n(x_{n-1}) = s'''_n(x_{n-1})$.

Before computer-aided graphic design programs, yacht designers and aerospace engineers used thin strips of wood called splines to help draw curves. The thin beams were fixed to the knots by hooks attached to heavy lead weights and either clamped at the boundary to prescribe the slope (complete splines) or unclamped so that the beam would be straight outside the interval (natural splines). If $y(x)$ describes the position of the thin wood beam, then the curvature is given by

$$\kappa(x) = \frac{y'''(x)}{(1 + y'(x)^2)^{3/2}}$$

and the deformation energy of the beam is

$$E = \int_a^b \kappa^2(x) dx = \int_a^b \left(\frac{y''(x)}{(1 + y'(x)^2)^{3/2}} \right)^2 dx.$$

For small deformations $E \approx \int_a^b [y''(x)]^2 dx = \|y''\|_2^2$. Physically, the beam takes the shape that minimizes the energy. This implies that the cubic spline is the smoothest interpolating polynomial.

Theorem 29. *Let s be an interpolating cubic spline of the function f at the nodes $a = x_0 < \dots < x_n = b$ and let $y \in C^2[a, b]$ be an arbitrary interpolating function on f . Then $\|s''\|_2 \leq \|y''\|_2$ if the spline has either complete, natural, or periodic boundary conditions.*

Proof. Note that $y'' = s'' + (y'' - s'')$ and hence

$$\int_a^b (y'')^2 dx = \int_a^b (s'')^2 dx + 2 \int_a^b s''(y'' - s'') dx + \int_a^b (y'' - s'')^2 dx.$$

The last term is nonnegative, so we just need to show that the middle term vanishes. Integrating by parts

$$\int_a^b s''(y'' - s'') dx = s''(y' - s') \Big|_a^b - \int_a^b s'''(y' - s') dx.$$

The boundary terms are zero by the imposed boundary conditions. Over the subinterval $x \in (x_i, x_{i+1})$ the cubic spline $s(x)$ is a cubic polynomial $s_i(x)$ with

$s'''(x) = s_i'''(x) = d_i$ for some constant d_i . Then

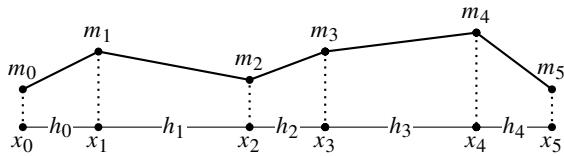
$$\begin{aligned} \int_a^b s'''(y' - s') dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} d_i (y' - s'_i) dx \\ &= \sum_{i=0}^{n-1} d_i [(y(x_{i+1}) - s_i(x_{i+1})) - (y(x_i) - s_i(x_i))] = 0. \quad \square \end{aligned}$$

► Computation of cubic splines

Let's find an efficient method to construct a cubic spline interpolating the values $y_i = f(x_i)$ for $a = x_0 < x_1 < \dots < x_n = b$. If $s(x)$ is a C^2 -piecewise cubic polynomial, then $s''(x)$ is a piecewise linear function. The second-order derivative of $s_i(x)$ over the subinterval $[x_i, x_{i+1}]$ is the line

$$s_i''(x) = m_{i+1} \frac{x - x_i}{h_i} + m_i \frac{x_{i+1} - x}{h_i}$$

where $h_i = x_{i+1} - x_i$ is the segment length and $m_i = s_i''(x_i)$:



We integrate twice to find

$$s_i(x) = \frac{1}{6} \frac{m_{i+1}}{h_i} (x - x_i)^3 + \frac{1}{6} \frac{m_i}{h_i} (x_{i+1} - x)^3 + A_i(x - x_i) + B_i, \quad (9.3)$$

where the constants A_i and B_i are determined by imposing the values at the ends of the subintervals $s_i(x_i) = y_i$ and $s_i(x_{i+1}) = y_{i+1}$:

$$y_i = \frac{1}{6} \frac{m_i}{h_i} h_i^3 + B_i \quad \text{and} \quad y_{i+1} = \frac{1}{6} \frac{m_{i+1}}{h_i} h_i^3 + A_i h_i + B_i.$$

From this, we have

$$B_i = y_i - \frac{1}{6} m_i h_i^2 \quad \text{and} \quad A_i = \frac{y_{i+1} - y_i}{h_i} - \frac{1}{6} (m_{i+1} - m_i) h_i. \quad (9.4)$$

We still need to determine m_i . To find these, we match the first derivatives $s(x)$ at x_i : $s'_{i-1}(x_i) = s'_i(x_i)$.

$$s'_i(x) = \frac{1}{2} \frac{m_{i+1}}{h_i} (x - x_i)^2 - \frac{1}{2} \frac{m_i}{h_i} (x_{i+1} - x)^2 + A_i.$$

for $i = 1, 2, \dots, n - 1$. Substituting in for A_{i-1} and A_i , we have

$$\begin{aligned}s'_{i-1}(x_i) &= \frac{1}{2} \frac{m_i}{h_{i-1}} (h_{i-1})^2 + \frac{y_i - y_{i-1}}{h_{i-1}} - \frac{1}{6} (m_i - m_{i-1}) h_{i-1}, \\ s'_i(x_i) &= -\frac{1}{2} \frac{m_i}{h_i} (h_i)^2 + \frac{y_{i+1} - y_i}{h_i} - \frac{1}{6} (m_{i+1} - m_i) h_i.\end{aligned}$$

Equating the two, we are left with

$$h_{i-1} m_{i-1} + 2(h_i + h_{i-1}) m_i + h_i m_{i+1} = 6 \left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right)$$

for $i = 1, 2, \dots, n - 1$. We have $n - 1$ equations and $n + 1$ unknowns. To close the system we need to add the boundary conditions. Let's impose natural boundary conditions: $s''_3(a) = s''_3(b) = 0$. Then $m_0 = 0$ and $m_n = 0$, and we have the following system:

$$\begin{bmatrix} 1 & & & & \\ \alpha_0 & \beta_1 & \alpha_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \alpha_{n-2} & \beta_{n-1} & \alpha_{n-1} \\ & & & & 1 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ \vdots \\ m_{n-1} \\ m_n \end{bmatrix} = \begin{bmatrix} 0 \\ \gamma_1 \\ \vdots \\ \gamma_{n-1} \\ 0 \end{bmatrix}$$

where

$$\alpha_i = h_i, \quad \beta_i = 2(h_i + h_{i-1}), \quad \text{and } \gamma_i = 6 \left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right). \quad (9.5)$$

Note that we can rewrite the $n + 1$ system as an $n - 1$ system by eliminating the first and last equations

$$\begin{bmatrix} \beta_1 & \alpha_1 & & & \\ \alpha_1 & \beta_2 & \alpha_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \alpha_{n-3} & \beta_{n-2} & \alpha_{n-2} \\ & & & \alpha_{n-2} & \beta_{n-1} \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_{n-2} \\ m_{n-1} \end{bmatrix} = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_{n-2} \\ \gamma_{n-1} \end{bmatrix}. \quad (9.6)$$

We can use (9.5) and (9.6) to determine the coefficients m of a spline with natural boundary conditions:

```
function spline_natural(x_i,y_i)
    h = diff(x_i)
    γ = 6*diff(diff(y_i)./h)
    α = h[2:end-1]
    β = 2(h[1:end-1]+h[2:end])
    [0;SymTridiagonal(β,α)\γ;0]
end
```

The following function computes the interpolating cubic spline using n points through the nodes given by the arrays x and y .

```
function evaluate_spline(x_i,y_i,m,n)
    h = diff(x_i)
    B = y_i[1:end-1] .- m[1:end-1].*h.^2/6
    A = diff(y_i)./h - h./6 .*diff(m)
    x = range(minimum(x_i),maximum(x_i),length=n+1)
    i = sum(x_i'.≤x,dims=2)
    i[end] = length(x_i)-1
    y = @. (m[i]*(x_i[i+1]-x)^3 + m[i+1]*(x-x_i[i])^3)/(6h[i]) +
        A[i]*(x-x_i[i]) + B[i]
    return(x,y)
end
```

► Splines in parametric form

We can adapt the work from the previous section to compute a spline through points in a plane. Consider a plane curve in parametric form $f(t) = (x(t), y(t))$ with $t \in [0, T]$ for some T . Take a set of points (x_i, y_i) for $i = 0, 1, \dots, n$ and introduce the partition $0 = t_0 < t_1 < \dots < t_n = T$. A simple way to parameterize the spline is by using the lengths of each straight-line segment

$$\ell_i = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

for $i = 1, 2, \dots, n$ and setting $t_0 = 0$ and $t_i = \sum_{j=1}^i \ell_j$. Now we simply need to compute splines for $x(t)$ and $y(t)$. This function is called the *cumulative length spline* and is a good approximation as long as the curvature of $f(t)$ is small.

• The Dierckx.jl function Spline1D returns a cubic spline.

The Julia Dierckx.jl package is a wrapper of the dierckx Fortran library developed by Paul Dierckx, providing an easy interface to building splines. Let's create a function and select some knots.

```
g = x-> @. max(1-abs(3-x),0)
x_i = 0:5; y_i = g(x_i)
x = LinRange(0,5,101);
```

Then we can plot the cubic spline.

```
using Dierckx, Plots
spline = Spline1D(x_i,y_i)
plot(x,spline(x)); plot!(x,g(x)); scatter!(x_i,y_i)
```

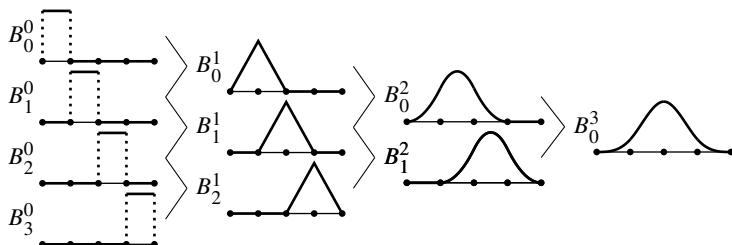
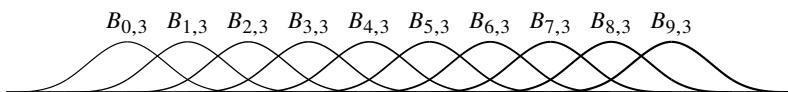


Figure 9.3: B-spline bases generated recursively by de Boor's algorithm.

► B-splines

An alternative approach to constructing interpolating splines is by building them using a linear combination of basis elements called basis splines or B-splines. Let $\{B_{0,p}(x), B_{1,p}(x), \dots, B_{n,p}(x)\}$ be a set of p th-order B-splines, where we use the second subscript to denote the order:



We can interpolate a spline curve $s(x) = \sum_{i=0}^n c_i B_{i,p}(x)$ through a set of knots $(x_0, f(x_0)), (x_1, f(x_1)), \dots, (x_n, f(x_n))$ by determining the coefficients c_0, c_1, \dots, c_n that satisfies the system of equations $f(x_j) = \sum_{i=0}^n c_i B_{i,p}(x_j)$ for $j = 1, 2, \dots, n$. B-splines have the following properties: each is nonnegative with local support, each is a piecewise polynomial of degree less than or equal to p over $p+1$ subintervals, and together they form a partition of unity. Degree-0 B-splines are piecewise constant, degree-1 B-splines are piecewise linear, degree-2 B-splines are piecewise quadratic, and so forth.

We can generate splines of any degree recursively starting with degree-0 splines using *de Boor's algorithm*. Let $a \leq x_0 < \dots < x_n \leq b$. The degree-0 B-splines are *characteristic functions*

$$B_{i,0}(x) = \begin{cases} 1, & \text{if } x_i \leq x < x_{i+1} \\ 0, & \text{otherwise.} \end{cases}$$

It's easy to see that the set $\{B_{0,0}(x), B_{1,0}(x), \dots, B_{n-1,0}(x)\}$ forms a partition of unity over $[a, b]$, that is, $\sum_{i=0}^{n-1} B_{i,0}(x) = 1$. We can construct higher degree B-splines that also form a partition of unity by recursively defining

$$B_{i,p}(x) = \left(\frac{x - x_i}{x_{i+p} - x_i} \right) B_{i,p-1}(x) + \left(\frac{x_{i+p+1} - x}{x_{i+p+1} - x_{i+1}} \right) B_{i+1,p-1}(x)$$

which we can rewrite as

$$B_{i,p} = V_{i,p} B_{i,p-1} + (1 - V_{i+1,p}) B_{i+1,p-1} \quad \text{with} \quad V_{i,p}(x) = \frac{x - x_i}{x_{i+p} - x_i}.$$

See Figure 9.3 on the facing page. The constant spline $B_{i,0}(x)$ has one subinterval for its nonzero support, the linear spline $B_{i,1}(x)$ has two subintervals for its nonzero support, the quadratic spline $B_{i,2}(x)$ has three subintervals for its nonzero support, and so forth.

Finally, suppose that the knots are equally spaced with separation h and a node at x_0 . A B-spline with equal separation between knots is called a cardinal B-spline. We can generate such a cubic B-spline

$$B_{i,3}(x) = B\left(\frac{x - x_0}{h} - i\right)$$

by translating and stretching the B-spline

$$B(x) = \begin{cases} \frac{2}{3} - \frac{1}{2}(2 - |x|)x^2, & |x| \in [0, 1) \\ \frac{1}{6}(2 - |x|)^3, & |x| \in [1, 2) \\ 0, & \text{otherwise.} \end{cases}$$

At the nodes $x = \{-1, 0, 1\}$, the B-spline takes the values $B(x) = \{\frac{1}{3}, \frac{2}{3}, \frac{1}{3}\}$. The values are zero at every other node. The coefficients c_j of the resulting tridiagonal system $f(x_j) = \sum_{i=0}^n c_i B_{i,3}(x_j)$ can be determined quite efficiently by inverting the corresponding matrix equation.

 The Interpolations.jl package, which is still under development, can be used to evaluate up through third-order B-splines, but the nodes must be equally spaced.

Let's fit a spline through the knots that we generated earlier:

```
using Interpolations
method = BSpline(Cubic(Natural(OnGrid())))
spline = scale(interpolate(yi, method), xi)
plot(x, spline(x)); plot!(x, g(x)); scatter!(xi, yi)
```

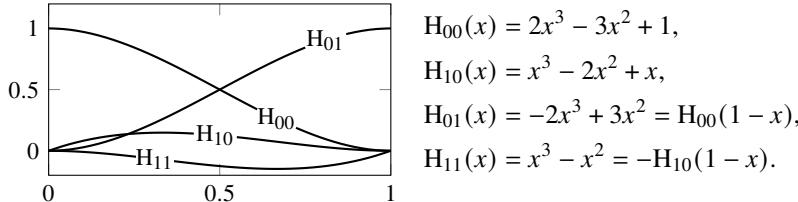
► Cubic Hermite spline

Another type of cubic spline is a piecewise cubic Hermite interpolating polynomial or PCHIP. Cubic Hermite polynomials are built using a linear combination of four basis functions, each one with either unit value or unit derivative on either end of an interval. Hence, we can uniquely determine a cubic Hermite spline on $[x_i, x_{i+1}]$ by prescribing the values $f(x_i)$, $f(x_{i+1})$, $f'(x_i)$ and $f'(x_{i+1})$. Consider

the unit interval, and let $p_0 = f(0)$, $p_1 = f(1)$, $m_0 = f'(0)$ and $m_1 = f'(1)$ at the nodes. In this case

$$p(x) = p_0 H_{00}(x) + m_0 H_{10}(x) + p_1 H_{01}(x) + m_1 H_{11}(x)$$

where the basis elements are given by



We can modify these bases to fit an arbitrary interval $[x_i, x_{i+1}]$ by taking

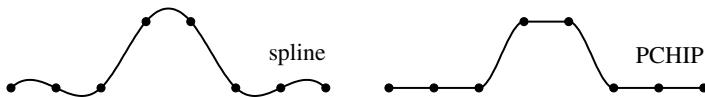
$$p(x) = p_i H_{00}(t) + m_i h_i H_{10}(t) + p_{i+1} H_{01}(t) + m_{i+1} h_i H_{11}(t)$$

where $h_i = x_{i+1} - x_i$, $t = (x - x_i)/h_i$, and the coefficients $p_i = f(x_i)$ and $m_i = f'(x_i)$. The coefficients m_i can be approximated by finite difference approximation

$$\frac{p_{i+1} - p_{i-1}}{x_{i+1} - x_{i-1}} \quad \text{or} \quad \frac{1}{2} \left(\frac{p_{i+1} - p_i}{x_{i+1} - x_i} + \frac{p_i - p_{i-1}}{x_i - x_{i-1}} \right),$$

using a one-sided derivative at the boundaries.

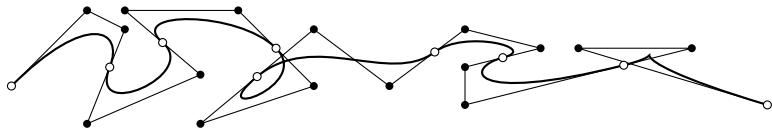
One practical difference between a PCHIP and a cubic spline interpolant is that the PCHIP has fewer wiggles and doesn't overshoot the knots like the spline is apt to do. On the other hand, the spline is smoother than the PCHIP. A cubic spline is C^2 -continuous whereas a PCHIP is only C^1 -continuous. A cubic spline is designed by matching the first and second derivatives of the polynomial segments at knots without explicit regard to the values of those derivatives. A PCHIP is designed by prescribing the first derivatives at knots, which are the same for each polynomial segment sharing a knot, while their second derivatives may be different. Take a look at both methods applied to the knots below:



9.4 Bézier curves

Another way to create splines is by using composite Bézier curves. While splines are typically constructed by fitting polynomials through nodes, Bézier curves are

constructed by using a set of control points to form a convex hull. Other than at the endpoints, a Bézier curve doesn't go through control points. A Bézier spline is built by matching position and slopes at the terminal control points (o) of several of Bézier curves:



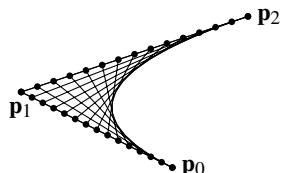
Bézier curves are all around us—just pick up a newspaper and you are bound to see Bézier curves. That's because composite Bézier curves are the mathematical encodings for glyphs used in virtually every modern font from quadratic ones in TrueType fonts to the cubic ones in the Times font used in each letter of text you are reading right now. Bézier curves are used to draw curves in design software like Inkscape and Adobe Illustrator, and they are used to describe paths in an SVG, the standard web vector image format.

A linear Bézier curve is a straight line segment through two control points $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$:

$$\mathbf{q}_{10}(t) = (1 - t)\mathbf{p}_0 + t\mathbf{p}_1$$

where $t \in [0, 1]$. Note that $\mathbf{q}_{10}(0) = \mathbf{p}_0$ and $\mathbf{q}_{10}(1) = \mathbf{p}_1$. By adding an additional control point \mathbf{p}_2 , we can build a quadratic Bézier curve. First, define the linear Bézier curve

$$\mathbf{q}_{11}(t) = (1 - t)\mathbf{p}_1 + t\mathbf{p}_2.$$



Now, combine $\mathbf{q}_{10}(t)$ and $\mathbf{q}_{11}(t)$:

$$\mathbf{q}_{20}(t) = (1 - t)\mathbf{q}_{10}(t) + t\mathbf{q}_{11}(t).$$

String art uses threads strung between nails hammered into a board to create geometric representations, such as of a ship's sail. You can think of the Bézier curve as the convex hull formed connecting threads from nails at $\mathbf{q}_{10}(t)$ to corresponding nails at $\mathbf{q}_{11}(t)$. The threads or line segments are tangent to the Bézier curve $\mathbf{q}_{20}(t)$. If we expand the expression for $\mathbf{q}_{20}(t)$ we have

$$\mathbf{q}_{20}(t) = (1 - t)^2\mathbf{p}_0 + 2t(1 - t)\mathbf{p}_1 + t^2\mathbf{p}_2$$

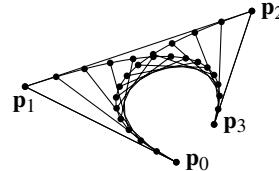
with a derivative

$$\begin{aligned}\mathbf{q}'_{20}(t) &= -2(1 - t)\mathbf{p}_0 + 2(1 - 2t)\mathbf{p}_1 + 2t\mathbf{p}_2 \\ &= -2((1 - t)\mathbf{p}_0 + t\mathbf{p}_1) + 2((1 - t)\mathbf{p}_1 + t\mathbf{p}_2) \\ &= 2(\mathbf{q}_{11}(t) - \mathbf{q}_{10}(t)).\end{aligned}$$

Note that $\mathbf{q}_{20}(0) = \mathbf{p}_0$ and $\mathbf{q}_{20}(1) = \mathbf{p}_2$, which says that the quadratic Bézier curve goes through \mathbf{p}_0 and \mathbf{p}_2 . And, $\mathbf{q}'_{20}(0) = 2(\mathbf{p}_1 - \mathbf{p}_0)$ and $\mathbf{q}'_{20}(1) = 2(\mathbf{p}_2 - \mathbf{p}_1)$, which says that the Bézier curve is tangent to the line segment $\overline{\mathbf{p}_0\mathbf{p}_1}$ and $\overline{\mathbf{p}_1\mathbf{p}_2}$ at \mathbf{p}_0 and \mathbf{p}_2 , respectively.

We can continue in the fashion by adding another control point \mathbf{p}_3 to build a cubic Bézier curve. Take

$$\begin{aligned}\mathbf{q}_{12}(t) &= (1-t)\mathbf{p}_2 + t\mathbf{p}_3 \\ \mathbf{q}_{21}(t) &= (1-t)\mathbf{q}_{11}(t) + t\mathbf{q}_{12}(t) \\ \mathbf{q}_{30}(t) &= (1-t)\mathbf{q}_{20}(t) + t\mathbf{q}_{21}(t).\end{aligned}$$



This time, the cubic Bézier curve $\mathbf{q}_{30}(t)$ is created from the convex hull formed by line segments connecting corresponding points on $\mathbf{q}_{20}(t)$ and $\mathbf{q}_{21}(t)$. These, in turn, are created from the segments connecting $\mathbf{q}_{10}(t)$ and $\mathbf{q}_{11}(t)$, and $\mathbf{q}_{11}(t)$ and $\mathbf{q}_{12}(t)$, respectively. See the QR link at the bottom of this page.

This recursive method of constructing the Bézier curve starting with linear Bézier curves then quadratic Bézier curves then cubic (and higher order) Bézier curves is called *de Casteljau's algorithm*. By expanding \mathbf{q}_{30} , we have the cubic Bernstein polynomial

$$\mathbf{q}_{30}(t) = (1-t)^3\mathbf{p}_0 + 3t(1-t)^2\mathbf{p}_1 + 3t^2(1-t)\mathbf{p}_2 + t^3\mathbf{p}_3.$$

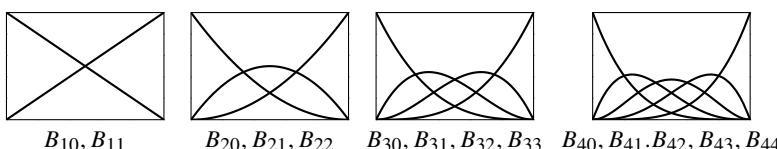
In general, the degree- n Bernstein polynomial is given by

$$\mathbf{q}_{n0}(t) = \sum_{k=0}^n \binom{n}{k} (1-t)^{n-k} t^k \mathbf{p}_k = \sum_{k=0}^n B_{nk}(t) \mathbf{p}_k,$$

or simply $\mathbf{q}_{n0}(t) = \mathbf{B}_n(t)\mathbf{p}$, where $\mathbf{B}_n(t)$ is an $m \times (n+1)$ Bernstein matrix constructed by taking m points along $[0, 1]$ and \mathbf{p} is an $(n+1) \times 2$ vector of control points. We can build a Bernstein matrix using the following function which takes a column vector such as $t = \text{LinRange}(0, 1, 20)$:

```
berNSTein(n, t) = @. binomial(n, 0:n)' * t^(0:n)' * (1-t)^(n:-1:0)'
```

The bases $\{B_{n0}(t), B_{n1}(t), \dots, B_{nn}(t)\}$ form a partition of unity. This says that the points along a Bézier curve are the weighted averages of the control points. From this, we know that the curve is contained in the convex hull formed by control points. The Bernstein polynomials are plotted below as a function of the parameter $t \in [0, 1]$. Notice how the weight changes on each control point as t moves from zero on the left of each plot to one on the right.



► Weierstrass approximation theorem

Bézier curves were popularized by Renault engineer Pierre Bézier in the 1960s for computer aided geometric design. Paul de Casteljau's contributions occurred around the same time at the competing automobile manufacturer Citroën. Bernstein polynomials, however, appeared much earlier. In 1912, Serge Bernstein introduced his eponymous polynomials in a probabilistic proof to the Weierstrass theorem. We'll conclude the chapter with Bernstein's proof. The proof is elegant and it provides an nice segue to the material in the next chapter.

It's no coincidence that the terms of the Bernstein polynomial bear striking similarity to the probability mass function of a binomial distribution. Bernstein's proof invokes Bernoulli's theorem, also known as the weak law of large numbers, which states that the frequency of success of a sequence of Bernoulli trials approaches the probability of success as the number of trials approaches infinity. A Bernoulli trial is a random experiment with one of two possible outcomes—a success or a failure. A coin flip is either heads or tails. A soccer shot is either a goal or a miss. A student's random guess on a multiple choice question is either right or wrong.

Theorem 30 (Weierstrass approximation theorem). *If $f(x)$ is a continuous function in the interval $[0, 1]$, then for any $\varepsilon > 0$, there is a polynomial $p_n(x)$ with a sufficiently large degree n such that $|f(x) - p_n(x)| < \varepsilon$ for every $x \in [0, 1]$.*

Proof. Consider an event whose probability equals x . Suppose that we perform n trials and agree to pay out $f(k/n)$ if that event occurs k times. Under these conditions, the expected payout is

$$E_n = \sum_{k=0}^n f\left(\frac{k}{n}\right) \binom{n}{k} x^k (1-x)^{n-k}$$

using a binomial distribution. Because $f(x)$ is continuous, we can choose a δ such that $|f(x) - f(\xi)| < \frac{1}{2}\varepsilon$ for any $\xi \in (x - \delta, x + \delta)$. Let $\widehat{f(x)}$ denote the maximum and $\underline{f(x)}$ denote the minimum of $f(x)$ over the interval $(x - \delta, x + \delta)$. Then $\widehat{f(x)} - f(x) < \frac{1}{2}\varepsilon$ and $f(x) - \underline{f(x)} < \frac{1}{2}\varepsilon$.

Let q be the probability that $|x - k/n| > \delta$, and let L be the maximum of $|f(x)|$ in $[0, 1]$. Then

$$\underline{f(x)}(1-q) - Lq < E_n < \widehat{f(x)}(1-q) + Lq,$$

which we can rewrite as

$$f(x) + (\underline{f(x)} - f(x)) - q(L + \underline{f(x)}) < E_n < f(x) + (\widehat{f(x)} - f(x)) - q(L + \widehat{f(x)}).$$

By Bernoulli's theorem, we can choose n sufficiently large such that $q < \varepsilon/4L$. It follows that

$$f(x) - \frac{\varepsilon}{2} - \frac{2L}{4L}\varepsilon < E_n < f(x) + \frac{\varepsilon}{2} + \frac{2L}{4L}\varepsilon,$$

and hence $|f(x) - E_n| < \varepsilon$. Of course, the expected payout E_n is simply a degree- n polynomial $p_n(x)$. \square

The Bernstein approximation of a function $f(x)$ is $\sum_{k=0}^n f(k/n)B_{nk}(x)$. To be clear, a Bernstein approximant does not interpolate given values, even if $f(x)$ is itself a polynomial. Let's plot the approximation of $f(x) = \sqrt{x}$:

```
n = 20; f = t -> sqrt(t)
y = bernstein(n,t)*f.(LinRange(0,1,n+1))
plot(t,y); plot!(t,f.(t))
```

This code does produce a polynomial approximation to $f(x)$, but it's not a very good one. While Bernstein polynomials are useful in de Casteljau's algorithm to generate Bézier curves, they are not practical in approximating arbitrary functions. Namely, they converge extremely slowly: $|f(x) - p_n(x)| = O(n^{-1})$. In the next chapter, we will examine methods that converge quickly. To dig deeper into Bernstein polynomials, check out Rida Farouki's centennial retrospective article.

9.5 Exercises

- ✍ 9.1. Prove theorem 24 by using the canonical basis $\{1, x, \dots, x^n\}$ to construct a Vandermonde matrix. Show that if $\{x_0, x_1, \dots, x_n\}$ are distinct, then the Vandermonde matrix \mathbf{V} is nonsingular. Hence, the Vandermonde system has a unique solution. Hint: Show that $\det(\mathbf{V}) = \prod_{j=0}^n \prod_{i \neq j} (x_i - x_j)$ by induction.
- ✍ 9.2. Write a program that constructs a closed parametric cubic spline $(x(t), y(t))$ using periodic boundary conditions. Use it to plot a smooth curve through several randomly selected knots.
- ✍ 9.3. A radial basis function is any function ϕ whose value depends only on its distance from the origin. We can use them as interpolating functions:

$$y(x) = \sum_{i=0}^n c_i \phi(x - x_i).$$

Compare interpolation using a polynomial basis with the Gaussian and cubic radial basis functions $\phi(x) = \exp(-x^2/2\sigma^2)$ and $\phi(x) = |x|^3$ for the Heaviside function using 20 equally spaced nodes in $[-1, 1]$. The Heaviside function $H(x) = 0$ when $x < 0$ and $H(x) = 1$ when $x \geq 0$.

- ✍ 9.4. Collocation is a method of solving a problem, such as a differential equation, by considering candidate solutions from a finite-dimensional space, such as the space of piecewise cubic polynomials, along with a set of points called

collocation points. The new, finite-dimensional problem is then solved exactly at the collocation points to get an approximate solution to the original problem.

Suppose that we have the differential equation $L u(x) = f(x)$ with boundary conditions $u(a) = u_a$ and $u(b) = u_b$ for some linear differential operator L . We approximate the solution $u(x)$ by $\sum_{j=0}^n c_j v_j(x)$ for some basis v_j . Take the collocation points $x_i \in [a, b]$ with $i = 0, 1, \dots, n$. Then solve the linear system of equations

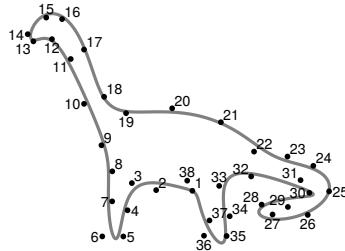
$$\sum_{j=0}^n c_j v_j(x_0) = u_a, \quad \sum_{j=0}^n c_j L v_j(x_i) = f(x_i), \quad \sum_{j=0}^n c_j v_j(x_n) = u_b.$$

Use collocation to solve Bessel's equation $xu'' + u' + xu = 0$ with $u(0) = 1$ and $u(b) = 0$ where b is the fourth zero of the Bessel function (approximately 11.7915344390142) using cubic B-splines with 10 or 20 equally spaced knots over the unit interval $[0, b]$. Comment on the order of convergence of the method as more collocation points are added.

- 9.5. A composite Bézier curve is sometimes used to approximate a circle. One way to do this is by piecing together four cubic Bézier curves that approximate quarter circles, ensuring that their tangents match at the endpoints. For a unit quarter circle with endpoints at $(0, 1)$ and $(1, 0)$, this means that we need to choose control points at $(c, 1)$ and $(1, c)$. What value c should we take to ensure that the radial deviation from the unit circle is minimized? How close of an approximation does this value give us?

Chapter 10

Approximating Functions



In the last chapter, we examined interpolation as a method of finding polynomials that coincide with a function at some given nodes or knots. This allowed us to model data and functions using smooth curves. In this chapter, we consider *best approximation*, which is frequently used in data compression, optimization, and machine learning. For a function f in some vector space F , we will find the function u in a subspace U that is closest to f with respect to some norm:

$$u = \arg \inf_{g \in U} \|f - g\|.$$

When the norm is the L^1 -norm, i.e., $\int_a^b |f(x) - g(x)| dx$, the problem is one of minimizing the average absolute deviation, the one that minimizes the area between curves. When the norm is the L^2 -norm, i.e., $\int_a^b (f(x) - g(x))^2 dx$, the problem is called least squares approximation or *Hilbert approximation*. When the norm is the L^∞ -norm, i.e., $\sup_{x \in [a,b]} |f(x) - g(x)|$, the problem is called uniform approximation, minimax, or *Chebyshev approximation*. The least squares approximation is popular largely because it is robust and typically the easiest to implement. So, we will concentrate on the least squares approximation.

10.1 Least squares approximation

An *inner product* is a mapping (\cdot, \cdot) of any two vectors of a vector space to the field over which that vector space is defined (typically real or complex numbers) satisfying three properties: linearity, positive-definiteness, and symmetry.

1. *Linearity* says that for any f, g , and h in the vector space and any scalar α of the field, $(f, \alpha g) = \alpha \cdot (f, g)$ and $(f, g + h) = (f, g) + (f, h)$.
2. *Positive-definiteness* says that $(f, f) > 0$ if f is non-zero (i.e., positive) and that $(f, f) = 0$ if f is identically zero (i.e., definite).
3. Finally, *symmetry* or *Hermiticity* says that $(f, g) = \overline{(g, f)}$ for a real vector space and for $(f, g) = \overline{(g, f)}$ for a complex one.

A vector space that has been assigned an inner product is called an *inner product space*. An inner product induces a natural norm $\|f\| = \sqrt{(f, f)}$, so every inner product space is also a normed vector space. Sometimes, as in the case of Fourier polynomials or functions with removable discontinuities, we loosen the notion of an inner product to a *pseudo-inner product* by removing the requirement of definiteness. In this case, the norm is replaced by a *semi-norm*.

By associating the dot product with the n -dimensional space \mathbb{R}^n , we get the Euclidean vector space. The Euclidean inner product $(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n x_i y_i$ has the geometric interpretation as a measure of the magnitude and closeness of two vectors: $(\mathbf{x}, \mathbf{y}) = \|\mathbf{x}\| \|\mathbf{y}\| \cos \theta$. In particular, the inner product of any two unit vectors is simply the angle between those two vectors. In this chapter, we will be interested in the space of continuous functions over an interval $[a, b]$ and its associated inner product $(f, g) = \int_a^b f(x)g(x)w(x) dx$ where $w(x)$ is a specified positive weight function. We can easily verify that the three properties of an inner product space hold. Like the Euclidean vector space, the weighted inner product for the space of functions can also be interpreted as a measure of the magnitude and closeness of the functions.

We say that f is *orthogonal* to g if $(f, g) = 0$. Sometimes, we use the term *w-orthogonal* to explicitly indicate that a weighting function is included in the definition of the inner product. We'll write this as $f \perp g$. If $f \perp g$ for all $g \in G$, we say that f is orthogonal to G and write it as $f \perp G$. For example, consider the space of square-integrable functions on the interval $[-1, 1]$ with the inner product $(f, g) = \int_{-1}^1 f(x)g(x) dx$. In this inner product space even functions, like as e^{-x^2} , are orthogonal to odd functions, like $\sin x$. Even functions form a subspace of square-integrable functions; odd functions form another subspace. In fact, the subspace of even functions is the *orthogonal complement* to the subspace of odd functions. That is to say, every function is the sum of an even function and an odd function.

Theorem 31. Let U be a subspace of the inner product space F . Take $f \in F$ and $u \in U$. Then u is the best approximation to f in U if and only if $f - u \perp U$.

Proof. Let's start by proving that if $f - u \perp U$ then u is the best approximation to f . If $f - u \perp U$, then for any $g \in U$

$$\begin{aligned}\|f - g\|^2 &= \|(f - u) + (u - g)\|^2 \\ &= \|f - u\|^2 + 2(f - u, f - g) + \|u - g\|^2 \xrightarrow{0} \geq \|f - u\|^2.\end{aligned}$$

To prove converse suppose that u is a best approximation to f . Let $g \in U$ and take $\lambda > 0$. Then $\|f - u + \lambda g\|^2 \geq \|f - u\|^2$. So,

$$\begin{aligned}0 &\leq \|f - u + \lambda g\|^2 - \|f - u\|^2 \\ &= \|f - u\|^2 + 2\lambda(f - u, g) + \lambda^2\|g\|^2 - \|f - u\|^2 \\ &= \lambda(2(f - u, g) + \lambda\|g\|^2).\end{aligned}$$

Because $\lambda > 0$, we have that $2(f - u, g) + \lambda\|g\|^2 \geq 0$. Letting λ shrink to zero, we get $(f - u, g) \geq 0$. Similarly, by replacing g with $-g$, we get $(f - u, -g) \geq 0$ from which it follows that $(f - u, g) \leq 0$. Therefore, $(f - u, g) = 0$. Because g is an arbitrary function of U , it follows that $f - u \perp U$. \square

Let $\{u_1, u_2, \dots, u_n\}$ be a basis for a subspace $U \subset F$. If we want to find the best approximation to $f \in F$ in U , then we must find the $u \in U$ such that $u - f \perp U$. That is, we must find the u such that $(u - f, u_i) = 0$ for every $i = 1, 2, \dots, n$. Take $u = \sum_{j=1}^n c_j u_j$. Then

$$\left(\sum_{j=1}^n c_j u_j - f, u_i \right) = 0,$$

from which it follows that

$$\sum_{j=1}^n c_j (u_j, u_i) = (f, u_i).$$

This system of equations is known as the *normal equation*. In matrix form, the normal equation is

$$\begin{bmatrix} (u_1, u_1) & (u_2, u_1) & \dots & (u_n, u_1) \\ (u_1, u_2) & (u_2, u_2) & \dots & (u_n, u_2) \\ \vdots & \vdots & \ddots & \vdots \\ (u_1, u_n) & (u_2, u_n) & \dots & (u_n, u_n) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (f, u_1) \\ (f, u_2) \\ \vdots \\ (f, u_n) \end{bmatrix}.$$

The matrix on the left is called the *Gram matrix*.

Example. Suppose that we want to find the least squares approximation to $f(x)$ over the interval $[0, 1]$ using degree- n polynomials. If we take the canonical basis $\{\varphi_i\} = \{1, x, x^2, \dots, x^n\}$, then

$$p_n(x) = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n = \sum_{i=0}^n c_i\varphi_i(x).$$

To determine $\{c_i\}$, we solve $(p_n, \varphi_i) = (f, \varphi_i)$ for $i = 1, 2, \dots, n$:

$$(\varphi_j, \varphi_i) = (x^j, x^i) = \int_0^1 x^{i+j} dx = \frac{1}{i+j+1}.$$

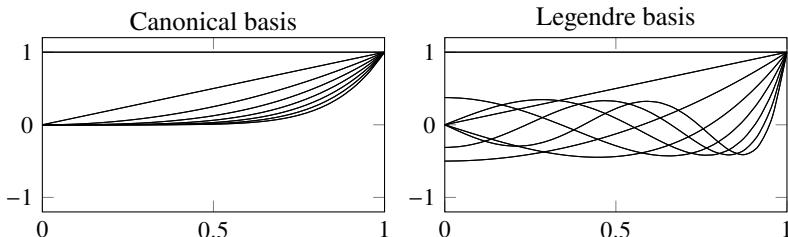
We now must solve the normal equation

$$\begin{bmatrix} 1 & 1/2 & 1/3 & \dots \\ 1/2 & 1/3 & 1/4 & \dots \\ 1/3 & 1/4 & 1/5 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (f, 1) \\ (f, x) \\ \vdots \\ (f, x^n) \end{bmatrix}$$

This Gram matrix, called the *Hilbert matrix*, is quite ill-conditioned. Even for relatively small systems, the computations can be unstable. For example, the condition number for 7×7 Hilbert matrix is roughly 10^8 , meaning that we can expect any error in our calculations to possibly be magnified by as much as 10^8 . And the condition number of a 12×12 Hilbert matrix exceeds floating-point precision. Typing

```
using LinearAlgebra; cond([1//(i+j-1) for i=1:12, j=1:12])*eps()
```

into Julia returns around 3.89. We can get an intuitive idea of why the canonical basis results in an ill-conditioned problem by just by looking at it. The first eight elements of the canonical basis and of the Legendre basis are shown below:



Differences between the elements of the canonical basis get smaller and smaller as the degree of the monomial increases. It is easy to distinguish the plots of 1 from x and x from x^2 , but the plots of x^7 and x^8 become almost indistinguishable. On the other hand, the plots of the Legendre polynomial basis elements are all largely different. So, heuristically, it seems that orthogonal polynomials like Legendre polynomials are simply better. \blacktriangleleft

10.2 Orthonormal systems

We say that a set of vectors $\{u_1, u_2, \dots, u_n\}$ is *orthonormal* in an inner product space if $(u_i, u_j) = \delta_{ij}$ for all i and j . Let's reexamine the normal equation when we use an orthonormal basis. Because $(u_i, u_j) = \delta_{ij}$, we now have

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (f, u_1) \\ (f, u_2) \\ \vdots \\ (f, u_n) \end{bmatrix}.$$

Wow! This means that if $\{u_1, u_2, \dots, u_n\}$ is an orthonormal system in an inner product space F , then the best approximation p_n to f is simply

$$p_n = \sum_{i=1}^n (f, u_i) u_i. \quad (10.1)$$

We call this best approximation the orthogonal projection of f . The inner product (f, u_i) is simply the component of f in the u_i direction. We'll denote the orthogonal projection operator by P_n :

$$P_n f = \sum_{i=1}^n (f, u_i) u_i.$$

We can summarize everything into the following theorem, which will be a main tool of the rest of the chapter.

Theorem 32. *Let $\{u_1, u_2, \dots, u_n\}$ be an orthonormal system in an inner product space F . Then the best approximation to f is the element $P_n f$, where the orthogonal projection operator $P_n = \sum_{i=1}^n (\cdot, u_i) u_i$.*

So far, we've only used a finite basis. Let's turn to infinite-dimensional vector spaces. To do this, we'll need a few extra terms. A *Cauchy sequence* is any sequence whose elements become arbitrarily close as the sequence progresses. That is to say, if we have a sequence of projections $P_0 f, P_1 f, P_2 f, \dots$, then the sequence is a Cauchy sequence if given some $\varepsilon > 0$, there is some integer N such that $\|P_n f - P_m f\| < \varepsilon$ for all $n, m > N$. A normed vector space is said to be *complete* if every Cauchy sequence of elements in that vector space converges to an element in that vector space. Such a vector space is called a *Banach space*. Similarly, a complete, inner product space is called a *Hilbert space*. An orthonormal basis $\{u_1, u_2, \dots\}$ of a Hilbert space F is called a complete, orthonormal system of F or a *Hilbert basis*.

Theorem 33 (Parseval's theorem). *If $\{u_1, u_2, \dots\}$ is a complete, orthonormal system in F , then $\|f\|^2 = \sum_{i=1}^{\infty} |(f, u_i)|^2$ for every $f \in F$.*

Proof. Because $\{u_1, u_2, \dots\}$ are orthonormal, we can define the projection operator $P_n = \sum_{i=1}^n (\cdot, u_i) u_i$. In the induced norm we simply have

$$\|P_n f\|^2 = \left\| \sum_{i=1}^n (f, u_i) u_i \right\|^2 = \left(\sum_{i=1}^n (f, u_i) u_i, \sum_{j=1}^n (f, u_j) u_j \right).$$

We expand the inner product to get

$$\|P_n f\|^2 = \sum_{i=1}^n \sum_{j=1}^n (f, u_i) (f, u_j) (u_i, u_j) = \sum_{i=1}^n |(f, u_i)|^2,$$

because $(u_i, u_j) = \delta_{ij}$. Now, because the system is complete,

$$\|f\|^2 = \lim_{n \rightarrow \infty} \|P_n f\|^2 = \sum_{i=1}^{\infty} |(f, u_i)|^2. \quad \square$$

We can generate a Hilbert basis (an orthonormal basis for an inner product space) by using the Gram–Schmidt method. The Gram–Schmidt method starts with an arbitrary basis and sequentially subtracts out the non-orthogonal components. Let $\{v_1, v_2, v_3, \dots\}$ be a basis for F with a prescribed inner product space.

1. Start by taking $u_1 = v_1$ and normalize $\hat{u}_1 = u_1 / \|u_1\|$.
2. Next, define $u_2 = v_2 - P_1 v_2$ where the projection operator $P_1 = (\cdot, \hat{u}_1) \hat{u}_1$, and normalize $\hat{u}_2 = u_2 / \|u_2\|$.
3. Now, define $u_3 = v_3 - P_2 v_3$ where the projection operator $P_2 = P_1 + (\cdot, \hat{u}_2) \hat{u}_2$, and normalize $\hat{u}_3 = u_3 / \|u_3\|$.
- n. We continue in this manner by defining $u_n = v_n - P_{n-1} v_n$ where the projection operator $P_{n-1} = \sum_{i=1}^{n-1} (\cdot, \hat{u}_i) \hat{u}_i$ and normalizing $\hat{u}_n = u_n / \|u_n\|$.

Then $\{\hat{u}_1, \hat{u}_2, \hat{u}_3, \dots\}$ is an orthonormal basis for F . Applying the Gram–Schmidt process directly can be a lot of work. Luckily, for generating orthogonal polynomials there's a shortcut. We can use a three-term recurrence relation known as Favard's theorem.¹

Theorem 34 (Favard's theorem). *For each weighted inner product, there are uniquely determined orthogonal polynomials with leading coefficient one satisfying the three-term recurrence relation*

$$\begin{cases} p_{n+1}(x) = (x + a_n) p_n(x) + b_n p_{n-1}(x) \\ p_0(x) = 1, \quad p_1(x) = x + a_1 \end{cases}$$

¹Jean Favard proved the theorem in 1935, although it was previously demonstrated by Aurel Wintner in 1926 and Marshall Stone in 1932.

where

$$a_n = -\frac{(xp_n, p_n)}{(p_n, p_n)} \quad \text{and} \quad b_n = -\frac{(p_n, p_n)}{(p_{n-1}, p_{n-1})}.$$

Proof. We'll use induction. First, the only polynomial of degree zero with leading coefficient one is $p_0(x) \equiv 1$. Suppose that the claim holds for the orthogonal polynomials p_0, p_1, \dots, p_{n-1} . Let p_{n+1} be an arbitrary polynomial of degree n and leading coefficient one. Then $p_{n+1} - xp_n$ is a polynomial with degree less than or equal to n . Because p_0, p_1, \dots, p_n form an orthogonal basis of \mathbb{P}_n

$$p_{n+1} - xp_n = \sum_{j=0}^n c_j p_j \quad \text{with} \quad c_j = \frac{(p_{n+1} - xp_n, p_j)}{(p_j, p_j)}.$$

Furthermore, if p_{n+1} is orthogonal to p_0, p_1, \dots, p_n :

$$c_j = \frac{(p_{n+1} - xp_n, p_j)}{(p_j, p_j)} = \frac{(p_{n+1}, p_j)}{(p_j, p_j)} - \frac{(xp_n, p_j)}{(p_j, p_j)} = 0 - \frac{(xp_n, p_j)}{(p_j, p_j)}$$

Let's expand numerator in the right-most term:

$$(xp_n, p_j) = (p_n, xp_j) = \left(p_n, p_{j+1} - \sum_{i=0}^{j-1} c_i p_i \right) = (p_n, p_{j+1}) - \sum_{i=0}^{j-1} c_i (p_n, p_i) \xrightarrow{0}$$

which equals zero for $j = 0, 1, \dots, n-2$ and equals (p_n, p_n) for $j = n-1$. So, $c_0 = \dots = c_{n-2} = 0$ and

$$a_n = c_n = -\frac{(xp_n, p_n)}{(p_n, p_n)}, \quad b_n = c_{n-1} = -\frac{(p_n, p_n)}{(p_{n-1}, p_{n-1})}.$$
□

Example. Legendre polynomials $P_n(x)$ are orthogonal polynomials defined by inner product $\int_{-1}^1 P_n(x)P_m(x)dx$. It's left as an exercise to show that the coefficients of the three-term recurrence relation are $a_n = 0$ and $b_n = -n^2/(4n^2 - 1)$ and the normalization $\|P_n\|^2 = 2$. We can write a recursive function to compute the Legendre polynomial

```
function legendre(x,n)
    n==0 && return(one.(x))
    n==1 && return(x)
    x.*legendre(x,n-1) .- (n-1)^2/(4(n-1)^2-1)*legendre(x,n-2)
end
```

◀

10.3 Fourier polynomials

So far, we've concentrated on subspaces constructed using polynomial basis elements. When approximating periodic functions, trigonometric polynomials are better suited as a basis. We can define the Fourier polynomials on $(0, 2\pi)$ using the exponential basis elements $\phi_k(x) = e^{ikx}$ for all integers k , positive and negative. We can define cosine polynomials and sine polynomials analogously by taking the real/even and imaginary/odd contributions.

Theorem 35. *The exponential basis elements $\phi_k(x) = e^{ikx}$ form an orthonormal system with the pseudo-inner product $(f, g) = \frac{1}{2\pi} \int_0^{2\pi} f(x)\overline{g(x)} dx$.*

Proof. We'll show that $(\phi_k, \phi_m) = \delta_{km}$.

$$\begin{aligned} (\mathbf{e}^{ikx}, \mathbf{e}^{imx}) &= \frac{1}{2\pi} \int_0^{2\pi} \mathbf{e}^{ikx} \mathbf{e}^{-imx} dx = \frac{1}{2\pi} \int_0^{2\pi} \mathbf{e}^{i(k-m)x} dx \\ &= \begin{cases} \frac{1}{2\pi} \int_0^{2\pi} 1 dx = 1, & \text{if } k = m \\ \frac{1}{2\pi} \left[\frac{\mathbf{e}^{i(k-m)x}}{i(k-m)} \right]_0^{2\pi} = \frac{1}{2\pi} \frac{1}{i(k-m)} [1 - 1] = 0, & \text{if } k \neq m \end{cases} \end{aligned}$$

So, $(\mathbf{e}^{ikm}, \mathbf{e}^{imx}) = \delta_{km}$. □

Because the Fourier polynomials are orthogonal, there is a projection operator

$$P_n f = \sum_{k=-n}^n c_k \mathbf{e}^{ikx} \quad \text{with} \quad c_k = (f, \mathbf{e}^{ikx}) = \frac{1}{2\pi} \int_0^{2\pi} f(x) \mathbf{e}^{-ikx} dx. \quad (10.2)$$

The projection $P_n f$ is called the truncated Fourier series, the set of coefficients $\{c_k\}$ are called the Fourier coefficients, and the inner product (f, ϕ_k) is the called the continuous Fourier transform. In practice, the continuous Fourier transform is approximated numerically by taking a finite set of points on the mesh $x_j = 2\pi j/n$ for $j = 0, 1, \dots, n-1$ and using the trapezoidal method to approximate the integral. By using the piecewise-constant approximation of f and g at $\{x_j\}$ in theorem 35 we arrive at a simple corollary:

Theorem 36. *The exponential basis elements $\phi_k(x) = e^{ikx}$ are orthonormal in the discrete inner product space $(f, g)_n = \frac{1}{n} \sum_{j=0}^{n-1} f(x_j) \overline{g(x_j)}$.*

A projection operator in the discrete inner product space, called the *discrete Fourier transform*, can be defined analogously to the continuous Fourier transform (10.2) by taking a discrete inner product at the meshpoints x_j and

$$P_n f(x_j) = \sum_{k=0}^{n-1} c_k \mathbf{e}^{ijk} \quad \text{with} \quad c_k = (f, \mathbf{e}^{ikx}) = \frac{1}{n} \sum_{j=0}^n f(x_j) \mathbf{e}^{-ijk}. \quad (10.3)$$

We call this the *inverse discrete Fourier transform*.

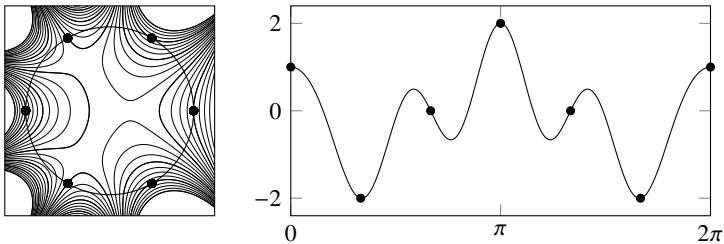


Figure 10.1: A polynomial over the complex plane $\sum_{k=0}^{n-1} c_k z^k$ restricted to the unit circle is a Fourier polynomial $\sum_{k=0}^{n-1} c_k e^{2\pi i k x/n}$.

► Interpolation with Fourier polynomials

Let's briefly revisit the topic of interpolation discussed in the previous chapter. Suppose that we have equally spaced nodes at $x_j = 2\pi j/n$ with values $f(x_j)$ for $j = 0, 1, \dots, n - 1$. Using the exponential basis functions $\phi_k(x) = e^{ikx}$, the interpolating Fourier polynomial satisfies the system of equations

$$f(x_j) = \sum_{j=0}^{n-1} c_k \phi_k(x_j) = \sum_{j=0}^{n-1} c_k e^{i2\pi j k / n} = \sum_{j=0}^{n-1} c_k \omega^{jk} \quad (10.4)$$

for some coefficients c_k . The value $\omega = e^{i2\pi/n}$ is an n th root of unity and we recognize from (10.3) that this transformation is the inverse discrete Fourier transform. So, the coefficients of the Fourier polynomial that interpolates a function f at equally spaced nodes $x_j = 2\pi j/n$ are given by $c_k = (f, \phi_k)$. That is, the solution to the interpolation problem using uniform mesh is the same as the solution to the best approximation problem using the discrete inner product.

We can also establish a connection between Fourier polynomials and algebraic polynomials. Suppose that we want to find the algebraic polynomial interpolant

$$p(z) = c_0 + c_1 z + c_2 z^2 + \cdots + c_{n-1} z^{n-1}$$

passing through the nodes $\{(z_0, y_0), (z_1, y_1), \dots, (z_{n-1}, y_{n-1})\}$ in $\mathbb{C} \times \mathbb{C}$. In the canonical basis $\{1, z, \dots, z^{n-1}\}$, we must solve the Vandermonde system $y_j = \sum_{k=0}^{n-1} c_k z^k$ to get the coefficients of the polynomial c_k . Now, restrict the nodes to equally spaced points along the unit circle, that is, take $z_k = \omega^k = e^{i2\pi k / n}$, which run counterclockwise starting with $z_0 = 1$. In this case, we have (10.4), which means that an algebraic polynomial in the complex plane is a Fourier polynomial on the unit circle. See the figure on this page.

► Approximation error

Let's finally examine the behavior of the Fourier coefficients for the continuous inner product

$$c_k = \left(f, e^{ikx} \right) = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-ikx} dx.$$

where $f(x)$ is periodic. The discrete inner product follows analogously. Note that c_0 is simply the mean value of $f(x)$ over $(0, 2\pi)$. Also, note that if $f(x)$ is differentiable, then by integration by parts

$$\left(f', e^{ikx} \right) = \frac{1}{2\pi} \int_0^{2\pi} f'(x) e^{-ikx} dx = \frac{ik}{2\pi} \int_0^{2\pi} f(x) e^{-ikx} dx = ik c_k$$

because $f(x)$ is periodic. In general, we have $\left(f^{(P)}, e^{ikx} \right) = (ik)^P c_k$ if the function $f(x)$ is sufficiently smooth. A *Sobolev space* is the space of square-integrable functions whose derivatives are also square-integrable functions. Let $H^p(0, 2\pi)$ denote the Sobolev space of periodic functions whose zeroth through p th derivatives are also square-integrable. We can think of Sobolev space as a refinement of the space of C^p -differentiable functions.

Theorem 37. *If $f \in H^p(0, 2\pi)$, then its Fourier coefficients $|c_k| = o(k^{-p-1/2})$ and the L^2 -error of its truncated Fourier series is $o(n^{-p})$.*

Proof. By Parseval's theorem

$$\frac{1}{2\pi} \int_0^{2\pi} |f^{(P)}(x)|^2 dx = \sum_{k=1}^{\infty} |k^p c_k|^2.$$

In order for the series to converge, it must follow that $|k^p c_k|^2 = o(k^{-1})$ as $k \rightarrow \infty$. In other words, $|c_k| = o(k^{-p-1/2})$. We'll use this bound to compute the error in the truncated Fourier series $P_n f$. We have that

$$\|f - P_n f\|^2 = \|P_\infty f - P_n f\|^2 = \sum_{k>|n|} |c_k|^2 = \sum_{k>|n|} o(k^{-2p-1}) = o(n^{-2p}).$$

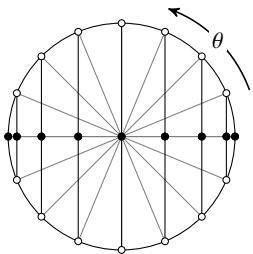
So, the error is $o(n^{-p})$. □

Note that if f is smooth (with infinitely many continuous derivatives), then the error is $o(n^{-p})$ for all integers p . So, the rate of convergence of the Fourier series will be faster than polynomial rate. In other words, it will have an exponential rate of convergence.

10.4 Chebyshev polynomials

Chebyshev polynomials are close cousins to Fourier polynomials. While Fourier polynomials are used to approximate periodic functions, Chebyshev polynomials are used to approximate functions with prescribed endpoints. Fourier polynomials are simply polynomials $\sum_{k=0}^n c_k z^k$ constrained to the unit circle $z = e^{i\theta}$ in the complex plane. If we take the real part of such a polynomial, we have a function that resides between $[-1, 1]$ along the real axis. Take

$$x = \operatorname{Re} z = \frac{1}{2}(z + z^{-1}) = \cos \theta.$$



For a general k th order monomial z^k , we can define the Chebyshev polynomial

$$T_k(x) = \operatorname{Re} z^k = \frac{1}{2}(z^k + z^{-k}) = \cos k\theta.$$

Then a polynomial approximant has the representation

$$p_n(x) = \sum_{k=0}^n a_k T_k(x) = \sum_{k=0}^n a_k \cos k\theta.$$

The derivative of a Chebyshev polynomial is

$$T'_k(x) = -k \sin k\theta \frac{d\theta}{dx} = \frac{k \sin k\theta}{\sin \theta} = \frac{k \sin k\theta}{\sqrt{1-x^2}}. \quad (10.5)$$

We use L'Hopital's rule to define the derivative $T'_k(\pm 1) = (\pm 1)^{k+1} k^2$ at the endpoints. Higher-order derivatives can be defined similarly. For example,

$$T''_k(x) = -\frac{k^2 \cos k\theta}{1-x^2} + \frac{kx \sin k\theta}{(1-x^2)^{3/2}} \quad (10.6)$$

with $T''_k(\pm 1) = (\pm 1)^{k+1} \frac{1}{3}(k^4 - k^2)$. In practice, to compute the Chebyshev derivatives (10.5) or (10.6), we can use discrete cosine and discrete sine transforms. Or, we can extend a function to a periodic domain using its reflection and then take the real components of a discrete Fourier transform. All of this can be done efficiently using a fast Fourier transform.

Another way to differentiate a Chebyshev polynomial is by building a full Chebyshev differentiation matrix with the Lagrange formulation of the polynomials using the Chebyshev extremum points. The Chebyshev extremum points $x_j \in [-1, 1]$ of $T_k(x)$ correspond to equally spaced points on the unit semicircle $\theta_j \in [0, \pi]$ where $\cos k\theta_j = \pm 1$. Computation using a full differentiation matrix is much less efficient than computation using a fast Fourier transform. But, such

an approach has the benefit of being easy to manipulate and explore. We'll develop the Chebyshev differentiation matrix over the remainder of this section. See Trefethen's book *Approximation Theory and Approximation Practice* for an in-depth discussion.²

In the previous chapter, we used Lagrange basis functions to fit a polynomial through a set of points. A polynomial $p_n(x)$ passing through the points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ can be expressed using a Lagrange polynomial basis

$$p_n(x) = \sum_{i=0}^n y_i \ell_i(x_j) \quad \text{with} \quad \ell_i(x) = \prod_{\substack{k=0 \\ k \neq i}}^n \frac{x - x_k}{x_i - x_k}.$$

In other words, the vectors $\mathbf{x} = (x_0, x_1, \dots, x_n)$ and $\mathbf{y} = (y_0, y_1, \dots, y_n)$ uniquely define the polynomial. At these points $\ell_i(x_j) = \delta_{ij}$, i.e., $\ell_i(x_j)$ is the identity operator. Let's derive the differentiation operator \mathbf{D} such that $\mathbf{y}' = \mathbf{D}\mathbf{y}$, where $\mathbf{y}' = (y'_0, y'_1, \dots, y'_n)$ are the values of the derivative p'_n at points \mathbf{x} .

Start with

$$p'_n(x) = \sum_{k=0}^n y_k \ell'_k(x) = \sum_{k=0}^n y'_k \ell_k(x).$$

By taking the logarithmic derivative of $\ell_i(x)$, we have

$$\ell'_i(x) = \ell_i(x) \cdot [\log \ell_i(x)]' = \ell_i(x) \sum_{\substack{k=0 \\ k \neq i}}^n \frac{1}{x - x_k}$$

where $x_k = \cos k\pi/n$ are Chebyshev extremum points. When $i \neq j$,

$$d_{ij} = \ell'_i(x_j) = \left(\prod_{\substack{k=0 \\ k \neq i}}^n \frac{x_j - x_k}{x_i - x_k} \right) \left(\sum_{\substack{k=0 \\ k \neq i}} \frac{1}{x_i - x_k} \right) = \frac{1}{x_i - x_j} \prod_{\substack{k=0 \\ k \neq i, j}}^n \frac{x_j - x_k}{x_i - x_k}.$$

We'll calculate the numerator—the denominator is almost the same:

$$\begin{aligned} \prod_{\substack{k=0 \\ k \neq i, j}}^n (x - x_k) &= \frac{\prod_{k=0}^n (x - x_k)}{(x - x_i)(x - x_j)} \\ &= \frac{(x^2 - 1) \prod_{k=1}^{n-1} (x - x_k)}{(x - x_i)(x - x_j)} \\ &= \frac{(x^2 - 1) T'_n(x)}{(x - x_i)(x - x_j) 2^{n-1}} \end{aligned}$$

²The book is also available at <http://www.chebfun.org/ATAP> as set of m-files consisting of \LaTeX markup and Matlab code. A pdf can be generated from them using the publish command in Matlab.

When x_j is an interior Chebyshev point, we take the limit $x \rightarrow x_j$ and use (10.6) to get the expression

$$\prod_{\substack{k=0 \\ k \neq i, j}}^n (x_j - x_k) = \frac{(x_j^2 - 1)T_n''(x_j)}{2^{n-1}(x_j - x_i)} = \frac{n^2(-1)^{j-1}}{2^{n-1}(x_j - x_i)}.$$

At endpoints $x_0 = 1$ and $x_n = -1$, we have

$$\prod_{\substack{k=0 \\ k \neq i, j}}^n (x_j - x_k) = \frac{(x_j^2 - 1)T_n'(x_j)}{(x_j - x_i)(x_j \pm 1)2^{n-1}} = \frac{(\pm 2)T_n'(\pm 1)}{(x_j - x_i)2^{n-1}} = \frac{(\pm 1)^j n^2}{2^{n-2}(x_j - x_i)}.$$

Hence, we can define

$$d_{ij} = \ell'_i(x_j) = \frac{2^{\delta_{i0} + \delta_{in}} (-1)^{i-j}}{2^{\delta_{j0} + \delta_{jn}} \alpha_{ij}}$$

where $\alpha_{ij} = (x_i - x_j + \delta_{ij})$. We haven't yet properly defined the diagonal elements d_{ii} , so let's make that fix. Because the derivative of a constant function is zero, i.e., $\sum_{k=0}^n 1 \cdot d_{ik} = 0$, it follows that

$$d_{ii} = - \sum_{\substack{k=0 \\ k \neq i}}^n d_{ik}.$$

So, we have the following function to compute the Chebyshev differentiation matrix and corresponding Chebyshev points

```
function chebdiff(n)
    x = -cos.(LinRange(0, π, n))
    c = [2; ones(n-2); 2].*(-1).^(0:n-1)
    D = c./c'./(x .- x' + I)
    D = Diagonal([sum(D, dims=2)...]), x
end
```

The Chebyshev differentiation matrix is an $n \times n$ matrix with rank $n - 1$. It is also a nilpotent matrix of index n , meaning that $\mathbf{D}^n = \mathbf{0}$. Nilpotency of the differentiation matrix is the same as saying that if we n -times differentiate a polynomial of degree less than n , we are left with zero.

Example. Let's use the Chebyshev differentiation matrix to compute the derivative of $u(x) = e^{-4x^2}$. If \mathbf{u} is a vector of values at Chebyshev points, then \mathbf{Du} is the derivative of \mathbf{u} .

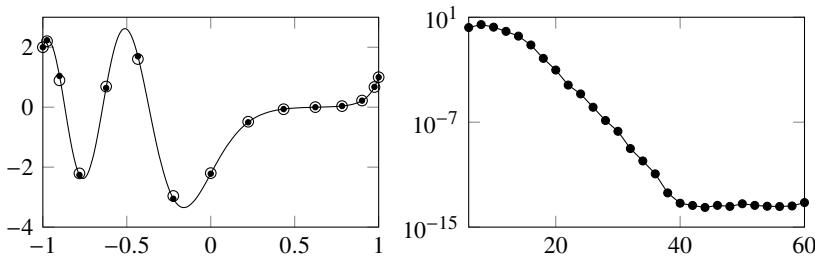


Figure 10.2: Left: Numerical solution (\circ) and exact solution (\cdot) to the Airy equation (10.7). Right: ℓ^∞ -error as a function of the number of Chebyshev points.

```
n = 15
D,x = chebdiff(n)
u = exp.(-4x.^2)
plot(x,D*u,marker=:o)
```

We can get higher-order derivatives by composing operators. For example, $\mathbf{D}^2\mathbf{u}$ is the second derivative of $u(x)$. By adding an additional constraint, we can use $\mathbf{D}^{-1}\mathbf{u}$ to compute the antiderivative of $u(x)$. For example, if $u(-1) = 2$, then the antiderivative of $u(x)$ is

```
B = zeros(1,n); B[1] = 1
plot(x,[B;D]\[2;u],marker=:o)
```

We can also use the Chebyshev differentiation matrix to solve a boundary value problem. Take the Airy equation

$$y'' - 256xy = 0 \quad \text{with} \quad y(-1) = 2 \quad \text{and} \quad y(1) = 1. \quad (10.7)$$

We'll prescribe two boundary conditions and then solve the system:

```
n = 15; k^2 = 256
D,x = chebdiff(n)
L = (D^2 - k^2*Diagonal(x))
L[[1,n],:] . = 0; L[1,1] = L[n,n] = 1
y = L\[2;zeros(n-2);1]
plot(x,y,marker=:dot)
```

How accurate is the solution? Figure 10.2 above shows a semi-log plot of the ℓ^∞ -error as a function of the number of nodes. The solution is already quite good using just 15 points. After that, for every two points added, the solution gains one digit of accuracy. The error is $O(10^{-n/2})$. Such a convergence rate is often

said to be exponential, geometric, or spectral. With about 40 points, we get a solution close to machine precision. ▶

These simple examples might spark a brilliant idea. We can represent functions using high-degree polynomials approximations, such as Chebyshev polynomials, to machine precision. Then function manipulation is the equivalent to performing operations on the polynomials. In the early 2000s, mathematicians Zachary Battles and Nick Trefethen applied this idea with the vision of creating computational systems that would “feel symbolic but run at the speed of numerics,” creating the open-source Matlab suite Chebfun. Numerical operators, such as `sum` and `diff`, are overloaded in Chebfun by operators using Chebyshev polynomial approximations, such as integration (using Clenshaw–Curtis quadrature) and differentiation. The backslash operator is overloaded to solving differential equations. Zeros are computed using a “colleague” matrix, the analog to the companion matrix for Chebyshev polynomial roots.³

 The ApproxFun.jl package includes methods for manipulating functions in Chebyshev and Fourier spaces, similar to the Matlab Chebfun package.

10.5 Wavelets

Wavelets provide yet another orthogonal system for approximating functions. As the “wave” in the word wavelet might suggest, wavelets are oscillatory and these oscillations balance each other out. Mathematically, their zeroth moments vanish. And as the “let” in the word wavelet might suggest, wavelets are small (like a piglet or a droplet) and localized in space or time. Mathematically, they have finite support or at least quickly decay. Wavelets are similar to Fourier polynomials, sharing many of the same features and applications, and Fourier polynomials themselves might be viewed as a special case of wavelets. A main difference is that while a Fourier polynomial is only localized in frequency, wavelets are localized in both space and frequency. While Fourier polynomials achieve orthogonality through scaling of basis functions, wavelets are orthogonal through both scaling and translation. And while there is just one family of Fourier polynomial, there are perhaps hundreds of families of wavelets.⁴ This section provides a condensed introduction to wavelets. Ingrid Daubechies’ “Ten Lectures on Wavelets,” Adhemar Bultheel’s “Learning to Swim in a Sea of Wavelets,” and

³The matrix analogue to the companion matrix for finding the roots of a general orthogonal polynomial is called a “comrade” matrix.

⁴Mathematician Laurent Duval has a compendium of over one hundred “starlets,” or wavelets ending in “let.”

Gilbert Strang's "Wavelets and Dilation Equations: A Brief Introduction" are all approachable primers of this important area of numerical mathematics.

The origins of wavelet theory date back to 1909 when mathematician Alfréd Haar proposed a simple system of orthogonal functions, now called Haar wavelets, in his doctoral dissertation. It took another 75 years for wavelets to emerge in their modern form and be named.⁵ In the late 1980s mathematicians Jean Morlet, Ingrid Daubechies, Stéphane Mallat, and Yves Meyer, motivated by problems in signals processing, developed a systematic theory of multiresolution or multiscale approximation.⁶ What is multiscale or multiresolution approximation intuitively? Well, right now, I'm looking across the room at a photograph of my beautiful wife. At a distance, I can tell that it is of a woman in a bright city at night. If I walk up to it, I can make out features of her face and smile, enough to confirm that this woman is indeed my wife. And if I examine it even more closely, I see minute details like the glint of a diamond earring I gave her on our wedding day. Each of these successive resolutions adds an additional layer of detail to the story in the photo. We can similarly find multiscales in music—a score, a motif, a measure, a note—and in literature—a novel, a paragraph, a sentence, a word.

Mathematically, multiresolution analysis provides a means of breaking up a function into the sum of approximations and details. Consider the following process of multiresolution approximation using a Haar wavelet. Start with the values:

$$1 \quad 3 \quad -3 \quad -1 \quad 9 \quad 11 \quad 5 \quad 7 \quad 23 \quad 25 \quad 21 \quad 19 \quad 16 \quad 16 \quad 19 \quad 21$$

First combine them pairwise recording the averages and the differences:

$$\begin{array}{ccccccccc} 1, 3 & -3, -1 & 9, 11 & 5, 7 & 23, 25 & 21, 19 & 16, 16 & 19, 21 \\ 2 \mp 1 & -2 \mp 1 & 10 \mp 1 & 6 \mp 1 & 24 \mp 1 & 20 \mp -1 & 16 \mp 0 & 20 \mp 1 \end{array}$$

Compute the pairwise averages and differences again:

$$\begin{array}{cccc} 2, -2 & 10, 6 & 24, 20 & 16, 20 \\ 0 \mp -2 & 8 \mp -2 & 22 \mp -2 & 18 \mp 2 \end{array}$$

And again:

$$\begin{array}{cc} 0, 8 & 22, 18 \\ 4 \mp 4 & 20 \mp 2 \end{array}$$

And one more time:

$$\begin{array}{c} 4, 20 \\ 12 \mp 8 \end{array}$$

⁵Wavelets were originally called *ondelettes* by the French pioneers Morlet and Grossman after a term frequently used in geophysics for an oscillating localized function.

⁶Electrical engineers had already developed more elaborate algorithms such as the subband decomposition algorithm before the rigorous function-analytic framework caught up.

The value 12 is the average of all the initial numbers. By adding the differences to it, we can reconstruct the original numbers.

$$\{12\} \mp \{8\} \mp \{4, 2\} \mp \{-2, 2, -2, 2\} \mp \{1, 1, 1, 1, 1, -1, 0, 1\}$$

For example, $12 - 8 - 4 - (-2) - 1 = 1$ and $12 - 8 - 4 - (-2) + 1 = 3$ give the first two numbers in the original sequence.

Multiresolution implies that basis functions act on multiple scales and are even self-similar in scale. Fourier polynomial bases $w_n(x) = e^{inx}$ are self-similar in scale. We can construct every one of them by scaling the function $w(x) = e^{ix}$, so that $w_n(x) = w(nx)$. But wavelets ought to also be compactly or almost compactly supported, and Fourier polynomials definitely do not have compact support. We can't construct all functions using merely a combination of scaled copies of a compactly supported function—we also need translations of this function. So, we want basis functions that are also self-similar in space. And, we'll want all of those basis functions to be orthogonal to their translates. Finding a class of functions that possesses all of these properties isn't easy. Before 1985, the Haar wavelet was the only orthogonal wavelet that people knew, and until Yves Meyer and Jan Olov Strömberg independently constructed a second orthogonal wavelet, many researchers believed that the Haar wavelet was the only one.

► Scaling function

Let's start with a formal definition and then make it actionable. Take the space of square-integrable functions L^2 . A *multiresolution approximation* is a sequence of subspaces $\emptyset \subset \dots \subset V_{-1} \subset V_0 \subset V_1 \subset \dots \subset L^2$ such that

1. $f(x) \in V_n$ if and only if $f(2x) \in V_{n+1}$ (dilation)
2. $f(x) \in V_n$ if and only if $f(x - 2^{-n}k) \in V_n$ for integer k (translation)
3. $\bigcup V_n$ is dense in L^2

From this definition, there should exist a basis function $\phi(x) \in V_0$, called the *scaling function* or *father wavelet*. Let's denote the translated copy of the father wavelet as $\phi_{0,k}(x) = \phi(x - k)$ for integer k . Then we can express any function $f \in V_0$ as $f(x) = \sum_{k=-\infty}^{\infty} a_k \phi_{0,k}(x)$ for some coefficients a_k . In particular, the scaled function $\phi(x/2) \in V_{-1} \subset V_0$ can be written as $\phi(x/2) = \sum_{k=-\infty}^{\infty} c_k \phi(x - k)$ for some c_k , and hence the father wavelet satisfies a dilation equation:

$$\phi(x) = \sum_{k=-\infty}^{\infty} c_k \phi(2x - k). \quad (10.8)$$

In general, we'll only consider scaling functions with compact support, and therefore the sum is effectively only over finite k . If we integrate both sides of

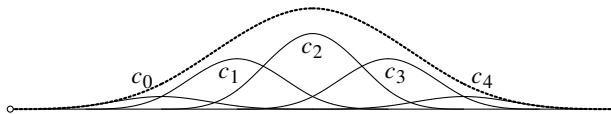
the dilation equation, then

$$\int_{-\infty}^{\infty} \phi(x) dx = \int_{-\infty}^{\infty} \sum_{k=-\infty}^{\infty} c_k \phi(2x - k) dx = \sum_{k=-\infty}^{\infty} c_k \int_{-\infty}^{\infty} \frac{1}{2} \phi(s) ds,$$

where we've made the change of variable $s = 2x - k$. If $\int_{-\infty}^{\infty} \phi(x) dx \neq 0$, then we have the condition

$$\sum_{k=-\infty}^{\infty} c_k = 2. \quad (10.9)$$

Example. Let's find solutions to dilation equation (10.8) that satisfy the constraint (10.9). One approach is to use splines. When $c_0 = 2$ and all other c_k are zero, the solution is a delta function $\phi(x) = \delta(x)$. When $c_0 = c_1 = 1$ and $c_k = 0$ otherwise, the solution is the rectangle function— $\phi(x) = 1$ if $x \in [0, 1]$ and $\phi(x) = 0$ otherwise. This function is the father wavelet of the Haar wavelets. If we set $\{c_0, c_1, c_2\} = \{\frac{1}{2}, 1, \frac{1}{2}\}$, we get a triangular function as the father wavelet. With coefficients $\{\frac{1}{4}, \frac{3}{4}, \frac{3}{4}, \frac{1}{4}\}$, we get the quadratic B-spline. And with $\{\frac{1}{8}, \frac{1}{2}, \frac{3}{4}, \frac{1}{2}, \frac{1}{8}\}$, we have the cubic B-spline. All of these can be demonstrated by construction. For example, the scaled cubic B-spline (depicted with a dotted line) is the sum of the five translated cubic B-splines:



Other than the B_0 Haar wavelet, B-spline wavelets are not orthogonal, which limits their usefulness. ◀

Let's consider scaling functions that are shift-orthonormal:

$$\int_{-\infty}^{\infty} \phi(x) \phi(x - m) dx = \delta_{0m}$$

for all integers m . What additional constraints do we need to put on the coefficients c_k ? To answer this question we'll need to take a short detour to examine a few properties of the scaling function. The quickest route is through Fourier space. If we take the Fourier transform of the scaling function

$$\hat{\phi}(\xi) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \phi(x) e^{-ix\xi} dx,$$

then the dilation equation is

$$\begin{aligned}\hat{\phi}(\xi) &= \sum_{k=-\infty}^{\infty} \frac{c_k}{2\pi} \int_{-\infty}^{\infty} \phi(2x - k) e^{-ix\xi} dx \\ &= \hat{H}\left(\frac{1}{2}\xi\right) \cdot \frac{1}{2\pi} \int_{-\infty}^{\infty} \phi(y) e^{-iy\xi/2} dy = \hat{H}\left(\frac{1}{2}\xi\right)\hat{\phi}\left(\frac{1}{2}\xi\right)\end{aligned}$$

where the scaled discrete Fourier transform

$$\hat{H}(\xi) = \frac{1}{2} \sum_{k=-\infty}^{\infty} c_k e^{-ik\xi}. \quad (10.10)$$

If we take $\int_{-\infty}^{\infty} \phi(x) dx = 1$, then $\hat{\phi}(0) \neq 0$ and it follows that $\hat{H}(0) = 1$. Hence, we must have that $\sum_{k=-\infty}^{\infty} c_k = 2$.

Theorem 38. $\sum_{k=-\infty}^{\infty} |\hat{\phi}(\xi + 2\pi k)|^2 = (2\pi)^{-1}$.

Proof. Note that for any function, the autocorrelation

$$\begin{aligned}\int_{-\infty}^{\infty} \phi(x)\phi(x-m) dx &= \int_{-\infty}^{\infty} \phi(x) \int_{-\infty}^{\infty} \hat{\phi}(\xi) e^{+i(x-m)\xi} d\xi dx \\ &= \int_{-\infty}^{\infty} \hat{\phi}(\xi) e^{-im\xi} \int_{-\infty}^{\infty} \phi(x) e^{+ix\xi} dx d\xi \\ &= \int_{-\infty}^{\infty} |\hat{\phi}(\xi)|^2 e^{-im\xi} d\xi.\end{aligned}$$

Using the orthogonality of the scaling function

$$\begin{aligned}\int_0^{2\pi} \frac{1}{2\pi} e^{-im\xi} d\xi &= \delta_{0m} = \int_{-\infty}^{\infty} \phi(x)\phi(x-m) dx \\ &= \int_{-\infty}^{\infty} |\hat{\phi}(\xi)|^2 e^{-im\xi} d\xi \\ &= \int_0^{2\pi} \sum_{k=-\infty}^{\infty} |\hat{\phi}(\xi + 2\pi k)|^2 e^{-im\xi} d\xi. \quad \square\end{aligned}$$

Corollary 39. $|\hat{H}(\xi)|^2 + |\hat{H}(\xi + \pi)|^2 = 1$.

Proof. Use the dilation equation $\hat{\phi}(2\xi) = \hat{H}(\xi)\hat{\phi}(\xi)$ with the result from the previous theorem:

$$\begin{aligned}\frac{1}{2\pi} &= \sum_{k=-\infty}^{\infty} |\hat{\phi}(2\xi + 2k\pi)|^2 = \sum_{k=-\infty}^{\infty} |\hat{H}(\xi + k\pi)|^2 |\hat{\phi}(\xi + k\pi)|^2 \\ &= |\hat{H}(\xi)|^2 \sum_{k=-\infty}^{\infty} |\hat{\phi}(\xi + 2k\pi)|^2 + |\hat{H}(\xi + \pi)|^2 \sum_{k=-\infty}^{\infty} |\hat{\phi}(\xi + 2k\pi + \pi)|^2.\end{aligned}$$

The last line results from splitting the sum over even and odd values of k and applying the 2π -periodicity of $\hat{H}(\xi)$. \square

Corollary 40. *The following conditions on the dilation equation are necessary for an orthogonal scaling function:*

$$\sum_{k=-\infty}^{\infty} (-1)^n c_k = 0, \quad \sum_{k=-\infty}^{\infty} c_k^2 = 2, \quad \text{and} \quad \sum_{k=-\infty}^{\infty} c_k c_{k-2m} = 0. \quad (10.11)$$

Proof. Because $\hat{H}(0) = 1$, it follows that $|\hat{H}(\pi)|^2 = 1 - |\hat{H}(0)|^2 = 0$, and the first condition follows directly. To show the second two conditions, expand $|\hat{H}(\xi)|^2 + |\hat{H}(\xi + \pi)|^2 = 1$ using the definitions of $\hat{H}(\xi)$:

$$\frac{1}{4} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} c_k c_j e^{-i(k-j)\xi} + \frac{1}{4} \sum_{j=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} c_k c_j (-1)^{k-j} e^{-i(k-j)\xi} = 1.$$

The terms for which $k - j$ is odd cancel and we are left with

$$\frac{1}{2} \sum_{m=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} c_k c_{k-2m} e^{-im\xi} = 1.$$

This expression is true for all ξ , so it follows that $\sum_{k=-\infty}^{\infty} c_k c_{k-2m} = 2\delta_{0m}$. \square

Example. The Haar scaling functions are orthonormal with $c_0 = 1$ and $c_1 = 1$. But they are piecewise constant. Let's construct a higher-order orthogonal wavelet by determining values c_0, c_1, c_2 , and c_3 that satisfy the constraints (10.11):

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 2 \\ c_2 c_0 + c_3 c_1 &= 0 \\ c_0 - c_1 + c_2 - c_3 &= 0 \end{aligned}$$

We'll need to add an additional constraint for a unique solution. Note that (10.9) can be derived from algebraic manipulation of (10.11), so it will not provide any additional constraint on the coefficients. For fast decaying wavelets, we want as many moments $\int_{-\infty}^{\infty} x^p \psi(x) dx$ of the wavelet function $\psi(x)$ discussed in the next section to vanish as possible. To approximate smooth functions with error $O(h^{-p})$, then $\hat{H}(\xi)$ must have zeros of order p at $\xi = \pi$. (See Strang [1989].) So, we'll enforce the additional constraint:

$$0c_0 - 1c_1 + 2c_2 - 3c_3 = 0.$$

The solution to this system of equations is

$$\{c_0, c_1, c_2, c_3\} = \left\{ \frac{1}{4} (1 + \sqrt{3}), \frac{1}{4} (3 + \sqrt{3}), \frac{1}{4} (3 - \sqrt{3}), \frac{1}{4} (1 - \sqrt{3}) \right\}$$

The corresponding wavelet is called D_4 from the work of Ingrid Daubechies. The D_2 Daubechies wavelet is the Haar wavelet. ◀

In practice, we don't ever need to explicitly construct the scaling functions, but doing so can help us understand them better. One way to construct the scaling functions $\phi(x)$ is by using the dilation equation $\phi(x) = \sum_{k=-\infty}^{\infty} c_k \phi(2x - k)$ as a fixed-point method starting with $\phi(x)$ initially equal to some guess function, e.g., the rectangle function, and iterating until the method converges. A faster approach is by using recursion. If we know the values of $\phi(x)$ at integer values of x , then we can use the dilation equation to compute $\phi(x)$ at half-integer values of x . Knowing these values, we use the dilation equation to compute the values at quarter-integer values, and so on to get $\phi(x)$ at dyadic rational values $x = i/2^j$. Here's a naïve implementation:

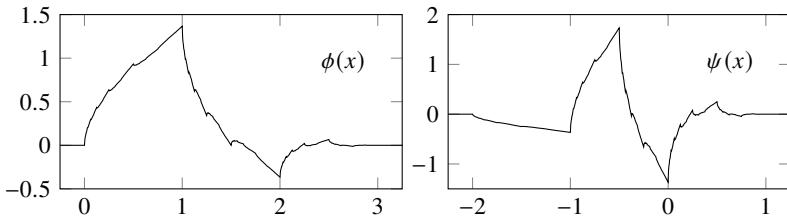
```
using OffsetArrays
function scaling(c,ϕₖ,n)
    m = length(c)-1; ℓ = 2^n
    ϕ = OffsetVector(zeros(3*m*ℓ),-m*ℓ)
    k = (0:m)*ℓ
    ϕ[k] = ϕₖ
    for j = 1:n
        for i = 1:m*2^(j-1)
            x = (2i-1)*2^(n-j)
            ϕ[x] = c ∙ ϕ[2x .- k]
        end
    end
    ((0:m*ℓ-1)/ℓ, ϕ[0:m*ℓ-1])
end
```

The `OffsetArrays.jl` package allows us to simplify notation by starting the scaling function array at $x = 0$. The domain has been padded on the left and right with zeros to simplify implementation.

Let's compute the scaling function for D_4 . We'll need to first determine the values $\phi(0)$, $\phi(1)$, $\phi(2)$, $\phi(3)$. The D_4 scaling function is identically zero except over the interval $[0, 3]$, so $\phi(0) = \phi(3) = 0$. From the dilation equation, we then have

$$\begin{aligned}\phi(1) &= c_0\phi(2) + c_1\phi(1) \\ \phi(2) &= c_2\phi(2) + c_3\phi(1),\end{aligned}$$

which says that $(\phi(1), \phi(2))$ is an eigenvector of the matrix with elements (c_1, c_0, c_3, c_2) . The eigenvectors of this matrix are $(-1, 1)$ and $(1 + \sqrt{3}, 1 - \sqrt{3})$. Which one do we choose? Here, we'll state a property without proof—the scaling functions form a partition of unity, i.e., the sum of any scaling function taken at integer values equals one: $\sum_{k=-\infty}^{\infty} \phi(k) = 1$. So, $(-1, 1)$ cannot be the right

Figure 10.3: Daubechies D_4 scaling function ϕ and wavelet function ψ .

choice. This means that we take $\phi(1) = \frac{1}{2}(1 + \sqrt{3})$ and $\phi(2) = \frac{1}{2}(1 - \sqrt{3})$. The Daubechies D_4 scaling function is plotted in the figure above using

```
c = [1+√3, 3+√3, 3-√3, 1-√3]/4
z = [0, 1+√3, 1-√3, 0]/2
(x,φ) = scaling(c, z, 8)
plot(x,φ)
```

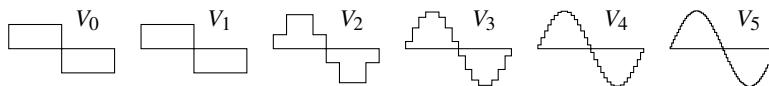
We can define normalized bases functions in V_n as

$$\phi_{nk}(x) = 2^{n/2} \phi(2^n x - k).$$

Furthermore, we can define an orthogonal projection of a function $f \in L^2$:

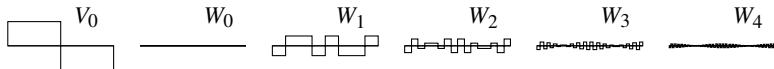
$$P_n f = \sum_{k=-\infty}^{\infty} (f, \phi_{nk}) \phi_{nk} = \sum_{k=-\infty}^{\infty} a_{nk} \phi_{nk} \quad (10.12)$$

for some coefficients $a_{nk} = (f, \phi_{nk})$. The projection into V_n acts to smooth the data by filtering out finer details. In the following figure, the function $f(x) = \sin \pi x$ is projected into the subspaces generated using Haar wavelets:



► Wavelet function

Notice that the scaling functions generate a sequence of nested subspaces $V_0 \subset V_1 \subset V_2 \subset \dots \subset L^2$. Let W_n be the orthogonal complement of V_n in V_{n+1} . That is, let $V_{n+1} = V_n \oplus W_n$. Then, $V_{n+1} = V_0 \oplus W_0 \oplus W_1 \oplus \dots \oplus W_n$. For any function $f_{n+1} \in V_n$, we can make the decomposition $f_{n+1} = f_n + g_n$ where $f_n \in V_n$ and $g_n \in W_n$. And continuing, $f_{n+1} = g_n + g_{n-1} + \dots + g_0 + f_0$ for $f_j \in V_j$ and $g_j \in W_j$. Were we to continue, we would have $V_{n+1} = \bigoplus_{k=-\infty}^n W_k$.



We can recursively decompose the V_{n+1} -projection of any function $f \in L^2$ into orthogonal components in V_n and W_n by using orthogonal projection operators

$$P_{n+1} f = P_n f + Q_n f,$$

where $P_n f$ is given by (10.12) and

$$Q_n f = \sum_{k=-\infty}^{\infty} (f, \psi_{nk}) \psi_{nk} = \sum_{k=-\infty}^{\infty} d_{nk} \psi_{nk}$$

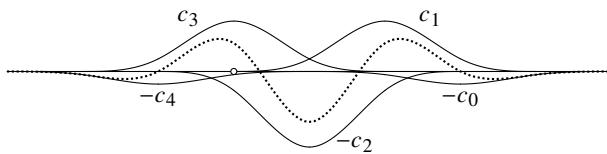
for $d_{nk} = (f, \psi_{nk})$ and basis elements $\psi_{nk} \in W_n$:

$$\psi_{nk}(x) = 2^{n/2} \psi(2^n x - k).$$

Specifically, we have that $Q_0 f = (P_1 - P_0)f$. From the dilation equation (10.8), we can derive a function $\psi(x) \in W_0$, called the *wavelet function* or *mother wavelet*:

$$\psi(x) = \sum_{k=-\infty}^{\infty} (-1)^k c_{1-k} \phi(2x - k). \quad (10.13)$$

Notice how the mother wavelet differs structurally from the father wavelet—alternating the signs the coefficients, reversing the order terms, and shifting them left along the x -axis. Also, unlike the father wavelet, which integrates to one, the mother wavelet integrates to zero. Using this definition we can generate the wavelet function (in the dotted line below) of the cubic B-spline scaling function on page 262:



Using the scaling function ϕ from the code on page 265, we can compute the wavelet function ψ with

```

 $\psi = \text{zero}(\phi); n = \text{length}(\phi)-1; \ell = \text{length}(\psi) \div 2n$ 
for k=0:n
     $\psi[(k*\ell+1):(k+n)*\ell] += (-1)^k * c[n-k+1] * \phi[1:2:end]$ 
end

```

which we can use to compute the Daubechies D_4 wavelet function in the figure on the preceding page.

► Discrete wavelet transform

Using the self-similar structure of wavelets, we can develop a recursive formulation for the discrete wavelet transform (DWT). Using the dilation equation (10.8), we have that

$$\phi_{nk}(x) = 2^{n/2} \sum_{i=-\infty}^{\infty} c_i \phi(2^n x - 2^n k - i) = \frac{1}{\sqrt{2}} \sum_{j=-\infty}^{\infty} c_{j-2k} \phi_{n+1,j}(x),$$

and similarly from (10.13) we have

$$\psi_{nk}(x) = \frac{1}{\sqrt{2}} \sum_{j=-\infty}^{\infty} (-1)^{1-j} c_{1-j+2k} \phi_{n+1,j}(x).$$

From these,

$$(\phi_{ni}, \phi_{n+1,j}) = \frac{1}{\sqrt{2}} c_{i-2j} \quad \text{and} \quad (\psi_{ni}, \phi_{n+1,j}) = \frac{(-1)^{1-i}}{\sqrt{2}} c_{1-i+2j}.$$

So,

$$\begin{aligned} a_{n+1,k} &= (f, \phi_{n+1,k}) = (P_{n+1} f, \phi_{n+1,k}) = (P_n f + Q_n f, \phi_{n+1,k}) \\ &= (P_n f, \phi_{n+1,k}) + (Q_n f, \phi_{n+1,k}) \\ &= \left(\sum_{i=-\infty}^{\infty} a_{ni} \phi_{ni}, \phi_{n+1,k} \right) + \left(\sum_{i=-\infty}^{\infty} a_{ni} \psi_{ni}, \phi_{n+1,k} \right) \\ &= \frac{1}{\sqrt{2}} \sum_{i=-\infty}^{\infty} c_{k-2i} a_{ni} + \frac{1}{\sqrt{2}} \sum_{i=-\infty}^{\infty} (-1)^{1-k+2i} c_{1-k+2i} d_{ni}. \end{aligned} \quad (10.14)$$

Similarly, we can write

$$d_{n+1,k} = \frac{1}{\sqrt{2}} \sum_{i=-\infty}^{\infty} c_{k-2i} a_{ni} - \frac{1}{\sqrt{2}} \sum_{i=-\infty}^{\infty} (-1)^{1-k+2i} c_{1-k+2i} d_{ni}. \quad (10.15)$$

These two expressions allow us to recursively reconstruct a function from its wavelet components. In practice, the algorithm can be computed recursively much like a fast Fourier transform. Reversing these steps allows us to deconstruct a function into its wavelet components.

We can represent (10.14) in matrix notation as

$$\mathbf{a}_{n+1} = [\mathbf{H}^\top \quad \mathbf{G}^\top] \begin{bmatrix} \mathbf{a}_n \\ \mathbf{d}_n \end{bmatrix} \quad \text{or alternatively} \quad \begin{bmatrix} \mathbf{H} \\ \mathbf{G} \end{bmatrix} \mathbf{a}_{n+1} = \begin{bmatrix} \mathbf{a}_n \\ \mathbf{d}_n \end{bmatrix}.$$

where \mathbf{H} and \mathbf{G} are orthogonal matrices: $\mathbf{H}^\top \mathbf{H} = \mathbf{I}$, $\mathbf{G}^\top \mathbf{G} = \mathbf{I}$, and $\mathbf{G}^\top \mathbf{H} = \mathbf{0}$. Both \mathbf{G} and \mathbf{H} are infinite-dimensional matrices because they extend across the

entire real line, but we can restrict ourselves to a finite domain while maintaining orthogonality by imposing periodic boundaries. Because the scaling function has compact support, the coefficients c_k are nonzero for finite values of k . Consider the case when $c_k \neq 0$ for $k = 0, 1, 2, 3$. The matrix structure is the same for other orders.

$$\mathbf{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ & c_0 & c_1 & c_2 & c_3 \\ & & c_0 & c_1 & c_2 & c_3 \\ c_2 & c_3 & & & c_0 & c_1 \end{bmatrix}$$

and

$$\mathbf{G} = \frac{1}{\sqrt{2}} \begin{bmatrix} c_3 & -c_2 & c_1 & -c_0 \\ & c_3 & -c_2 & c_1 & -c_0 \\ & & c_3 & -c_2 & c_1 & -c_0 \\ c_1 & -c_0 & & & c_3 & -c_2 \end{bmatrix}.$$

It's common to write the discrete wavelet transform as a square matrix

$$\mathbf{T} = \mathbf{P} \begin{bmatrix} \mathbf{H} \\ \mathbf{G} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & -c_2 & c_1 & -c_0 \\ & c_0 & c_1 & c_2 & c_3 \\ & c_3 & -c_2 & c_1 & -c_0 \\ & & c_0 & c_1 & c_2 & c_3 \\ & & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & & & c_0 & c_1 \\ c_1 & -c_0 & & & c_3 & -c_2 \end{bmatrix}$$

where \mathbf{P} is the “perfect shuffle” permutation matrix that interweaves the rows of \mathbf{H} and \mathbf{G} . Notice that for \mathbf{T} to be orthogonal, we must have that $\frac{1}{2}(c_0^2 + c_1^2 + c_2^2 + c_3^2) = 1$ and $c_2c_0 + c_3c_1 = 0$. Furthermore, $c_0 + c_1 + c_2 + c_3 = 2$. And, for vanishing zeroth moment $c_0 + c_2 = 0$ and $c_1 + c_4 = 0$.

 The Wavelets.jl package includes several utilities for discrete wavelet transforms.

Example. One application of discrete wavelet transforms is image compression. The JPEG 2000 format was developed in the late 1990s as a replacement to the discrete-cosine-transform-based JPEG format developed in the early 1990s. JPEG 2000 uses Cohen–Daubechies–Feauveau (CDF) 9/7 wavelets for lossy compression and LeGall–Tabatabai (LGT) 5/3 wavelets for lossless compression. Because of the multiresolution nature of wavelets, lower resolution subsections of JPEG 2000 images can be extracted and downloaded rather than the entire image. This feature is particularly useful when working with massive gigapixel medical, remote sensing, and scientific images, especially where high resolution and low edge-compression artifacts are essential. ICER, a similar wavelet-based



Figure 10.4: Discrete wavelet transform (right) of the image (left).

image compression format, is used by NASA to transmit two-dimensional images and three-dimensional, hyperspectral images back from the Mars Exploration Rovers. (See Kiely et al. [2006].) While JPEG 2000 is arguably a better format than the original JPEG format, it never succeeded in replacing the JPEG format for common applications. One of the reasons that it failed to catch on was that it required more memory than a typical late 1990s computer could efficiently process. And because it uses a different algorithm from the JPEG format, there was no backward compatibility. So, industry was hesitant to become adopt it, especially when the JPEG format already worked quite well.

Let's examine the two-dimensional DWT of a 512×512 -pixel image using the Wavelets.jl library. We'll clamp the output values between 0 and 1 and adjust the levels so that small values are more easily seen, especially on a printed page.

```
using Wavelets, Images
img = float.(Gray.(load(download(bucket*"laura_square.png"))))
B = dwt(Float32.(img), wavelet(WT.haar))
[img Gray.(1 .- clamp01.(sqrt.(abs.(B))))]
```

See Figure 10.4 above. Beyond being a little trippy, what's going on in it? Recall the structure of the one-dimensional multiresolution decomposition discussed on page 261

$$\{12\} \mp \{8\} \mp \{4, 2\} \mp \{-2, 2, -2, 2\} \mp \{1, 1, 1, 1, 1, -1, 0, 1\}.$$

This decomposition starts with the sum of the values of the original sequence (the contribution of the father wavelet). The next term tells us the gross fluctuation (the contribution of the mother wavelet). After this, each resolution provides finer detail with each subsequent generations of daughter wavelets. Figure 10.4

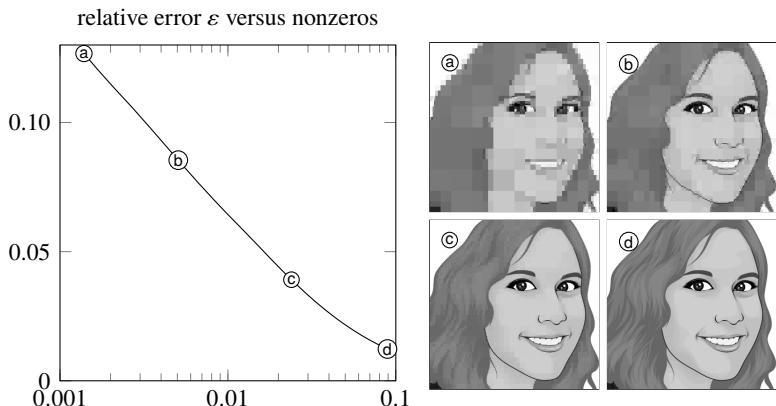


Figure 10.5: The relative Frobenius error of a DWT (Haar) compressed image as a function of the filter threshold. Also, see the QR link at the bottom of this page.

shows a similar decomposition in two-dimensions. The tiny pixel in the top-left corner has a value of 325, the sum of the pixel values across the original image. It's surrounded by three one-pixel-sized components that provide the horizontal, vertical, and diagonal details to the next higher resolution. Around these are 2×2 pixels details. Then 4×4 , and so on until we get to 256×256 -pixel blocks. These three largest blocks show the fluctuations that occur at the pixel level of the original image.

We can compress an image by filtering or zeroing out the coefficients of the DWT of that image whose magnitudes fall below a given threshold. Let's set up such a function.

```
function dwtfilter(channels,wt,k)
    map(1:3) do i
        A = dwt(channels[i,:,:], wavelet(wt))
        threshold!(A, HardTH(), k)
        clamp01!(idwt(A, wavelet(wt)))
    end
end
```

We'll examine the effect of varying the filter threshold on lossy compression error. The following code creates an interactive widget to compare different wavelets and thresholds:

```
using Interact, Wavelets, Images
img = float.(load(download(bucket*"laura_square.png")))
channels = channelview(img)
func = Dict("Haar"=>WT.haar, "D4"=>WT.db4, "Coiflet"=>WT.coif4)
```



```
@manipulate for wt in togglebuttons(func; label="Transform"),
  k in slider(0:0.05:1.0; value = 0, label = "Threshold")
  colorview(RGB,dwtfilter(channels,wt,k)...)
end
```

The relative Frobenius error, as a function of the percentage of non-zero elements, is plotted in Figure 10.4 on the previous page. Compare the DWT image compression with the DCT image compression in Figure 6.5 on page 162. DWT image compression maintains sharp edges without the ringing artifacts that are characteristic of DCT image compression. Even at extremely high levels of compression such details are well preserved. This feature in particular makes DWT especially relevant in applications such as fingerprint analysis. ◀

10.6 Neural networks

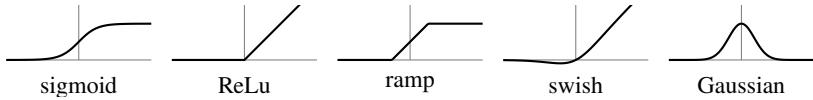
Artificial neural networks provide yet another way of approximating functions. In 1943, neurophysiologist Warren McCulloch and mathematician Walter Pitts, inspired by Bertrand Russell's formal logic and Alan Turing's recent research on computability, teamed up to develop a basic computational model of the brain.⁷ They surmised that a brain was simply a biological Turing machine with logic encodings in neurons, and they demonstrated that arbitrary Boolean functions could be represented as a mathematical "nervous net." While McCulloch and Pitt's model sought to mimic the brain, today's artificial neural networks bear as much resemblance to the biological neural networks that inspired them as airplanes bear resemblance to birds.⁸

An artificial neuron is a function consisting of the composition of an affine transformation and a nonlinear function $\phi(x)$, called an activation function. Often, a second affine transformation is added to the composition. The activation function $\phi(x)$ acts as a basis function. The first affine transformation affects the input values by stretching and sliding the activation function along the x -axis.

⁷Bullied as a boy, Walter Pitts spent hours hiding in the library, reading books like Whitehead and Russell's *Principia Mathematica*. When he was 15, he ran away from home and showed up at the University of Chicago, where Bertrand Russell was teaching. He remained at the university for several years, never registering as a student, often homeless and working menial jobs. It's here he met Warren McCulloch and they developed the foundational theory of neural nets. Walter Pitts, enticed to continue his research on modeling the brain, left Chicago to complete his PhD at MIT. Over time he grew increasingly lonely and distraught, eventually setting fire to his dissertation and all of his notes. He tragically died of alcoholism at age 46. To dig deeper, read Amanda Gefter's article in *Nature Quarterly*.

⁸Biomimicry looks to nature for insights and inspiration in solving engineering problems. Early unsuccessful pioneers of artificial flight made wings from wood and feathers attached to their arms. Leonardo da Vinci's study of birds led him to devise his ornithopter. Otto Lilienthal did the same in developing his glider. By the time that Orville and Wilbur Wright engineered the first successful powered aircraft, it looked very little like a bird other than having wings.

The second affine transformation affects the output values by stretching and sliding the activation function in the y -direction. An activation could be any number of functions—a radial basis function such as a B-spline or Gaussian, a monotonically increasing sigmoid function, a threshold function, a rectifier function, and so on.



McCulloch and Pitts used a step function \square for an activation function to model neurons as logical operators. Sigmoid activation functions \swarrow , which naturally arise from logistic regression, were the most popular until fairly recently. Now, rectified linear units or ReLUs \nearrow are in vogue, largely do their simplicity and effectiveness in deep learning.

By taking a linear combination of scaled, translated activation functions

$$f_n(x) = \sum_{i=1}^n c_i \phi(w_i x + b_i)$$

we can approximate any continuous function $f(x)$. Furthermore, it doesn't matter which activation function we choose beyond practical considerations, because we can linearly combine activation functions to create new ones. For example, consider a rectified linear unit $x^+ = \text{ReLU}(x) = \max(0, x) \nearrow$. Combining two ReLUs $(x+1)^+ - x^+$ will make a ramp function \swarrow , an approximation to a sigmoid function. Combining two ramps—or three ReLUs $(x+1)^+ - 2x^+ + (x-1)^+$ —will make a linear B-spline basis function \wedge . Likewise, two step functions will make a rectangle function \square and two sigmoid functions will make a sigmoid bump \sim .

Theorem 41 (Universal approximation theorem). *Let $f(x)$ be a sufficiently smooth function and let $f_n(x)$ be a neural net approximation of $f(x)$ over an interval of length ℓ using a ReLU activation function. For any $\varepsilon > 0$, the error $\|f_n(x) - f(x)\|_\infty \leq \varepsilon$ if we use at least $\ell \sqrt{\max |f''(x)| / 8\varepsilon}$ neurons.*

Proof. Let x_1, x_2, \dots, x_n be n equally spaced nodes over the interval. Then the mesh size h is ℓ/n . A linear B-spline basis function can be built using three ReLU functions, so $f_n(x) = \sum_{i=1}^n f(x_i) B_1(h^{-1}(x - x_i))$ is piecewise interpolating polynomial with $f_n(x_i) = f(x_i)$. From theorem 26 on page 228, regarding polynomial interpolation error, on each subinterval

$$|f_n(x) - f(x)| = \frac{1}{2} f''(\xi_i)(x - x_i)(x_{i+1} - x) \leq \frac{1}{8} |f''(\xi)| h^2$$

for some ξ_i in the subinterval. Choose n such that $\frac{1}{8} \max |f''(x)| h^2 \leq \varepsilon$. In practice, we need one ReLU function for every line segment. So, we need n ReLU neurons, where $n \geq \ell \sqrt{|f''(\xi)| / 8\varepsilon}$. \square

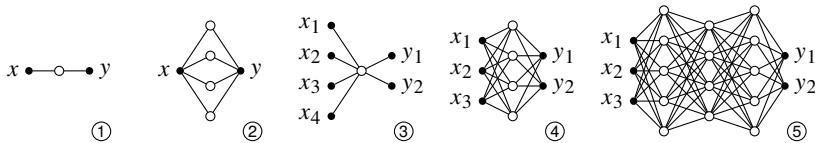


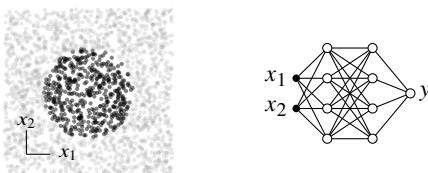
Figure 10.6: Neural networks: ① a simple artificial neuron; ② a simple neural net; ③ a multiple-input/multiple-output neuron; ④ a single layer neural net; and ⑤ a deep neural net.

The function $f(x)$ need not be smooth—the proof of a similar result for Lipschitz functions is left as an exercise. George Cybenko’s influential 1989 paper “Approximation by superpositions of a sigmoidal function” generalizes the universal approximation theorem for an arbitrary sigmoid function. From the universal approximation theorem, it may appear that neural networks are little different than directly using B-splines, radial basis functions, or any other method we have already discussed. Namely, we can better approximate a function by adding additional neurons into a single layer. The real magic of neural nets emerges by using multiple layers. We can feed the output of one neuron into another neuron. And, we can feed that neuron into yet another one, and that one into another, and so on.

Consider the graphs in Figure 10.6. Input and output nodes are denoted with \bullet . The neurons are denoted with \circ . The weighted edges, often called synapses, connect the nodes together. The simplest artificial neuron is $y = w_2\phi(w_1x + b)$. A neuron can have multiple inputs and multiple outputs $\mathbf{y} = \mathbf{w}_2\phi(\mathbf{w}_1^\top \mathbf{x} + b)$. For simplicity of notation and implementation, the bias can be incorporated by prepending b into a new zeroth element of \mathbf{w}_1 and a corresponding 1 into a zeroth element of \mathbf{x} , giving us $y = \mathbf{w}_2\phi(\mathbf{w}_1^\top \mathbf{x})$. We can join neurons in a layer to form a single-layer neural net $\mathbf{y} = \mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x})$ where \mathbf{W}_1 and \mathbf{W}_2 are matrices of synaptic weights. The input and output nodes form input and output layers, and the neurons form a single hidden layer. By feeding one neuron into another, we create a multi-layer neural net called a deep neural net

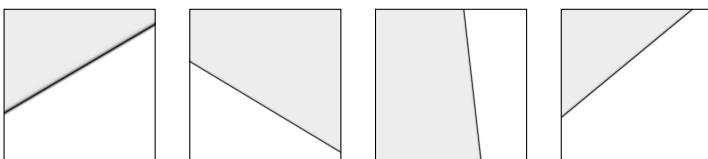
$$\mathbf{y} = \mathbf{W}_k\phi(\mathbf{W}_{k-1}\phi(\cdots\phi(\mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x}))\cdots)).$$

While McCulloch and Pitts forged their neural nets out of the mathematical certainty of formal logic, modern neural nets reign as tools of statistical uncertainty and inference. Deep neural networks are able to classify immense and complicated data sets, thanks in large part to specialized computing hardware and stochastic optimization techniques that allow them work efficiently in high dimensions. To better understand how a deep neural net partitions data, consider a collection of two thousand (x_1, x_2) points, each one labeled either -1 if it lies outside of the unit circle or $+1$ if it lies inside of it.

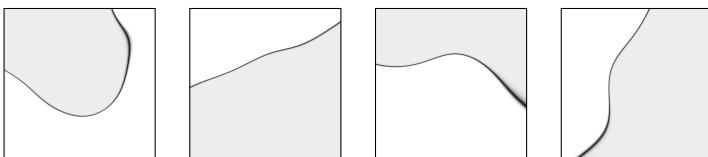


Let's partition the points using the neural net $y = \phi(\mathbf{W}_3\phi(\mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x})))$, with two inputs, two hidden layers each with four neurons, and one output for the label. We'll use $\phi(x) = \tanh x$ as the activation function.

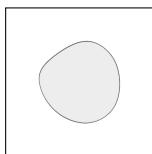
A single neuron partitions the plane with a line, like a crisp fold in a sheet of paper, called a decision boundary.⁹ Each of the four neurons in the first hidden layer produces different partitions in the plane—four separate folds in the paper.



When combined linearly together, these decision boundaries partition a space into a simple, possibly unbound polytope. In a deep neural net, these initial partitions are the building blocks for more complicated geometries with each subsequent layer. The second hidden layer forms curved boundaries by combining the output of the first neural layer using smooth sigmoid and tanh activation functions. The ReLU activation function produces polytopes with flat sides.



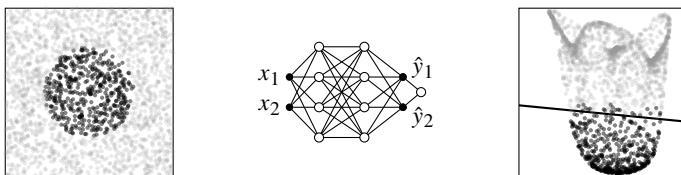
The neural network uses these new shapes to construct the final partition, an almost circular blob that approximates the original data. Had we included more data, used more neurons or more layers, or trained the neural network for a longer period of time, the blob would have been more circular.



⁹Not to be misled by metaphor, the real decision boundary is the result of a smooth sigmoid function, so there is a layer of fuzziness surrounding the line.

Adding more neurons to a layer increases the dimensionality. Just as a four-dimensional space allows us to untie a three-dimensional knot, a wide neural net provides more freedom to move around. Adding more layers increases the depth of recursion. We've seen the power of recursion in dynamical systems and multiresolution analysis. The self-similarity in Mandelbrot sets and Daubechies wavelets comes from recursively feeding output into input. As a loose analogy, folding and unfolding a sheet of paper will produce one crease at a time, but folding that sheet again and again before unfolding it will produce a complex pattern of creases.

The manifold hypothesis provides another heuristic for understanding deep neural nets. The manifold hypothesis posits that natural data lie (or almost lie) on lower-dimensional manifolds embedded in high-dimensional space. Intuitively, correlations that exist in data lead to the implicit organization of that data.¹⁰ Each hidden layer successively stretches and squishes space, warping and untangling the manifolds to make them linearly separable. Think about the original light and dark dots from the figure on the preceding page, reproduced on the left below.



There's no straight line that can partition this data. But, by embedding the data in higher dimensions, we can homeomorphically stretch and fold the space until it can be partitioned with a hyperplane. Imagine that you print the original light and dark dot pattern onto a spandex handkerchief. How might you remove the dark dots with one straight cut using a pair of scissors? Push the middle of the handkerchief up to make a poof, and then cut across that poof. This is what our neural net does. We can visualize the actions of the neural net by examining intermediate outputs (\hat{y}_1, \hat{y}_2) obtained from splitting \mathbf{W}_3 into a product of 1×2 and 2×4 matrices. The rightmost figure above shows how the neural net has stretched, folded, and squashed the data points into the plane, like a used tissue thoughtlessly tossed onto the roadway. The decision boundary can be determined using the 1×2 matrix along with the bias term. Just as the circular blob didn't perfectly partition the original space, the decision boundary doesn't provide a perfect cut between light and dark dots.

¹⁰For example, handwritten digits might be digitized as 28 by 28 pixel images, which are embedded in a 784 dimensional space (one dimension for each pixel). The intrinsic dimension—the minimum degrees of freedom required to describe an object in space—is much smaller the extrinsic dimensionality. A “3” looks like another “3” and a “7” looks like another “7.” Neural networks manipulate low-intrinsic-dimensional data in higher dimensional space.

We'll return to neural networks when we discuss fitting data in the next section. To dig deeper, see Goodfellow, Bengio, and Courville's text *Deep Learning*, Friedman, Hastie, and Tibshirani's text *Elements of Statistical Learning*, the open-source book project *Dive into Deep Learning* (<https://d2l.ai>) developed by several Amazon computer scientists, or Christopher Olah's blog post "Neural Networks, Manifolds, and Topology." Also, check out Daniel Smilkov and Shan Carter's interactive visualization of neural networks at <http://playground.tensorflow.org>.

10.7 Data fitting

We often want to find the "best" function that fits a data set of, say, m input-output values $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$. Suppose that we choose a model function $y = f(x; c_1, c_2, \dots, c_n)$ that has n parameters c_1, c_2, \dots, c_n and that the number of observations m is at least as large as the number of parameters n . With a set of inputs and corresponding outputs, we can try to determine the parameters c_i that fit the model, but the input and output data likely come from observations with noise or hidden variables that overcomplicate our simplified model. So, the problem is inconsistent unless we perhaps (gasp!) overfit the data by changing the model function. Luckily, the problem can be handled nicely using the least squares best approximation.

► Linear least squares approximation

Suppose that our model is linear with respect to the parameters c_1, c_2, \dots, c_n . That is,

$$f(x; c_1, c_2, \dots, c_n) = c_1\varphi_1(x) + c_2\varphi_2(x) + \dots + c_n\varphi_n(x)$$

where $\{\varphi_1, \varphi_2, \dots, \varphi_n\}$ are basis elements that generate a subspace F . To find the best approximation to y in F , we must find the $f \in F$ such that $y - f \perp F$. In this case, we have the normal equation

$$(f, \varphi_i) = \sum_{j=1}^n c_j (\varphi_j, \varphi_i) = (y, \varphi_i) \text{ for } i = 1, 2, \dots, n,$$

where the inner product $(f, \varphi_i) = \sum_{k=1}^m f(x_k)\varphi_i(x_k)$. The matrix form of the normal equation is

$$\begin{bmatrix} (\varphi_1, \varphi_1) & (\varphi_2, \varphi_1) & \dots & (\varphi_n, \varphi_1) \\ (\varphi_1, \varphi_2) & (\varphi_2, \varphi_2) & \dots & (\varphi_n, \varphi_2) \\ \vdots & \vdots & \ddots & \vdots \\ (\varphi_1, \varphi_n) & (\varphi_2, \varphi_n) & \dots & (\varphi_n, \varphi_n) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (y, \varphi_1) \\ (y, \varphi_2) \\ \vdots \\ (y, \varphi_n) \end{bmatrix},$$

which in vector notation is simply $\mathbf{A}^\top \mathbf{Ac} = \mathbf{A}^\top \mathbf{y}$ where \mathbf{A} is an $m \times n$ matrix with elements given by $A_{ij} = \varphi_j(x_i)$. In the solution $\mathbf{c} = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \mathbf{y}$, the term

$(\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top$ is called the *pseudoinverse* of \mathbf{A} and denoted \mathbf{A}^+ . Formally, we have replaced the solution $\mathbf{c} = \mathbf{A}^{-1} \mathbf{y}$ with $\mathbf{c} = \mathbf{A}^+ \mathbf{y}$. In practice, we don't need to explicitly compute the pseudoinverse. Instead, we can solve $\mathbf{A}\mathbf{y} = \mathbf{c}$ using the QR method. For more discussion of linear least squares and the pseudoinverse refer back to Chapter 3.

► Nonlinear least squares approximation

What do we do if our model cannot be expressed as a linear combination of the basis elements of an inner product space? For example, we may want to determine the Gaussian distribution $y = c_1 e^{-(x-c_2)^2/2c_3^2}$ that best matches some given data by determining the amplitude c_1 , the mean c_2 , and the standard deviation c_3 . In general, for the given function f , we want to find the parameters $\mathbf{c} = (c_1, c_2, \dots, c_n)$ such that $y_i = f(x_i; c_1, c_2, \dots, c_n)$ at given the points $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$.

Let's switch the roles of \mathbf{c} and \mathbf{x} as variable and parameter and define $f_i(\mathbf{c}) = f(c_1, c_2, \dots, c_n; x_i)$. This allows us to write the system of residuals as

$$\begin{aligned} r_1 &= y_1 - f_1(c_1, c_2, \dots, c_n) \\ r_2 &= y_2 - f_2(c_1, c_2, \dots, c_n) \\ &\vdots \\ r_m &= y_m - f_m(c_1, c_2, \dots, c_n). \end{aligned}$$

In vector notation, this system is simply $\mathbf{r} = \mathbf{y} - f(\mathbf{c})$. If this system were consistent and $m = n$, we could use Newton's method that we developed in Section 8.6 to find a \mathbf{c} such that $\mathbf{r} = \mathbf{0}$. In such a case, we would iterate

$$\mathbf{J}_f(\mathbf{c}^{(k)}) \Delta \mathbf{c}^{(k+1)} = -\mathbf{r}^{(k)},$$

where $\Delta \mathbf{c}^{(k+1)} = \mathbf{c}^{(k+1)} - \mathbf{c}^{(k)}$ and $\mathbf{J}_f(\mathbf{c})$ is the $m \times n$ Jacobian matrix whose (i, j) -component is $\partial r_i / \partial c_j$. The Jacobian matrix $\mathbf{J}_f(\mathbf{c}) = -\mathbf{J}_f(\mathbf{c})$, so we equivalently have

$$\mathbf{J}_f(\mathbf{c}^{(k)}) \Delta \mathbf{c}^{(k+1)} = \mathbf{r}^{(k)} \quad (10.16)$$

where the (i, j) -component of the Jacobian $\mathbf{J}_f(\mathbf{c})$ is $\partial f_i / \partial c_j$. We can solve this overdetermined system at each iteration using QR factorization and then update $\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + \Delta \mathbf{c}^{(k+1)}$. The method is called the *Gauss–Newton method*.

By multiplying both sides of (10.16) by $\mathbf{J}_f^\top(\mathbf{c}^{(k)})$ and then formally solving for $\mathbf{c}^{(k+1)}$, we get an explicit formulation for the Gauss–Newton method:

$$\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + \mathbf{J}_f^+(\mathbf{c}^{(k)}) \mathbf{r}^{(k)}, \quad (10.17)$$

where the pseudoinverse $\mathbf{J}_f^+ = (\mathbf{J}_f^\top \mathbf{J}_f)^{-1} \mathbf{J}_f^\top$. Because method can be unstable, we can regularizes the pseudoinverse by making the $\mathbf{J}_f^\top \mathbf{J}_f$ term more diagonally

dominant. This has the effect of weakly decoupling the system of equations. The *Levenberg–Marquardt method* is given by

$$\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + (\mathbf{J}^T \mathbf{J} + \lambda \mathbf{D})^{-1} \mathbf{J}^T \mathbf{r}^{(k)}, \quad (10.18)$$

where \mathbf{J} is $\mathbf{J}_f(\mathbf{c}^{(k)})$, the matrix \mathbf{D} is a non-negative diagonal matrix, and the damping parameter λ is positive. Typical choices for \mathbf{D} include the identity matrix \mathbf{I} or the diagonal matrix $\text{diag}(\mathbf{J}^T \mathbf{J})$. The damping parameter is typically changed at each iteration, increasing it if the residual increases and decreasing it if the residual decreases. The Levenberg–Marquardt method is analogous to *Tikhonov regularization* used to solve underdetermined, inconsistent linear systems. As mentioned in section 3.2, we don't need to explicitly construct the pseudoinverse to implement the method—we can use a stacked matrix instead. But the explicit formulation (10.18) is more efficient, especially when the number of data points is significantly larger than the number of parameters.

Example. Suppose that we've been handed some noisy data collected from an experiment, which from theory, we expect to be modeled by two Gaussian functions

$$f(x; \mathbf{c}) = c_1 e^{-c_2(x-c_3)^2} + c_4 e^{-c_5(x-c_6)^2} \quad (10.19)$$

for some unknown parameters $\mathbf{c} = \{c_1, c_2, \dots, c_6\}$. Let's use Newton's method to recover the parameters.

Newton's method is $\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} - (\mathbf{J}^{(k)})^+ (\mathbf{f}(\mathbf{x}; \mathbf{c}^{(k)}) - \mathbf{y})$ where $(\mathbf{J}^{(k)})^+$ is the pseudoinverse of the Jacobian. The numerical approximation to the Jacobian using the complex-step derivative is

```
function jacobian(f,x,c)
    J = zeros(length(x),length(c))
    for k in (n = 1:length(c))
        J[:,k] .= imag(f(x,c+1e-8im*(k .== n)))/1e-8
    end
    return J
end
```

Let's implement the Levenberg–Marquardt method:

```
function gauss_newton(f,x,y,c)
    r = y - f(x,c)
    for j = 1:100
        G = jacobian(f,x,c)
        M = G'*G
        c += (M+Diagonal(M))\ (G'*r)
        norm(r-(r=y-f(x,c))) < 1e-12 && return(c)
    end
    print("Gauss-Newton did not converge.")
```

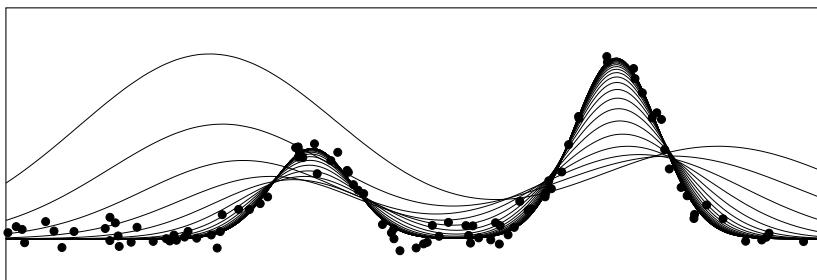


Figure 10.7: The least squares solution for (10.19). Solutions are depicted at intermediate iterations. The input data is represented by •.

```
end
```

For the purposes of this example, we'll take the measured data to be

```
f = (x,c) -> @. c[1]*exp(-c[2]*(x-c[3])^2) +
    c[4]*exp(-c[5]*(x-c[6])^2)
x = 8*rand(100)
y = f([1, 3, 3, 2, 3, 6],x) + 0.1*randn(100);
```

Let's take $\{2, 0.3, 2, 1, 0.3, 7\}$ as the initial guess. The problem is solved using

```
c₀ = [2, 0.3, 2, 1, 0.3, 7]
c = gauss_newton(f,x,y,c₀)
```

The plot above shows the intermediate solutions at each iteration, ending with the parameters $\mathbf{c} = \{0.98, 3.25, 98, 3.00, 1.96, 2.88, 5.99\}$.

Another way to control stability in Newton's method is with a learning rate α . The Newton step $\mathbf{c} += \alpha \mathbf{G} \setminus \mathbf{r}$ with $\alpha=0.5$ could be substituted into the method in place of the Levenberg–Marquardt step. Because the Levenberg–Marquardt method is more stable, we can use a learning rate as large as 2 with it.

In practice, we might use the LsqFit.jl package which solves nonlinear least squares problems:

```
using LsqFit
fit = curve_fit(f,x,y,c₀)
```

The parameters are given by `fit.param`. The residuals are given by `fit.resid`. ◀

• LsqFit.jl uses the Levenberg–Marquardt algorithm to fit a function to data.

► Logistic regression

A *likelihood function* is a probability distribution interpreted as a function of its parameters. By maximizing the likelihood function, we determine parameters that make a distribution most likely to fit observed data. We call such values the maximum likelihood estimate.

Consider an experiment that has a Boolean-valued outcome—pass or fail, yes or no, true or false—in other words, a Bernoulli trial. We can model such an experiment using a Bernoulli distribution $p^y(1-p)^{1-y}$ where $p(\mathbf{x}, \mathbf{w})$ is the probability for some data \mathbf{x} and some parameters \mathbf{w} and $y \in \{0, 1\}$ is the possible outcome. With several trials, we can define a (scaled) likelihood function as

$$L(\mathbf{w}) = \prod_{i=1}^n p^{y_i} (1-p)^{1-y_i}$$

for a probability $p(\mathbf{x}_i, \mathbf{w})$. The log-likelihood function is given by the logarithm of the likelihood:

$$\ell(\mathbf{w}) = \sum_{i=1}^n y_i \log p + (1 - y_i) \log(1 - p). \quad (10.20)$$

Because the logarithm is a monotonically increasing function, the maximum likelihood estimate is the same as the maximum of the log-likelihood. So we need to just find the parameters \mathbf{w} that maximizes $\ell(\mathbf{w})$. To do this, we'll first need a model for the probabilities $p(\mathbf{x}_i, \mathbf{w})$.

A generalized linear model (GLM) is one that relates the expectation of the response $\mu_i = E(y_i)$ to a linear predictor

$$f(\mu_i) = \eta_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_n x_{in} = \mathbf{w}^\top \mathbf{x}_i.$$

The function f is called a *link function*. We want a link function that is smooth, monotonically increasing, and maps the unit interval $(0, 1)$ to the real line $(-\infty, \infty)$. The link function most commonly associated with the Bernoulli distribution is the logit function

$$\text{logit } \mu = \log \frac{\mu}{1 - \mu}.$$

While there are other link functions,¹¹ the logit function is easy to manipulate, it has the nice interpretation of being the log odds ratio, and it is the inverse of

¹¹ Statistician and physician Joseph Berkson introduced the logit function in 1944 as a close approximation to the inverse cumulative normal distribution used ten years earlier by biologist Chester Bliss. Bliss called his curve the probit, a portmanteau of probability and unit, and Berkson followed analogously with logit, a portmanteau of logistical and unit.

the logistic function $\sigma(\eta) = 1/(1 + e^{-\eta})$. Using the inverse logit (the logistic function) in the log-likelihood function (10.20) yields

$$\ell(\mathbf{w}) = \sum_{i=1}^n y_i \log \sigma(\mathbf{w}^\top \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i)).$$

Now, we can calculate the maximum likelihood estimate. The derivative of the logistic function $\sigma(\eta)$ is $\sigma(\eta)(1 - \sigma(\eta))$, from which it follows that the derivative of $y \log \sigma(\eta) + (1 - y) \log(1 - \sigma(\eta))$ is the remarkably simple expression $y - \sigma(\eta)$. Define \mathbf{X} as the matrix whose i th row is \mathbf{x}_i^\top . The contribution from \mathbf{x}_i to the gradient of $\ell(\mathbf{w})$ is

$$\mathbf{g}_i = (y_i - \sigma(\mathbf{w}^\top \mathbf{x}_i)) \mathbf{x}_i.$$

The corresponding gradient using all data points is

$$\mathbf{g} = \mathbf{X}^\top (\mathbf{y} - \sigma(\mathbf{X}\mathbf{w})).$$

The contribution from \mathbf{x}_i to the Hessian is

$$\mathbf{H}_i = \mathbf{x}_i^\top \sigma(\mathbf{w}^\top \mathbf{x}_i) (1 - \sigma(\mathbf{w}^\top \mathbf{x}_i)) \mathbf{x}_i.$$

After summing over all data points, the Hessian is $\mathbf{H} = \mathbf{X}^\top \mathbf{S} \mathbf{X}$, with the diagonal matrix

$$\mathbf{S} = \sigma(\mathbf{X}\mathbf{w}^{(k)}) (1 - \sigma(\mathbf{X}\mathbf{w}^{(k)})).$$

Therefore, following the discussion in section 8.8, Newton's method is

$$\begin{aligned} \mathbf{w}^{(k+1)} &= \mathbf{w}^{(k)} - (\mathbf{H}^{(k)})^{-1} \mathbf{g}^{(k)} \\ &= \mathbf{w}^{(k)} + (\mathbf{X}^\top \mathbf{S}^{(k)} \mathbf{X})^{-1} \mathbf{X}^\top (\mathbf{y} - \sigma(\mathbf{w}^\top \mathbf{X}^{(k)})). \end{aligned}$$

Often, the terms on the right-hand side are regrouped as a formula called *iteratively reweighted least squares* (IRLS):

$$\mathbf{w}^{(k+1)} = (\mathbf{X}^\top \mathbf{S}^{(k)} \mathbf{X})^{-1} \mathbf{X}^\top \left(\mathbf{S}^{(k)} \mathbf{X} \mathbf{w}^{(k)} + \mathbf{y} - \sigma(\mathbf{w}^\top \mathbf{X}^{(k)}) \right).$$

Let's see what the method looks like in Julia. We'll first define the logistic function and generate some synthetic data.

```
σ = x -> @. 1/(1+exp(-x))
x = rand(30); y = ( rand(30) .< σ(10x.-.5) );
```

Now, we apply Newton's method.

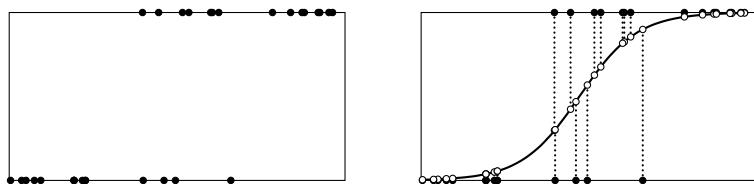


Figure 10.8: Classification data (left). The function fitting the data (right).

```
X, w = [one.(x) x], zeros(2,1)
for i=1:10
    S = σ(X*w).* (1 .- σ(X*w))
    w += (X'*(S.*X))\ (X'*(y - σ(X*w)))
end
```

A more than minimal working example would set a condition to break out the loop once the maximum absolute change in w is less than some threshold. The solution is plotted in Figure 10.8 above. In practice, we might use Julia's GLM library to solve the logistic regression problem.

```
using GLM, DataFrames
data = DataFrame(X=x, Y=y)
logit = glm(@formula(Y ~ X), data, Bernoulli(), LogitLink())
```

• The GLM.jl library fits a generalized linear model to provided data.

► Neural networks

Let's pick up from the previous section and examine how to solve a data fitting problem using neural networks. Consider a two-layer neural net $\mathbf{y} = \mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x}_*)$ for some data \mathbf{x}_* and \mathbf{y}_* and some unknown matrices of parameters \mathbf{W}_1 and \mathbf{W}_2 . The vector \mathbf{x} has m elements— $m - 1$ of these elements contain the input variables (called features) and one element, set to 1, is for the bias. The vector \mathbf{y} is has n dimensions for the labels. The parameters \mathbf{W}_1 and \mathbf{W}_2 are $k \times m$ and $n \times k$ matrices, where k is a hyperparameter of the algorithm, independent of the problem. All-in-all, we need to determine kmn parameters, although many of these parameters may be zero.

We'll determine the parameters by finding a \mathbf{y} that minimizes a loss function, also known as a cost function. A typical loss function is the mean squared error—the normalized ℓ^2 -error—of the residual

$$L(\mathbf{y}) = N^{-1} \|\mathbf{y} - \mathbf{y}_*\|_2^2,$$

which is minimized when its gradient $[\partial L/\partial \mathbf{y}] = 2N^{-1}(\mathbf{y} - \mathbf{y}_*)$ is zero. The notation $[\cdot]$ is used to help emphasize that $\partial L/\partial \mathbf{y}$ is a matrix or a vector.

We can solve the minimization problem using the gradient descent method, which iteratively marches values along the local gradient some distance and then reevaluates the direction. Start with some random initial guess for the parameters \mathbf{W}_1 and \mathbf{W}_2 . Use these values to compute $\mathbf{y}(\mathbf{x}_*) = \mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x}_*)$. Then use \mathbf{y} to determine the loss and the gradient of the loss. Update the values of the parameters

$$\begin{aligned}\mathbf{W}_1 &\leftarrow \mathbf{W}_1 - \alpha_1 [\partial L/\partial \mathbf{W}_1] \\ \mathbf{W}_2 &\leftarrow \mathbf{W}_2 - \alpha_2 [\partial L/\partial \mathbf{W}_2].\end{aligned}$$

The constants α_1 and α_2 , called learning rates, are chosen to ensure stability of the iterative method. Iterate until the method converges to a local minimum. Hopefully, that local minimum is also a global minimum.

We'll use two identities for matrix derivatives in computing the gradients. If $f(\mathbf{C}) = f(\mathbf{AB})$, then

$$[\partial f/\partial \mathbf{A}] = [\partial f/\partial \mathbf{C}] [\mathbf{B}]^T \text{ and } [\partial f/\partial \mathbf{B}] = [\mathbf{A}]^T [\partial f/\partial \mathbf{C}],$$

If $\mathbf{C} = g(\mathbf{A})$, then we the chain rule for $f(g(\mathbf{A}))$ is

$$[\partial f/\partial \mathbf{A}] = [g'(\mathbf{A})]^T [\partial f/\partial \mathbf{C}],$$

where $[g'(\mathbf{A})]$ is a Jacobian matrix. So,

$$[\partial L/\partial \mathbf{W}_2] = [\partial L/\partial \mathbf{y}] [\phi(\mathbf{W}_1\mathbf{x}_*)]^T$$

and

$$[\partial L/\partial \mathbf{W}_1] = [\mathbf{W}_2\phi'(\mathbf{W}_1\mathbf{x}_*)]^T [\partial L/\partial \mathbf{y}] [\mathbf{x}_*]^T.$$

The Jacobian matrix $[\phi']$ is a diagonal matrix. In practice, we'll use a training set consisting of N data points \mathbf{x}_* and \mathbf{y}_* . We can treat \mathbf{x}_* as an $m \times N$ matrix and $\mathbf{y}(\mathbf{x}_*)$ and \mathbf{y}_* both as $n \times N$ matrices.

Example. Let's use neural networks to model the data given by the noisy semicircle $y = \sqrt{1 - x^2} + \sigma(x)$, where $\sigma(x)$ is some random perturbation.¹² Let's pick 100 points.

¹²A neural network on a one-dimensional problem with a small number parameters is wholly inefficient. Any number of universal approximators like Chebyshev polynomials or B-splines would provide a better solution in a fraction of the compute time. However, these methods quickly get bogged down as the size of the problem increases. We'll return to the curse of dimensionality when we discuss Monte Carlo methods in the next chapter.

```
N = 100; θ = LinRange(0,π,N)'
x = cos.(θ); ī = [one.(x);x]
y = sin.(θ) + 0.05*randn(1,N)
```

We'll select a neural network with one hidden layer with n neurons, an input layer with 2 nodes (one for the bias), and one output node. We'll also use a ReLU activation function.

```
n = 20; W1 = rand(n,2); W2 = randn(1,n)
ϕ = x -> max.(x,0)
dϕ = x -> (x.>0)
```

We solve the problem using gradient descent with a sufficiently large number of iterations, called epochs. To ensure stability, we take a learning rate $\alpha = 0.1$.

```
α = 0.1
for epoch = 1:1000
    ŷ = W2 * ϕ(W1*ī)
    ∂L∂y = 2(ŷ-y)/N
    ∂L∂W1 = dϕ(W1*ī) .* (W2' * ∂L∂y) * ī'
    ∂L∂W2 = ∂L∂y * ϕ(W1*ī)'
    W1 -= 0.1α * ∂L∂W1
    W2 -= α * ∂L∂W2
end
```

Now that we have our trained parameters \mathbf{W}_1 and \mathbf{W}_2 , we construct the solution using our neural network model.

```
scatter(ī[2,:],y',opacity=0.3)
ī = LinRange(-1.2,1.2,200)'; ī = [one.(ī);ī]; ŷ = W2 * ϕ(W1*ī)
plot!(ī[2,:],ŷ',width=3)
```

The solution is plotted in the figure on the next page and at the QR link at the bottom of this page. We can also solve the same problem using a sigmoid activation function

```
ϕ = x -> @. 1 / (1 + exp(-x))
dϕ = x -> @. ϕ(x)*(1-ϕ(x))
```

along with alternative loss function $L(\mathbf{y}) = \|\mathbf{y} - \mathbf{y}_*\|_2$. The gradient is of this loss function is $[\partial L/\partial \mathbf{y}] = 2(\mathbf{y} - \mathbf{y}_*)/L(\mathbf{y})$.

```
L = norm(ŷ - y)
∂L∂y = (ŷ-y)/L
```

In practice, one uses a machine learning package. Let's use Flux.jl to solve the same problem. The recipe is as before: build a model with initially unknown



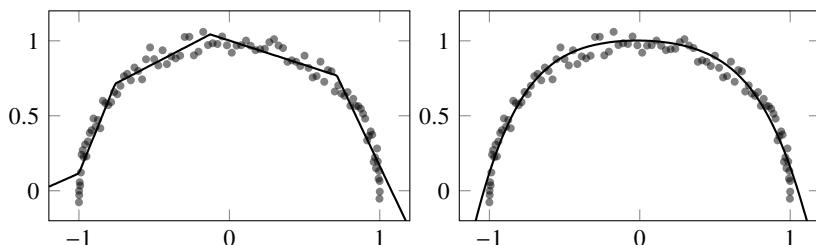


Figure 10.9: The neural network approximation for the data • using ReLU activation functions (left) and sigmoid activation functions (right).

parameters, prescribe a loss function, add some data, choose an optimizer such as gradient descent, and then iterate.

```
using Flux
model = Chain(Dense(1, n, relu), Dense(n, 1))
loss(x,y) = Flux.Losses.mse(model(x), y)
parameters = params(model)
data = [(x,y)]
optimizer = Descent(0.1)
for epochs = 1:10000
    Flux.train!(loss, parameters, data, optimizer)
end
```

The variable `parameters` is an array of four things: the weights and bias terms of \mathbf{W}_1 and the weights and a bias terms of \mathbf{W}_2 . The object `model` is the trained model, which we can now use as a function `model(x)` for an arbitrary input `x`.

Using neural network on a one-dimensional problem that has twenty parameters might seem like a. Yes, any number of universal approximators such as Chebyshev polynomials or B-splines will provide much better solution in a fraction of the the compute time. However, these methods can quickly get bogged down as the number of dimensions increases. We'll return to the curse of dimensionality when we discuss Monte Carlo methods in the next chapter. ◀

 Flux.jl is a deep learning ecosystem built entirely in Julia.

Mathematician Hannah Fry aptly describes a neural network as “an enormous mathematical structure that features a great many knobs and dials” that must be tweaked and tuned so as not to be a “complete pile of junk.” Tweaking and tuning all these knobs and dials is no easy task. Even relatively simple neural nets used for handwritten digit classification can have tens of thousand parameters. One of the largest deep neural nets, GPT-3 (Generative Pre-trained Transformer 3) used

for natural language generation, has 96 layers with over 175 billion parameters. On top of that, gradient descent is slow, it can stall on any number of local minimums and saddle points, and the problem is increasingly ill conditioned as the number of parameters rises. Fortunately, there are several approaches to these challenges.

We can drastically reduce the number of parameters using convolutional layers and pooling layers, rather than using fully connected layers. Convolutional neural nets (CNNs), frequently used in image recognition, use convolution matrices with relatively small kernels. Such kernels—common in image processing as edge detection, sharpen, blur, and emboss filters—consist of tunable parameters for novel pattern filtering. Pooling layers reduce the dimensionality by taking a local mean or maximum of the input parameters.

Significant research has led to the development of specialized model architectures. Residual networks (ResNets), used in image processing, allow certain connections to skip over intermediate layers. Recurrent neural networks (RNNs), used in text processing or speech recognition, use sequential data as updates to intermediate layers. Recent research has explored implicitly defined layers and neural differential equations.

Another way to speed up model training is by starting with a better initial guess of the parameters than simply choosing a random initial state. Transfer learning uses similar, previously trained models to start the parameters closer to a solution. Alternatively, we can embrace randomness. Machine learning often requires massive amounts of data. Stochastic gradient descent uses several batches of smaller, randomly selected subsets of the training data.

We can modify gradient descent to get around its shortcomings such as adding momentum to overcome shallow local minimum and saddle points or using adaptive learning rates such as AdaGrad and Adam to speed up convergence. Finally, to speed up the computation of gradients and avoid numerical stability, we can use automatic differentiation, which we discuss in the next chapter.

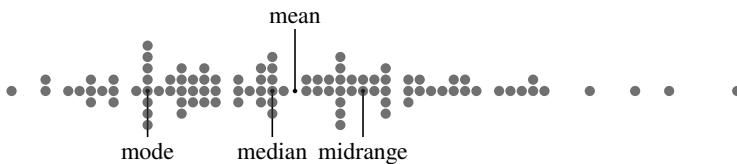
► Statistical best estimates

While the focus of this chapter has been finding on the “best” estimate in the ℓ^2 -norm, for completeness let’s conclude with a comment on the “best” estimate of sample data in a general ℓ^p -norm. Suppose that we have a set of data $\mathbf{x} = (x_1, x_2, \dots, x_n)$ consisting of real values. And, suppose that we’d like to find a value \bar{x} that is the “best” estimate of that data set. In this case, we find \bar{x} that is closest to each element of \mathbf{x} in the ℓ^p -norm

$$\bar{x} = \arg \min_s \|\mathbf{x} - s\|_p = \arg \min_s \left(\sum_{i=1}^n |x_i - s|^p \right)^{1/p}$$

where p ranges from zero to infinity. If we further define 0^0 to be 0, then we can strictly include $p = 0$. The ℓ^0 -“norm” is simply the count of nonzero elements in

the set.¹³ Quotation marks are used to emphasize that the ℓ^0 -“norm” is not truly a norm because it lacks homogeneity: $\|\alpha \cdot \mathbf{x}\|_0 \neq \alpha \cdot \|\mathbf{x}\|_0$. The value \bar{x} closest to \mathbf{x} in ℓ^0 , ℓ^1 , ℓ^2 , and ℓ^∞ -norms is one of the four statistical averages—the mode, median, mean, and midrange, respectively. The mode is the value which occurs most frequently in the data set regardless of its relative magnitude. The midrange is the midpoint of the two extrema of the set, regardless of any other elements of the set. For example, consider the following beeswarm plot of data set sampled from a Rayleigh distribution:



Notice the locations of the mode, median, mean, and midrange relative to the data. How well does each do in summarizing the distribution of the data? Where might they located be on other data? What are the potential biases that arise from using a specific average?

10.8 Exercises

10.1. We can think about orthogonality in geometric terms by using cosine to define the angle between two functions

$$\cos \theta = \frac{(f, g)}{\|f\| \|g\|}$$

and even between two subspaces

$$\cos \theta = \sup_{f \in F, g \in G} \frac{(f, g)}{\|f\| \|g\|}.$$

- (a) Find the closest quadratic polynomial to x^3 over $(0, 1)$ with the L^2 -inner product

$$(f, g) = \int_0^1 f(x)g(x) dx.$$

- (b) Use the definition of the angle between two subspaces to show that when n is large, x^{n+1} is close to the subspace spanned by $\{1, x, x^2, \dots, x^n\}$.

¹³Not to be confused with the generalized mean $n^{-1/p} \|\mathbf{x}\|_p$, which in the limit as $p \rightarrow 0$ is the geometric mean of \mathbf{x} .

10.2. Show that the coefficients in the three-term recurrence relation of the Legendre polynomials are given by a_n and $b_n = -n^2/(4n^2 - 1)$.

10.3. When we learn arithmetic, we are first taught integer values. Only later are we introduced to fractions and then irrational numbers which fill out the number line. Learning calculus is similar. We start by computing with whole derivatives and rules of working with them such as

$$\frac{d}{dx} \frac{d}{dx} f(x) = \frac{d^2}{dx^2} f(x).$$

In advanced calculus, we may be introduced to fractional derivatives. For example,

$$\frac{d^{1/2}}{dx^{1/2}} \frac{d^{1/2}}{dx^{1/2}} f(x) = \frac{d}{dx} f(x)$$

One way to compute a fractional derivative is with the help of a Fourier transform. If $\hat{f}(k)$ is the Fourier transform of $f(x)$, then $(ik)^p \hat{f}(k)$ is Fourier transform of the p th derivative of $f(x)$. Compute the fractional derivatives of the sine, piecewise-quadratic, piecewise-linear, and Gaussian functions:

- $\sin x$ for $x \in [-\pi, \pi]$
- $-\frac{1}{4} \operatorname{sign}(x) \cdot ((2|x| - 1)^2 - 1)$ for $x \in [-1, 1]$
- $\frac{1}{2} - |x|$ for $x \in [-1, 1]$
- e^{-6x^2} for $x \in [-2, 2]$

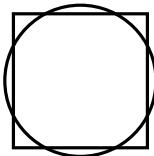
The function e^{-6x^2} is not periodic, but as long as the domain is sufficiently large, it can be approximated as being periodic across the boundaries. Similarly, the piecewise-linear and piecewise-quadratic functions are not smooth, but as long as the domain has sufficient resolution to account for the slow decay of the Fourier transform, we minimize the effects of aliasing. Repeat the exercise using a Chebyshev differentiation matrix.

10.4. In 1998, Yann LeCun and fellow researchers published one of the earliest examples of a convolutional neural network, called LeNet-5, in their article “Gradient-based learning applied to document recognition.” LeNet-5, which consists of five trainable parameter layers with over 60 thousand parameters combined, was designed to classify handwritten digits. Modern CNN architecture has evolved since LeNet-5 debuted almost 25 years ago—ReLU activation functions have displaced sigmoid activation functions, max pooling or even no pooling is typically used instead average pooling, depth can reach into the hundreds of layers rather than a handful of them, and GPUs have largely replaced CPUs. This exercise examines the classic LeNet-5, sometimes described as the “Hello, World” program of deep neural nets. Read the LeCun article and

implement LeNet-5 to classify the digits in the MNIST dataset. The dataset is available through the `MLDatasets.jl` package.

Chapter 11

Differentiation and Integration



Numerical approximations to derivatives and integrals appear throughout scientific computing. For example, root-finding methods typically require a derivative. Backpropagation, a necessary step in training neural networks, uses gradients. And collocation computes high-order polynomial approximations of derivative and integral operators. This chapter rounds out many of the topics from other chapters by examining numerical and automatic differentiation as well as Newton-Cotes, Gaussian, and Monte Carlo integration.

11.1 Numerical differentiation

In the previous chapter, we discussed numerical differentiation using Fourier and Chebyshev polynomials. Now, let's turn to numerical differentiation using Taylor polynomials.

► Taylor polynomial approach

Suppose that we want to find the derivative of some sufficiently smooth function f at x . Let h be a small displacement. From Taylor series expansion, we have

$$\begin{aligned} f(x + h) &= f(x) + hf'(x) + h^2 f''(x) + O(h^3) \\ f(x) &= f(x) \end{aligned}$$

Taking the difference of these equations gives us

$$f(x + h) - f(x) = hf'(x) + h^2 f''(x) + O(h^3),$$

from which we have the approximation

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

with an error of $hf''(x) + O(h^2)$. The error term says that the approximation is first order. To get higher-order methods, we add more nodes. With three equally spaced nodes:

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + O(h^3)$$

$$f(x) = f(x)$$

$$f(x-h) = f(x) - hf'(x) + \frac{1}{2}h^2f''(x) + O(h^3).$$

We want to eliminate as many terms that could contribute to error. Consider the unknowns a, b, c

$$af(x+h) = af(x) + ahf'(x) + a\frac{1}{2}h^2f''(x) + a\frac{1}{6}h^3f'''(x) + O(h^4)$$

$$bf(x) = bf(x)$$

$$cf(x-h) = cf(x) - chf'(x) + c\frac{1}{2}h^2f''(x) - c\frac{1}{6}h^3f'''(x) + O(h^4).$$

To ensure consistency, we must eliminate the $f(x)$ terms and keep the $f'(x)$ terms on the right-hand side. To get a second-order method, we must further eliminate the $f''(x)$ terms. So, we have the system of equations

$$a + b + c = 0$$

$$a - c = 1$$

$$\frac{1}{2}a + \frac{1}{2}c = 0$$

The solution to the system $a = \frac{1}{2}$, $b = 0$, and $c = -\frac{1}{2}$ gives us

$$\frac{1}{2}f(x+h) - \frac{1}{2}f(x-h) = hf'(x) + \frac{1}{6}h^3f'''(x) + O(h^4)$$

from which we have

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

with an error of $\frac{1}{6}h^2f'''(x) + O(h^3)$.

Of course, we can extend this idea to four nodes to get a third-order method, five nodes to get a fourth-order method, and so on. And we could use non-equally spaced nodes. Let's use the Vandermonde matrix to automate the process of computing the coefficients. The system of Taylor polynomial approximations is

$$f(x + \delta_i h) = \sum_{j=0}^{n-1} c_{ij} \frac{1}{j!} (\delta_i h)^j f^{(j)}(x) + O(h^n),$$

where δ_i scales the step size h , can be written in matrix form

$$[f(x + \delta_i h)] = \mathbf{V} \mathbf{C} [h^j f^{(j)}(x)] + O(h^{n+1}).$$

where the matrix \mathbf{V} is the scaled Vandermonde matrix

$$\mathbf{V} = \begin{bmatrix} 1 & \delta_0 & \cdots & \delta_0^{n-1} \\ 1 & \delta_1 & \cdots & \delta_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \delta_{n-1} & \cdots & \delta_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} 1/0! & & & \\ & 1/1! & & \\ & & \ddots & \\ & & & 1/(n-1)! \end{bmatrix}$$

and $\mathbf{C} = [c_{ij}]$ is the matrix of undetermined coefficients. To determine the coefficient matrix \mathbf{C} , we simply need to calculate \mathbf{V}^{-1} . The first row of \mathbf{C} is the set of coefficients $\{c_{0i}\}$ for a $O(h^n)$ approximation of $f(x)$, the second row is the set of coefficients $\{c_{1i}\}$ for an $O(h^{n-1})$ approximation of $f'(x)$, and so on. The coefficients of the truncation error is simply $\mathbf{C} [\delta_i^n] / n!$.

Example. Let's compute the third-order approximation to $f'(x)$ using nodes at $x - h, x, x + h$ and $x + 2h$:

```
d = [-1, 0, 1, 2]
n = length(d)
V = d.^(@(0:n-1)' .// factorial.(0:n-1)'
C = inv(V)
```

The coefficients of the truncation error are $C * d.^n ./ factorial(n)$:

$$\begin{array}{rrrrr} 0 & 1 & 0 & 0 & 0 \\ -1/3 & -1/2 & 1 & -1/6 & -1/12 \\ 1 & -2 & 1 & 0 & 1/12 \\ -1 & 3 & -3 & 1 & 1/2 \end{array}$$

From the second row we have

$$f'(x) = \frac{-\frac{1}{3}f(x-h) - \frac{1}{2}f(x) + f(x+h) - \frac{1}{6}f(x+2h)}{h} + O(\frac{1}{12}h^3). \quad \blacktriangleleft$$

• Julia has a rational number type that expresses the ratio of integers. For example, $3//4$ is the Julia representation for $\frac{3}{4}$. You can also convert a float 0.75 to a rational using `Rational(0.75)`.

► Richardson extrapolation

To improve the accuracy of the polynomial interpolant, we simply need to add more points by subdividing the interval into smaller and smaller subintervals. Rather than restarting with a whole new set of nodes and recomputing the approximation to derivative, let's come up with a way to add nodes that updates the approximation from the previous nodes. We'll use a method called *Richardson extrapolation*. Suppose that we have any expansion

$$\phi(h) = a_0 + a_1 h^p + a_2 h^{2p} + a_3 h^{3p} + \dots \quad (11.1)$$

for which we want to determine the value a_0 in terms of function on the left-hand side of the equation. For example, we could have the central difference approximation

$$\phi(h) = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{1}{3!} f''(x)h^2 + \frac{1}{3!} f^{(5)}(x)h^4 + \dots .$$

Richardson extrapolation allows us to successively eliminate the h^p , h^{2p} , and higher-order terms. Take

$$\phi(\delta h) = a_0 + a_1 \delta^p h^p + a_2 \delta^{2p} h^{2p} + a_3 \delta^{3p} h^{3p} + \dots .$$

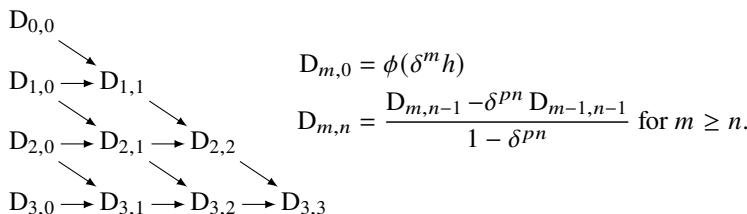
where $0 < \delta < 1$. Typically, $\delta = \frac{1}{2}$. Then

$$\phi(\delta h) - \delta^p \phi(h) = a_0(1 - \delta^p) + 0 + a_2(\delta^{2p} - \delta^p)h^2 + \dots ,$$

from which

$$\frac{\phi(\delta h) - \delta^p \phi(h)}{1 - \delta^p} = a_0 + \tilde{a}_1 h^{2p} + \tilde{a}_2 h^{3p} + \tilde{a}_3 h^{4p} + \dots$$

for some new coefficients $\tilde{a}_1, \tilde{a}_2, \dots$. This new equation has the same form as the original equation (11.1), so we can repeat the process to eliminate the h^{2p} term. This gives us a recursion formula for $a_0 = D_{m,n} + o((\delta^m h)^{pn})$ where



Julia code for Richardson extrapolation taking $\delta = \frac{1}{2}$ is

```

function richardson(f,x,m,n=m)
    n > 0 ?
        (4^n*richardson(f,x,m,n-1) - richardson(f,x,m-1,n-1))/(4^(n-1)) :
        φ(f,x,2^m)
end

```

where the finite difference operator is

$$\phi = (f, x, n) \rightarrow (f(x+1/n) - f(x-1/n))/(2/n)$$

We can compute the derivative of $\sin x$ at zero as `richardson(x->sin(x), 0, 4)`, resulting in an error of 2.4×10^{-14} .

11.2 Automatic differentiation

There are three common methods of calculating the derivative using a computer: numerical differentiation, symbolic differentiation, and automatic differentiation. Numerical differentiation, which is discussed extensively throughout this book, evaluates the derivative of a function by calculating finite difference approximations using that function or by computing the actual derivative of a simpler approximation of the original function. Because numerical differentiation takes differences of approximate values or approximate functions, it is affected by truncation error and round-off error. Symbolic differentiation applies rules of differential calculus—the product rule, the power rule, the chain rule, the sine rule, and so on—to manipulate mathematical expressions of symbolic variables. It expands and then simplifies the derivative of that expression in terms of the symbolic variable. By explicitly manipulating the terms, symbolic differentiation produces the closed-form expression of the derivative without numerical approximation error. While such an approach works well for simple expressions, computation time and memory often grow exponentially as a function of the expression size. Automatic differentiation or autodiff shares features of both numerical and symbolic differentiation. Like symbolic differentiation, automatic differentiation uses explicit rules of calculus to evaluate a derivative of a mathematical expression. Like numerical differentiation, it operates on numerical variables instead of symbolic ones.

Every function evaluated by a computer is a composition of a few basic operations: addition and subtraction, multiplication, exponentiation, some elementary functions, and looping and branching. In Julia even common functions like `sin` and `exp` are computed using high-order Taylor polynomials. Because operations can be represented using a few basic functions, we can develop an arithmetic that differentiates those operators and links them together using the chain rule.

Let's start with a short excursion into nonstandard analysis. A dual number is a number of the form $a + b\epsilon$ where $\epsilon^2 = 0$. Dual numbers can be thought of as

the adjoining of a real or complex number a with an infinitesimal number $b\varepsilon$. Similar to how a complex number $a + bi$ can be represented as an ordered pair of real numbers (a, b) , for which multiplication and other operations are defined in a special way, a dual number $a + b\varepsilon$ can be represented as the ordered pair (a, b) . Let's first consider Taylor series expansion:

$$f(x + \varepsilon) = f(x) + \varepsilon f'(x) + \frac{1}{2}\varepsilon^2 f''(x) + \dots = f(x) + \varepsilon f'(x) \equiv (f, f'),$$

where all ε^2 and higher terms are zero. This expression is the formal infinitesimal definition of the derivative using nonstandard analysis

$$f'(x) = \frac{f(x + \varepsilon) - f(x)}{\varepsilon}.$$

Note that the derivative is not defined as the limit as a real number ε goes to zero. Instead, it is defined entirely with respect to an infinitesimal ε . Dual numbers follow the same rules of addition and multiplication with the additional rule that $\varepsilon^2 = 0$. For example, for addition we have

$$\begin{aligned} (f, f') + (g, g') &\equiv (f + f'\varepsilon) + (g + g'\varepsilon) = (f + g) + (f' + g')\varepsilon \\ &\equiv (f + g, f' + g'), \end{aligned}$$

for multiplication

$$\begin{aligned} (f, f') \cdot (g, g') &\equiv (f + f'\varepsilon) \cdot (g + g'\varepsilon) = fg + (fg' + g'f)\varepsilon \\ &\equiv (fg, fg' + g'f), \end{aligned}$$

and for division

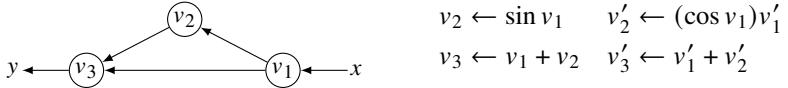
$$\begin{aligned} \frac{(f, f')}{(g, g')} &\equiv \frac{f + f'\varepsilon}{g + g'\varepsilon} = \frac{(f + f'\varepsilon)}{(g + g'\varepsilon)} \frac{(g - g'\varepsilon)}{(g - g'\varepsilon)} = \frac{fg + (f'g - g'f)\varepsilon}{g^2} \\ &\equiv \left(\frac{f}{g}, \frac{f'g - g'f}{g^2} \right), \end{aligned}$$

giving us the familiar addition, product, and quotient rules of calculus. The values of the derivatives can be computed at the same time as values of the functions are evaluated with only a few additional operations. When implemented in a programming language such as Julia or Matlab, each function can be overloaded to return not only the value of the function but also its derivative. For the chain rule

$$\begin{aligned} (f \circ g, (f \circ g)') &\equiv f(g(x) + \varepsilon g'(x)) = f(g(x)) + \varepsilon g'(x)f'(g(x)) \\ &\equiv (f \circ g, f'g'). \end{aligned}$$

When applying the chain rule to several functions $f \circ g \circ h$, we can either compute from the inside out $f \circ (g \circ h)$, a process called forward accumulation, or we can compute from the outside in by taking $(f \circ g) \circ h$, a process called reverse accumulation.

Example. Consider the function $y = x + \sin x$ with the computational graph:



We can build a minimal working example of forward accumulation automatic differentiation by defining a structure and overloading the base operators:

```
struct Dual
    value
    deriv
end
```

Let's define a few helper functions:

```
Dual(x) = Dual(x, 1)
value(x) = x isa Dual ? x.value : x
deriv(x) = x isa Dual ? x.deriv : 0
```

Now, we'll overload some base functions:

```
Base.:+(u, v) = Dual(value(u)+value(v), deriv(u)+deriv(v))
Base.:-(u, v) = Dual(value(u)-value(v), deriv(u)-deriv(v))
Base.:*(u, v) = Dual(value(u)*value(v),
                      value(u)*deriv(v)+value(v)*deriv(u))
Base.:sin(u) = Dual(sin(value(u)), cos(value(u))*deriv(u))
```

We'll define $x = 0$ as a dual number and y as the function of that dual number:

```
x = Dual(0)
y = x + sin(x)
```

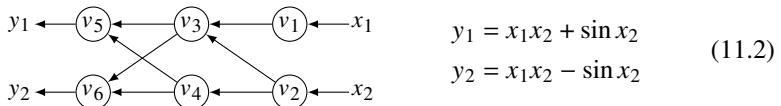
which returns $\text{Dual}(0.0, 2.0)$ as expected. ◀

For a system $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ we have the dual form

$$(f(\mathbf{x}), f'(\mathbf{x})) = f(\mathbf{x} + \boldsymbol{\varepsilon}) = f(\mathbf{x}) + \mathbf{J}_f(\mathbf{x})\boldsymbol{\varepsilon} = (f(\mathbf{x}), \mathbf{J}_f(\mathbf{x})).$$

where $\mathbf{J}_f(\mathbf{x})$ is the $m \times n$ Jacobian matrix evaluated at \mathbf{x} . In constructing the Jacobian matrix, we have a choice between using forward accumulation and reverse accumulation. We'll examine each in the following example.

Example. Consider the system of equations and its accompanying computational graph



variables	v		v'			
(v_1, v'_1)	x_1	2	x'_1		1	0
(v_2, v'_2)	x_2	π	x'_2		0	1
(v_3, v'_3)	$v_1 v_2$	2π	$D_{31}v'_1 + D_{32}v'_2 \rightarrow v_2 v'_1 + v_1 v'_2$	π	2	
(v_4, v'_4)	$\sin v_2$	0	$D_{42}v'_2 \rightarrow v'_2 \cos v_2$	0	-1	
(v_5, v'_5)	$v_4 + v_3$	2π	$D_{53}v'_3 + D_{54}v'_4 \rightarrow v'_3 + v'_4$	π	1	
(v_6, v'_6)	$v_4 - v_3$	2π	$D_{63}v'_3 + D_{64}v'_4 \rightarrow v'_3 - v'_4$	π	3	

Figure 11.1: Forward accumulation automatic differentiation of the system (11.2) evaluated at $(x_1, x_2) = (2, \pi)$.

We can express the derivatives as

$$\begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \begin{bmatrix} v'_5 \\ v'_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} v'_3 \\ v'_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} v_2 & v_1 \\ 0 & \cos v_2 \end{bmatrix} \begin{bmatrix} v'_1 \\ v'_2 \end{bmatrix}$$

or equivalently

$$\mathbf{y}' = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} v_2 & v_1 \\ 0 & \cos v_2 \end{bmatrix} \mathbf{x}' = \mathbf{J}\mathbf{x}'.$$

Using a standard unit vector ξ_i as our input for \mathbf{x}' will return a column of the Jacobian matrix evaluated at \mathbf{x} . In general, for m input variables, we'll need to compute m forward accumulation passes to compute the m columns. Because there are two input variables x_1 and x_2 in the system (11.2), we'll need to do two sweeps. We first start with $(x'_1, x'_2) = (1, 0)$ to get the first column of the Jacobian matrix. We next start with $(x'_1, x'_2) = (0, 1)$ to get the second column of the Jacobian matrix. Figure 11.1 shows each node (v_i, v'_i) of the computational graph starting with $(x_1, x_2) = (2, \pi)$. The term D_{ij} represents the partial derivative $\partial v_i / \partial v_j$.

Let's use the code developed in the previous example on system (11.2). We can define the dual numbers as:

```

x1 = Dual(2,[1 0])
x2 = Dual(pi,[0 1])
y1 = x1*x2 + sin(x2)
y2 = x1*x2 - sin(x2)

```

The variables $y_1.value$, $y_2.value$, $y_1.deriv$, $y_2.deriv$ returns 6.2832, 6.2832, [3.1416 1.0000], and [3.1416 3.0000] as expected.

Now let's examine reverse accumulation. Reverse accumulation computes the transpose (or adjoint) of the Jacobian matrix

$$\begin{bmatrix} v_2 & 0 \\ v_1 & \cos v_2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \mathbf{J}^\top \bar{\mathbf{x}}$$

variables	v	\bar{v}		
(v_1, \bar{v}_1)	x_1	2	$D_{31}\bar{v}_3 \rightarrow v_2\bar{v}_3$	π
(v_2, \bar{v}_2)	x_2	π	$D_{32}\bar{v}_3 + D_{42}\bar{v}_4 \rightarrow v_1\bar{v}_3 + \bar{v}_4 \cos v_2$	1
(v_3, \bar{v}_3)	$v_1 v_2$	2π	$D_{53}\bar{v}_5 + D_{63}\bar{v}_6 \rightarrow \bar{v}_5 + \bar{v}_6$	1
(v_4, \bar{v}_4)	$\sin v_2$	0	$D_{54}\bar{v}_5 + D_{64}\bar{v}_6 \rightarrow \bar{v}_5 - \bar{v}_6$	-1
(v_5, \bar{v}_5)	$v_4 + v_3$	2π	y'_1	1
(v_6, \bar{v}_6)	$v_4 - v_3$	2π	y'_2	0

Figure 11.2: Reverse accumulation automatic differentiation of the system (11.2) evaluated at $(x_1, x_2) = (2, \pi)$.

where $\bar{\mathbf{x}}$ is called the adjoint value. Using a standard unit vector ξ_i as our input for $\bar{\mathbf{x}}$ will return a row of the Jacobian matrix evaluated at \mathbf{x} . In general, for n output variables, we'll need to compute n reverse accumulation passes to compute the n rows. When the number of input variables m is much larger than the number of output variables n , which is often the case in neural networks, using reverse accumulation is much more efficient than forward accumulation. We define the adjoint $\bar{v}_i = \partial y / \partial v_i$ as the derivative of the dependent variable with respect to y_i . To compute the Jacobian for a point (x_1, x_2) , we first run through a forward pass to get the dependencies and graph structure. The forward pass is the same as forward accumulation and results in the same expression for y_1 and y_2 . We then run backward once for each dependent variable, first taking $(y'_1, y'_2) = (1, 0)$ to get the first row of the Jacobian matrix and then taking $(y'_1, y'_2) = (0, 1)$ to get the second row of the Jacobian matrix. See Figure 11.2 on this page. ▶

• Julia has several packages that implement forward and reverse automatic differentiation including Zygote.jl, ForwardDiff.jl, and ReverseDiff.jl.

11.3 Newton–Cotes quadrature

Numerical integration is often called *quadrature*, a term that like many others in mathematics seems archaic. Historically, the quadrature of a figure described the process of constructing a square with the same areas as some given figure. The word “quadrature” comes from the Latin *quadratura* (to square). It wasn’t until the late nineteenth century that the classical problem of ancient geometers of “squaring the circle”—finding a square with the same area as a circle using only a compass and straightedge—was finally proved to be impossible when the Lindemann–Weierstrass theorem demonstrated that π is a transcendental number. The name quadrature stuck with us, and now most major scientific programming

languages use some variant of the term `quad` for numerical integration.¹ It wasn't until recently that Matlab started to use the function name `integral`, because many users were having trouble finding `quad`.

The typical method for numerically integrating a smooth function is to first approximate the function by a polynomial interpolant and then integrate the polynomial exactly. Recall from Chapter 9, that given a set of $n + 1$ nodes we can determine the Lagrange basis for a polynomial

$$f(x) \approx p_n(x) = \sum_{i=0}^n f(x_i) \ell_i(x) \quad \text{where} \quad \ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}.$$

Therefore, we can approximate the integral of $f(x)$ by

$$\begin{aligned} \int_a^b p_n(x) dx &= \int_a^b \left(\sum_{i=0}^n f(x_i) \ell_i(x) \right) dx \\ &= \sum_{i=0}^n f(x_i) \int_a^b \ell_i(x) dx \\ &= \sum_{i=0}^n w_i f(x_i) \quad \text{where} \quad w_i = \int_a^b \ell_i(x) dx. \end{aligned} \quad (11.3)$$

For uniformly spaced nodes, the method is called a *Newton–Cotes formula*.

Suppose that we take two nodes at $x = \{0, 1\}$. Then

$$\begin{aligned} w_0 &= \int_0^1 \ell_0(x) dx = \int_0^1 (1-x) dx = \frac{1}{2} \\ w_1 &= \int_0^1 \ell_1(x) dx = \int_0^1 x dx = \frac{1}{2}. \end{aligned}$$

We have that $\int_0^1 f(x) dx \approx \frac{1}{2}(f(0) + f(1))$. We call this the *trapezoidal rule*. Over an arbitrary interval $[a, b]$,

$$\int_a^b f(x) dx \approx \frac{b-a}{2}(f(a) + f(b)).$$

Now, suppose that we divide the interval $[0, 1]$ in half, giving us three nodes at

¹The even more archaic sounding term *cubature* (finding the volume of solids) is sometimes used in reference to multiple integrals.

$x = \{0, \frac{1}{2}, 1\}$. This time the polynomial interpolant will be quadratic. Then

$$\begin{aligned} w_0 &= \int_0^1 \ell_0(x) dx = \int_0^1 2(1-x)(\frac{1}{2}-x) dx = \frac{1}{6}, \\ w_1 &= \int_0^1 \ell_1(x) dx = \int_0^1 4x(1-x) dx = \frac{2}{3}, \\ w_2 &= \int_0^1 \ell_2(x) dx = \int_0^1 2x(x-\frac{1}{2}) dx = \frac{1}{6}. \end{aligned}$$

We have that $\int_0^1 f(x) dx \approx \frac{1}{6} [f(0) + 4f(\frac{1}{2}) + f(1)]$. We call this *Simpson's rule*. Over an arbitrary interval $[a, b]$,

$$\int_a^b f(x) dx \approx \frac{b-a}{6} \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right].$$

We could continue by dividing each subsegment $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 0]$ in half, giving us five equally spaced nodes. This time the polynomial interpolant is quartic, and the approximation is called *Boole's rule*:

$$\int_0^1 f(x) dx \approx \frac{2}{45} [7f(0) + 32f(\frac{1}{4}) + 12f(\frac{1}{2}) + 32f(\frac{3}{4}) + 7f(1)].$$

From theorem 26 on page 228, if an n th degree polynomial $p_n(x)$ interpolates a sufficiently smooth function $f(x)$ at the points x_0, x_1, \dots, x_n in an interval, then for each x in that interval there is a ξ in the interval such that the pointwise error

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^n (x - x_i).$$

The error for the trapezoidal rule over an interval of length h is

$$\int_0^h f(x) - p_1(x) dx = \frac{1}{2} f^{(2)}(\xi) \int_0^h (x-h) dx = -\frac{1}{12} h^3 f^{(2)}(\xi)$$

for some $\xi \in [0, h]$. Now, consider Simpson's rule over $[-h, h]$ —two subintervals each of length h . Simpson's rule is derived using a quadratic interpolating polynomial, so it is exact if $f(x)$ is a quadratic polynomial. Simpson's rule is in fact also exact if $f(x)$ is cubic, because x^3 is an odd function and its contribution integrates out to zero. So, let's consider a cubic polynomial over the interval $[-h, h]$ with nodes at $-h$, 0 , and h and a fourth node at an arbitrary point c . The error of Simpson's rule is

$$\int_{-h}^h f(x) - p_3(x) dx = \frac{1}{24} f^{(4)}(\xi) \int_{-h}^h x(x-c)(x^2 - h^2) dx = -\frac{1}{90} h^5 f^{(4)}(\xi)$$

for some $\xi \in [-h, h]$. While we can get more accurate methods by adding nodes and using a higher-order polynomials such as Boole's rule, such an approach is not generally a good approach because of the Runge phenomenon. Instead, we can either use a piecewise polynomial approximation (a spline) or we pick nodes so that the uniform error is minimized (such as using Chebyshev nodes). We'll look at both of these options in turn.

► Composite methods

Composite methods use splines to approximate the functions being integrated. The simplest type of spline is a piecewise constant spline. Using these we get Riemann sums. By using a piecewise linear spline, we get a more accurate method—the *composite trapezoidal rule*—that applies the trapezoidal rule to each subinterval. For $n + 1$ equally-space nodes

$$\int_a^b f(x) dx \approx h \left[\frac{1}{2}f(a) + \sum_{i=1}^{n-1} f(a + ih) + \frac{1}{2}f(b) \right] \quad (11.4)$$

where $h = (b - a)/n$.

```
function trapezoidal(f,x,n)
    F = f.(LinRange(x[1],x[2],n+1))
    (F[1]/2 + sum(F[2:end-1]) + F[end]/2)*(x[2]-x[1])/n
end
```

The error for the trapezoidal rule is bounded by $\frac{1}{12}h^3|f^{(2)}(\xi)|$ for some ξ in a subinterval, so the error for the composite trapezoidal rule is bounded by $\frac{1}{12}(b-a)^3|f^{(2)}(\xi)|/n^2$ for some $\xi \in (a, b)$. With a little extra work, which we'll do in a couple of pages, we can derive a much better error estimate. Because the composite trapezoidal rule is a second-order method, if we use twice as many intervals, we can expect the error to be reduced by about a factor of four. And with four times as many intervals, we can reduce the error by about a factor of sixteen.

Using a quadratic spline, Simpson's rule becomes the *composite Simpson's rule*. For $n + 1$ equally spaced nodes, Simpson's rule says

$$\int_{x_{i-1}}^{x_{i+1}} f(x) dx \approx \frac{2h}{6} (f(x_{i-1}) + 4f(x_i) + f(x_{i+1})).$$

Therefore, over the entire interval $[a, b]$

$$\int_a^b f(x) dx \approx \frac{h}{3} \left(f(x_0) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + 2 \sum_{i=2}^{n/2} f(x_{2i-2}) + f(x_n) \right).$$

The error of Simpson's rule is $\frac{1}{90}h^5f^{(4)}(\xi)$, so the error of the composite Simpson's rule is bounded by $\frac{1}{180}(b-a)^5f^{(4)}(\xi)/n^4$ for some ξ over the interval (a, b) . By using twice as many intervals, we should expect the error to decrease by a factor of 16. In practice, one often uses an *adaptive* composite trapezoidal or Simpson's rule. But, if we can choose the positions of the nodes at which we integrate, then it's much better to choose Gaussian quadrature.

We can also apply Richardson extrapolation discussed on page 294 to the composite trapezoidal rule. Such an approach is called *Romberg's method*. Take $\phi(h)$ to be the composite trapezoidal rule using nodes separated by a distance h . Then $D_{m,0}$ extrapolation is equivalent to the composite trapezoidal rule, the $D_{m,1}$ extrapolation is equivalent to the composite Simpson's rule, and the $D_{m,2}$ extrapolation is equivalent to the composite Boole's rule—all with 2^m subsegments. We can implement a naïve Romberg's method by replacing the definition of ϕ in the Julia code on page 294 with a composite transpose method:

```
 $\phi = (f, x, n) \rightarrow \text{trapezoidal}(f, x, n)$ 
```

Now, `richardson(x->sin(x), [0, π/2], 4)` returns the integral of $\sin x$ from zero to $\pi/2$, with an error of 1.98×10^{-12} .

Example. Let's examine the error of the composite trapezoidal rule applied to the function $f(x) = x + x^p(2-x)^p$ over the interval $[0, 2]$ with $p = 1, 2, \dots, 7$.

```
n = [floor(Int, 10^y) for y in LinRange(1, 2, 10)]
error = zeros(10, 7)
f = (x, p) -> x + x.^p.*(2-x).^p
for p ∈ 1:7,
    S = trapezoidal(x->f(x, p), [0, 2], 10^6)
    for i ∈ 1:length(n)
        Sn = trapezoidal(x->f(x, p), [0, 2], n[i])
        error[i, p] = abs(Sn - S)/S
    end
end
slope = ([log.(n) one.(n)]\log.(error))[1, :]
info = .*(string.((1:7)'), ": slope=", string.(round.(slope'))))
plot(n, error, xaxis=:log, yaxis=:log, labels = info)
```

See Figure 11.3 on the next page. We can determine the convergence rate by computing $[\mathbf{x} \ 1]^{-1} \mathbf{y}$ where \mathbf{x} is a vector of logarithms of the number of nodes, \mathbf{y} is a vector of logarithms of the errors, and $\mathbf{1}$ is an appropriately-sized vector of ones. We had earlier determined that the error of a composite trapezoidal rule should be $O(n^2)$ or better for a smooth function. We do find that the error of the trapezoidal rule is $O(n^2)$ when $p = 1$. But, we find that the error is $O(n^4)$ when p is 2 or 3, $O(n^6)$ when p is 4 or 5, and $O(n^8)$ when p is 6 or 7. So, what's going on? ◀

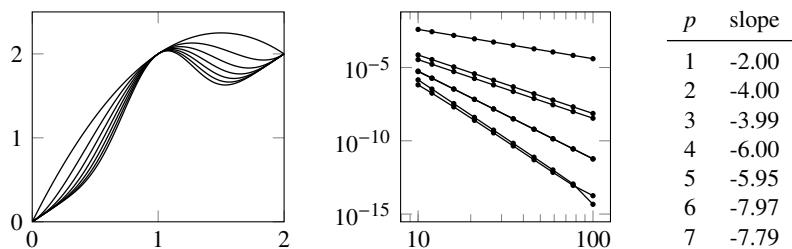


Figure 11.3: Left: Plot of $f_p(x) = x + x^p(2-x)^p$ for $p = 1, 2, \dots, 7$. Middle: Error using composite trapezoidal rule for $f_p(x)$ as a function of the number of nodes. Right: Slopes of the curves.

► Error of the composite trapezoidal rule

Let's reexamine the quadrature error of the composite trapezoidal rule a little more rigorously. Apply the composite trapezoidal rule to a function $f(x)$ on the interval $[0, \pi]$ with $n + 1$ nodes:

$$S_n = \frac{\pi}{n} \left[\frac{f(0)}{2} + \sum_{i=1}^{n-1} f\left(\frac{i\pi}{n}\right) + \frac{f(\pi)}{2} \right].$$

The interval $[0, \pi]$ might seem a little arbitrary, and in fact, any finite interval would do, but using $[0, \pi]$ will help the mathematical feng shui further on. If we expand $f(x)$ as a cosine series

$$f(x) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos kx \quad \text{with} \quad a_k = \frac{2}{\pi} \int_0^{\pi} f(x) \cos kx \, dx,$$

then

$$S = \int_0^{\pi} f(x) \, dx = \frac{a_0\pi}{2}$$

and

$$S_n = \frac{a_0\pi}{2} + \frac{\pi}{n} \sum_{k=1}^{\infty} a_k \left(\frac{1 + (-1)^k}{2} + \sum_{i=1}^{n-1} \cos \frac{ik\pi}{n} \right) = S + \pi \sum_{k=1}^{\infty} a_{2nk}.$$

The last equality follows directly from the identity

$$\sum_{j=0}^{n-1} \cos \frac{jk\pi}{n} = \begin{cases} n & \text{if } k \text{ is an even multiple of } n \\ 0 & \text{if } k \text{ is otherwise even} \\ 1 & \text{if } k \text{ is odd.} \end{cases}$$

This identity itself follows from computing the real part of the geometric series

$$\sum_{j=0}^{n-1} e^{ijk\pi/n} = \frac{1 - (-1)^k}{1 - e^{ik\pi/n}}$$

and noting that the real part of $(1 - e^{ix})^{-1}$ equals $\frac{1}{2}$ for any x .

The quadrature error $S_n - S$ is $\pi \sum_{k=1}^{\infty} a_{2nk}$. So the convergence rate of the composite trapezoidal method is determined by the convergence rate of coefficients of the cosine series. If $f(x)$ is a smooth function, then by repeatedly integrating by parts we get

$$a_{2nk} = \frac{2}{\pi} \int_0^\pi f(x) \cos 2nkx \, dx = \frac{2}{\pi} \sum_{j=1}^{\infty} (-1)^j \frac{f^{(2j-1)}(x)|_0^\pi}{(2nk)^{2j}}.$$

Therefore, the error of the trapezoidal rule can be characterized by the odd derivatives at the endpoints $x = 0$ and $x = \pi$. If $f'(0) \neq f'(\pi)$, then we can expect a convergence rate of $O(n^2)$. If $f'(0) = f'(\pi)$ and $f'''(0) \neq f'''(\pi)$, then we can expect a convergence rate of $O(n^4)$. If the function $f(x)$ is periodic or if each of the odd derivatives of $f(x)$ vanishes at the endpoints, then the composite trapezoidal rule gives us exponential or spectral convergence.² If the function $f(x)$ is not periodic or does not have matching odd derivatives at the endpoints, we can still make a change of variable to put it in a form that does. This approach called Clenshaw–Curtis or Frejér quadrature will allow us to integrate any smooth function with spectral convergence.

► Clenshaw–Curtis quadrature

Take any smooth function over an interval $[-1, 1]$ and make a change of variables $x = \cos \theta$,

$$\int_{-1}^1 f(x) \, dx = \int_0^\pi f(\cos \theta) \sin \theta \, d\theta.$$

We can express $f(\cos \theta)$ as a cosine series

$$f(\cos \theta) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(k\theta) \quad \text{where} \quad a_k = \frac{2}{\pi} \int_0^\pi f(\cos \theta) \cos k\theta \, d\theta$$

²The Euler–Maclaurin formula, which relates a sum to an integral, provides an alternative method estimating the error for the composite trapezoidal rule. The Euler–Maclaurin formula states that if $f \in C^P(a, b)$ where a and b are integers, then

$$\sum_{i=a+1}^b f(i) = \int_a^b f(x) \, dx + \sum_{k=0}^{P-1} \frac{B_{k+1}}{(k+1)!} \left(f^{(k)}(b) - f^{(k)}(a) \right) + R_P$$

where B_k are the Bernoulli numbers and R_P is a remainder term. Odd Bernoulli numbers are zero except for B_1 . Ada Lovelace is credited for writing the first computer program in 1843 to calculate the Bernoulli numbers using Charles Babbage's never-built Analytic Engine.

and integrate analytically

$$S = \int_0^\pi \left(\frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(k\theta) \right) \sin \theta \, d\theta = a_0 + \sum_{k=1}^{\infty} \frac{2a_{2k}}{1 - (2k)^2}$$

where we now need to evaluate the coefficients a_{2k} . The integrand $f(\cos \theta) \cos k\theta$ is an even function at $\theta = 0$ and $\theta = \pi$, so we can expect spectral convergence. The trapezoidal rule is just the (type-1) discrete cosine transform

$$a_{2k} = \frac{2}{n} \left[\frac{f_0}{2} + \sum_{j=1}^{n-1} f_j \cos \left(\frac{2\pi j k}{n} \right) + \frac{f_n}{2} \right] \quad \text{where } f_j = \cos \left(\frac{\pi j}{n} \right)$$

Extend the function $f(\cos \theta)$ as an even function at π : $f_{n+j} = f_{n-j}$. Then the discrete cosine transform over n nodes is exactly one-half the discrete Fourier transform of $2n$ nodes because

$$\sum_{j=0}^{2n-1} f_j \exp \left(-\frac{i2\pi j k}{n} \right) = \sum_{j=0}^{n-1} f_j \cos \left(\frac{2\pi j k}{n} \right) + \sum_{j=n}^{2n-1} f_{2n-j} \cos \left(-\frac{2\pi j k}{n} \right)$$

after the sine terms cancel each other out. We can now easily compute the sum using a fast Fourier transform that can be evaluated in $O(n \log n)$ operations.

We can implement the Clenshaw–Curtis quadrature in Julia by computing the discrete cosine transform in

```
using FFTW, LinearAlgebra
function clenshaw_curtis(f,n)
    x = cos.(π*(0:n)' / n)
    w = zeros(n+1,1); w[1:2:n+1] = 2 ./ (1 .- (0:2:n).^2)
    1/n * dctI(f.(x)) * w
end
```

Julia's FFTW package does not have a named type-1 discrete cosine transform—the `dct` function is a type-2 DCT. We can formulate a type-1 DCT using

```
function dctI(f)
    g = FFTW.r2r!([f...],FFTW.REDFT00)
    [g[1]/2; g[2:end-1]; g[end]/2]
end
```

or we can explicitly construct one as above

```
function dctI(f)
    n = length(f)
    g = real(fft(f[[1:n; n-1:-1:2]]))
    [g[1]/2; g[2:n-1]; g[n]/2]
end
```

Aspects of Clenshaw–Curtis quadrature might seem familiar from the discussion of Chebyshev polynomials over the past two chapters. Note that

$$f(\cos \theta) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty} a_k \cos(k \cos^{-1} x) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty} a_k T_k(x).$$

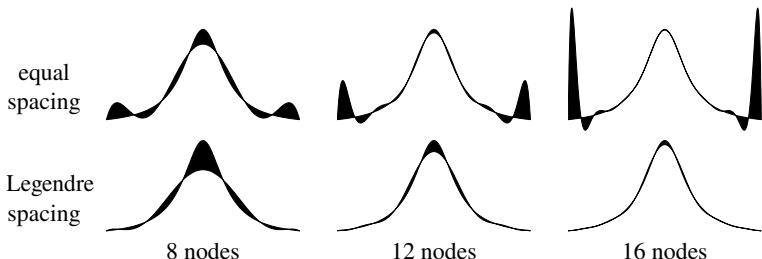
where $T_k(x)$ is the k th Chebyshev polynomial. So, expanding $f(\cos \theta)$ in a cosine series is the same as expanding $f(x)$ as using Chebyshev polynomials. In the next section, we'll discuss the expansion of functions in orthogonal polynomial bases.

11.4 Gaussian quadrature

Let's return to the Newton–Cotes formula

$$\int_a^b p(x) dx = \sum_{i=0}^n w_i f(x_i) \quad \text{where} \quad w_i = \int_a^b \ell_i(x) dx. \quad (11.5)$$

By adding more nodes, we can construct higher-order polynomials to approximate the integrand more closely. The equally spaced nodes of Newton–Cotes quadrature are problematic, specifically because of Runge's phenomenon. In Chapter 9, we used Chebyshev nodes (the zeros of the Chebyshev polynomial) to minimize Runge's oscillations. In this section, we'll consider broader families of orthogonal polynomials. The figures below show the difference between the function $(1 + 16x^2)^{-1}$ and polynomials that fit the function at either equally spaced nodes or Legendre–Lobatto nodes:



Choosing quadrature nodes from the zeros of orthogonal polynomials does more than simply mitigate Runge's phenomenon. Remarkably, using these carefully chosen nodes also allows us to determine a polynomial interpolant whose degree is almost *double* the number of nodes.

Suppose that we let the nodes x_i vary freely, still computing the weights w_i based on their positions using the Newton–Cotes formula (11.5). One might ask “what is the maximum degree of a polynomial $p(x)$ that can be constructed?”

<i>polynomial</i>	<i>interval</i>	<i>weight</i>	$\ 1\ ^2$	a_k	b_k
Legendre	$[-1, 1]$	1	2	0	$k^2/(4k^2 - 1)$
Chebyshev	$[-1, 1]$	$1/\sqrt{1-x^2}$	$\pi/2$	0	$(1 + \delta_{k0})/4$
Hermite	$(-\infty, \infty)$	$\exp(-x^2)$	$\sqrt{\pi}$	0	$k/2$
Laguerre	$(0, \infty)$	$\exp(-x)$	1	$2k$	k^2

Figure 11.4: Parameters of common Gaussian quadrature methods.

This is precisely the question Carl Friedrich Gauss raised in 1814. With $n + 1$ nodes and $n + 1$ weights along with $2n + 2$ constraints, one might rightfully conjecture that $p(x)$ would be uniquely determined using $2n + 2$ coefficients. Hence, the maximum degree is $2n + 1$. Quadrature that carefully chooses nodes to construct a maximal polynomial is called *Gaussian quadrature* (and sometimes Gauss–Christoffel quadrature).

Recall from Chapter 10 that we can generate orthogonal polynomials by using different inner product spaces. These orthogonal polynomials are the most important classes of polynomials for Gaussian quadrature and lend their names to Gauss–Legendre, Gauss–Chebyshev, Gauss–Hermite, Gauss–Laguerre, and Gauss–Jacobi quadrature. See the table above. Gauss–Jacobi quadrature, with a weight $(1-x)^\alpha(1+x)^\beta$ with $\alpha, \beta > -1$, is a generalization of Gauss–Legendre quadrature ($\alpha = \beta = 0$) and Gauss–Chebyshev quadrature ($\alpha = \beta = -\frac{1}{2}$) and is suited for functions with endpoint singularities.

Theorem 42 (Fundamental Theorem of Gaussian Quadrature). *Let w be a positive weight function and let $q \in \mathbb{P}_{n+1}$ be a polynomial of degree $n + 1$ that is orthogonal to the space of n th-degree polynomials \mathbb{P}_n . Let x_0, x_1, \dots, x_n be the zeros of q . Then for any $f \in \mathbb{P}_{2n+1}$*

$$\int_a^b f(x)\omega(x) dx = \sum_{i=0}^n w_i f(x_i) \quad \text{where} \quad w_i = \int_a^b \ell_i(x)\omega(x) dx$$

with Lagrange basis $\ell_i(x)$ with nodes at x_0, x_1, \dots, x_n .

Proof. Suppose that $f \in \mathbb{P}_{2n+1}$ and $q \in \mathbb{P}_{n+1}$. Then by dividing f by q

$$f(x) = q(x)p(x) + r(x)$$

for some $p, r \in \mathbb{P}_n$. Therefore,

$$\int_a^b f(x)\omega(x) dx = \int_a^b q(x)p(x)\omega(x) dx + \int_a^b r(x)\omega(x) dx \xrightarrow{0} \sum_{i=0}^n w_i r(x_i)$$

because $(p, q) = \int_a^b q(x)p(x)\omega(x) dx = 0$ for all $p \in \mathbb{P}_n$ and because $r \in \mathbb{P}_n$. At the zeros of q , $f(x_i) = q(x_i)p(x_i) + r(x_i) = r(x_i)$, and the result follows directly. \square

This theorem says that an n -node Gaussian quadrature yields the exact result for a polynomial of degree $2n - 1$ or less. Therefore, we can get a high-order accuracy with relatively few points—especially nice when $f(x)$ is expensive to compute.

Theorem 43. *For any function $f \in C^{2n+2}[a, b]$, the Gaussian quadrature error*

$$\int_a^b f(x)\omega(x) dx - \sum_{i=0}^n w_i f(x_i) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \int_a^b q^2(x)\omega(x) dx$$

where $q(x) = \prod_{i=0}^n (x - x_i)$ and ξ is some point in the interval of integration.

Proof. There exists a polynomial of degree $p(x)$ of degree at most $2n + 1$ with $p(x_i) = f(x_i)$ and $p'(x_i) = f'(x_i)$. From theorem 27

$$f(x) - p(x) = \frac{f^{(2n+2)}(\xi(x))}{(2n+2)!} q^2(x)$$

from which it follows (after integrating both sides and applying the mean value theorem for integrals to the right-hand side):

$$\int_a^b f(x)\omega(x) dx - \int_a^b p(x)\omega(x) dx = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \int_a^b q^2(x)\omega(x) dx.$$

By the fundamental theorem of Gaussian quadrature

$$\int_a^b p(x)\omega(x) dx = \sum_{i=0}^n w_i p(x_i) = \sum_{i=0}^n w_i f(x_i).$$

And the error formula holds after substitution. \square

► Determining the nodes and the weights

We still need to determine the nodes x_i and the weights w_i . Let's first find the nodes for an arbitrary family of orthogonal polynomials. Recall Favard's theorem (page 250), which says that for each weighted inner product there exist uniquely determined orthogonal polynomials $p_k \in \mathbb{P}_k$ with leading coefficient one satisfying the three-term recurrence relation

$$\begin{cases} p_{k+1}(x) = (x - a_k)p_k(x) - b_k p_{k-1}(x) \\ p_0(x) = 1, \quad p_1(x) = x - a_1 \end{cases}$$

where

$$a_k = \frac{(xp_k, p_k)}{(p_k, p_k)} \quad \text{and} \quad b_k = \frac{(p_k, p_k)}{(p_{k-1}, p_{k-1})}.$$

It will be useful to use an orthonormal basis. Take $\hat{p}_k = p_k / \|p_k\|$. Then we can rewrite the three-term recurrence relation as

$$\|p_{k+1}\| \hat{p}_{k+1}(x) = (x - a_k) \|p_k\| \hat{p}_k(x) - b_k \|p_{k-1}\| \hat{p}_{k-1}(x).$$

Dividing through by $\|p_k\|$ gives us

$$\frac{\|p_{k+1}\|}{\|p_k\|} \hat{p}_{k+1}(x) = (x - a_k) \hat{p}_k(x) - b_k \frac{\|p_{k-1}\|}{\|p_k\|} \hat{p}_{k-1}(x),$$

and by noting that $\|p_{k+1}\|/\|p_k\| = \sqrt{b_k}$, we have

$$\sqrt{b_{k+1}} \hat{p}_{k+1}(x) = (x - a_k) \hat{p}_k(x) - \sqrt{b_k} \hat{p}_{k-1}(x).$$

We can rearrange the terms

$$\sqrt{b_{k+1}} \hat{p}_{k+1}(x) + a_k \hat{p}_k(x) + \sqrt{b_k} \hat{p}_{k-1}(x) = x \hat{p}_k(x),$$

which is simply the linear system $\mathbf{J}_n \mathbf{p} = x \mathbf{p} + \mathbf{r}$:

$$\begin{bmatrix} a_0 & \sqrt{b_1} & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & \\ & \ddots & \ddots & \ddots \\ & & \sqrt{b_{n-1}} & a_{n-1} & \sqrt{b_n} \\ & & & \sqrt{b_n} & a_n \end{bmatrix} \begin{bmatrix} \hat{p}_0(x) \\ \hat{p}_1(x) \\ \vdots \\ \hat{p}_{n-1}(x) \\ \hat{p}_n(x) \end{bmatrix} = x \begin{bmatrix} \hat{p}_0(x) \\ \hat{p}_1(x) \\ \vdots \\ \hat{p}_{n-1}(x) \\ \hat{p}_n(x) \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \sqrt{b_{n+1}} \hat{p}_{n+1}(x) \end{bmatrix}.$$

The tridiagonal matrix \mathbf{J}_n is called the *Jacobi operator*. This equation says that $p_{n+1}(x) = 0$ if and only if $\mathbf{r} = \mathbf{0}$ if and only if $\mathbf{J}_n \mathbf{p} = x \mathbf{p}$. That is, the roots of p_{n+1} are simply the eigenvalues of \mathbf{J}_n .

Now, let's get the weights. Note that by the fundamental theorem of Gaussian quadrature

$$\sum_{k=0}^n w_k \hat{p}_i(x_k) \hat{p}_j(x_k) = \int \hat{p}_i(x) \hat{p}_j(x) \omega(x) dx = (\hat{p}_i, \hat{p}_j) = \delta_{ij},$$

because the degree of $\hat{p}_i(x) \hat{p}_j(x)$ is strictly less than $2n$. But this simply says that $\mathbf{Q}^T \mathbf{Q} = \mathbf{I}$ where the elements $Q_{ik} = \sqrt{w_k} \hat{p}_i(x_k)$. Hence, \mathbf{Q} is an orthogonal matrix. So, \mathbf{QQ} also equals the identity operator. Therefore,

$$\sum_{k=0}^n \sqrt{w_i w_j} \hat{p}_k(x_i) \hat{p}_k(x_j) = \delta_{ij}$$

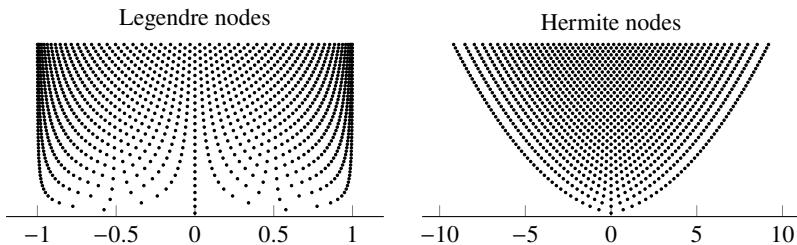


Figure 11.5: Location of Legendre and Hermite nodes for $n = 1, 2, \dots, 50$.

which says (for $i = j$) that

$$w_j \sum_{k=0}^n \hat{p}_k^2(x_j) = 1.$$

That is, $\sqrt{w_j}[\hat{p}_0(x_j), \hat{p}_1(x_j), \dots, \hat{p}_n(x_j)]^\top$ is a unit eigenvector of \mathbf{J}_n . Let's examine the first component of this unit eigenvector $\sqrt{w_j}\hat{p}_0(x_j)$. Because $p_0(x) \equiv 1$, we have $\hat{p}_0(x) \equiv 1/\|1\|$. Therefore, $\sqrt{w_j}$ simply equals $\|1\|$ times the first component of the unit eigenvector of \mathbf{J}_n . We can summarize all of this in the following algorithm.

Theorem 44 (Golub–Welsch algorithm). *The nodes x_i for Gaussian quadrature are given by the eigenvalues of the Jacobi operator*

$$\begin{bmatrix} a_0 & \sqrt{b_1} & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & \\ & \ddots & \ddots & \ddots \\ & & \sqrt{b_{n-1}} & a_{n-1} & \sqrt{b_n} \\ & & & \sqrt{b_n} & a_n \end{bmatrix}.$$

The weights are given by $\|1\|^2$ times the square of the first element of the unit eigenvectors.

Parameters for common Gaussian quadrature rules are summarized in Figure 11.4 on page 308. The position of the nodes of Gauss–Legendre and Gauss–Hermite quadrature are shown in the figure above. Note that nodes and weights of Gauss–Chebyshev quadrature can be directly and simply computed from the expression $x_k = \cos((2k - 1)\pi/2n)$ and $w_k = \pi/n$, so there is no need to use the Golub–Welsch algorithm. The Golub–Welsch algorithm takes $O(n^2)$ operations to compute the eigenvalue-eigenvector pairs of a symmetric matrix. More recently, faster $O(n)$ algorithms have been developed that use

Newton's method to solve $p_n(x) = 0$ and evaluate $p_n(x)$ and $p'_n(x)$ by three-term recurrence or use asymptotic expansion.³

We can implement a naïve Gauss–Legendre quadrature with n nodes using

```
f = x -> cos(x)*exp(-x^2)
nodes, weights = gauss_legendre(n)
f.(nodes) * weights
```

where the nodes and weights are computed using the Golub–Welsch algorithm

```
function gauss_legendre(n)
    a = zeros(n)
    b = (1:n-1).^2 ./ (4*(1:n-1).^2 .- 1)
    l2 = 2
    λ, v = eigen(SymTridiagonal(a, sqrt.(b)))
    (λ, l2*v[1,:].^2)
end
```

Alternatively, the Julia library FastGaussQuadrature.jl gives a fast, accurate method of computing the nodes and weights

```
using FastGaussQuadrature
nodes, weights = gausslegendre(n)
```

Gaussian quadrature can be extended to any bounded domain or unbounded domain. To integrate over an interval $[a, b]$, use the affine transformation $x = \varphi(\xi) = \frac{1}{2}(b - a)\xi + \frac{1}{2}(a + b)$ which maps $[-1, 1] \rightarrow [a, b]$. Then we have

$$\int_a^b f(x) dx = \frac{b-a}{2} \int_{-1}^1 f(\varphi(\xi)) d\xi,$$

and we perform Gauss–Legendre quadrature as usual. To integrate $\int_{-\infty}^{\infty} f(x) dx$, we might try a change of variable using a transformation $x = \tanh^{-1} \xi$ which maps $(-1, 1) \rightarrow (-\infty, \infty)$. Then

$$\int_{-\infty}^{\infty} f(x) dx = \int_{-1}^1 \frac{f(\tanh^{-1} \xi)}{1 - \xi^2} d\xi,$$

and we perform Gauss–Legendre quadrature as usual. Or, instead we might try $x = \tan(\pi\xi/2)$ or $x = \xi/(1 - \xi^2)$. Gauss–Hermite quadrature can itself be thought of as a change of variable from $(-1, 1) \rightarrow (-\infty, \infty)$ with the scaled error function $w = (\sqrt{\pi}/2) \operatorname{erf} x$ and $d\xi = e^{-x^2} dx$. Similarly, Gauss–Laguerre quadrature is a change of variable from $(0, 1) \rightarrow (0, \infty)$ with $\xi = -e^{-x}$ and $d\xi = e^{-x} dx$.

³Algorithms using Ignace Bogaert's explicit asymptotic formulas can easily generate over a billion Gauss–Legendre nodes and weights to 16 digits of accuracy. See Townsend [2015]

► Gauss–Kronrod quadrature

Gauss–Kronrod quadrature extends Gaussian quadrature by augmenting the original quadrature nodes with an additional set of nodes. Usual Gaussian quadrature is first computed using the original set of nodes, for a quadrature rule that is order $2n - 1$. Then quadrature is repeated over both sets of nodes, reusing the function evaluations from the first set of nodes. The resulting quadrature is of order $3n + 1$ and can be used in estimating quadrature error. Let's take a closer look.

Consider a polynomial $p(x) \in \mathbb{P}_n$ with n roots in the interval (a, b) and a second polynomial $q(x) \in \mathbb{P}_{n+1}$ with $n + 1$ roots in the same interval. Then $p(x)q(x) \in \mathbb{P}_{2n+1}$. Any polynomial $f(x) \in \mathbb{P}_{3n+1}$ can be written as

$$f(x) = r(x) + p(x)q(x)s(x)$$

for some $s(x) \in \mathbb{P}_n$ and $r(x) \in \mathbb{P}_{2n}$, because the number of coefficients of $f(x)$ must equal the number of coefficients of $r(x)$ plus the number of coefficients of $s(x)$. Furthermore, if

$$\int_a^b p(x)q(x)x^k \omega(x) dx = 0 \quad \text{for each } k = 0, 1, \dots, n \quad (11.6)$$

for some weight function $\omega(x)$, then

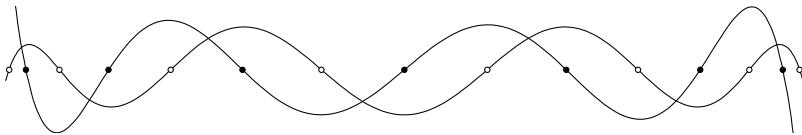
$$\int_a^b f(x)\omega(x) dx = \int_a^b r(x)\omega(x) dx.$$

We just need to find a suitable $q(x)$ so that (11.6) holds. In the 1960s, Aleksandr Kronrod examined this problem for the interval $[-1, 1]$ and unit weight function by taking $p(x) = P_n(x)$ to be a degree- n Legendre polynomial and $q(x) = K_{n+1}(x)$ to be the degree- $(n + 1)$ Stieltjes polynomial associated with $P_n(x)$. In general, if $P_n(x)$ is any degree- n orthogonal polynomial over the interval (a, b) with the associated weight $\omega(x)$, then the *Stieltjes polynomial* $K_{n+1}(x)$ associated with $P_n(x)$ is a degree- $(n + 1)$ polynomial defined by the orthogonality condition

$$\int_a^b K_{n+1}(x)P_n(x)x^k \omega(x) dx = 0 \quad \text{for } k = 0, 1, \dots, n.$$

The family of Stieltjes polynomials $K_0(x), K_1(x), \dots$ is associated to a family of orthogonal polynomials $P_0(x), P_1(x), \dots$. While Stieltjes polynomials are not guaranteed to have real zeros, there are important families of Stieltjes polynomials that not only have real zeros but have interlacing zeros, meaning that between any two zeros of P_n there is a zero of K_{n+1} . The Stieltjes polynomial associated

with Legendre polynomials is one such family.⁴ The following figure shows the Legendre polynomial $P_7(x)$ along with the associated Stieltjes polynomial $K_8(x)$, whose roots interlace one another:



We can generate a Kronrod quadrature rule once we determine the $2n + 1$ nodes x_i of $K_{n+1}(x)P_n(x)$ along with the associated quadrature weights w_i . One method is using Jacobi matrices similar to the Golub–Welsch algorithm described in Laurie [1997]. Because $r(x)$ is a degree- $2n$ polynomial, $r(x)$ can be integrated exactly using $2n + 1$ quadrature nodes. At the roots x_i of $p(x)q(x)$, the function $f(x_i) = r(x_i)$, and we can integrate $f(x)$ as $\sum_{i=0}^{2n} w_i f(x_i)$. A popular implementation of the adaptive Gauss–Kronrod rule is a (15,7)-point pair that couples a 7-point Gauss–Legendre rule with a 15-point Gauss–Kronrod rule, requiring 15 function evaluations. The integral is computed using the Kronrod rule, and error is estimated by subtracting the two rules. The weights and nodes are simply hard-coded into the algorithm.

► Gauss–Radau and Gauss–Lobatto quadratures

Gaussian quadrature nodes are strictly in the interior of the interval of integration. We can modify Gaussian quadrature to add a node to one endpoint, called *Gauss–Radau quadrature*, or to both end endpoints, called *Gauss–Lobatto quadrature*. The nodes and weights for Gauss–Radau and Gauss–Lobatto quadratures can be determined using the Golub–Welsh algorithm with a modified Jacobi matrix or by using Newton’s method. (See Gautschi [2006].)

The Golub–Welsh algorithm can generate the nodes and weights of an $(n + 1)$ -point Gauss–Legendre–Radau quadrature by replacing $a_n = \pm n / (2n - 1)$ for a node at ± 1 . And, the algorithm can generate the nodes and weights of an $(n + 1)$ -point Gauss–Legendre–Lobatto quadrature by replacing $b_n = (n - 1) / (2n - 3)$ for nodes at ± 1 . Because Gauss–Radau quadrature fixes a node to be an endpoint, it removes one degree of freedom from the set of nodes and weights. So, using it we can fit at most a degree- $2n$ polynomial with a set of $n + 1$ nodes. Gauss–Lobatto quadrature fixes nodes at both endpoints, removing two degrees of freedom. So, we can fit at most a degree- $(2n - 1)$ polynomial with a set of $n + 1$ nodes.

⁴Another is the Stieltjes polynomials associated with the Chebyshev polynomials of the first kind $T_n(x)$, which happens to be $(1 - x^2)U_{n-2}(x)$, where $U_n(x)$ is a Chebyshev polynomial of the second kind—with weight $\omega(x) = (1 - x^2)$ over $[-1, 1]$. A third—the associated Stieltjes polynomials for the Chebyshev polynomials of the second kind $U_n(x)$ —are the Chebyshev polynomials of the first kind $T_n(x)$.

► Quadrature in practice

Every scientific programming language has some sort of quadrature method, typically including a variant of Gaussian quadrature such as Gauss–Lobatto or Gauss–Kronrod quadrature. Julia has several quadrature packages. QuadGK.jl uses adaptive Gauss–Kronrod quadrature. FastGaussQuadrature.jl computes the nodes and weights for n -point Gaussian quadrature in $O(n)$ operations. The routine can compute a billion Gauss–Legendre nodes in under a minute. Matlab includes several methods. The functions quad, quadv, and quadl, which perform the adaptive Simpson’s rule, vectorized adaptive Simpson’s rule, and Gauss–Lobatto quadrature, are no longer recommended in the documentation. It advises using integral instead, which is “just an easier to find and easier to use version of quadgk.” The function quadgk is an implementation of adaptive Gauss–Kronrod quadrature. Matlab also has trapz, which implements the composite trapezoidal rule. The Octave commands are similar to Matlab commands with a few exceptions. The function quad numerically integrates a function using the Fortran routines from QUADPACK, and the Octave command quadcc uses adaptive Clenshaw–Curtis quadrature. Python’s scipy.integrate function quad and quad_vec (for vector-valued functions) use the QUADPACK library, providing a Gauss–Kronrod 21-point rule, Gauss–Kronrod 15-point rule, and a trapezoidal rule. The library also includes simple routines such as simpson, which implements Simpson’s rule (for sampled data), and romberg and romb, which implement Romberg’s method.

11.5 Monte Carlo integration

Gaussian quadrature uses precisely determined nodes to compute a highly accurate solution. But, sometimes when there is a great deal of complexity or high numbers of dimensions, you may just want any solution. Monte Carlo integration takes an entirely different approach from the polished grace of orthogonal polynomial quadrature. Instead, it relies on almost brute force application of statistical law, allowing it to tackle problems that other techniques can’t touch.

Example. Buffon’s needle is one of the earliest problems to pair probability together with geometry. In 1733, French naturalist Georges–Louis Leclerc, Comte de Buffon posed the problem: “Suppose we have a floor made of parallel strips of wood, each the same width, and we drop a needle onto the floor. What is the probability that the needle will lie across a line between two strips?” Let’s let ℓ be the length of a needle, d be the width of the strip of wood, θ be the angle of the needle with respect to the length of the strip, and x be the distance of the center of the needle from the line. Let’s also take the length of the needle smaller than the width of the strip. Then the needle crosses the line if $x \leq (\ell/2) \sin \theta$. By dropping the needle at random, the position x and the angle θ will each come

from uniform distributions $U(0, d/2)$ and $U(0, \pi/2)$, where $U(a, b) = 1/(b - a)$ if $x \in [a, b]$ and zero otherwise. To find the probability that the needle lies across a line we simply integrate over the joint probability density function $U(0, d/2) \cdot U(0, \pi/2)$ to get $4/\pi d$:

$$\int_0^{\pi/2} \int_0^{(\ell/2) \sin \theta} \frac{4}{\pi d} dx d\theta = \frac{2\ell}{\pi d}.$$

When the length of a needle is exactly half the width of a strip of wood, the probability equals $1/\pi$, giving us an experimental way of computing π albeit not a very efficient one. Take a large number of needles, perhaps a thousand or more. Drop them on the floor and count those that cross a line. Then, the number of needles divided by that count should give us a rough approximation of π . The more needles we use, the better the approximation. ◀

To compute multidimensional integrals we could apply Fubini's rule to repeated one-dimensional integrals. This approach is perfect for low-dimensional domains, but the complexity grows exponentially as the number of dimensions increases. With N total nodes in d dimensions, we have an equivalent of $N^{1/d}$ nodes per dimension, and a p th-order quadrature method has a convergence rate of $O(N^{-p/d})$. For example, in statistical mechanics, we can compute the energy of a system of particles by integrating over the probability distribution in phase space $f(\mathbf{x})$ where \mathbf{x} is the generalized coordinates of position and velocity. For six particles (each with three position and velocity coordinates), we would need to integrate over a 36-dimensional space. Using a uniform mesh with only two grid points in each direction, we would still need to compute 2^{36} nodes. With a petaflop supercomputer (one that does 10^{15} floating-point operations per second), we could expect a solution in under a second. But with only two nodes in each dimension, we are likely to get a poor approximation. To reduce the error in the approximation by half, we could use twice as many points in each dimension. Now, we would need 2^{72} nodes—in other words, more than two months using the same computer. The problem is afflicted with the so-called *curse of dimensionality*.

The Monte Carlo method was developed by Stanislaw Ulam in 1946. While convalescing and playing solitaire, Ulam asked himself what was the likelihood that a game of Canfield was winnable. After giving up on solving the problem using pure combinatorial calculations, Ulam wondered whether it might be more practical to simply lay out say a hundred hands and then just count the number of wins. With newly developed electronic computers such as ENIAC, the mundane task of statistical sampling could be automated. Furthermore, the approach seemed well suited for exploring neutron diffusion and other problems of mathematical physics.⁵

⁵The name Monte Carlo, proposed by Ulam's colleague Nicholas Metropolis, was inspired by

The *law of the unconscious statistician* provides the expected value of a function $f(X)$ of a random variable X

$$\mathbb{E}[f(X)] = \int_{\Omega} f(\mathbf{x}) p_X(\mathbf{x}) d\mathbf{x} \quad (11.7)$$

where p_X is the probability density function of the random variable X . A probability density function is a non-negative function that tells us probability that the random variable lies within a region $P(X \in \Omega) = \int_{\Omega} p_X(\mathbf{x}) d\mathbf{x}$. Equation (11.7) is intuitive and we'll take it as given—for a rigorous proof see a standard text in probability. Consider the one-dimensional case for a uniform distribution $U(a, b)$. Its probability density function $p_X(x)$ equals $(b - a)^{-1}$ for $x \in (a, b)$ and zero otherwise. Using (11.7), the integral

$$F = \int_a^b f(x) dx = (b - a) \mathbb{E}[f(X)].$$

Now, we need to just estimate the expected value $\mathbb{E}[f(X)]$, which we can do in practice by sampling. In general, we want to compute $F = \int_{\Omega} f(\mathbf{x}) d\mathbf{x}$. We'll define the N -sample Monte Carlo estimate for F as

$$\langle F^N \rangle = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(X_i)}{p_X(X_i)}.$$

A Monte Carlo estimate has the expected value

$$\begin{aligned} \mathbb{E}[\langle F^N \rangle] &= \mathbb{E}\left[\frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p_X(X_i)}\right] = \frac{1}{N} \sum_{i=1}^N \mathbb{E}\left[\frac{f(X_i)}{p_X(X_i)}\right] \\ &= \frac{1}{N} \sum_{i=1}^N \int_{\Omega} \frac{f(\mathbf{x})}{p_X(\mathbf{x})} p_X(\mathbf{x}) d\mathbf{x} = \frac{1}{N} \sum_{i=1}^N \int_{\Omega} f(\mathbf{x}) d\mathbf{x} = \int_{\Omega} f(\mathbf{x}) d\mathbf{x} = F. \end{aligned}$$

The law of large numbers says that sample average converges to the expected value as the sample size approaches infinity. For arbitrarily small ε , the probability

$$\lim_{N \rightarrow \infty} P(|\langle F^N \rangle - \mathbb{E}[\langle F^N \rangle]| > \varepsilon) = 0.$$

Hence, the Monte Carlo estimate $\langle F^N \rangle$ converges to the integral F . To integrate a function $f(\mathbf{x})$ over the bounded domain Ω using a uniform distribution, we take $F = \int_{\Omega} f(\mathbf{x}) d\mathbf{x} = V(\Omega) \mathbb{E}[f]$ where $V(\Omega)$ is the volume of the domain.

Ulam's uncle who would borrow money from relatives because he "just had to go to Monte Carlo." The probabilistic programming language Stan (with interfaces in Julia, Python, and Matlab) was named in honor of Stanislaw Ulam.

The variance of a Monte Carlo estimate is

$$\begin{aligned}\sigma^2 [\langle F^N \rangle] &= \sigma^2 \left[\frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p_X(X_i)} \right] \\ &= \frac{1}{N^2} \sum_{i=1}^N \sigma^2 \left[\frac{f(X_i)}{p_X(X_i)} \right] = \frac{1}{N^2} \sum_{i=1}^N \sigma^2 [Y_i] = \frac{1}{N} \sigma^2 [Y],\end{aligned}$$

where $\sigma^2 [Y_i]$ is the sample variance of $Y_i = f(X_i)/p_X(X_i)$. The second equality comes from two properties of variance—that $\sigma^2 [aX] = a^2 \sigma^2 [X]$ and that $\sigma^2 [X_1 + X_2] = \sigma^2 [X_1] + \sigma^2 [X_2]$ when X_1 and X_2 are independent random variables. The last equality holds because X_i are independent and identically distributed variables. The standard error of the mean is then

$$\sigma [\langle F^N \rangle] = \frac{1}{\sqrt{N}} \sigma [Y],$$

and the convergence rate is $O(N^{-1/2})$. Such a convergence rate is rather poor in comparison with even a simple Riemann sum with a convergence rate of $O(N^{-1})$ for one-dimensional problems. But, the error of the Monte Carlo method does not depend on the number of dimensions!

Using N nodes to integrate a function over a one-dimensional domain using a simple Riemann sum approximation results in an error of $O(1/N)$. For the d -dimensional problem, we would need N^d quadrature points to get an error $O(1/N)$. That is, for N points, we get an error of $O(N^{-1/d})$. When the dimension is greater than two, the Monte Carlo method wins out over simple first-order Riemann sums. Furthermore, we can use techniques to reduce the sampling variance $\sigma^2 [Y]$. Specifically, by choosing samples from a probability density function $p_X(\mathbf{x})$ that has a similar shape to $f(\mathbf{x})$.

► Sampling

Often, say in statistical mechanics and molecular dynamics, we want to sample from a probability distribution that is not simply the uniform distribution. In general, for a probability density function, $E[f] = \int_{\Omega} f(\mathbf{x}) p(\mathbf{x}) d\mathbf{x}$. Let's examine two sampling methods—*inversion sampling* and *acceptance-rejection sampling*.

Inversion sampling A function $p(x)$ is a probability density function if $p(x)$ is nonnegative and $\int_{-\infty}^{\infty} p(x) dx = 1$. The cumulative distribution function $P(x) = \int_{-\infty}^x p(s) ds$, a monotonically increasing function, is the probability that a sampled value will be less than x . The inverse distribution function $P^{-1}(y)$ allows us to sample from $p(x)$ by sampling y from a uniform distribution. For example, the standard normal distribution $p(x) = e^{-x^2/2}/\sqrt{2/\pi}$ has the error

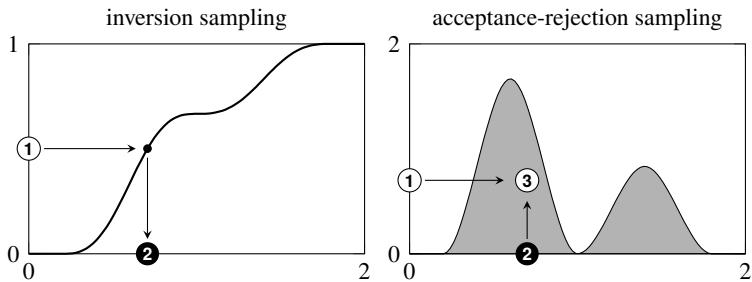


Figure 11.6: Techniques for sampling a variable ② from a distribution. Using inversion sampling, a value ① is sampled from $U(0, 1)$ and mapped to ② using the inverse cumulative probability function. Using acceptance-rejection sampling, ② is uniformly sampled from the domain and ① is sampled from another easy to compute distribution, possibly a uniform distribution. If the corresponding ③ lies within the accept region □, then choose ②. Otherwise, draw ② and ① again, repeating until you have one that does.

function $P(x) = \text{erf } x$ as its cumulative distribution. To sample from a normal distribution, we would compute $P^{-1}(y) = \text{erf}^{-1} y$, where y is sampled from a uniform distribution. To sample from an exponential distribution $p(x) = \alpha e^{-\alpha x}$, we first calculate $P(x) = \int_0^x \alpha e^{-\alpha s} ds = 1 - e^{-\alpha x}$. Then taking its inverse we get $P^{-1}(y) = -\alpha^{-1} \log(1 - y)$.

Acceptance-rejection sampling Computing the inverse of a cumulative density function is not always easy, especially if the distribution is empirical. Instead, another way to sample from a distribution p is by sampling from a different but close distribution g and use the acceptance-rejection algorithm (developed by von Neumann in 1951). Take M such that $p(x) \leq Mg(x)$ for all x in our domain.

1. Sample x using $g(x)$ and sample y from $U(0, 1)$.
2. If $y < p(x)/Mg(x)$, then accept the value x as a sample drawn from p . Otherwise, reject it and start again with the first step, repeating until you have an x that is accepted.

Metropolis algorithm The Metropolis algorithm uses acceptance-rejection sampling to generate a sequence of random samples from a distribution $p(x)$. The Metropolis algorithm works sampling from some arbitrary symmetric test distribution $g(x|y)$. That is, $g(x|y) = g(y|x)$ such as a normal distribution $g(x|y) = e^{-(x-y)^2/2}/\sqrt{2/\pi}$. The extension of the Metropolis algorithm to nonsymmetric test distributions is called the Metropolis–Hastings algorithm.

Also, we can use a function $f(x)$ that is proportional to $p(x)$ rather than $p(x)$ itself. In this way, we don't explicitly need to know the normalization constants of $p(x)$. Choose an arbitrary x_0 , and for each iteration i :

1. Sample x' by from the distribution $g(x'|x_i)$ and compute an acceptance ratio $\alpha = f(x')/f(x_i)$.
2. Now, sample u from $U(0, 1)$. If $u \leq \alpha$, then accept the candidate x' and set $x_{i+1} = x'$. Otherwise, reject the candidate and set $x_{i+1} = x_i$.

Through this process, we get a sequence of values that either remain in place or move according to the acceptance ratio α . If $\alpha \geq 1$, i.e. if the density $p(x)$ at the candidate point x' is greater than the density at the current point x_i , then we always move to the candidate point. But, if the density at the candidate point is less than the density at the current point, we only move with probability α .

Example. We don't need to directly compute the inverse error function to sample from the standard normal distribution. We could instead use the Box–Muller transform.⁶ Consider

$$p(x)p(y) = \frac{1}{2\pi} e^{-(x^2+y^2)/2}.$$

By making the change of variable $x = r \cos \theta$ and $y = r \sin \theta$, we have the differential element

$$p(x)p(y) dx dy = \frac{1}{2\pi} e^{-r^2/2} r dr d\theta = \left(r e^{-r^2/2} dr \right) \left(\frac{1}{2\pi} d\theta \right).$$

We want to sample r from the density $re^{-r^2/2}$ and θ from $1/2\pi$. To do this we will use inversion sampling:

$$u = \int_0^r re^{-r^2/2} dr = 1 - e^{-r^2/2} \quad \text{and} \quad v = \int_0^\theta \frac{1}{2\pi} d\theta = \frac{\theta}{2\pi}.$$

Solving for r and θ gives us $r = \sqrt{-2 \log(1-u)}$ and $\theta = 2\pi v$ where u and v are independently sampled from the uniform distribution $U(0, 1)$. Because u is taken uniformly from $(0, 1)$, we could also sample $1-u$ and instead take $r = \sqrt{-2 \log u}$. Therefore,

$$x = \sqrt{-2 \log u} \cos(2\pi v) \quad \text{and} \quad y = \sqrt{-2 \log u} \sin(2\pi v)$$

gives two independent normally distributed samples. ◀

⁶The transform was introduced in 1958 by statistician George E. P. Box, famously known for his aphorism “all models are wrong, but some are useful,” and mathematician Mervin Muller.

 The Distrtributions.jl package has a large collection of distributions. For example, to draw 20 samples from a Poisson distribution with $\lambda = 6$ use `rand(Poisson(6), 20)`.

Example. Monte Carlo integration chooses quadrature nodes according to some probability distribution using a random number generator. John von Neumann once quipped “anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin,” but he conceded that it was far faster than feeding a computer random numbers from punched cards. A sequence of pseudo-random numbers is a sequence of statistically independent numbers sampled from the uniform distribution over the interval $[0, 1]$. There are several ways of generating such a sequence. During the latter half of the twentieth century, linear congruence generators were commonly used pseudo-random number generators until they were for the most part replaced by better methods like the Mersenne twister developed in 1998 by Makoto Matsumoto and Takuji Nishimura. A linear congruence generator is given by

$$x_{i+1} = (ax_i + c) \mod m$$

for some integers a , c , m , and seed x_0 . The Lehmer formulation, developed by Derrick Lehmer in 1949, takes $c = 0$ and m to be a prime number. By theorem 20 on page 155, by choosing a to be primitive root of m , the sequence has maximal period $m - 1$. For example, by taking $m = 13$ and $a = 2$ with the seed $x_0 = 5$, we get the sequence

$$5 \rightarrow 10 \rightarrow 7 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 11 \rightarrow 9 \rightarrow$$

which then repeats. The minimal standard generator developed by Stephen Park and Keith Miller in 1988 takes m as the Mersenne prime $M_{31} = 2^{31} - 1$ and $a = 7^5$ as a primitive root of M_{31} . Responding to criticism of the method, Park later updated to a to the prime 48271. 

11.6 Exercises

 11.1. Occasionally, one may need to compute a one-side derivative to, say, enforce a boundary condition or modeling a discontinuity.

1. Find a third-order approximation to the derivative $f'(x)$ with nodes at x , $x + h$, $x + 2h$ and $x + 3h$ for some stepsize h .
2. What choice of stepsize h minimizes the total (round-off and truncation) error of the derivative of $\sin x$ using this approximation with double-precision floating-point numbers?
3. Find a second-order approximation for the second derivative $f''(x)$ for the same set of nodes.

- 11.2. The recursive implementation of Richardson extrapolation on page 294 is inefficient when the number of steps n is high. The number of function calls doubles with each step using recursion. We'd need 2^n function calls to evaluate just $n^2 - n$ terms, resulting in excessive duplication. Rewrite the code for Richardson extrapolation as a more efficient nonrecursive function.
- 11.3. Find a zero of the function $f(x) = 4 \sin x + \sqrt{x}$ near $x = 4$ using Newton's method with automatic differentiation. Then modify Newton's method to find the local minimum of $f(x)$ near $x = 4$. To compute $f''(x)$ we can use $f(\text{Dual}(\text{Dual}(x)).\text{deriv}.\text{deriv})$.
- 11.4. The Cauchy differentiation formula relates the derivative of an analytic function with a contour integral

$$f^{(p)}(a) = \frac{p!}{2\pi i} \oint_{\gamma} \frac{f(z)}{(z-a)^{p+1}} dz,$$

where γ is any simple, counter-clockwise contour around $z = a$ in the complex plane. Compute sixth derivative of $e^x(\cos^3 x + \sin^3 x)^{-1}$ evaluated at $x = 0$. The exact value is 64.

- 11.5. Use Newton's method as an alternative to the Golub–Welsh algorithm to generate nodes and weights for Gauss–Legendre quadrature. Bonnet's recursion formula

$$nP_n(x) = (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x),$$

where $P_0(x) = 1$ and $P_1(x) = x$, is useful in generating the Legendre polynomials. The derivatives of the Legendre polynomials can be computed using the following recursion formula

$$(x^2 - 1)P'_n(x) = nxP_n(x) - nP_{n-1}(x).$$

Quadrature weights can be computed using either $w_k = 2/(nP'_n(x_k)P_{n-1}(x_k))$ or $w_k = 2/((1-x_k^2)(P'_n(x_k))^2)$. To ensure that Newton's method converges to the right nodes, you'll need to start sufficiently close to each of them. Take $x_j^{(0)} = -\cos((4j-1)\pi/(4n+2))$.

- 11.6. Use Gauss–Legendre quadrature to numerically compute

$$\int_{-1}^1 e^{-16x^2} dx = \frac{\sqrt{\pi}}{4} \operatorname{erf}(4)$$

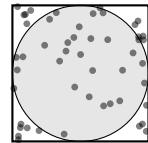
with an error of less than 10^{-14} . Repeat the exercise using composite Simpson's quadrature. Determine the convergence rate of the numerical solution to the exact solution by looping over several different numbers of nodes and plotting the error as a function of the number of nodes.

- 11.7. The fundamental solution to the heat equation $u_t = u_{xx}$ with an initial distribution $u(0, x) = u_0(x)$ is given by

$$u(t, x) = \frac{1}{\sqrt{4\pi t}} \int_{-\infty}^{\infty} u_0(s) e^{-(x-s)^2/4t} ds.$$

Use Gauss–Hermite quadrature to compute the solution to the heat equation at time $t = 1$ for an initial distribution $u_0(x) = \sin x$.

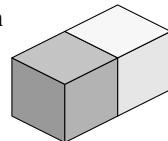
- 11.8. One simple application of Monte Carlo integration is computing π by noting that the area of a circle of unit radius is π . Take n samples $(x, y) \in [-1, 1] \times [-1, 1]$ and count the number m of samples where (x, y) is in the unit circle $x^2 + y^2 < 1$. Then $\pi \approx m/n$.



1. Use Monte Carlo integration to compute an approximation for π .
2. Use a log-log plot of the error to confirm the convergence rate.
3. Modify your solution in part 1 to use Monte Carlo integration to compute the volume of a 9-dimensional unit hypersphere.

- 11.9. Two cubes, each with equal size and equal uniform density, are in contact along one face. The gravitational force between two point masses separated by a distance r is

$$F = G \frac{m_1 m_2}{r^2},$$

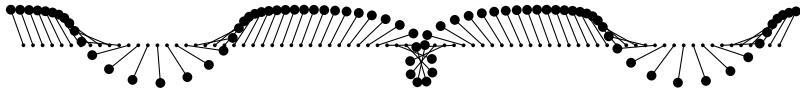


where G is the gravitational constant and m_1 and m_2 are the magnitudes of the two masses. What is the total gravitational force between the two cubes? (This question was proposed by mathematician Nick Trefethen as one of his “ten-digit problems.”)

Numerical Methods for Differential Equations

Chapter 12

Ordinary Differential Equations



Any study of numerical methods for partial differential equations (PDEs) likely starts by examining numerical methods for ordinary differential equations (ODEs). One reason is that numerical techniques used to solve time-dependent partial differential equations often employ the *numerical method of lines*, in which the spatial component is discretized and the new semi-discrete problem is solved using standard ODE solvers. Consider the heat equation $\frac{\partial}{\partial t} u(\mathbf{x}, t) = \Delta u(\mathbf{x}, t)$ where $\Delta = \sum_i \frac{\partial^2}{\partial x_i^2}$ is the Laplacian operator. The heat equation can be formally solved as a first-order linear ordinary differential equation to get the solution $u(\mathbf{x}, t) = e^{t\Delta} u(\mathbf{x}, 0)$ where $e^{t\Delta}$ is a linear operator acting on the initial conditions $u(\mathbf{x}, 0)$. A typical numerical solution involves discretizing the Laplacian operator Δ in space to create a system of ODEs and then using an ODE solver in time. A second reason to dig deep into ODEs is that the theory of PDEs builds on the theory of ODEs. Understanding stability, consistency, and convergence of the numerical schemes for ordinary differential equations will help us to understand stability, consistency, and convergence of the numerical schemes for partial differential equations.

12.1 Well-posed problems

Consider the initial value problem $u' = f(t, u)$ with $u(0) = u_0$. An initial value problem is said to be *well-posed* if (1) the solution exists, (2) the solution is unique, and (3) the solution depends continuously on the initial conditions. Let's examine these conditions for well-posedness.

A function $f(t, u)$ is *Lipschitz continuous* if there exists some constant $L < \infty$ such that $|f(t, u) - f(t, u^*)| \leq L|u - u^*|$ for all $u, u^* \in \mathbb{R}$. Lipschitz continuity is stronger than continuity but weaker than continuous differentiability. For example, the function $|u|$ is Lipschitz continuous on \mathbb{R} , although it isn't differentiable at $u = 0$. The function u^2 is not globally Lipschitz continuous (although it is locally Lipschitz continuous) because its slope becomes unbounded as u approaches infinity. Similarly, the function $\sqrt{|u|}$ is not globally Lipschitz continuous, because its slope is unbounded at $u = 0$.

To see why Lipschitz continuity is desirable, let's look at what happens to the solution of $u' = f(t, u)$ when $f(t, u)$ is not Lipschitz continuous. First, take the problem $u' = u^2$ with $u(0) = 1$ over the interval $t \in [0, 2]$. This problem can be solved by separation of variables to get

$$u(t) = \frac{1}{1-t}.$$

The solution blows up at $t = 1$, so it does not even exist on the interval $[1, 2]$. Now, consider the problem $u' = \sqrt{u}$ with $u(0) = 0$ over the interval $t \in [0, 2]$. This problem has the solution $u(t) = \frac{1}{4}t^2$. It also has the solution $u(t) = 0$. In fact, it has infinitely many solutions of the form

$$u(t) = \begin{cases} 0 & t < t_c \\ \frac{1}{4}(t - t_c)^2 & t \geq t_c \end{cases}$$

where the critical parameter is arbitrary. So, when $f(t, u)$ is not Lipschitz continuous in u , the solution may fail to exist; or if it exists, it may fail to be unique. We state the following theorem without proof

Theorem 45. *If $f(t, u)$ is continuous in t and Lipschitz continuous in u , then the initial value problem $u'(t) = f(t, u(t))$ with $u(0) = u_0$ has a unique solution.*

We say that an initial value problem is *stable* if a small perturbation in the initial value or in $f(t, u)$ produces a bounded change in the solution. That is, $u'(t) = f(t, u)$ with $u(0) = u_0$ is stable if there exists a positive value K such that for any sufficiently small ε the solution $v(t)$ to the perturbed problem $v'(t) = f(t, v) + \delta(t)$ with $v(0) = u_0 + \varepsilon_0$ satisfies $|v(t) - u(t)| \leq K\varepsilon$ whenever $|\varepsilon_0|, |\delta(t)| < \varepsilon$ for all $t \in [0, T]$. We can clarify this definition with a theorem.

Theorem 46. *If $f(t, u)$ is Lipschitz continuous in u , then the initial value problem $u'(t) = f(t, u(t))$ with $u(0) = u_0$ is stable.*

Proof. For an error $e(t) = v(t) - u(t)$, we have

$$e'(t) = v'(t) - u'(t) = f(t, v) - f(t, u) - \delta(t)$$

and $e(0) = v(0) - u(0) = \varepsilon_0$. If $f(t, u)$ is Lipschitz continuous in u , then

$$\begin{aligned}|e'(t)| &= |f(t, v) - f(t, u) - \delta(t)| \\&\leq |f(t, v) - f(t, u)| + |\delta(t)| \\&\leq L|v(t) - u(t)| + |\delta(t)| \\&\leq L|e(t)| + |\delta(t)|.\end{aligned}$$

Let ε be the maximum of ε_0 and the supremum of $|\delta(t)|$ over $[0, T]$. Then $|e'(t)| \leq L|e(t)| + \varepsilon$ and $|e(0)| \leq \varepsilon$. It follows that

$$|e(t)| \leq \frac{\varepsilon}{L} \left[(L+1)e^{Lt} - 1 \right] \leq \frac{\varepsilon}{L} \left[(L+1)e^{LT} - 1 \right] = \varepsilon K.$$

The error is bounded for $0 \leq t \leq T$, so the initial value problem is stable. \square

Keep in mind the following two takeaways both over the rest of this chapter and also the remainder of this course. First, well-posedness requires existence, uniqueness, and stability. Second, if $f(t, u)$ is continuous in t and Lipschitz continuous in u , then the problem $u' = f(t, u)$ is well-posed.

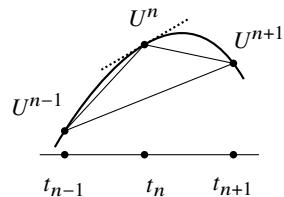
12.2 Single-step methods

Let $u \equiv u(t)$. Suppose we are given with the initial value problem

$$u' = f(u) \quad \text{where} \quad u(0) = u_0. \quad (12.1)$$

Let's discretize t by taking n uniform steps of size k , i.e., $t_n = nk$. Except for Chapter 16, the remainder of this book will adopt a fairly standard notation with k representing a step size in time and h representing a step size in space. Using k in place of Δt and h in place of Δx will help tidy some otherwise messier expressions. The derivative u' can be approximated by difference operators:

$$\begin{aligned}\text{forward difference} \quad \delta_+ U^n &= \frac{U^{n+1} - U^n}{k} \\ \text{backward difference} \quad \delta_- U^n &= \frac{U^n - U^{n-1}}{k} \\ \text{central difference} \quad \delta_0 U^n &= \frac{U^{n+1} - U^{n-1}}{2k}\end{aligned}$$



where U^n is the numerical approximation to $u(t_n)$. Using these operators, we can formulate several finite difference schemes to numerically solve the initial value problem (12.1):



Forward Euler (explicit)

 $O(k)$

$$\frac{U^{n+1} - U^n}{k} = f(U^n) \quad (12.2)$$



Backward Euler (implicit)

 $O(k)$

$$\frac{U^{n+1} - U^n}{k} = f(U^{n+1}) \quad (12.3)$$



Leapfrog (explicit)

 $O(k^2)$

$$\frac{U^{n+1} - U^{n-1}}{2k} = f(U^n) \quad (12.4)$$



Trapezoidal (implicit)

 $O(k^2)$

$$\frac{U^{n+1} - U^n}{k} = \frac{f(U^{n+1}) + f(U^n)}{2} \quad (12.5)$$

A finite difference scheme can be graphically represented using a *stencil*, which indicates which terms are employed. We'll use \bigcirc to depict the terms used on the left hand side to discretize the time derivative and \bullet to indicate the terms used on the right hand side to approximate the function $f(u)$. For ODEs, the stencils run vertically with the t_{n+1} terms at the top, the t_n terms below those, then the t_{n-1} terms, and so on. When we move to PDEs, we'll add a horizontal component to the stencil to represent discrete spatial derivatives. In each of the schemes, we can express U^{n+1} in terms of U^n or U^{n-1} either explicitly or implicitly. A numerical method of U^{n+1} is *implicit* if the right-hand side contains a term $f(U^{n+1})$. Otherwise, the numerical method is *explicit*.

Before digging into the consistency and stability of finite difference methods, let's develop a little more intuition of their behavior. An *integral curve* is a parametric curve that represents the solution to an ordinary differential equation for a specified initial condition. We can imagine the set of integral curves as streamlines of a flow. The slope of the tangent at any point is the right-hand-side term $f(t, u)$. A perturbation such as truncation error will bump us from one integral curve to a neighboring one. Existence and uniqueness require that each integral curve be non-intersecting, although they might get really close to one another. Let's plot the solutions over one step of the forward Euler, backward Euler, leapfrog, and trapezoidal methods. See the figure on the facing page.

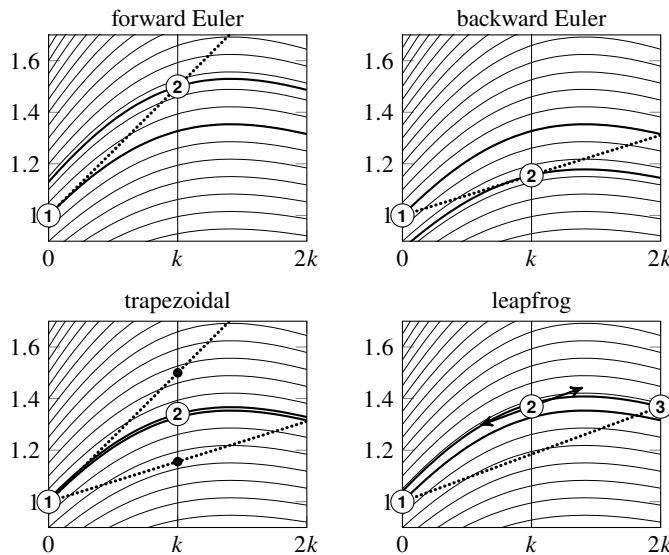


Figure 12.1: Integral curves and numerical solutions using the forward Euler, backward Euler, leapfrog, and trapezoidal methods.

The forward Euler method uses the slope of the integral curve at ① to take a step forward. We are bumped off the original curve and find ourselves at ② on a neighboring one. The backward Euler uses the slope of the integral curve at ② to take a step forward. Finding the slope of the integral curve at ② without explicitly knowing value of integral curve at ② is not trivial. The backward Euler method, like the forward Euler method, bumps off the original integral curve. The trapezoidal method averages the slopes of the forward Euler and backward Euler to take a step forward. We are still bumped off the original integral curve, but this time it's not as far. The leapfrog method is not a single step method. It uses the slope of the integral curve at ② to take a double step from ① to ③. Like the trapezoidal method, the leapfrog method performs pretty much better than either of the Euler methods.

► Consistency

To determine $u(t + k)$ for some time step k given a known value $u(t)$, we can use the Taylor series approximation of $u(t + k)$:

$$u(t + k) = u(t) + ku'(t) + \frac{1}{2}k^2u''(t) + O(k^3).$$

Using the relation $u'(t) = f(u(t))$ gives

$$u(t+k) = u(t) + kf(u(t)) + k\tau_k$$

where the truncation $\tau_k = \frac{1}{2}ku'(t) + O(k^2)$. When $k\tau_k \approx 0$, we simply have $u(t+k) \approx u(t) + kf(u(t))$, from which we have the forward Euler method $U^{n+1} = U^n + kf(U^n)$ or equivalently

$$\frac{U^{n+1} - U^n}{k} = f(U^n).$$

The error for each step, given by the local truncation error $k\tau_k$, is $O(k^2)$. The error for the method, given by τ_k , is $O(k)$. We say that a numerical scheme is *consistent* if $\lim_{k \rightarrow 0} \tau_k = 0$, that is, we can make the numerical scheme approximate the original problem to arbitrary accuracy.

We can improve the method by using a better approximation to the derivative. Take the Taylor series

$$\begin{aligned} u(t_{n+1}) &= u(t_n) + ku'(t_n) + \frac{1}{2}k^2u''(t_n) + \frac{1}{6}k^3u'''(t_n) + O(k^4) \\ u(t_{n-1}) &= u(t_n) - ku'(t_n) + \frac{1}{2}k^2u''(t_n) - \frac{1}{6}k^3u'''(t_n) + O(k^4) \end{aligned}$$

Subtracting the two gives us:

$$u(t_{n+1}) - u(t_{n-1}) = 2ku'(t_n) + \frac{1}{3}k^3u'''(t_n) + O(k^4).$$

And for $u'(t_n) = f(u(t_n))$

$$u(t_{n+1}) - u(t_{n-1}) = 2kf(t_n) + k\tau_k$$

where the truncation $\tau_k = \frac{1}{3}k^2u'''(t_n) + O(k^3)$. From this, we have the leapfrog scheme

$$\frac{U^{n+1} - U^{n-1}}{2k} = f(U^n).$$

with truncation error $O(k^2)$.

Figure 12.2 on the next page gives another graphical depiction of consistency in numerical methods. If L is a linear operator, then the differential equation $u' = Lu$ has the solution $u(t) = e^{tL}u(0)$. In particular, $u' = \lambda u$ has the solution $u(t) = e^{\lambda t}u(0)$, and starting with U^n , the exact solution after time k is $U^{n+1} = e^{\lambda t}U^n$. Substituting the ratio $r = U^{n+1}/U^n$ into the numerical schemes (12.2)–(12.5) for the model equation $u' = \lambda u$ and solving for r gives us several approximations for the multiplier $e^{\lambda t}$:

exact solution	forward Euler	backward Euler	trapezoidal	leapfrog
$e^{\lambda k}$	$1 + \lambda k$	$\frac{1}{1 - \lambda k}$	$\frac{1 + \frac{1}{2}\lambda k}{1 - \frac{1}{2}\lambda k}$	$\lambda k \pm \sqrt{(\lambda k)^2 + 1}$

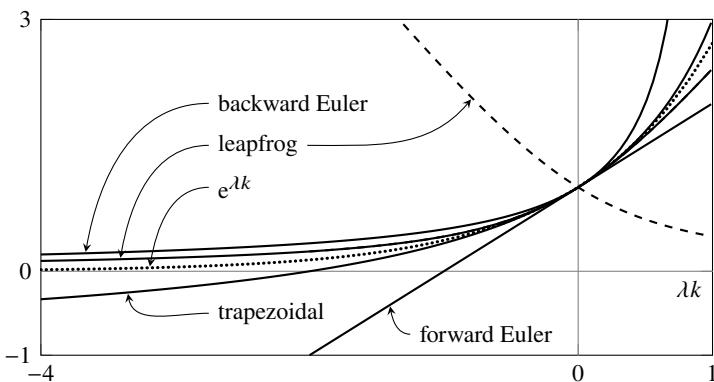


Figure 12.2: Plot of the multiplier $r(\lambda k)$ for several numerical schemes along the real λk -axis compared to the exact solution multiplier $e^{\lambda k}$.

The exact solution $e^{\lambda k}$ is depicted as a dotted curve. Approximations are good when λk is close to zero. The backward Euler and forward Euler methods are both first-order methods. The leapfrog and trapezoidal methods as second-order methods are better approximations. The leapfrog method has two solutions—one which matches the exact solution fairly well and another one which doesn't—the absolute value of this other solution is depicted using a dashed line. This second branch leads to stability issues for the leapfrog method when the real part of λk is negative and not close to zero. The backward Euler method blows up at $\lambda k = \frac{1}{2}$ and the trapezoidal method blows up at $\lambda k = 2$. Note that the backward Euler method limits 0 and the trapezoidal method limits -1 as $\lambda k \rightarrow -\infty$. We'll return to this limiting behavior when we discuss L-stable and A-stable methods. We'll examine consistency in more depth when we look at multistep methods.

► Stability

A numerical method is *stable* if, for any sufficiently small step size k , a small perturbation in the numerical solution (perhaps the initial value) produces a bounded change in the numerical solution at given subsequent time. Furthermore, a numerical method is *absolutely stable* if, for any sufficiently small step size k , the change due to a perturbation of size δ is not larger than δ at any of the subsequent time steps. In other words, a method is stable if perturbations are bounded over finite time, and it is absolutely stable if perturbations shrink or at least do not grow over time. See the figure on the following page.

Do not equate stability with consistency—a method can be both quite stable and quite wrong. Still, absolute stability is desirable because numerical errors will not only not grow, they will diminish over time. Let's determine a condition under

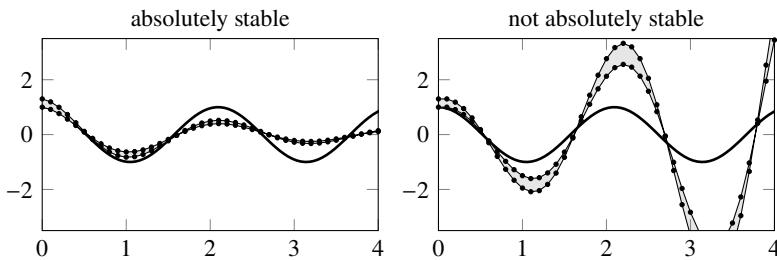


Figure 12.3: A perturbation decreases over time using an absolutely stable method and increases using an absolutely unstable one. Neither one is particularly accurate.

which a numerical method is absolutely stable. We'll limit the discussion to linear methods $U^{n+1} = L(U^n, U^{n-1}, \dots)$. The discussion applies to nonlinear methods insofar as those methods can be linearized in a meaningful and representative way. Let U^0 be an initial condition and V^0 be a perturbation of that initial condition. The error in the initial condition is $e^0 = V^0 - U^0$ and the error at time $t_n = nk$ is $e^n = V^n - U^n$. Since

$$|e^{n+1}| = \left| \frac{e^{n+1}}{e^n} \right| \cdot \left| \frac{e^n}{e^{n-1}} \right| \cdots \left| \frac{e^1}{e^0} \right| \cdot |e^0|,$$

a sufficient condition for the error to be nonincreasing is for $|e^{n+1}/e^n| \leq 1$ for all n . Because the method is a linear,

$$e^{n+1} = V^{n+1} - U^{n+1} = L(U^n, \dots) - L(V^n, \dots) = L(e^n, \dots).$$

Therefore, to determine the region of absolute stability, we need to determine under what conditions $|r| = |U^{n+1}/U^n| \leq 1$.

Consider the simple test problem $u'(t) = \lambda u(t)$ for an arbitrary complex value λ . For a nonlinear equation $u'(t) = f(t, u)$, we might consider this test problem as a linearization $u'(t) = f'(0, u_0)u(t)$. For a system of equations, $f'(0, u)$ is the Jacobian matrix that might be diagonalized to decouple a set of equations $u'_i(t) = \lambda_i u_i(t)$. For what time step k is a numerical scheme absolutely stable? To answer this question, we'll determine for what values $z = \lambda k$ the growth factor $|r| = |U^{n+1}/U^n| \leq 1$. The region in the λk -plane where $|r| \leq 1$ is called the *region of absolute stability*, and it can be used to select an appropriate numerical method. Let's find the regions of absolute stability for the forward Euler scheme, the backward Euler scheme, the leapfrog scheme, and the trapezoidal scheme.

► Regions of absolute stability

Consider the simple test problem $u'(t) = \lambda u(t)$ for an arbitrary complex value λ . For what time step k is a numerical scheme absolutely stable? To answer

this question, we'll determine for what values $z = \lambda k$ the growth factor $|r| = |U^{n+1}/U^n| \leq 1$. The region in the λk -plane where $|r| \leq 1$ is called the *region of absolute stability*, and it can be used to choose an appropriate numerical method. Let's find the regions of absolute stability for the forward Euler scheme, the backward Euler scheme, the leapfrog scheme, and the trapezoidal scheme.

Forward Euler. The forward Euler method for $u' = \lambda u$ is

$$\frac{U^{n+1} - U^n}{k} = \lambda U^n$$

from which we have $U^{n+1} = (1 + \lambda k)U^n$ and therefore $|U^{n+1}| \leq |1 + \lambda k||U^n|$. Absolute stability requires $|1 + \lambda k| \leq 1$. That is, the region inside the unit circle centered at $\lambda k = -1$. See Figure 12.4 on the next page. For a given value λ , we may need to limit the size of our step k to ensure that λk is inside the region of absolute stability. Except for the origin, the region of absolute stability for the forward Euler scheme does not contain any part of the imaginary axis, so the method is not absolutely stable for purely imaginary λ .

Backward Euler. The backward Euler method for $u' = \lambda u$ is

$$\frac{U^{n+1} - U^n}{k} = \lambda U^{n+1}.$$

From this we have $(1 - \lambda k)U^{n+1} = U^n$ and therefore $|U^{n+1}| \leq |1 - \lambda k|^{-1}|U^n|$. Absolute stability requires $|1 - \lambda k| \geq 1$. This means that we can take any size step size as long as we are outside of the unit circle centered at $\lambda k = 1$. A numerical method is said to be *A-stable* if the region of absolute stability includes the entire left half λk -plane. The backward Euler method is A-stable.

Note that absolute stability says that the error in a perturbed numerical solution decreases over time (numerical solution minus numerical solution). It does not say anything about the accuracy of a numerical method (numerical solution minus analytic solution). For example, the numerical method $U^{n+1} = 0$ is stable but the scheme is always inconsistent. Nontrivially, consider the harmonic oscillator $u'' = -9u$ with $u(0) = 1$ and $u'(0) = 0$. The solution is $u(t) = \cos 3t$. This problem can be written as a system by defining $v = u'$ and

$$\frac{d}{dt} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -9 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \quad \text{with} \quad \begin{bmatrix} u(0) \\ v(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

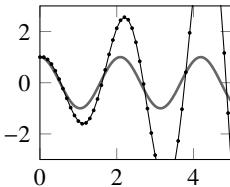
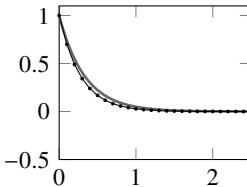
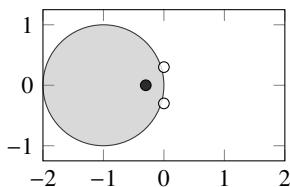
The solution using a backward Euler method

$$\begin{bmatrix} U^{n+1} \\ V^{n+1} \end{bmatrix} = \begin{bmatrix} 1 & -k \\ 9k & 1 \end{bmatrix}^{-1} \begin{bmatrix} U^n \\ V^n \end{bmatrix}$$

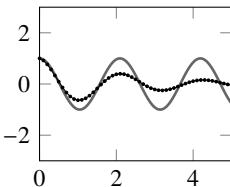
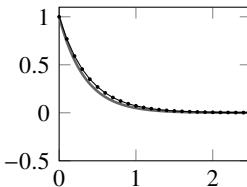
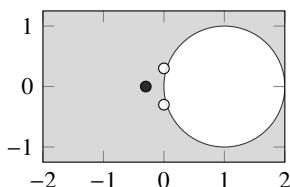
is a damped sine wave that dissipates within a few oscillations when k is large.

Region of absolute stability $\bullet \quad u' = -3u$ $\circ \quad u'' = -9u$

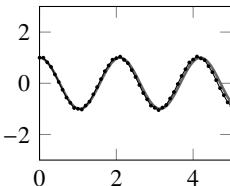
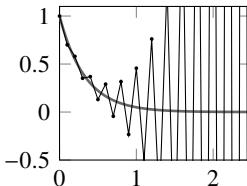
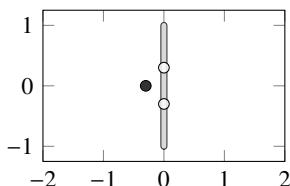
forward Euler



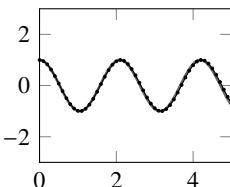
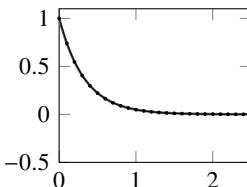
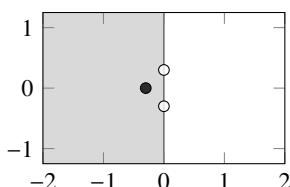
backward Euler



leapfrog



trapezoidal



Leapfrog. We can rewrite

$$\frac{U^{n+1} - U^{n-1}}{k} = 2\lambda U^n$$

as $U^{n+1} - 2\lambda k U^n - U^{n-1} = 0$. By letting $r = U^{n+1}/U^n = U^n/U^{n-1}$, then $r^2 = U^{n+1}/U^{n-1}$ and it follows that

$$r^2 - 2\lambda kr - 1 = 0.$$

We want to find when $|r| \leq 1$ in the complex plane, so let's take $r = e^{i\theta}$ as the boundary. From this we have

$$e^{2i\theta} - 2\lambda k e^{i\theta} - 1 = 0.$$

So $e^{i\theta} - e^{-i\theta} = 2\lambda k$, from which $i \sin \theta = \lambda k$. Therefore, the leapfrog scheme is absolutely stable only on the imaginary axis $\lambda k \in [-i, +i]$.

Trapezoidal. Letting $r = U^{n+1}/U^n$ in the trapezoidal method

$$\frac{U^{n+1} - U^{n-1}}{k} = \lambda \frac{U^{n+1} + U^{n-1}}{2}$$

gives us $r^2 - 1 = \frac{1}{2}\lambda k(r^2 + 1)$ or equivalently

$$\lambda k = 2 \frac{r^2 - 1}{r^2 + 1}.$$

As before, take $r = e^{i\theta}$ to identify the boundary of the region of absolute stability. Then

$$\lambda k = 2 \frac{e^{2i\theta} - 1}{e^{2i\theta} + 1} = 2i \tan \theta.$$

That is $\lambda k \in (-i\infty, +i\infty)$. So the domain boundary $|r| = 1$ along the entire imaginary axis, and the region of absolute stability includes the entire left half λk -plane. Therefore, the trapezoidal scheme is A-stable. A method is consistent if and only if the region of stability contains the origin $\lambda k = 0$. We call such a condition *zero-stability*.

12.3 Lax equivalence theorem

Before continuing to discuss specific numerical methods in depth, let's pause to consider the Lax equivalence theorem, which says that a consistent and stable method converges. “Consistency” means that we can get two equations to agree enough, “stability” means that we can control the size of the error enough, and “convergence” means that we can get two solutions to agree enough. The theorem itself is important enough that it's sometimes called the fundamental theorem of

numerical analysis. It tells us when we can trust a numerical method for solving linear ordinary or partial differential equations to give us the right results.

Consider the initial value problem $u_t = L u$ where L is a linear operator. In the next chapter, we'll consider L to be the Laplacian operator, and we'll get the heat equation $u_t = \Delta u$. In Chapter 14, we'll consider $L = \mathbf{c} \cdot \nabla$, and we'll get the advection equation $u_t = \mathbf{c} \cdot \nabla u$. Suppose that $u(t, \cdot)$ is the analytic solution to $u_t = L u$. We'll use the dot \cdot as a non-specific placeholder. It could stand-in for some spatial dimensions if we are dealing with a partial differential equation. Or, it could be nothing at all if we have an ordinary differential equation, in which case u is simply a function of t . Let $t_n = nk$ be the uniform discretization of time, and let U^n denote the finite difference approximation of $u(t_n, \cdot)$. Define a finite difference operator H_k such that $U^{n+1} = H_k U^n$. The subscript k in H_k denotes the dependence on the step size k . Of course, $U^{n+1} = H_k^{n+1} U^0$. For a first-order differential equation, U^n and H_k are scalars. For a system of differential equations, U^n is a vector and H_k is a matrix.

Let's restate the definitions convergence, consistency and stability with more rigor. Let the error $e_k(t_n, \cdot) = U^n - u(t_n, \cdot)$. A method is *convergent* in a norm $\|\cdot\|$ if the error $\|e_k(t, \cdot)\| \rightarrow 0$ as $k \rightarrow 0$ for any $t \in [0, T]$ and any initial data $u(0, \cdot)$. Define the *truncation error* as

$$\tau_k(t, \cdot) = \frac{u(t+k, \cdot) - H_k u(t, \cdot)}{k}.$$

A method is *consistent* if $\|\tau_k(t, \cdot)\| \rightarrow 0$ as $k \rightarrow 0$. Furthermore, a method is consistent of order p if, for all sufficiently smooth initial data, there exists a constant c_τ such that $\|\tau_k(t, \cdot)\| \leq c_\tau k^p$ for all sufficiently small step sizes k . Define the operator norm $\|H_k\|$ as the relative maximum $\|H_k u\|/\|u\|$ over all vectors u . A method is *stable* if, for each time T , there exists a c_s and $k_s > 0$ such that $\|H_k^n\| \leq c_s$ for all $nk \leq T$ and $k \leq k_s$. In particular, if $\|H_k\| \leq 1$, then $\|H_k^n\| \leq \|H_k\|^n \leq 1$, which implies stability. But, we can also relax conditions on H_k to $\|H_k\| \leq 1 + \alpha k$ for any constant α , since in this case

$$\|H_k^n\| \leq \|H_k\|^n \leq (1 + \alpha k)^n \leq e^{\alpha kn} \leq e^{\alpha T}.$$

Theorem 47 (Lax equivalence). *A consistent, stable method is convergent.*

Proof. For a finite difference operator H_k , the truncation error at t_{n-1} is

$$\tau_k(t_{n-1}, \cdot) = \frac{u(t_n, \cdot) - H_k u(t_{n-1}, \cdot)}{k}.$$

We can write the exact solution as $u(t_n, \cdot) = H_k u(t_{n-1}, \cdot) + k \tau_k(t_{n-1}, \cdot)$. Using this expression, the error at t_n is

$$\begin{aligned} e_k(t_n, \cdot) &= U^n - u(t_n, \cdot) \\ &= H_k U^{n-1} - H_k u(t_{n-1}, \cdot) - k \tau_k(t_{n-1}, \cdot) \\ &= H_k e_k(t_{n-1}, \cdot) - k \tau_k(t_{n-1}, \cdot) \\ &= H_k [H_k e_k(t_{n-2}, \cdot) - k \tau_k(t_{n-2}, \cdot)] - k \tau_k(t_{n-1}, \cdot), \end{aligned}$$

and continuing the iteration

$$e_k(t_n, \cdot) = H_k^n e_k(0, \cdot) - k \sum_{j=1}^n H_k^{j-1} \tau_k(t_{j-1}, \cdot).$$

By the triangle inequality

$$\|e_k(t_n, \cdot)\| \leq \|H_k^n\| \cdot \|e_k(0, \cdot)\| + k \sum_{j=1}^n \|H_k^{j-1}\| \cdot \|\tau_k(t_{j-1}, \cdot)\|.$$

From the stability condition, we have $\|H_k^j\| \leq c_s$ for all $j = 1, 2, \dots, n$ and all $nk \leq T$. Therefore,

$$\|e_k(t_n, \cdot)\| \leq c_s \left(\|e_k(0, \cdot)\| + k \sum_{j=1}^n \|\tau_k(t_{j-1}, \cdot)\| \right).$$

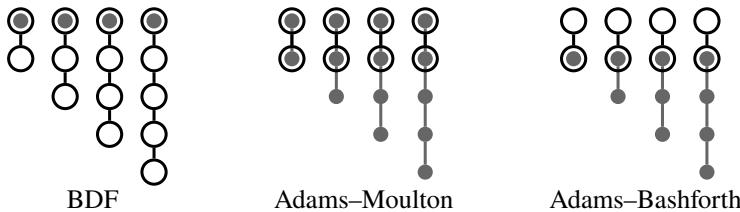
From the consistency condition, we have $\|\tau_k(t_{j-1}, \cdot)\| \rightarrow 0$ as $k \rightarrow 0$. And if the method is p th order consistent, then $\|\tau_k(t_{j-1}, \cdot)\| \leq c_\tau k^p$. So,

$$\|e_k(t_n, \cdot)\| \leq c_s (\|e_k(0, \cdot)\| + c_\tau T k^p).$$

If the initial error $\|e_k(0, \cdot)\| \leq c_0 k^p$, then $\|e_k(t_n, \cdot)\| \leq c_s (c_0 + c_\tau T) k^p$. Hence, the method is convergent. \square

12.4 Multistep methods

The forward and backward Euler methods both give first-order approximations to the solution by using piecewise linear functions. We can get a more accurate approximation to the derivative by using higher-order piecewise polynomials—piecewise quadratic, piecewise cubic, and so forth. For an n th-order polynomial approximation, we require n terms of the Taylor series and hence n steps in time. In this section, we examine general multistep methods. Stencils for the backward differentiation formulas (BDF), sometimes called Gear's method, and Adams methods are given on the following page. Note that the first-order BDF method is the same as the backward Euler method, the second-order Adams–Moulton method is the same as the trapezoidal method, and the first-order Adams–Bashforth method is the same as the forward Euler method.



► Backward differentiation formulas

We can get a better approximation to the derivative of $u(t)$ by canceling out more terms of the Taylor series. We've already derived a first-order *backward differentiation formula*, the backward Euler method. Let's derive a second-order implicit method by using a piecewise quadratic approximation for u . Consider the nodes $u(t_{n+1})$, $u(t_n)$, and $u(t_{n-1})$:

$$\begin{aligned} u(t) &= u(t) \\ u(t - k) &= u(t) - ku'(t) + \frac{1}{2}k^2u''(t) + O(k^3) \\ u(t - 2k) &= u(t) - 2ku'(t) + \frac{4}{2}k^2u''(t) + O(k^3) \end{aligned}$$

We want to cancel as many terms as possible by combining the three equations in some fashion. Let a_{-1} , a_0 , and a_1 be unknowns, then

$$\begin{aligned} a_{-1}u(t) &= a_{-1}u(t) \\ a_0u(t - k) &= a_0u(t) - a_0ku'(t) + \frac{1}{2}a_0k^2u''(t) + O(k^3) \\ a_1u(t - 2k) &= a_1u(t) - 2a_1ku'(t) + \frac{4}{2}a_1k^2u''(t) + O(k^3). \end{aligned}$$

We have three variables, so we are allowed three constraints. For *consistency*, we require the coefficient of $u(t)$ to be zero and the coefficient of $u'(t)$ to be one; for *accuracy*, we require the coefficient $u''(t)$ to be zero. Therefore

$$\begin{aligned} a_{-1} + a_0 + a_1 &= 0 \\ 0 - a_0 - 2a_1 &= k^{-1} \\ 0 + \frac{1}{2}a_0 + \frac{4}{2}a_1 &= 0. \end{aligned}$$

We solve this linear system for (a_{-1}, a_0, a_1) and we get

$$a_{-1} = \frac{3}{2}k^{-1}, \quad a_0 = -2k^{-1}, \quad a_1 = \frac{1}{2}k^{-1}.$$

So

$$\frac{\frac{3}{2}u(t_{n+1}) - 2u(t_n) + \frac{1}{2}u(t_{n-1})}{k} = u'(t_{n+1}) + O(k^2).$$

By using the this approximation for $u'(t)$, we obtain the second-order backward differentiation formula (BDF2)

$$\frac{3}{2}U^{n+1} - 2U^n + \frac{1}{2}U^{n-1} = kf(U^{n+1}).$$

An r -step (and r th order) backward differentiation formula takes the form

$$a_{-1}U^{n+1} + a_0U^n + \cdots + a_{n-r+1}U^{n-r+1} = kf(U^{n+1})$$

where the coefficients $a_{-1}, a_0, \dots, a_{n-r+1}$ are chosen appropriately.

Now, let's determine the region of absolute stability for BDF2. We take $f(u) = \lambda u$. Then letting $r = U^{n+1}/U^n$ in

$$\frac{3}{2}U^{n+1} - 2U^n + \frac{1}{2}U^{n-1} = kf(U^{n+1}),$$

we have

$$\frac{3}{2}r^2 - 2r + \frac{1}{2} = k\lambda r^2.$$

For which λk is $|r| \leq 1$? While we can easily compute this answer analytically using the quadratic formula, it becomes difficult or impossible for higher-order methods. So, instead, we will simply plot the boundary of the region of absolute stability. Since we want to know when $|r| \leq 1$, i.e., when r is inside the unit disk in the complex plane, we will consider $r = e^{i\theta}$. Then the variable λk can be expressed as the rational function

$$\lambda k = \frac{\frac{3}{2}r^2 - 2r + \frac{1}{2}}{r^2}.$$

We can plot this function in Julia using

```
r = exp.(2im*pi*(0:.01:1))
plot(@. (1.5r^2 - 2r + 0.5)/r^2)
```

The regions of absolute stability for backward differentiation formulas are *unbounded* and always contain the negative real axis (well, at least up through BDF6). See Figure 12.5 on the next page. This stability feature makes them particularly nice methods for the stiff problems we'll encounter later. Note that BDF3 and higher are not A-stable, because their regions of absolute stability do not contain the entire left half-plane. Such methods are often referred to as almost A-stable or *A(α)-stable*. A method is *A(α)-stable* if the region of stability contains the sector in the negative λk -plane between $\pi \pm \alpha$. The regions of absolute stability for the BDF methods get progressively smaller with increased accuracy, and BDF7 and higher are no longer zero-stable. This trade-off between stability and accuracy is typical but not necessary—higher-order Runge–Kutta methods are often more stable than their lower order counterparts.

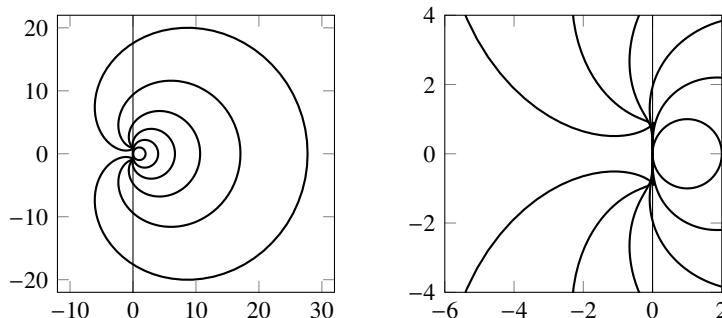


Figure 12.5: Left: internal contours of the regions of absolute stability for BDF1–BDF6. The regions of absolute stability are the areas outside these contours. Right: a close-up of the same contours.

► General multistep methods

The focus of BDF methods is entirely on the left-hand-side term u' . In designing a broader class of methods, we ought to think about the whole equation $u' = f(u)$ and not simply the left-hand side. A general s -step method may be written as

$$\sum_{j=-1}^{s-1} a_j U^{n-j} = k \sum_{j=-1}^{s-1} b_j f(U^{n-j})$$

where the coefficients a_j and b_j may be zero. When $b_{-1} = 0$, the method is explicit; when $b_{-1} \neq 0$, the method is implicit.

The *local truncation error* of the finite difference approximation is defined as

$$\tau_{\text{local}} = \sum_{j=-1}^{s-1} a_j u(t_{n-j}) - k \sum_{j=-1}^{s-1} b_j f(u(t_{n-j})),$$

and it is used to quantify the error for one time step. Error accumulates at each iteration. Because $t_n = nk$, it follows that $n = O(1/k)$. The *global truncation error* of a finite difference approximation is defined as $\tau_{\text{global}} = \tau_{\text{local}}/k$ and quantifies the error over several time steps. For example, the local truncation error of the leapfrog scheme is $O(k^3)$, and the global truncation error is $O(k^2)$.

From Taylor series expansion with $t_{n-j} = t_n - jk$, we have

$$\begin{aligned} u(t_{n-j}) &= u(t_n) - jku'(t_n) + \frac{1}{2}(jk)^2 u''(t_n) + \dots \\ f(u(t_{n-j})) &= u'(t_{n-j}) = u'(t_n) - jku''(t_n) + \frac{1}{2}(jk)^2 u'''(t_n) + \dots \end{aligned}$$

Then the local truncation error at t_n is

$$\begin{aligned} \sum_{j=-1}^{s-1} a_j u(t_n) - k \sum_{j=-1}^{s-1} (ja_j + b_j) u'(t_n) + k^2 \sum_{j=-1}^{s-1} \left(\frac{1}{2} j^2 a_j + jb_j\right) u''(t_n) - \dots \\ \dots - \frac{(-1)^p}{p!} k^p \sum_{j=-1}^{s-1} \left(j^p a_j + p j^{p-1} b_j\right) u^{(p)}(t_n) + \dots \end{aligned}$$

For a consistent method, the global truncation error must limit zero as the step size limits zero. Therefore,

$$\sum_{j=-1}^{s-1} a_j = 0 \quad \text{and} \quad \sum_{j=-1}^{s-1} ja_j + b_j = 0. \quad (12.6a)$$

To further get a method that is $O(k^p)$ accurate, we need to set the first $p+1$ coefficients of the truncation error to 0:

$$\sum_{j=-1}^{s-1} \left(j^i a_j + i j^{i-1} b_j\right) = 0 \quad \text{for } i = 1, 2, \dots, p. \quad (12.6b)$$

We can write this system in matrix form $\mathbf{A}\mathbf{a} + \mathbf{B}\mathbf{b} = 0$, where \mathbf{A} is the transpose of a Vandermonde matrix and \mathbf{B} is the transpose of a confluent Vandermonde matrix:

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & s \\ 1 & 2^2 & 3^2 & \cdots & s^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2^s & 3^s & \cdots & s^s \end{bmatrix} \begin{bmatrix} 1 \\ a_0 \\ a_1 \\ \vdots \\ a_{s-1} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 1 \\ 2 & 4 & 6 & \cdots & 2s \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s & s2^{s-1} & s3^{s-1} & \cdots & s^s \end{bmatrix} \begin{bmatrix} b_{-1} \\ b_0 \\ b_1 \\ \vdots \\ b_{s-1} \end{bmatrix} = 0.$$

This system can be solved with a few lines of code in Julia. The following function returns the multistep coefficients for stencils given by m and n :

```
function multistepcoeffs(m,n)
    s = length(m) + length(n)
    A = (m.+1).^(0:s-1)
    B = (0:s-1).*(n.+1).^(0:s-1).//(n.+1)
    c = -[A B]\ones(Int64,s)
    [1;c[1:length(m)]], c[(length(m)+1):end]
end
```

The input m is a row vector indicating nonzero (free) indices of $\{a_j\}$ and n is a row vector of nonzero indices of $\{b_j\}$. The value a_{-1} is always implicitly one and \emptyset should not be included in m . By formatting B as an array of rational numbers

using `//` and formatting the `ones` array as integers, the coefficients `c` will be formatted as rational numbers. For example, the input `m = [1]` and `n = [0 1 2]`, which corresponds to the third-order Adams–Moulton method, results in the two arrays `[1 -1]` and `[5/12 2/3 -1/12]`. We can use these coefficients to plot the boundary of the region of absolute stability:

```
function plotstability(a,b)
    λk(r) = (a . r.^{0:length(a)-1}) ./ (b . r.^{0:length(b)-1})
    r = exp.(im*LinRange(0,2π,200))
    plot(λk.(r),label="",aspect_ratio=:equal)
end
```

Suppose that we want to maximize the order of a multistep method. The system (12.6) has $s + 1$ equations with have $s + 1$ unknown and $s + 1$ prescribed variables. We might be tempted to try something like this

$$\begin{bmatrix} \mathbf{a}^\top \\ \mathbf{b}^\top \end{bmatrix} = \begin{bmatrix} 1 & a_1 & a_2 & \cdots & a_{s/2} & 0 & \cdots & 0 \\ b_0 & b_1 & b_2 & \cdots & a_{s/2} & 0 & \cdots & 0 \end{bmatrix}.$$

After all, it works for the trapezoidal method with $\mathbf{a} = (1, -1, 0)$ and $\mathbf{b} = (\frac{1}{2}, \frac{1}{2}, 0)$, giving a second-order, single-step method. But, in general, such an approach does not ensure the stability of the scheme. In 1956, Germund Dahlquist proved that the order of accuracy of a stable s -step linear multistep formula can be no greater than $s + 2$ if s is even, $s + 1$ if s is odd, and s if the formula is explicit. This theorem is now called the *first Dahlquist barrier*.¹

Multistep methods may require several starting values, i.e., to compute U^s for an s -step method, we first need U^0, U^1, \dots, U^{s-1} . For example, to implement the leapfrog method, first use the forward Euler method to compute U^1 . Then use the leapfrog after that. The local truncation error from the Euler method is $O(k^2)$ and the contribution to the global truncation error from that one step is $O(k^2)$. Therefore, the overall global error from the leapfrog method is $O(k^2)$.

► Adams methods

Whereas BDF methods have complicated approximations for the derivative on the left-hand side, Adams methods have simple left-hand sides and complicated right-hand sides. A BDF method evaluates the derivative at t_{n+1} to match the right-hand side. Adams methods rely on the mean value theorem to find an intermediate time somewhere in between t_n and t_{n+1} . To do this, they work by adjusting the right-hand side. The Adams methods have the form $a_{-1} = 1$ and $a_0 = -1$ and $a_j = 0$ for $1 \leq j \leq s - 1$. That is,

$$U^{n+1} = U^n + k \sum_{j=-1}^{s-1} b_j f(U^{n-j}).$$

¹In 1963, Dahlquist established his second barrier stating that no explicit scheme can be A-stable.

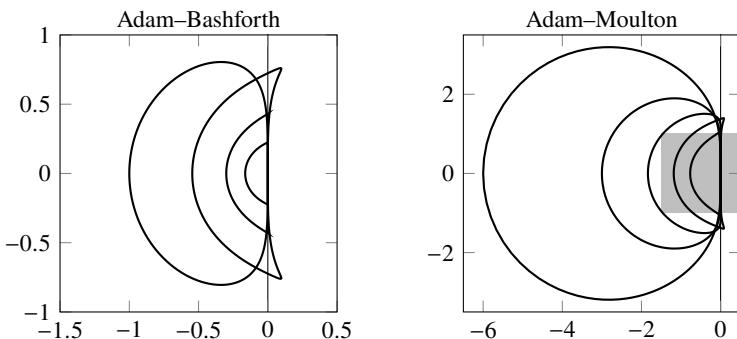


Figure 12.6: External contours of the regions of absolute stability for the explicit Adams–Bashforth methods AB2–AB5 (left) and implicit Adams–Moulton methods AM3–AM7 (right). The plot region of the Adams–Bashforth is shown with the gray rectangle overlaying the plot region of the Adams–Moulton.

The explicit *Adams–Bashforth method* takes $b_{-1} = 0$ and the implicit *Adams–Moulton method* takes $b_{-1} \neq 0$.

Another difference between Adams methods and BDF methods is their regions of absolute stability. Unlike the BDF methods which have unbounded regions, both the Adams–Bashforth and the Adams–Moulton methods have bounded regions. See Figure 12.6 above. That the region of absolute stability for an explicit method like the Adams–Bashforth is bounded is expected, but that the region of absolute stability of an implicit method like the Adams–Moulton method is bounded is a little disappointing, especially because the implementation of implicit methods require significantly more effort. Still, the region of absolute stability of an Adams–Moulton method is about three times larger than that of the corresponding Adams–Bashforth method.

► Predictor-corrector methods

Nonlinear implicit methods can be difficult to implement because they require an iterative nonlinear solver. One method is to use an explicit method for one time step to “predict” the implicit term and then use an implicit method to “correct” the solution of the explicit method. Adams methods are particularly well-suited

for combining as predictor-corrector methods:

$$\text{predictor: } \tilde{U}^{n+1} = U^n + k \sum_{j=0}^{s-1} b_j f(U^{n-j}) \quad (12.7a)$$

$$\text{corrector: } U^{n+1} = U^n + kb_0^* f(\tilde{U}^{n+1}) + k \sum_{j=0}^{s-1} b_j^* f(U^{n-j}). \quad (12.7b)$$

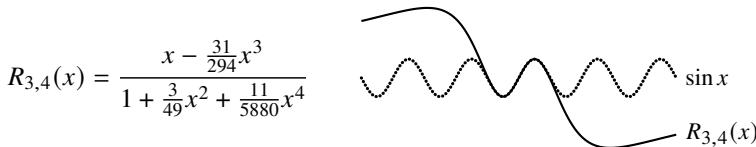
The two steps are often expressed as predict–evaluate–correct–evaluate (PECE). The corrector step can always be run a second or third or more times using its output as input to achieve better convergence. In this case, the method is often expressed as PE(CE)^m. If the corrector is run until convergence, PE(CE)[∞] is simply an implementation of the implicit method.

► Padé approximation

A *Padé approximation* is a rational-function extension to the Taylor polynomial approximation of an analytic function:

$$R_{m,n}(x) = \frac{P_m(x)}{Q_n(x)} = \frac{\sum_{i=0}^m p_i x^i}{1 + \sum_{i=1}^n q_i x^i}.$$

For example, the order-(3,4) Padé approximation to $\sin x$ is



When $n = 0$, the denominator $Q_n(r) = 1$ and the Padé approximation is identical to a Taylor polynomial. Linear multistep methods can be interpreted as Padé approximations to the logarithm function. In particular, $u' = \lambda u$ has the solution $u(t) = e^{\lambda t} u(0)$, and starting with U^n , the exact solution after time k is $U^{n+1} = e^{\lambda k} U^n$. Define the ratio $r = U^{n+1}/U^n = e^{\lambda k}$. If $r(\lambda k)$ is an approximant of the exponential function $e^{\lambda k}$, then the inverse $\lambda k(r)$ is approximant of the logarithm.

We can compute the Padé approximant of $\log r$ using a symbolic mathematics language like Wolfram.² For example, $R_{1,3}(x)$ is constructed using

```
m = 1; n = 3;
Simplify[PadeApproximant[Log[r], {r, 1, {m, n}}]]
```

²WolframCloud (<https://www.wolframcloud.com>) provides free, online access to Mathematica.

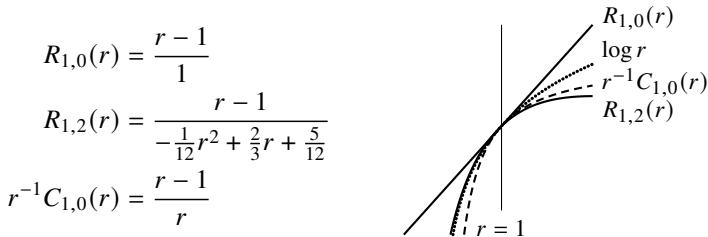


Figure 12.7: Padé approximant representations of the backward Euler, third-order Adams–Moulton, and forward Euler methods.

Another way to compute the Padé approximant to $\log r$ is by starting with the Taylor polynomial $T(x)$ of $\log(x+1)$; then, determining the coefficients of $P_m(x)$ and $Q_n(x)$ from $P_m(x) = Q_n(x)T(x) + O(x^{m+n+1})$; and finally, substituting $r - 1$ back in for x and grouping coefficients by powers of r . But, perhaps the easiest way to compute the Padé approximant is simply by solving the linear system outlined for general multistep methods using `multistepcoeffs` on page 343.

Implicit methods correspond to the Padé approximations of $\log r$. An order- p BDF methods corresponds to $R_{p,0}(r)$ and an order- p Adams–Moulton method corresponds to $R_{1,p-1}(r)$. Explicit methods are associated with $r^{-1}C_{m,n}(r)$, where $C_{m,n}(r)$ is the Padé approximations of $r \log r$. The leapfrog method is $r^{-1}C_{2,0}(r)$ and order- p Adams–Bashforth method corresponds to $r^{-1}C_{1,p-1}(r)$. The figure above shows the Padé approximants that correspond to the backward Euler, the third-order Adams–Moulton, and the forward Euler methods. Each of these rational functions approximates $\log r$ at $r = 1$ when $\lambda k = 0$.

12.5 Runge–Kutta methods

By directly integrating $u'(t) = f(u, t)$ we can change the differential equation into an integral equation

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(t, u(t)) dt.$$

The challenge now is to numerically evaluate the integral. One way of doing this is to use the Runge–Kutta method. Unlike multistep methods, Runge–Kutta methods can get high-order results without saving the solutions of previous time steps, making them good choices for adaptive step size. Because of this, Runge–Kutta methods are often methods of choice.

We can approximate the integral using quadrature

$$\int_{t_n}^{t_{n+1}} f(t, u(t)) dt \approx k \sum_{i=1}^s b_i f(t_i^*, u(t_i^*))$$

with appropriate weights b_i and nodes $t_i^* \in [t_n, t_{n+1}]$. In fact, by choosing the s nodes and weights so that we have a Gaussian quadrature, we can get a method that has order $2s$. We must already know the values $u(t_i^*)$ to apply this method. Such an approach seems circular, but of course, we can always use quadrature at intermediate stages to approximate each of the $u(t_i^*)$.

Example. The simplest quadrature rule uses a Riemann sum:

$$\int_a^b f(x) dx \approx (b - a)f(a).$$

Let's use this to compute U^{n+1} given U^n . From

$$u(t_{n+1}) = u(t_n) + \int_{t_n}^{t_{n+1}} f(t, u(t)) dt$$

we have $U^{n+1} = U^n + kf(t_n, U^n)$, which is the forward Euler method. ◀

Example. The midpoint rule is a more accurate one-point quadrature rule

$$\int_a^b f(x) dx \approx (b - a)f\left(\frac{b + a}{2}\right).$$

This time we'll need to approximate $f(t_{n+1}/2, U^{n+1/2})$ —we can use the forward Euler method. Take $K_1 = f(t_n, U^n)$ and

$$K_2 = f(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kf(t_n, U^n)) = f(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kK_1).$$

The midpoint rule says

$$u(t_{n+1}) = u(t_n) + \int_{t_n}^{t_{n+1}} f(t, u(t)) dt = u(t_n) + kf(t_{n+1}/2, u(t_{n+1}/2))$$

from which we have $U^{n+1} = U^n + kK_2$. This second-order method is known as Heun's method. ◀

Example. The trapezoidal rule is another one-point quadrature rule

$$\int_a^b f(x) dx \approx (b-a) \left[\frac{1}{2}f(a) + \frac{1}{2}f(b) \right].$$

To compute U^{n+1} given U^n , we'll need to approximate $f(t_{n+1}, U^{n+1})$ using the forward Euler method. Take $K_1 = f(t_n, U^n)$ and

$$K_2 = f(t_n + k, U^n + kf(t_n, U^n)) = f(t_n + k, U^n + kK_1).$$

Combining these we have a second-order rule $U^{n+1} = U^n + k \left[\frac{1}{2}K_1 + \frac{1}{2}K_2 \right]$. ▶

Example. To get high-order methods, we add quadrature points. The next step up is Simpson's rule, which says

$$\int_a^b f(x) dx \approx (b-a) \left[\frac{1}{6}f(a) + \frac{2}{3}f\left(\frac{b+a}{2}\right) + \frac{1}{6}f(b) \right].$$

We'll need to approximate $f(t_{n+1/2}, U^{n+1/2})$ and $f(t_{n+1}, U^{n+1})$. ▶

An s -stage Runge–Kutta method is given by

$$U^{n+1} = U^n + k \sum_{i=1}^s b_i K_i \quad \text{with} \quad K_i = f\left(t_n + c_i k, U^n + k \sum_{j=1}^s a_{ij} K_j\right) \quad (12.8)$$

In general, computing the coefficients a_{ij} is not a trivial exercise. For consistency, we need to take $c_i = \sum_{j=1}^s a_{ij}$ and $\sum_{i=1}^s b_i = 1$. These coefficients are often conveniently given as a *Butcher tableau*

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array}$$

where $[\mathbf{A}]_{ij} = a_{ij}$. If \mathbf{A} is a strictly lower triangular matrix, then the Runge–Kutta method is explicit. Otherwise, the method is implicit. If \mathbf{A} is a lower triangular matrix, then the method is a diagonally implicit Runge–Kutta Method (DIRK).

The second-order Heun method (RK2) is given by

$$\begin{array}{c|c} & \begin{matrix} 0 & \\ \hline \end{matrix} \\ \begin{matrix} K_1 = f(t_n, U^n) \\ K_2 = f(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kK_1) \\ U^{n+1} = U^n + kK_2 \end{matrix} & \begin{matrix} \frac{1}{2} & \frac{1}{2} \\ \hline 0 & 1 \end{matrix} \end{array}$$

The Butcher tableau of the trapezoidal rule is

$$\begin{array}{l}
 K_1 = f(t_n, U^n) \\
 K_2 = f(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kK_1) \\
 U^{n+1} = U^n + k(\frac{1}{2}K_2 + \frac{1}{2}K_1)
 \end{array}
 \quad \left| \begin{array}{c|cc}
 0 & 0 \\
 1 & 0 & 1 \\
 \hline
 \frac{1}{2} & \frac{1}{2} & \frac{1}{2}
 \end{array} \right.$$

The trapezoidal rule is a DIRK method with an explicit first stage. The fourth-order Runge–Kutta (RK4), sometimes called the classical Runge–Kutta method or simply “the” Runge–Kutta method, is derived from Simpson’s rule.

$$\begin{array}{l}
 K_1 = f(t_n, U^n) \\
 K_2 = f(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kK_1) \\
 K_3 = f(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kK_2) \\
 K_4 = f(t_n + k, U^n + kK_3) \\
 U^{n+1} = U^n + \frac{1}{6}k(K_1 + 2K_2 + 2K_3 + K_4).
 \end{array}
 \quad \left| \begin{array}{c|cccc}
 0 & 0 & & & \\
 \frac{1}{2} & \frac{1}{2} & & & \\
 \frac{1}{2} & 0 & \frac{1}{2} & & \\
 1 & 0 & 0 & 1 & \\
 \hline
 \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} &
 \end{array} \right.$$

Example. Plot the regions of stability for the Runge–Kutta methods given by the following tableaus:

$$\left| \begin{array}{c|cc}
 0 & 0 & \\
 \hline
 1 & 1 &
 \end{array} \right. \quad \left| \begin{array}{c|cc}
 0 & \frac{1}{2} & \\
 \hline
 1 & -1 & 2 \\
 \hline
 \frac{1}{6} & \frac{2}{3} & \frac{1}{6}
 \end{array} \right. \quad \left| \begin{array}{c|ccc}
 0 & \frac{1}{2} & \frac{1}{2} & & \\
 \hline
 \frac{1}{2} & 0 & \frac{1}{2} & & \\
 \frac{1}{2} & 0 & 0 & 1 & \\
 1 & 0 & 0 & 1 & \\
 \hline
 \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} &
 \end{array} \right.$$

The region of absolute stability is bounded by the $|r| = 1$ contour, where $r = U^{n+1}/U^n$ and U^n is the solution to $u' = \lambda u$. For a Runge–Kutta method we can get an explicit expression for λk in terms of r . For $f(t, u) = \lambda u$, a general Runge–Kutta method is by (12.8)

$$U^{n+1} = U^n + k \mathbf{b}^T \mathbf{K},$$

with \mathbf{K} given implicitly by

$$\mathbf{K} = \lambda(U^n \mathbf{E} + k \mathbf{A} \mathbf{K}) \tag{12.9}$$

where \mathbf{K} is an $s \times 1$ vector, \mathbf{b}^T is a $1 \times s$ vector, \mathbf{A} is an $s \times s$ matrix, and \mathbf{E} is an $s \times 1$ vector of ones. We solve (12.9) for \mathbf{K} to get

$$\mathbf{K} = \lambda U^n (\mathbf{I} - \lambda k \mathbf{A})^{-1} \mathbf{E}$$

where \mathbf{I} is an $s \times s$ identity matrix. Then

$$r = 1 + \lambda k \mathbf{b}^T (\mathbf{I} - \lambda k \mathbf{A})^{-1} \mathbf{E}. \tag{12.10}$$

We can use Julia to compute the $|r| = 1$ contours of (12.10) for each tableau. The regions of absolute stability are plotted in Figure 12.8 on the facing page. ◀

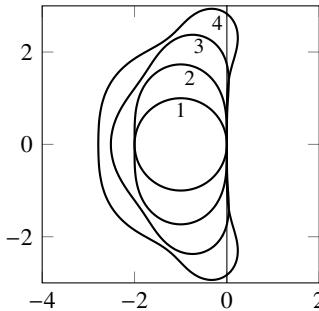


Figure 12.8: External contours of the regions of absolute stability for the explicit Runge–Kutta methods RK1–RK4. The regions of stability get progressively *larger* with increasing order.

Example. We can build a second-order Runge–Kutta method by combining the trapezoidal method with BDF2 in two stages, first taking the trapezoidal method for half a time step followed by the BDF method for the remaining half time step:

$$\begin{aligned} U^{n+1/2} &= U^n + \frac{1}{4}k(f(U^{n+1/2}) + f(U^n)) \\ U^{n+1} &= \frac{4}{3}U^{n+1/2} - \frac{1}{3}U^n + \frac{1}{3}kf(U^{n+1}) \end{aligned}$$

Substituting the first equation into the second gives us

$$U^{n+1} = U^n + \frac{1}{3}k(f(U^{n+1}) + f(U^{n+1/2}) + f(U^n))$$

from which we have the DIRK method.

$$\begin{array}{l} K_1 = f(U^n) \\ K_2 = \frac{1}{4}f(U^n) + \frac{1}{4}f(U^{n+1/2}) \\ K_3 = \frac{1}{3}f(U^n) + \frac{1}{3}f(U^{n+1/2}) + \frac{1}{3}f(U^{n+1}) \\ U^{n+1} = U^n + k\left(\frac{1}{3}K_1 + \frac{1}{3}K_2 + \frac{1}{3}K_3\right) \end{array} \quad \begin{array}{c|ccc} 0 & & & \\ \hline \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & \\ 1 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & \end{array}$$

Instead of breaking the time step into two equal subintervals, we could also take the intermediate point at some fraction α of the time step. In this case, we take the trapezoidal method over a subinterval αk and the BDF2 method over the two subintervals αk and $(1 - \alpha)k$ with $0 < \alpha < 1$. Finding an optimal $\alpha = 2 - \sqrt{2}$ is left as an exercise. \blacktriangleleft

Runge–Kutta methods can seem complicated at first glance. Because of this, it is good to develop an intuition about how they work. For a unit step size, we can interpret c_i as nodes along the interval $[0, 1]$. The terms K_i are the

slope of $u(t)$ at the node. Runge–Kutta methods can be viewed as successive approximations combined together at each stage to get a more accurate solution. See Figure 12.9 on the next page.

► Adaptive step size

One advantage that Runge–Kutta methods have over multistep methods is the relative ease to which the step size can be varied, taking smaller steps when needed to minimize error and maintain stability and larger steps otherwise. Often, a Runge–Kutta method embeds a lower-order method inside a higher-order method, sharing the same Butcher tableau. For example, the Bogacki–Shampine method (implemented in Julia using the function `BS3`) combines a second-order and third-order Runge–Kutta method using the same quadrature points:

	0			
$\frac{1}{2}$		$\frac{1}{2}$		
$\frac{3}{4}$		0	$\frac{3}{4}$	
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	
	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	0
	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

The K_i are the same for the second-order and the third-order methods. By subtracting the solutions we can approximate the truncation error. If the truncation error is above a given tolerance, we set the step size k to $k/2$. If the truncation error is below another given tolerance, we set the step size k to $2k$.

Occasionally, the notation $RKm(n)$ is used to describe a Runge–Kutta method where m is the order of the method to obtain the solution and n is the order of the method to obtain the error estimate. This notation isn't consistent and some authors instead use (m, n) or $n(m)$. At other times a method may have multiple error estimators. For example, the $8(5,3)$ uses a fifth-order error estimator and bootstraps a third-order estimator.

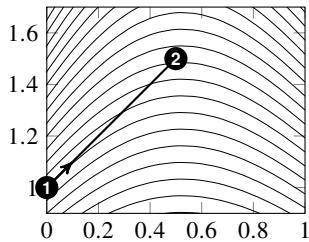
► Implicit Runge–Kutta methods

Another, modern approach to Runge–Kutta methods is through collocation. Collocation is a method of problem-solving that chooses a lower-dimensional subspace for candidate solutions, often polynomials of a specified degree, along with a set of collocation points. The desired solution is the one that satisfies the problem exactly at the collocation points. With s carefully chosen points, Gaussian quadrature exactly integrates a polynomial of degree $2s - 1$. For an ODE, we define the s -degree collocation polynomial $u(t)$ that satisfies

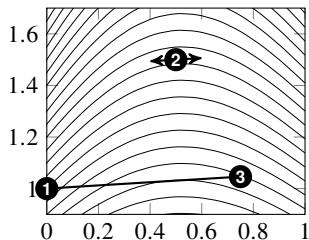
$$u'(t_0 + c_i k) = f(t_0 + c_i k, u(t_0 + c_i k)) \quad (12.11)$$

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{3}{4}$	0	$\frac{3}{4}$		
$\frac{4}{9}$		$\frac{1}{3}$	$\frac{4}{9}$	
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	
	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

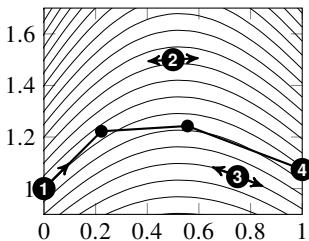
The Bogacki–Shampine scheme given by the Butcher tableau on the left is implemented in Julia using BS3. It can be implemented as an adaptive step size routine.



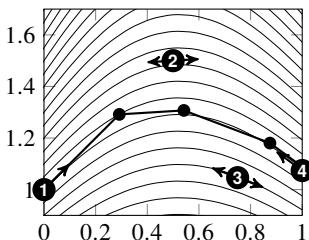
We start with the second row of the Butcher tableau. Using the slope $f(0, u(0))$ at ① we find the approximate solution at $t = \frac{1}{2}$ by using a simple forward Euler approximation to arrive at ②.



The third row of the Butcher tableau says we now try to find the solution at $t = \frac{3}{4}$ using the slope we just found at ② and nothing from ①. Again, we use a simple linear extrapolation.



The fourth row says we now try to find the solution at $t = 1$ by first traveling $\frac{2}{9}$ with a slope given by ①, then a distance $\frac{1}{3}$ with a slope given by ②, and then a distance $\frac{4}{9}$ with slope from ③.



The final row tells us to first travel $\frac{7}{24}$ with a slope from ①, then $\frac{1}{4}$ with a slope from ②, then $\frac{1}{3}$ with a slope from ③, and finally $\frac{1}{8}$ with a slope from ④. The solution provides a third-order correction over the solution from ④.

Figure 12.9: Runge–Kutta methods can be viewed as successive approximations combined together at each stage to get a more accurate solution. In this example, we take step size $k = 1$ with $u(0) = 1$.

for $i = 1, 2, \dots, s$ where $u(t_0) = u_0$ and the collocation points $c_i \in [0, 1]$. The solution is given by $u(t_0 + k)$. For example, when $s = 1$, the polynomial is

$$u(t) = u_0 + (t - t_0)K_1 \quad \text{where} \quad K_1 = f(t_0 + c_1 k, u_0 + c_1 K_1 k).$$

When $c_1 = 0$ we have the explicit Euler method, when $c_1 = 1$ we have the implicit Euler method, and $c_1 = \frac{1}{2}$ we have the implicit midpoint method:

$$\begin{array}{c|c} 0 & 0 \\ \hline 1 & \\ \hline \end{array} \quad \begin{array}{c|c} 1 & 1 \\ \hline 1 & \\ \hline \end{array} \quad \begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline 1 & \\ \hline \end{array}.$$

Theorem 48. *An s -point collocation method is equivalent to an s -stage Runge–Kutta method (12.8) with coefficients*

$$a_{ij} = \int_0^{c_i} \ell_j(z) dz \quad \text{and} \quad b_i = \int_0^1 \ell_i(z) dz,$$

where $\ell_i(z)$ is the Lagrange polynomial basis function for a node at c_i .

Proof. Let $u(t)$ be the collocation polynomial and let $K_i = u'(t_0 + c_i k)$. The Lagrange polynomial

$$u'(t_0 + zk) = \sum_{j=1}^s K_j \ell_j(z) \quad \text{where} \quad \ell_i(z) = \prod_{\substack{l=0 \\ l \neq i}}^s \frac{z - c_l}{c_i - c_l}.$$

Integrating with respect to z over $[0, c_i]$ gives us

$$u(t_0 + c_i k) = u_0 + k \sum_{j=1}^s K_j \int_0^{c_i} \ell_j(z) dz = u_0 + k \sum_{j=1}^s a_{ij} K_j.$$

Similarly, integrating over $[0, 1]$ gives us $u(t_0 + k) = u_0 + k \sum_{j=1}^s b_j K_i$. Substituting both of these expressions into the collocation method (12.11) gives us the Runge–Kutta method (12.8). \square

A Gauss–Legendre method is an implicit Runge–Kutta method that uses collocation based on Gauss–Legendre quadrature points from the roots of shifted Legendre polynomials. From theorem 43, the error in an n -point Gauss–Legendre quadrature is

$$\frac{f^{(2n)}(\xi)}{(2n)!} \int_a^b \prod_{i=0}^{s-1} (x - x_i) dx$$

for nodes x_0, x_1, \dots, x_{s-1} and for some $\xi \in [a, b]$. Over an interval of length k , the integral is bounded by k^{2s+1} . So, it follows that the local truncation error

is $O(k^{2s+1})$ and the global truncation error is $O(k^{2s})$. Therefore, an s -stage Gauss–Legendre method is order $2s$.

Radau methods are Gauss–Legendre methods that include one of the endpoints as quadrature points—either $c_1 = 0$ (Radau IA methods) or $c_s = 1$ (Radau IIA methods). Such methods have maximum order $2s - 1$. Gauss–Lobatto quadrature chooses the quadrature points that include both ends of the interval. Lobatto methods (Lobatto IIA) have both $c_1 = 0$ and $c_s = 1$ and have maximum order $2s - 2$.

Implicit Runge–Kutta methods offer higher accuracy and greater stability profiles, but they are more difficult to implement because they require solving a system of nonlinear equations. Diagonally implicit Runge–Kutta methods limit the Runge–Kutta method (12.8) to one implicit term

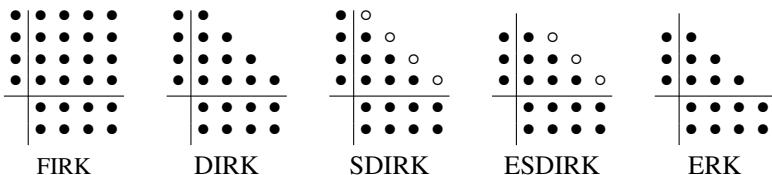
$$K_i = f\left(U^n + k \sum_{j=1}^i a_{ij} K_j\right).$$

We can solve these equations successively by applying Newton’s method at each stage to handle the one implicit term. We can also go one step further. By linearizing the equation with respect to the implicit term at each stage, we would only need to invert a linear operator. Such an approach, called a Rosenbrock method, is effectively the same as performing one Newton step for each stage. We lose some stability and accuracy but gain speed. In practice, the Rosenbrock method is

$$K_i = f\left(U^n + k \sum_{j=1}^{i-1} a_{ij} K_j\right) + k \mathbf{J} \sum_{j=1}^i d_{ij} K_j$$

where the Jacobian $\mathbf{J} = f'(U^n)$ and the coefficients d_{ij} that arise out of linearization need to be determined.

One can assemble an impressive bestiary of acronyms that have been used to classify implicit Runge–Kutta methods. The figure below summarizes the Butcher tableau structures for common Runge–Kutta methods: fully implicit (FIRK); diagonally implicit (DIRK); single-diagonal-coefficient, diagonally implicit (SDIRK); explicit-first-stage, single-diagonal-coefficient, diagonally implicit (ESDIRK); and explicit (ERK). The glyph \bullet indicates values and the glyph \circ indicates identical non-zero values.



In general, stability, accuracy, and difficulty and computational cost of implementation decrease from left to right. Implicit methods are often constructed to be

stiffly accurate requiring a Newton solver. Because the diagonal elements of an SDIRK method are identical, the Jacobian matrix may be reused in the Newton iterations. To dig deeper, see the review article by Kennedy and Carpenter.

12.6 Nonlinear equations

Up to now, we've focused on linear differential equations, which can be often be solved analytically. In this section, we briefly discuss nonlinear differential equations, starting with the stability condition. For linear differential equations, we used the eigenvalue λ along with the region of absolute stability of the numerical method to determine a stability condition for the step size. If we linearize the right-hand side of a nonlinear differential equation $u' = f(u)$ near a value u^* , we have $f(u) \approx f(u^*) + (u - u^*)f'(u^*)$. Our linearized equation near u^* is

$$u' = f'(u^*)u + (f(u^*) - f'(u^*)u^*).$$

This says that we should examine $\lambda = |f'(u^*)|$ as our eigenvalue to determine stability and step size. For a system $\mathbf{u}' = \mathbf{f}(\mathbf{u})$ we have the linearization near a value \mathbf{u}^*

$$\mathbf{u}' = \nabla\mathbf{f}(\mathbf{u}^*)\mathbf{u} + (\mathbf{f}(\mathbf{u}^*) - \nabla\mathbf{f}(\mathbf{u}^*)\mathbf{u}^*),$$

where $\nabla\mathbf{f}(\mathbf{u}^*)$ is the Jacobian matrix $\partial f_i / \partial u_j$ evaluated at \mathbf{u}^* . The spectral radius of the Jacobian matrix (the magnitude of its largest eigenvalue) now determines the stability condition.

Implicit methods are more difficult to implement, but they are more stable. When the system of differential equations is linear, this often means inverting that system using Gaussian elimination, discrete Fourier transforms, or some other relatively direct method. For large, sparse systems we may opt for an iterative technique like SOR. Implementing implicit methods on a nonlinear system almost always means that we will need to use an iterative solver. Perhaps the simplest iterative solver is functional iteration. Here, one time step of an implicit method like the Adams–Moulton method

$$U^n = U^{n-1} + k \sum_{i=0}^s b_i f(U^{n-i})$$

is replaced with a fixed-point iteration

$$U^{n(j+1)} = kb_0 f(U^{n(j)}) + U^{n-1} + k \sum_{i=1}^s b_i f(U^{n-i}),$$

initially taking $U^{n(0)} = U^{n-1}$ and iterating until convergence. From theorem 22, a fixed-point iteration $u^{(j+1)} = \phi(u^{(j)})$ only converges if $\phi(u^{(j)})$ is a contraction mapping, i.e., if $|\phi'(u^{(j)})| < 1$. This requirement itself sets a stability condition

on any functional iteration method. Namely, for the Adams–Moulton method with functional iteration, we will need to take the time step $k < |b_0 f'(U^{n(j)})|^{-1}$. This approach would be ill-suited for stiff problems, which we'll discuss in the next system. For such problems, Newton's method is preferred.

Newton's method solves $\phi(u) = 0$ by taking

$$u^{(j+1)} = u^{(j)} - [\phi'(u^{(j)})]^{-1} \phi(u^{(j)})$$

where $\phi'(u^{(j)})$ is the derivative or Jacobian matrix of $\phi(u)$. The Jacobian matrix can be calculated analytically or numerically using a finite difference approximation. We don't necessarily need to compute the Jacobian matrix at every iteration within a time step. The modified Newton's method

$$u^{(j+1)} = u^{(j)} - [\phi'(u^{(0)})]^{-1} \phi(u^{(j)})$$

only calculates the Jacobian matrix at the start of each time step and then reuses it for each iteration.

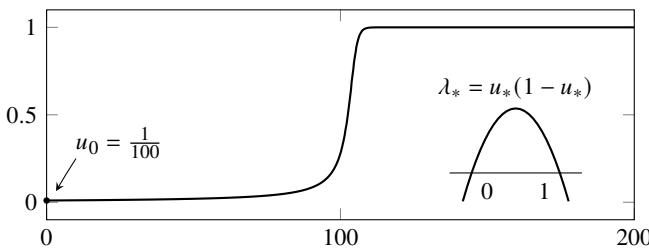
Both Newton's method and fixed-point iteration may not converge to the correct solution if the initial guess is not sufficiently close to the target. Therefore, explicit method is sometimes used to get a sufficiently close guess and then an implicit method is used to get even closer for each time step. Such methods are called predictor-corrector methods and often use a combination of Adams–Bashforth and Adams–Moulton methods.

12.7 Stiff equations

A differential equation is said to be *stiff* if the dynamics of the solution react on at least two largely different time scales. As we shall see in the next chapter, diffusion equations are stiff. The solution evolves quickly in the presence of a large spatial gradient and then slows to a crawl once it smooths out. Because numerical errors often result in large gradients, the problem remains stiff even after the fast scale is no longer apparent in the solution dynamics.

Example. In his book *Numerical Computing with MATLAB*, Cleve Moler introduces a simple model of combustion, which he credits to Larry Shampine.³ Imagine striking a match. The ball of flame expands quickly until the amount of oxygen available through the surface balances the amount of oxygen consumed in combustion. Let $u(t)$ be the radius of the ball of flame. The surface area is proportional to u^2 and the volume is proportional to u^3 . Therefore, we can write the toy model as $u' = u^2 - u^3$ where with $u(0) = u_0$. By taking an initial condition of $u_0 = \frac{1}{100}$, we have the following dynamics:

³Shampine himself credits Robert O'Malley, who himself credits Edward Reiss. An explicit solution, given in terms of a Lambert W function, can be found in an article by Corless et al.



The equation has an unstable equilibrium at $u = 0$ and a stable equilibrium at $u = 1$. The solution evolves slowly when $u(t)$ is near zero until $t \approx 1/u_0$. At this point, the solution rapidly rises to just below 1. Then, the change in the solution slows and asymptotically approaches 1. To get a better idea of what's going on in the nonlinear problem, consider the quasilinear form $u' = \lambda_* u$ with $\lambda_* = u_*(1 - u_*)$ for a value u_* . When u_* is near zero, λ_* is close to zero. As u_* increases, λ_* increases to a maximum of $\frac{1}{4}$ at $u_* = \frac{1}{2}$ and then decreases back to zero as u_* approaches one.

Let's compare solutions using the adaptive Dormand–Prince Runge–Kutta method (RK45) and the adaptive fifth-order BDF methods over the interval $t \in [0, 200]$ starting with $u_0 = 0.999$:



While the dynamics of the equation near $u = 1$ are relatively benign, we need to take a lot of tiny steps using the Runge–Kutta method—124 in total. But, the BDF method takes just 11. And almost all of these steps are early on—the last step is an enormous leap of $k = 157$. Both methods adjust the step size to ensure stability. To determine the stability conditions for a numerical method, we examine the derivative of the right-hand side $f(u) = u^2(1 - u)$, which is $f'(u) = 2u - 3u^2$. Specifically, when $u(t) \approx 1$, the derivative $f'(u) \approx -1$. The lower limit of the region of absolute stability of the Dormand–Prince Runge–Kutta method is -3 . If we use this method, we can take steps as large as $k = 3$ near $u = 1$ and still maintain absolute stability—the solver took an average step size of 1.6. On the other hand, if we use the BDF scheme, which has no lower limit on its region of absolute stability, we can take as large a step as we'd like. \blacktriangleleft

Chemists Charles Curtiss and Joseph Hirschfelder first referenced “stiff” equations in their 1952 article on the topic. They motivated their discussion describing free radicals that are rapidly created and destroyed when compared to an over-all chemical reaction. The dynamics of the equations occur faster than the desired resolution of the solution, but while these dynamics are not a point of interest, they nonetheless drive the behavior of the numerical solution.

Seventy years after Curtiss and Hirschfelder introduced the notion of stiffness, mathematicians still have not settled on its definition.⁴

Like the Reiss combustion model, stiff equations often have a quickly decaying transient solution and an equilibrium solution. A linear system of equations $\mathbf{u}' = \mathbf{A}\mathbf{u}$ is stiff if the eigenvalues all lie in the left half-plane and the ratio of the magnitudes of the real parts of the largest and smallest eigenvalues is large. A system of equation $\mathbf{u}' = \mathbf{f}(\mathbf{u})$ is stiff if the same holds for the eigenvalues of its Jacobian matrix $\mathbf{A} = \nabla \mathbf{f}(\mathbf{u})$. This *stiffness ratio*,

$$\kappa = \frac{\max |\operatorname{Re} \lambda(\mathbf{A})|}{\min |\operatorname{Re} \lambda(\mathbf{A})|},$$

is analogous to the condition number of a matrix.

Stiffness is also a property of stability. Ernst Hairer and Gerhard Wanner, in their treatise on stiff differential equations, state simply, “Stiff equations are problems for which explicit methods don’t work.” Using an explicit method on a stiff problem is like running on cobblestones. You might want to run quite quickly and carefree and maybe you think that you can, but a slight misstep will trip you up and send you flying onto the pavement.

Example. Consider the behavior of $u' = \lambda(\sin t - u) + \cos t$ where $u(0) = u_0$ and λ is a large, positive value. The solution $u(t) = u_0 e^{-\lambda t} + \sin t$ evolves over two time scales—a slow time scale of $\sin t$ and a fast time scale of $u_0 e^{-\lambda t}$.



Take $\lambda = 500$ and $u_0 = 0$. The differential equation has the simple solution $u(t) = \sin t$. Over the interval $t \in [0, 20]$ an adaptive fifth-order BDF method takes 102 time steps. Over the same interval, an adaptive fifth-order Runge–Kutta method takes 3022 steps—roughly 30 times as many as the BDF! ◀

Because stiff equations have relatively large negative eigenvalues, implicit solvers are natural choices for them. In particular, an A-stable method such as the backward Euler or trapezoidal method will ensure stability no matter where the eigenvalues lie in the left-half plane. But, for stiff problems, A-stability is usually not enough. While the trapezoidal method is unconditionally stable in the left-half plane, it typically produces transient oscillations that overshoot and overcorrect along the path of exponential decay for large, negative eigenvalues.

⁴Söderlind, Jay, and Calvo’s review article “Stiffness 1952–2012: Sixty years in search of a definition” addresses the lack of consensus on an adequate definition.

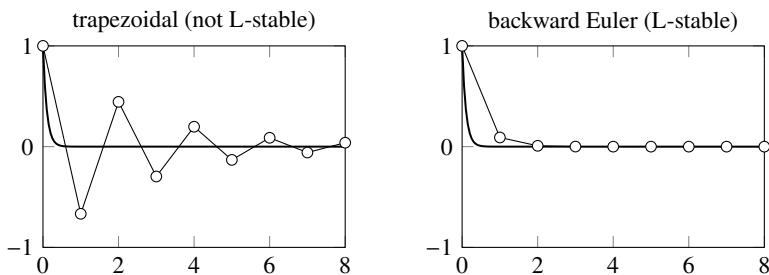


Figure 12.10: The numerical solutions to $u' = -10u$ using the second-order trapezoidal method and first-order backward Euler method with step size $k = 1$.

Consider the initial value problem $u' = -\lambda u$ for a positive factor λ . The trapezoidal method $U^{n+1} - U^n = -\frac{1}{2}\lambda k (U^{n+1} + U^n)$ can be written as

$$U^{n+1} = \left(\frac{1 - \frac{1}{2}\lambda k}{1 + \frac{1}{2}\lambda k} \right)^n U^0 = (r(\lambda k))^n U^0.$$

Because the trapezoidal method is A-stable, we can take a large time step size k and still ensure stability. But when λk is large, the multiplying factor $r(\lambda k) \approx -1$. Consequently, the numerical solution $U^n \approx (-1)^n U^0$ will oscillate for a long time as it slowly decays. See the figure on the current page.

The backward Euler method $U^n - U^{n-1} = -\lambda k U^n$ can be written as

$$U^n = \left(\frac{1}{1 + \lambda k} \right)^n U^0 = (r(\lambda k))^n U^0.$$

Now, when λk is large, the multiplying factor $r(\lambda k) \approx 0$, and the numerical solution decays quickly.

We say that a method is *L-stable* if it is A-stable and $r(\lambda k) \rightarrow 0$ as $\lambda k \rightarrow -\infty$. We can weaken the definition to say that a method is *almost L-stable* if it is almost A-stable and $|r(\lambda k)| < 1$ as $\lambda k \rightarrow -\infty$. In other words, a method is almost L-stable if it is zero-stable and the interior of its region of absolute stability contains the point $-\infty$. Backward differentiation formulas BDF1 and BDF2 are L-stable and BDF3–BDF6 are almost L-stable. Several third- and fourth-order DIRK and Rosenbrock methods are also L-stable.

12.8 Splitting methods

Often, PDEs have stiff, linear terms coupled with non-stiff, nonlinear terms. Examples include

Navier–Stokes equation	$\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla) \mathbf{v} = -\frac{1}{\rho} \nabla p + \nu \Delta \mathbf{v} + \mathbf{g}$
reaction-diffusion equation	$\frac{\partial u}{\partial t} = \Delta u + u(1 - u^2)$
nonlinear Schrödinger equation	$i \frac{\partial \psi}{\partial t} = -\frac{1}{2} \Delta \psi + \psi ^2 \psi$

The stiff, linear terms are best treated implicitly, while the nonlinear terms are best treated explicitly to simplify implementation. Such an approach is called *splitting*.

► Operator splitting

Let's examine the stiff, nonlinear differential equation

$$u' = \cos^2 u - 10u \quad \text{with } u(0) = u_0. \quad (12.12)$$

We can write the problem as $u' = N u + L u$, where $N u = \cos^2 u$ and $L u = -10u$. The solution u is being forced by $N u$ and $L u$ *concurrently*. We can't solve this problem analytically, but we can approximate the solution for small times by considering $N u$ and $L u$ as acting *successively*. We can do this in two ways:

1. First solve $u' = N u$. Then use the solution as initial conditions of $u' = L u$.
2. First solve $u' = L u$. Then use the solution as initial conditions of $u' = N u$.

Let's evaluate each of these approximate solutions after a time k .

1. The solution to $u' = \cos^2 u$ with initial condition u_0 is $u(t) = \tan^{-1}(t + u_0)$.
The solution to $u' = -10u$ with initial condition $\tan^{-1}(k + u_0)$ is

$$u(k) = e^{-10k} \tan^{-1}(k + \tan u_0).$$

2. The solution to $u' = -10u$ with initial condition u_0 is $u(t) = e^{-10t} u_0$. The solution to $u' = \cos^2 u$ with initial condition $e^{-10k} u_0$ is

$$u(k) = \tan^{-1}(k + \tan(e^{-10k} u_0)).$$

A natural question to ask is “how close are these approximations to the exact solution?” Both of these solutions are plotted along curves ❶ and ❸ in Figure 12.11 on the following page.

Suppose we are given the problem $u' = A u + B u$ where A and B are arbitrary operators. Consider a simple splitting where we alternate by solving $u' = A u$ and then solving $u' = B u$ each for one time step k . We can write and implement this procedure as $U^* = S_1(k) U^n$ followed by $U^{n+1} = S_2(k) U^*$, or equivalently as $U^{n+1} = S_2(k) S_1(k) U^n$, where $S_1(k)$ and $S_2(k)$ are solution operators.

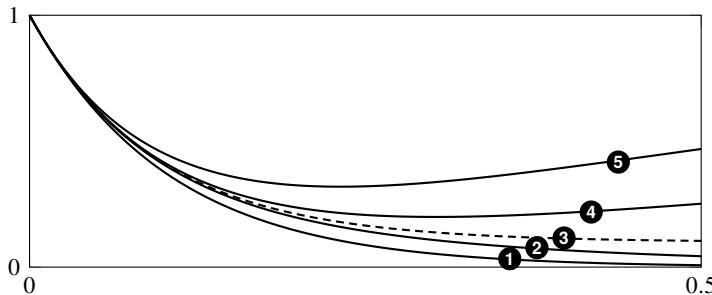


Figure 12.11: Solution to (12.12) using general splitting in ① and ⑤, Strang splitting in ② and ④, and exactly in ③. Each operator is inverted without numerical error over a time step $k = 0.5$.

What is the splitting error of applying the operators A and B successively rather than concurrently? If $S_1(k)$ and $S_2(k)$ are exact solution operators $S_1(k)U^n = e^{kA}U^n$ and $S_2(k)U^n = e^{kB}U^n$, then

$$\begin{array}{ll} \text{concurrently:} & U^{n+1} = e^{k(A+B)}U^n \\ \text{successively:} & \tilde{U}^{n+1} = e^{kB}e^{kA}U^n \end{array}$$

The splitting error is $|U^{n+1} - \tilde{U}^{n+1}|$. By Taylor series expansion, we have

$$\begin{aligned} e^{kA} &= I + kA + \frac{1}{2}k^2A^2 + O(k^3) \\ e^{kB} &= I + kB + \frac{1}{2}k^2B^2 + O(k^3) \end{aligned}$$

and hence

$$\begin{aligned} e^{kB}e^{kA} &= I + k(A+B) + \frac{1}{2}k^2(A^2 + B^2 + 2BA) + O(k^3) \\ e^{k(A+B)} &= I + k(A+B) + \frac{1}{2}k^2(A^2 + B^2 + AB + BA) + O(k^3). \end{aligned}$$

Subtracting these two operators gives us

$$e^{k(A+B)} - e^{kB}e^{kA} = \frac{1}{2}k^2(AB + BA - 2BA) + O(k^3)$$

There is no splitting error if $AB = BA$. But, in general, A and B do not commute. So, the splitting error is $O(k^2)$ for each time step. After n time steps, the error is $O(k)$.

► Strang splitting

We can reduce the error in operator splitting by using *Strang splitting*. For each time step, we solve $u' = Au$ for a half time step, then solve $u' = Bu$ for a full

time step, and finally solve $u' = A u$ for a half time step. Because

$$e^{\frac{1}{2}kA} e^{kB} e^{\frac{1}{2}kA} - e^{k(A+B)} = O(k^3),$$

Strang splitting is second order. (The proof of this is left as an exercise.) Note that

$$\begin{aligned} U^{n+1} &= S_1\left(\frac{1}{2}k\right) S_2(k) S_1\left(\frac{1}{2}k\right) U^n \\ &= S_1\left(\frac{1}{2}k\right) S_2(k) S_1\left(\frac{1}{2}k\right) S_1\left(\frac{1}{2}k\right) S_2(k) S_1\left(\frac{1}{2}k\right) U^{n-1}. \end{aligned}$$

But since $S_1\left(\frac{1}{2}k\right) S_1\left(\frac{1}{2}k\right) = S_1(k)$, this expression becomes

$$= S_1\left(\frac{1}{2}k\right) S_2(k) S_1(k) S_2(k) S_1\left(\frac{1}{2}k\right) U^{n-1}.$$

Continuing like this, we get

$$= S_1\left(\frac{1}{2}k\right) [S_2(k) S_1(k)]^n S_2(k) S_1\left(\frac{1}{2}k\right) U^0$$

So, to implement Strang splitting, we only need to solve $u' = A u$ for a half time step on the initial and final time steps. In between, we can use simple operator splitting. In this manner, the global splitting error is $O(k^2)$ without much more computation. It is impossible to have an operator splitting method with $O(k^3)$ or smaller error.

► IMEX methods

Another such approach a stiff problem is an implicit-explicit (IMEX) method. Such a method pairs the same discretization for the time derivative with an implicit discretization for the stiff linear term and an explicit discretization for the nonlinear term.

A typical second-order IMEX method combines the second-order Adams–Bashforth and the second-order Adams–Moulton methods. Recall that the second-order Adams–Moulton method is just the trapezoidal method. For both of these methods, the time derivatives are approximated to second-order at $t_{n+1/2}$. For $u' = L u + N u$, we have

$$\frac{U^{n+1} - U^n}{k} = L U^{n+1/2} + N U^{n+1/2} \approx \frac{1}{2} L U^{n+1} + \frac{1}{2} L U^n + \frac{3}{2} N U^n - \frac{1}{2} N U^{n-1}. \blacktriangleleft$$

We can alternatively build a second-order IMEX scheme using the BDF2 method for the linear part $u' = L u$:

$$\frac{\frac{3}{2} U^{n+1} - 2 U^n + \frac{1}{2} U^{n-1}}{k} = L U^{n+1}.$$

A BDF method approximates the derivative at t_{n+1} . To avoid a splitting error, we should also evaluate the nonlinear term $N U$ at t_{n+1} by extrapolating using the

terms U^n and U^{n-1} to approximate U^{n+1} . We can use Taylor series to find the approximation:

$$\begin{aligned} u(t_n) &= u(t_{n+1}) - ku'(t_{n+1}) + \frac{1}{2}k^2u''(t_{n+1}) + O(k^3) \\ u(t_{n-1}) &= u(t_{n+1}) - 2ku'(t_{n+1}) + 2k^2u''(t_{n+1}) + O(k^3). \end{aligned}$$

By combining these equations, we have

$$2u(t_n) - u(t_{n-1}) = u(t_{n+1}) - k^2u''(t_{n+1}) + O(k^3).$$

So, the IMEX method is

$$\frac{\frac{3}{2}U^{n+1} - 2U^n + \frac{1}{2}U^{n-1}}{k} = L U^{n+1} + N [2U^n - U^{n-1}].$$

We can get higher orders by using higher-order extrapolation. To dig deeper into other ways of implementing IMEX methods, see the article by Ascher, Ruuth, and Spiteri.

► Integrating factors

Integrating factors provide another way to deal with stiff problems. Consider the problem $u' = Lu + Nu$ with $u(0) = u_0$, where L is a linear operator and N is a nonlinear operator. If we set $v = e^{-tL}u$, then

$$v' = e^{-tL}N(e^{tL}v) \quad \text{with} \quad v(0) = e^{-tL}u_0.$$

There is no splitting error, but such an approach may only lessen the stiffness.

12.9 Symplectic integrators

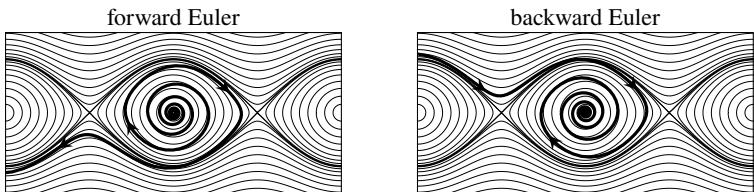
The equation of motion of a simple pendulum is $\theta'' + \kappa \sin \theta = 0$ where $\theta(t)$ is the angle and the coefficient $\kappa = g/\ell$ is the acceleration due to gravity divided by the length of the pendulum. We'll take unit mass, length, and gravitational acceleration to simplify the discussion. In this nondimensionalized system with $\kappa = 1$ and with the position $q(t)$ and the angular velocity $p(t)$, we have the system of equations

$$\frac{dq}{dt} = p \quad \text{and} \quad \frac{dp}{dt} = -\sin q.$$

This problem can be solved analytically in terms of elliptic functions.⁵ To simply find the trajectory in phase space (q, p) , we use conservation of energy. The sum of the kinetic energy T and potential energy V , given by the Hamiltonian

⁵See Karlheinz Ochs' article "A comprehensive analytical solution of the nonlinear pendulum."

$H = T + V = \frac{1}{2}p^2 - \cos q$, is constant along a trajectory. For initial conditions (q_0, p_0) , the total energy is given by $H_0 = \frac{1}{2}p_0^2 - \cos q_0$. Because total energy is constant, $p^2 = 2H_0 + 2\cos q$, from which we can plot the trajectories in the phase plane. Suppose that we solve the problem using the forward Euler and backward Euler methods taking $q_0 = \pi/3$ and $p_0 = 0$ with timestep $k = 0.1$. See the figures below or the QR link at bottom of this page. In both cases, the pendulum starts in the same position. When the pendulum has sufficient momentum $|p_0| > 2$, there is enough energy for it to swing over the top. Otherwise, the pendulum swings back and forth with $|q(t)| < \pi$.



The forward Euler method is unstable and the total energy grows exponentially in time. Even if the pendulum starts with very little movement, the pendulum will speed up over time, getting higher and higher with each swing until eventually it swings over the top. The backward Euler method is stable, but the total energy decays over time. Even if the pendulum starts with enough energy to swing over the top, over time energy dissipates until the pendulum all but stops at the bottom of its swing. In both methods, the energy is not conserved giving an unphysical solution. In problems such as planetary motion and molecular dynamics, using a scheme that conserves energy over a long period of time is important to get a physically correct solution. Using smaller timesteps and higher-order methods can help, but over time energy may still not be conserved. So, let's look at a class of solvers designed to conserve the Hamiltonian.

The pendulum belongs to the general class of Hamiltonian systems

$$\frac{dp}{dt} = -\nabla_q H \quad \text{and} \quad \frac{dq}{dt} = +\nabla_p H.$$

By defining $\mathbf{r} \equiv (p, q)$:

$$\frac{d\mathbf{r}}{dt} = D_H \mathbf{r} \equiv \mathbf{J} \nabla H(\mathbf{r}) \quad \text{where} \quad \mathbf{J} = \begin{bmatrix} 0 & -\mathbf{I} \\ +\mathbf{I} & 0 \end{bmatrix}.$$

For a time-independent Hamiltonian $H(p, q)$

$$\frac{dH}{dt} = \nabla_p H \frac{dp}{dt} + \nabla_q H \frac{dq}{dt} = 0,$$

and we see that the Hamiltonian is a conserved quantity.



Symplectic methods are numerical methods that seek to preserve the volume of a differential 2-form $|dp \wedge dq|$ in phase space and conserve the Hamiltonian. Let's start with the semi-implicit or symplectic Euler method, a first-order scheme that combines the purely implicit forward Euler method and the purely explicit backward Euler method:

$$P^{n+1} = P^n - k \nabla_q H(P^{n+1}, Q^n) \quad (12.13a)$$

$$Q^{n+1} = Q^n + k \nabla_p H(P^{n+1}, Q^n) \quad (12.13b)$$

or alternatively

$$P^{n+1} = P^n - k \nabla_q H(P^n, Q^{n+1}) \quad (12.14a)$$

$$Q^{n+1} = Q^n + k \nabla_p H(P^n, Q^{n+1}). \quad (12.14b)$$

Let's restrict ourselves to a separable Hamiltonian $H(p, q) = T(p) + V(q)$. In this case, $\nabla_p H(p, q) = \nabla_p T(p)$ and $\nabla_q H(p, q) = \nabla_q V(q)$. It seems reasonable that we could improve the symplectic Euler method by implementing (12.13) for a half-timestep followed by (12.14) for a half-timestep:

$$\begin{aligned} P^{n+1/2} &= P^n - \frac{1}{2} k \nabla_q V(Q^n) \\ Q^{n+1} &= Q^n + k \nabla_p T(P^{n+1/2}) \\ P^{n+1} &= P^{n+1/2} - \frac{1}{2} k \nabla_q V(Q^{n+1}) \end{aligned}$$

This second-order approach, called the *Verlet method*, was reintroduced by Loup Verlet in 1967 and can be viewed as an application of Strang splitting.

We can continue to build a higher-order method by composing Verlet methods together. For a fourth-order method, take a Verlet with a timestep of αk followed by a Verlet with timestep $(1 - 2\alpha)k$ followed by a Verlet with timestep αk where $\alpha = (2 - 2^{-1/3})^{-1}$. To dig deeper, see Hairer, Lubich, and Wanner's treatise on geometric numerical integration.

12.10 Practical implementation

It should be completely unsurprising that any scientific computing language would have any number of versatile ODE solvers. While Matlab and Python focus on a few robust ODE solvers, Julia's DifferentialEquations.jl module contains over 200—far too many to discuss in any depth here.⁶ Several of these methods correspond to the standard solvers in Matlab and Python—see

⁶For a complete list of Julia's ODE solvers, see https://diffeq.sciml.ai/latest/solvers/ode_solve/. Christopher Rackauckas, the lead developer of DifferentialEquations.jl, also provides a comparison of differential equation solvers for the major scientific programming language in a post in the online journal *The Winnower*.

<i>Method</i>	<i>Julia</i>	<i>Matlab</i>	<i>Python</i>	<i>stiff</i>	<i>accuracy</i>
Radau IIA 5(3)	RadauIIA5	—	Radau	●	●
BDF/NDF	IDA	ode15i	BDF	●	○
Rosenbrock (order 4)	Rodas4	—	—	●	○
Rosenbrock (order 3)	Rosenbrock23	ode23s [†]	—	●	○
TR-BDF2 SDIRK	TRBDF2	ode23tb [†]	—	●	○
Adams–Moulton/BDF	lsoda	lsode [‡]	LSODA	○	●
Klopfenstein–Shampine	QNDF	ode15s	—	○	○
Trapezoidal ESDIRK	Trapezoid	ode23t [†]	—	○	○
ABM PECE	VCABM	ode113	—	○	●
Dormand–Prince 8(5,3)	DP8	—	DOP853	○	●
Verner 7(6)	Vern7	—	—	○	●
Tsitouras 5(4)	Tsit5	—	—	○	○
Dormand–Prince 4(5)	DP5	ode45	RK45	○	○
Bogacki–Shampine 2(3)	BS3	ode23	RK23	○	○

○ low ● medium ● high

Figure 12.12: Equivalent ODE routines in Julia, Matlab, and Python. This list includes a fraction of all methods available in Julia. Sundials provides additional routines to Python and Julia. [†]Exclusive to Matlab. [‡]Exclusive to Octave.

the table in Figure 12.12. This section summarizes these common solvers and their suitability for different problem types. Given Matlab’s influential role on Python’s SciPy and Julia, it makes some sense to start the discussion with it.

For a full account of the development of Matlab’s ODE suite, see any of the articles authored by Lawrence Shampine in the References. When Lawrence Shampine developed Matlab’s ODE suite, he had a principal design goal that the suite should have a uniform interface so the different specialized and robust solvers could be called in *exactly* the same way. It’s this trade-off between efficiency and convenience that he felt differentiated a “problem solving environment” such as Matlab from “general scientific computation.” Instead of simply returning the solution at each timestep, for example, Shampine designed Matlab’s ODE suite to provide a smooth interpolation between timesteps (especially) when the high-order solvers took large timesteps. Such a design consideration indeed simplifies plotting nice-looking solutions.

The `ode45` routine is the popular explicit Dormand–Prince Runge–Kutta method (sometimes called the DOPRI method), using six function evaluations to compute a fourth-order solution with a fifth-order error correction. This routine is recommended as the go-to solver for non-stiff problems. The `ode23` routine is the second-order Bogacki–Shampine Runge–Kutta method presented on page 352. The `ode113` routine is a variable-step, variable-order Adams–Bashforth–Moulton PECE. It computes a solution to order 12 and an error estimate up to order 13 to

control the variable step size. The `ode15s` routine uses Klopfenstein–Shampine numerical differentiation formulas (NDF). These formulas are modifications of BDF methods with variable-step, variable-order to order 5 and have better stability properties, especially at higher orders, making them more efficient than BDF methods on stiff problems. The `ode23i` routine is a second-order BDF method designed for fully implicit problems $f(t, u, u') = 0$. All of the stiff solvers must solve an implicit difference equation using a Newton method or variation of it, so it is helpful to provide the Jacobian matrix for the right-hand-side function $f(u)$ whenever possible. Otherwise, the method will calculate it using a finite difference approximation. The `ode23s` routine uses a second-order modified Rosenbrock formula that is designed especially for stiff problems with sharp changes (quasi-discontinuities) in the solutions. The `ode23t` routine implements the trapezoidal Runge–Kutta method. The method is not L-stable—it's not a good choice for really stiff problems, but it also doesn't have excessive numerical damping. Because the method doesn't have an embedded error estimate, the routine differentiates a cubic polynomial through prior nodes to estimate the error and adjust the step size. The `ode23tb` routine is an L-stable trapezoidal-BDF2 SDIRK method discussed on page 351 taking $\alpha = 2 - \sqrt{2}$.

Octave's ODE solvers have some notable differences from the Matlab solvers. In addition to four Matlab-compatible solvers `ode23`, `ode45`, `ode15s`, and `ode15i`, Octave's primary routine is `lsode`. The routine is an acronym of Livermore Solver for Ordinary Differential Equations and comes from Alan Hindmarsh's family of Fortran ODE solvers developed at Lawrence Livermore National Laboratory (LLNL). It uses BDF methods for stiff problems and Adams–Moulton methods for non-stiff problems, using functional iteration to evaluate the implicit terms.

Python's `scipy.integrate` library has several ODE solvers. Each of these methods can also be called for one time step, allowing them to be integrated into time-splitting routines. They can also be used as part of `solve_ivp`, which solves an ODE using adaptive step size over a prescribed interval of integration, through the `method` option. The `RK45` (default), `RK23`, and `BDF` routines are equivalent to Matlab's `ode45`, `ode23`, and `ode15i` routines. `LSODA`, a wrapper for the ODEPACK solver, is similar to the `lsode` solver available in Octave, except that it automatically and dynamically switches between the nonstiff Adams–Moulton and stiff BDF solvers. The `Radau`, `BDF`, and `LSODA` routines all require a Jacobian matrix of the right-hand side of the system. If none is provided, the Jacobian will be approximated by the routine using a finite difference approximation. Other packages include `scikits.odes` which provides a wrapper to the LLNL's Sundials⁷ library for BDF and Adams–Moulton routines. In particular, the Sundials routine `CVODE` is similar to `LSODA` but achieves higher stability.

Julia's flagship ODE package is `DifferentialEquations.jl`, developed by Chris Rackauckas and Qing Nie. Other packages include `SciPyDiffEq.jl`—a wrapper

⁷A portmanteau of SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers

for Python’s `scipy.integrate` solvers—and `Sundials.jl`—a wrapper to the Sundials library, containing six solvers for stiff and nonstiff nonlinear problems. `DifferentialEquations.jl` includes routines equivalent to those in Matlab and Python, as well as many methods that are more efficient than the standard routines. In particular, `Tsit5`—the Tsitouras 5(4) Runge–Kutta method—is often more efficient than Dormand–Prince `DP5`; `Rodas4`—a fourth order A-stable stiffly stable Rosenbrock method—is often more efficient than `Rosenbrock23` and `IDA`; and `Vern7`—Verner’s “most efficient” 7(6) Runge–Kutta method—is often more efficient than the higher order `VCABM` and `DP8`. For more details and more routines, check out the documentation at <https://diffeq.sciml.ai>.

Let’s look at a recipe using `DifferentialEquations.jl`. Suppose that we wish to solve the equation for a pendulum $u'' = \sin u$ with initial conditions $u(0) = \pi/9$ and $u'(0) = 0$ over $t \in [0, 8\pi]$. For this problem, we’ll want to use a method that is symplectic or almost symplectic, so let’s use the trapezoidal method. We can solve the problem in Julia using the following steps (see the Back Matter for implementation in Python and Matlab):

1. Load the module
 2. Set up the parameters
 3. Define the problem
 4. Choose the method
 5. Solve the problem
 6. Present the solution
- ```
using DifferentialEquations, Plots
pendulum(u,p,t) = [u[2]; -sin(u[1])]
u₀ = [8π/9, 0]; tspan = [0, 8π]
problem = ODEProblem(pendulum, u₀, tspan)
method = Trapezoid()
solution = solve(problem, method)
plot(solution, xaxis="t", label=[θ"" "ω"])
```

If a method is not explicitly stated, Julia will automatically choose one when it solves the problem. You can help Julia choose a method by telling it whether the problem is stiff (`alg_hints = [:stiff]`) or nonstiff (`alg_hints = [:nonstiff]`). The solution can be addressed either as an array where `solution[i]` indicates the  $i$ th element or as a function where `solution(t)` is the interpolated value at time  $t$ . Implicit-explicit (IMEX) problems such as  $u_t = f(u, t) + g(u, t)$  can be specified using the `SplitODEProblem` function and choosing an appropriate solver. When solving a semilinear system of equations, you can define a function  $f(u, t) \equiv Lu$  as a linear operator using the `DiffEqArrayOperator` function from the `DiffEqOperators.jl` library.

## 12.11 Exercises

12.1. A  $\theta$ -scheme is of the form

$$\frac{U^{n+1} - U^n}{k} = (1 - \theta)f(U^{n+1}) + \theta f(U^n)$$

Find the regions of absolute stability for the  $\theta$ -scheme.

12.2. Plot stability contours for the forward Euler, backward Euler, trapezoidal, and leapfrog methods in the  $\lambda k$ -plane using several values of  $|r|$  besides  $|r| = 1$ .

12.3. Show that applying one step of the trapezoidal method to a linear differential equation is the same as applying a forward Euler method for one half time step followed by the backward Euler method for one half time step.

12.4. Plot the region of stability for the Merson method with Butcher tableau

|               |               |               |               |                |               |
|---------------|---------------|---------------|---------------|----------------|---------------|
|               | 0             |               |               |                |               |
| $\frac{1}{3}$ |               | $\frac{1}{3}$ |               |                |               |
| $\frac{1}{3}$ |               | $\frac{1}{6}$ | $\frac{1}{6}$ |                |               |
| $\frac{1}{2}$ |               | $\frac{1}{8}$ | 0             | $\frac{3}{8}$  |               |
| 1             |               | $\frac{1}{2}$ | 0             | $-\frac{3}{2}$ | 2             |
|               | $\frac{1}{6}$ | 0             | 0             | $\frac{2}{3}$  | $\frac{1}{6}$ |

12.5. Show that  $e^{\frac{1}{2}kA}e^{kB}e^{\frac{1}{2}kA} - e^{k(A+B)} = O(k^3)$ , for two arbitrary, linear operators A and B. From this it follows from which it follows that error using Strang splitting is  $O(k^2)$ .

12.6. The multistage trapezoidal and BDF2 introduced in the example on page 351 are both implicit methods. To implement each stage typically requires computing a Jacobian matrix for Newton method when the right-hand-side function  $f(u)$  is nonlinear. Thankfully, by choosing  $\alpha$  appropriately we can reuse the same Jacobian matrix across both stages.

1. Write the formulas for the trapezoidal method over a subinterval  $\alpha k$  and the BDF2 method over the two subintervals  $\alpha k$  and  $(1 - \alpha)k$  with  $0 < \alpha < 1$ .
2. Determine the Jacobian matrices for these methods.
3. Find an  $\alpha$  such that the Jacobian matrices are proportional.
4. Plot the region of absolute stability.
5. Show that the multistage trapezoidal–BDF2 method is L-stable for this choice of  $\alpha$ .

12.7. Develop a third-order L-stable IMEX scheme. Study the stability of the method by writing the characteristic polynomial and plotting the regions of absolute stability in the complex plane for the implicit part and the explicit part of the IMEX scheme.

12.8. Plot the regions of absolute stability for the Adams–Bashforth–Moulton predictor-corrector methods: AB1-AM2, AB2-AM3, AB3-AM4, and AB4-AM5 with PE(CE)<sup>m</sup> for  $s = 0, 1, \dots, 4$ .

12.9. Compute the Padé approximant  $R_{3,2}(x)$  of  $\log r$  starting with a Taylor polynomial as outlined on page 346.

12.10. Show that the Verlet method for the system  $u' = p$  and  $p' = f(u)$  is the same as the central difference scheme  $U^n - 2U^{n-1} + U^{n-2} = kf(U^n)$  for  $u'' = f(u)$ .

12.11. A coupled pendulum is created by connecting two pendula together with a spring. The Hamiltonian is

$$H = \frac{1}{2}(p_1^2 + p_2^2) - \cos q_1 - \cos q_2 - \varepsilon \cos(q_1 - q_2 - 2)$$

where  $\varepsilon$  is a coupling parameter. Use a symplectic integrator to solve the system with suitable initial conditions and coupling parameter.

12.12. In 1963, mathematician and meteorologist Edward Lorenz developed a simple model of atmospheric dynamics. The now extensively studied Lorenz equation is

$$\frac{dx}{dt} = \sigma(y - x), \quad \frac{dy}{dt} = \rho x - y - xz, \quad \frac{dz}{dt} = -\beta z + xy. \quad (12.15)$$

Plot the solution  $(x(t), z(t))$  when  $\sigma = 10$ ,  $\beta = 8/3$ ,  $\rho = 28$ .

12.13. The SIR model is a simple epidemiological model for infectious diseases that tracks the change over time in the percentage of susceptible, infected, and recovered individuals of a population

$$\frac{dS}{dt} = -\beta IS, \quad \frac{dI}{dt} = \beta IS - \gamma I, \quad \frac{dR}{dt} = \gamma I$$

where  $S(t) + I(t) + R(t) = 1$ ,  $\beta > 0$  is the infection rate, and  $\gamma > 0$  is the recovery rate. The basic reproduction number  $R_0 = \beta/\gamma$  tells us how many other people on average an infectious person might infect at the onset of the epidemic time  $t = 0$ . Plot the curves  $\{S(t), I(t), R(t)\}$  of the SIR model over  $t \in [0, 15]$  starting within an initial state  $\{0.99, 0.01, 0\}$  with  $\beta = 2$  and  $\gamma = 0.4$ .

12.14. The Duffing equation

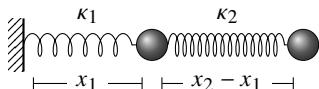
$$x''(t) + \gamma x'(t) + \alpha x + \beta x^3 = \delta \cos \omega t$$

models the motion of a nonlinear damped driven oscillator. Think of a weight attached to a spring attached to a piston. The spring becomes stiffer or softer as it is stretched by adding a cubic term to the linear Hooke's law. The parameter  $\gamma$  is a damping coefficient,  $\alpha$  is the stiffness constant of the spring,  $\beta$  is the additional nonlinearity of the spring constant, and  $\delta$  and  $\omega$  are the amplitude and

angular frequency of the oscillator. Solve the Duffing equation for parameters  $\{\alpha, \beta, \gamma, \delta, \omega\}$  equal to  $\{-1, 1, 0.37, 0.3, 1\}$ . Plot the solution in phase space  $(x(t), x'(t))$  for  $t \in [0, 200]$ . What happens when the damping coefficient is changed from 0.37 to another value?

- 12.15. One way to solve a boundary value problem  $y''(x) = f(y, y', x)$  with boundary values  $y(x_0) = y_0$  and  $y(x_1) = y_1$  is by using a shooting method. A shooting method solves the initial value problem  $y''(x) = f(y, y', x)$  using one of the boundary conditions  $y(x_0) = y_0$  and  $y'(x_0) = s$  for some guess  $s$ . We then compare the difference in the solution at  $x_1$  with the prescribed boundary value  $y_1$  to make updates to our guess  $s$  as an initial condition. With this new guess, we again solve the initial value problem and compare its solution with the correct boundary value to get another updated guess. We continue in this manner until our solution  $y(x_1)$  converges to  $y_1$ . This problem is equivalent to finding the zeros  $s$  to the error function  $e(s) = y(x_1; s) - y_1$ . Use the Roots.jl function `find_zero` along with an ODE solver to find the solution to the Airy equation  $y'' + xy = 0$  with  $y(-12) = 1$  and  $y(0) = 1$ .

- 12.16. Two balls of equal mass are attached in series to a wall using two springs—one soft and one stiff.



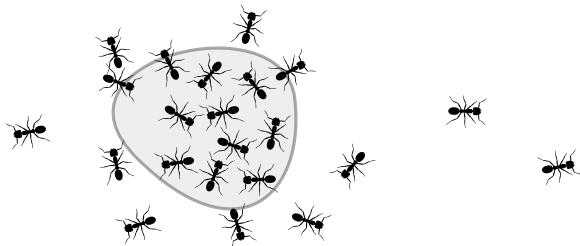
$$\begin{aligned} x_1'' &= -\kappa_1 x_1 + \kappa_2(x_2 - x_1) \\ x_2'' &= -\kappa_2(x_2 - x_1) \end{aligned}$$

Suppose that the the soft spring is initially stretched to  $x_1(0) = 1$  and the stiff spring is in equilibrium at  $x_2(0) = x_1(0)$ . Discuss the limiting behavior when  $\kappa_2 \rightarrow \infty$ . Solve the problem numerically when  $\kappa_2 \gg \kappa_1$ .

## Chapter 13

---

# Parabolic Equations



A group of molecules, bacteria, insects, or even people will often move in a random way. As a result these “particles” tend to spread out. On the large scale, this evolution can be modeled by the diffusion process. Let  $u(t, \mathbf{x})$  be the concentration of particles in  $\mathbb{R}^n$  and consider an arbitrarily shaped but fixed domain  $\Omega \subset \mathbb{R}^n$ . Then the mass of particles at some time  $t$  in  $\Omega$  is  $\int_{\Omega} u(t, \mathbf{x}) dV$ . Since the particles are moving, the mass of particles in  $\Omega$  changes as some particles enter the domain and other particles leave it. Flux is the number of particles passing a point or through an area per unit time. The change in the number of particles in  $\Omega$  equals the integral of the flux of particles  $\mathbf{J}(t, \mathbf{x})$  through the boundary

$$\frac{d}{dt} \int_{\Omega} u(t, \mathbf{x}) dV = - \int_{\partial\Omega} \mathbf{J}(t, \mathbf{x}) \cdot \mathbf{n} dA.$$

By invoking the divergence theorem, we have

$$\int_{\Omega} \frac{\partial}{\partial t} u(t, \mathbf{x}) dV = - \int_{\Omega} \nabla \cdot \mathbf{J}(t, \mathbf{x}) dV.$$

Since the domain  $\Omega$  was arbitrarily chosen, it follows that

$$\frac{\partial}{\partial t} u(t, \mathbf{x}) = -\nabla \cdot \mathbf{J}(t, \mathbf{x}).$$

Fickian diffusion says that the flux  $\mathbf{J}(t, \mathbf{x})$  is proportional to the gradient of the concentration  $u(t, \mathbf{x})$ . Imagine a one-dimensional random walk, in which particles move equally to the right or to the left. Suppose that we have two adjacent cells, one at  $x - \frac{1}{2}h$  with a particle concentration  $u(x - \frac{1}{2}h)$  and one at  $x + \frac{1}{2}h$  with a concentration  $u(x + \frac{1}{2}h)$ , separated by an interface at  $x$ . More particles from the cell with higher concentration will pass through the interface than particles from the cell with lower concentration. If a particle moves through the interface with a positive diffusivity  $\alpha(x)$ , then the flux across the interface can be approximated by

$$J(t, x) = \frac{-\alpha(x)(u(t, x + \frac{1}{2}h) - u(t, x - \frac{1}{2}h))}{h}.$$

In the limit as  $h \rightarrow 0$ , we have



$$J = -\alpha(x) \frac{\partial u}{\partial x}.$$

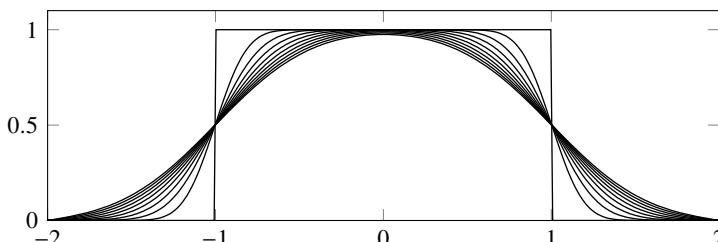
And, in general,

$$\frac{\partial u}{\partial t} = \nabla \cdot (\alpha(\mathbf{x}) \nabla u).$$

Consider an insulated heat conducting bar. Temperature along the bar can be modeled as  $u_t = (\alpha(x)u_x)_x$ , where  $\alpha(x) > 0$  is the heat conductivity and  $u(0, x) = u_0(x)$  is the initial temperature distribution. Because the bar is insulated, the heat flux at the ends of the bar is zero. For a uniform heat conductivity  $\alpha(x) = \alpha$ , the heat equation is simply

$$u_t = \alpha u_{xx}, \quad u(0, x) = u_0(x), \quad u_x(t, x_L) = 0 \quad u_x(t, x_R) = 0. \quad (13.1)$$

This problem is well-posed and can be solved analytically by separation of variables or Fourier series. In the next section, we'll use it as a starting place to examine numerical methods for parabolic partial differential equations, equations that have one derivative in time and two-derivatives in space. For now, let's examine the behavior of the solution over time. Take the heat conductivity  $\alpha = 1$ , and take an initial distribution with  $u(x, 0) = 1$  for  $x \in [-1, 1]$  and  $u(x, 0) = 0$  otherwise along  $[-2, 2]$ . Snapshots of the solution taken at equal intervals from  $t \in [0, 0.1]$  are shown in the following figure and at the QR link at the bottom of this page:



Notice how the solution evolves very quickly at first and then gradually slows down over time. The height of the distribution decreases and its width increases, steep gradients become shallow ones, sharp edges become smooth curves, and the distribution seems to seek a constant average temperature along the entire domain.

There are several prototypical properties of the heat equation  $u_t = \alpha \Delta u$  to keep in mind. It obeys a *maximum principle*—the maximum value of a solution always decreases in time unless it lies on a boundary. Equivalently, the minimum value of a solution always increases unless it lies on a boundary. Simply stated, warm regions get cooler, while cool regions get warmer. If the boundaries are insulating, then the total heat or energy is conserved, i.e., the  $L^1$ -norm of the solution is constant in time. The heat equation is also dissipative. i.e., the  $L^2$ -norm of the solution—often confusingly called the “energy”—decreases in time. A third property is that the solution at any positive time is smooth and grows smoother in time. For a complete review, see Lawrence Evans’ textbook *Partial Differential Equations*.

Parabolic equations are more than just the heat equation. One important class of parabolic equations is the reaction-diffusion equation that couples a reactive growth term with a diffusive decay term. These equations often model the transitions between disorder and order in physical systems. Reaction-diffusion equations are also used to model the pattern formation in biological systems. Take chemotaxis. Animals and bacteria often communicate by releasing chemicals. Not only do the bacteria spread out due to normal diffusion mechanisms, the bacteria may also move along a direction of a concentration gradient, following the strongest attractant, which itself is subject to diffusion. The dispersive mechanisms behind parabolic equations also drive stabilizing behavior in viscous fluid dynamics like the Navier–Stokes equation, ensuring that it doesn’t blow-up like its inviscid cousin, the Euler equation.<sup>1</sup>

### 13.1 Method of lines

A typical way to numerically solve a time-dependent PDE is by using the *method of lines*. We discretize space while keeping time continuous to derive a semidiscrete formulation. The advantage of this approach is that we can decouple time and space discretizations, thereby combining the best methods for each. Take the grid points  $0 = x_0 < x_1 < \dots < x_m = 1$ . For simplicity, define the a uniform mesh  $x_j = j h$  where  $h$  is the grid spacing. Let  $U_j(t)$  be a finite difference approximation to  $u(t, x_j)$  in space. To get a second-order approximation of the

---

<sup>1</sup>That the Navier–Stokes equation does or doesn’t blow-up is a matter of contention. Olga Ladyzhenskaya, Jacques–Louis Lions, and Giovanni Prodi independently proved the existence of smooth solutions to the two-dimensional Navier–Stokes equation in the 1960s. In 2000, the Clay Mathematics Institute added the three-dimensional Navier–Stokes equation as a one-million dollar Millennium Prize. It remains unsolved.

second derivative, we can use a central difference approximation

$$D^2 U_j = D \left( \frac{U_{j+1/2} - U_{j-1/2}}{h} \right) = \frac{U_{j+1} - 2U_j + U_{j-1}}{h^2}.$$

Hence,  $u_t = \alpha u_{xx}$  becomes

$$\frac{\partial}{\partial t} U_j = \alpha \frac{U_{j+1} - 2U_j + U_{j-1}}{h^2} \quad (13.2)$$

with  $j = 0, 1, \dots, m$ . We now have a system of  $m + 1$  ordinary differential equations, and we can use any numerical methods for ODEs to solve the problem. Note that the system is not closed. In addition to the interior mesh points at  $j = 1, 2, \dots, m - 1$  and the explicit boundary mesh points at  $j = 0$  and  $j = m$ , we also have mesh points outside of the domain at  $j = -1$  and  $j = m + 1$ . We will use the boundary conditions to explicitly remove these *ghost points* and close the system.

This approach of employing the numerical method of lines to solve a partial differential equation is often called *time-marching*. We can combine any consistent ODE solver with any consistent discretization of the Laplacian. In the previous chapter, we discussed the forward Euler, the backward Euler, leapfrog and the trapezoidal methods to solve ODEs. Let's apply these methods to the linear diffusion equation. The following stencils correspond to the stencils on page 329. The trapezoidal method applied to the diffusion equation is called the Crank–Nicolson method.



Forward Euler

 $O(k + h^2)$ 

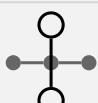
$$\frac{U_j^{n+1} - U_j^n}{k} = \alpha \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \quad (13.3)$$



Backward Euler

 $O(k + h^2)$ 

$$\frac{U_j^{n+1} - U_j^n}{k} = \alpha \frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{h^2} \quad (13.4)$$

Leapfrog (*unstable!*) $O(k^2 + h^2)$ 

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = \alpha \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \quad (13.5)$$



Crank–Nicolson (trapezoidal)  $O(k^2 + h^2)$

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{1}{2}\alpha \frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{h^2} + \frac{1}{2}\alpha \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \quad (13.6)$$

## ► Boundary conditions

The method of lines converted the heat equation into a system of  $m + 1$  differential equations with  $m + 3$  unknowns. In addition to the mesh points at  $x_0, x_1, \dots, x_m$ , we also have ghost points outside the domain. We use the boundary constraints to explicitly remove the ghost points and close the system by eliminating the two unknowns  $U_{-1}$  and  $U_{m+1}$ . There are several common boundary conditions.

A *Dirichlet boundary condition* specifies the value of the solution on the boundary. For example, for a heat conducting bar in contact with heat sinks of temperatures  $u_L$  and  $u_R$  at the ends of the bar, we have the boundary conditions  $u(t, 0) = u_L$  and  $u(t, 1) = u_R$ . We can eliminate the left ghost point  $U_{-1}$  by using a second-order approximation  $\frac{1}{2}(U_{-1} + U_1) \approx U_0 = u_L$ . In this case, substituting  $U_{-1} = 2u_L - U_1$  into (13.2) gives us

$$\frac{\partial}{\partial t} U_0 = \alpha \frac{2U_1 - U_0}{h^2} + 2\alpha \frac{u_L}{h^2}. \quad (13.7)$$

The right ghost point can similarly be eliminated.

**Example.** Let's implement a backward Euler scheme for (13.1) that uses Dirichlet boundary conditions. Let  $\nu = \alpha k / h^2$ . From (13.4) and (13.7) we have

$$\begin{aligned} U_0^{n+1} - U_0^n &= -2\nu U_0^{n+1} + 2\nu u_L \\ U_j^{n+1} - U_j^n &= \nu \left( U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1} \right) \\ U_m^{n+1} - U_m^n &= -2\nu U_m^{n+1} + 2\nu u_R. \end{aligned}$$

Move  $U^{n+1}$  terms to the left-hand side and  $U^n$  terms to the right-hand side:

$$\begin{bmatrix} 1 + 2\nu & & & \\ -\nu & 1 + 2\nu & -\nu & \\ & \ddots & \ddots & \ddots \\ & & -\nu & 1 + 2\nu & -\nu \\ & & & & 1 + 2\nu \end{bmatrix} \begin{bmatrix} U_0^{n+1} \\ U_1^{n+1} \\ \vdots \\ U_{m-1}^{n+1} \\ U_m^{n+1} \end{bmatrix} = \begin{bmatrix} U_0^n \\ U_1^n \\ \vdots \\ U_{m-1}^n \\ U_m^n \end{bmatrix} + \begin{bmatrix} 2\nu u_L \\ 0 \\ \vdots \\ 0 \\ 2\nu u_R \end{bmatrix}.$$

This system is simply  $(\mathbf{I} - \nu \mathbf{D}) \mathbf{U}^{n+1} = \mathbf{U}^n + \mathbf{b}$  where  $\mathbf{D}$  is the modified tridiagonal matrix  $(1, -2, 1)$ . At each time step, we need to invert a tridiagonal matrix by using a tridiagonal solver. A tridiagonal solver efficiently uses Gaussian

elimination by only operating on the diagonal and two off-diagonals, taking only  $O(3m)$  operations to solve a linear system instead of  $O(\frac{2}{3}m^3)$  operations for general Gaussian elimination.

The following Julia code implements the backward Euler method over the domain  $[-2, 2]$  with zero Dirichlet boundary conditions and initial conditions  $u(0, x) = 1$  for  $|x| < 1$  and  $u(0, x) = 0$  otherwise.

```
Δx = .01; Δt = .01; L = 2; ν = Δt/Δx^2; u₁ = 0; uᵣ = 0;
x = -L:Δx:L; m = length(x)
u = (abs.(x).<1)
u[1] += 2ν*u₁; u[n] += 2ν*uᵣ
D = Tridiagonal(ones(m-1), -2ones(m), ones(m-1))
D[1,2] = 0; D[n,n-1] = 0
A = I - ν*D
for i in 1:20
 u = A\u
end
```

We can compute the runtime by adding `@time begin` before and `end` after the code (or the similar `@btime` macro from the `BenchmarkTools.jl` package). This code takes roughly 0.0003 seconds on a typical laptop. If a nonsparse matrix is used instead of the `Tridiagonal` one by setting `D = Matrix(D)`, then the runtime is 0.09 seconds—significantly slower but not unreasonably slow. However, for a larger system with  $\Delta x = .001$ , the runtime using a `Tridiagonal` matrix is still about 0.004 seconds, whereas for a nonsparse matrix it is almost 15 seconds. ◀

• The `LinearAlgebra.jl` library contains types such as `Tridiagonal` that have efficient specialized linear solvers. Such matrices can be converted to regular matrices using the command `Matrix`. The identity operator is simply `I` regardless of size.

A *Neumann boundary condition* specifies the value of the derivative or gradient of the solution at the boundary. For example, to model a bar that is insulated at both ends, we set the heat flux  $J(t, x)$  to be zero on the boundary giving  $u_x(t, 0) = u_x(t, 1) = 0$ . Zero Neumann boundary conditions are also called reflecting boundary conditions.

**Example.** Let's implement a Crank–Nicolson scheme with reflecting boundary conditions. The Crank–Nicolson scheme is second-order in space, so we should also approximate the boundary conditions with the same order. The second-order approximations for the derivatives  $u_x(t, x_0)$  and  $u_x(t, x_m)$  are  $(U_1 - U_{-1})/h$  and  $(U_{m+1} - U_{m-1})/h$ . For reflecting boundary conditions  $u_x(t, x_1) = 0$  and  $u_x(t, x_m) = 0$ , so we can eliminate the left ghost points by setting  $U_{-1} = U_1$  and

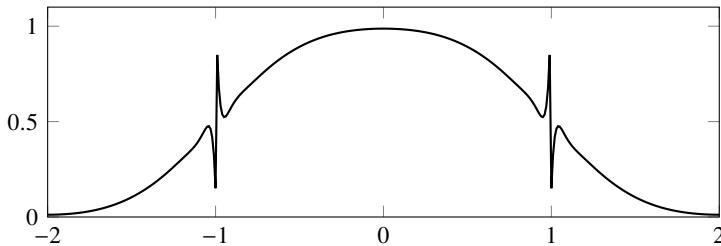
$U_{m+1} = U_{m-1}$ . Let  $\nu = \alpha k/h^2$ . Then the Crank–Nicolson method (13.6) is

$$\begin{aligned}(2 + 2\nu)U_0^{n+1} - 2\nu U_1^{n+1} &= (2 - 2\nu)U_0^n + 2\nu U_1^n \\ -\nu U_{j-1}^{n+1} + (2 + 2\nu)U_j^{n+1} - \nu U_{j+1}^{n+1} &= \nu U_{j-1}^n + (2 - 2\nu)U_j^n + \nu U_{j+1}^n \\ 2\nu U_{m-1}^{n+1} - (2 + 2\nu)U_m^{n+1} &= 2\nu U_{m-1}^n + (2 - 2\nu)U_m^n\end{aligned}$$

Now, we need to solve the system  $(\mathbf{I} - \nu\mathbf{D})\mathbf{U}^{n+1} = (\mathbf{I} - \nu\mathbf{D})\mathbf{U}^n$  where  $\mathbf{D}$  is a modified tridiagonal matrix  $(1, -2, 1)$ . The following Julia code implements the Crank–Nicolson method with reflecting boundary conditions over the domain  $[-2, 2]$  with initial conditions  $u(0, x) = 1$  for  $|x| < 1$  and  $u(0, x) = 0$  otherwise.

```
Δx = .01; Δt = .01; L = 2; ν = Δt/Δx^2
x = -L:Δx:L; m = length(x)
u = (abs.(x).<1)
D = Tridiagonal(ones(m-1), -2ones(m), ones(m-1))
D[1,2] = 2; D[m,m-1] = 2
A = 2I + ν*D
B = 2I - ν*D
anim = @animate for i in 1:40
 plot(x,u, legend=:none, ylims=(0,1))
 u = B\ (A*u)
end
gif(anim, "heat_cn.gif", fps = 5)
```

Compare this code with that of the backward Euler method on the preceding page, noting similarities and differences. Running this code, we observe transient spikes at  $x = -1$  and  $x = 1$ , where the initial distribution had discontinuities. Also, see the QR link at the bottom of this page.



These are clearly numerical artifacts. What causes them and how can we get a better solution? The Crank–Nicolson method is A-stable but not L-stable. Large eigenvalue components, associated with steep gradients, only slowly decay. See Figure 12.10 on page 360 for comparison. We'll examine numerical stability further in the next section and in the next chapter when we discuss dispersive and dissipative schemes. As a practical matter, we can regularize a problem



by replacing a discontinuity in an initial condition with a smooth and rapidly-changing surrogate function. For example, instead of the discontinuous step function, we can take  $\frac{1}{2} + \frac{1}{2} \tanh(\varepsilon^{-1}x)$ , where the thickness of the interface is  $O(\varepsilon)$ .  $\blacktriangleleft$

A third type of boundary condition, the *Robin boundary condition*, is a combination of Dirichlet and Neumann boundary conditions  $u(t, 0) + au_x(t, 0) = b$  and  $u(t, 1) - au_x(t, 1) = b$  for some constants  $a$  and  $b$ . A Robin boundary condition occurs when boundaries absorbs some of the mass or heat and reflects the rest of it. By contrast, *mixed boundary conditions* apply a Dirichlet boundary condition to one boundary and a Neumann boundary condition to the other.

If the problem has *periodic boundary conditions*  $u(t, 0) = u(t, 1)$ , the system can no longer be made to be tridiagonal. Instead, we are left with a circulant matrix which can be inverted quickly using a discrete Fourier transform as we will see in Chapter 16.

A problem can also have *open boundary conditions*, in which the diffusion occurs over an infinite or semi-infinite domain. While we may only be interested in the dynamics in a finite region, we still need solve the problem over the entire domain. One approach to solving the problem over  $(-\infty, \infty)$  is with a nonuniform mesh generated using an inverse sigmoid function such as  $\tanh^{-1} x$  or  $x/(1 - |x|^p)$  for some  $p \geq 1$ . A similar approach is mapping the original problem to a finite domain by using a sigmoid function.

## 13.2 Von Neumann analysis

In the previous chapter, we found the region of absolute stability of a numerical method for the problem  $u' = \lambda u$  by determining the values  $\lambda k$  for which perturbations of the solution decrease over time. We now want to determine under what conditions a numerical method for a partial differential equation is stable. We can apply a discrete Fourier transform to decouple a linear PDE into a system of ODEs. Because eigenvalues are unaffected by a change in basis, we can use the linear stability analysis we developed in the previous chapter. Furthermore, because numerical stability is a local behavior, assuming periodic boundary conditions will not greatly change the analysis.

Take the domain  $[0, 2\pi]$  and uniform discretization in space  $x_j = jh$  with  $h = 2\pi/(2m + 1)$ . The discrete Fourier transform is defined as

$$\hat{u}(t, \xi) = \frac{1}{2m + 1} \sum_{j=0}^{2m} u(t, x_j) e^{-i\xi j h},$$

and the discrete inverse Fourier transform is defined as

$$u(t, x_j) = \sum_{\xi=-m}^m \hat{u}(t, \xi) e^{i\xi j h},$$

where the wave number  $\xi$  is an integer (because the domain is bounded).

Use the method of lines to spatially discretize the heat equation  $u_t = \alpha u_{xx}$  for  $x \in [0, 2\pi]$  to get

$$\frac{\partial}{\partial t} u(t, x_j) = \alpha \frac{u(t, x_{j+1}) - 2u(t, x_j) + u(t, x_{j-1})}{h^2}. \quad (13.8)$$

Substituting the definition of the discrete inverse Fourier transform into this expression gives

$$\sum_{\xi=-m}^m \frac{\partial}{\partial t} \hat{u}(t, \xi) e^{i\xi j h} = \sum_{\xi=-m}^m \alpha \frac{e^{i\xi(j+1)h} - 2e^{i\xi j h} + e^{i\xi(j-1)h}}{h^2} \hat{u}(t, \xi).$$

Equivalently,

$$\sum_{\xi=-m}^m \frac{\partial}{\partial t} \hat{u}(t, \xi) e^{i\xi j h} = \sum_{\xi=-m}^m \alpha \frac{e^{i\xi h} - 2 + e^{-i\xi h}}{h^2} \hat{u}(t, \xi) e^{i\xi j h}.$$

This equality is true for each integer  $j$ , so

$$\frac{\partial \hat{u}}{\partial t} = \alpha \frac{e^{i\xi h} - 2 + e^{-i\xi h}}{h^2} \hat{u}$$

for each  $\xi = -m, -m+1, \dots, m$ . By simplifying the expression

$$\alpha \frac{e^{i\xi h} - 2 + e^{-i\xi h}}{h^2} = \frac{2\alpha}{h^2} (\cos \xi h - 1) = -4 \frac{\alpha}{h^2} \sin^2 \frac{\xi h}{2},$$

we have

$$\frac{\partial \hat{u}}{\partial t} = \left( -4 \frac{\alpha}{h^2} \sin^2 \frac{\xi h}{2} \right) \hat{u}. \quad (13.9)$$

Note that (13.9) is now a linear ordinary differential equation  $\hat{u}' = \lambda_\xi \hat{u}$ , where the eigenvalues  $\lambda_\xi = -4\alpha/h^2 \sin^2(\xi h/2)$  are all non-positive real numbers.

The regions of stability for the forward Euler, backward Euler, leapfrog, and trapezoidal methods are shown in Figure 13.1 on the next page. A method is stable as long as all  $\lambda_\xi k$  lie in the region of stability. The eigenvalues  $\lambda_\xi$  can be as large as  $|\lambda_\xi| = -4\alpha/h^2$ . By using this bound along with associated the region of stability we can determine the stability conditions for the different numerical schemes.

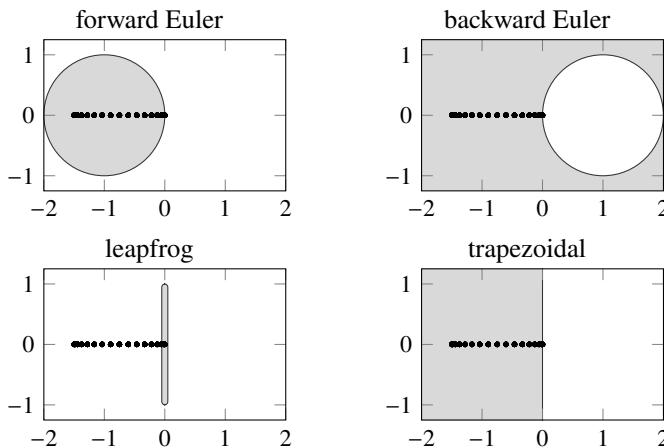


Figure 13.1: Regions of absolute stability (shaded) in the  $\lambda k$ -plane for the forward Euler, backward Euler, leapfrog, and trapezoidal methods along with eigenvalues  $\lambda_\xi$  of the central difference approximation with  $\alpha k / h^2 = \frac{3}{2}$ .

Let's start by determining the stability conditions for the forward Euler method. In the previous chapter, we found that the region of absolute stability for the forward Euler method is the unit circle centered at  $-1$ , containing the interval  $[-2, 0]$  along the real axis. A numerical method is stable if  $\lambda_\xi k$  is in the region of stability for all  $\xi$ . Because the magnitude of  $\lambda_\xi$  can be as large as  $4\alpha/h^2$ , it follows that  $k \leq h^2/2\alpha$  in order that  $\lambda_\xi k$  stay within the region of absolute stability. Such a stability condition is called the *Courant–Friedrichs–Lowy condition* or *CFL condition*. This CFL condition says that if we take a small grid spacing  $h$ , then we need to take a much smaller time step  $k$  to maintain stability. If we double the gridpoints to reduce the error, we need to quadruple the number of time steps to maintain stability. If we increase the number of gridpoints a tenfold, we need to increase the number of time steps a hundredfold. Such a restriction is impractical.

Now, let's determine the stability conditions when we use the backward Euler method to implement (13.8). For the backward Euler method, there is no constraint on the time step  $k$  when the eigenvalues are anywhere along the negative real axis. So, the backward Euler method is *unconditionally stable*. We need to use a tridiagonal solver at each time step, but this extra effort to get unconditional stability is usually worth it. While the backward Euler method is  $O(h^2)$  in space, it is only  $O(k)$  in time. So, to achieve second-order accuracy overall, we'll still need to keep  $k = O(h^2)$ . We'd be better to pair (13.8) with a scheme that is  $O(k^2)$  in time.

In the above analysis, we used the property that the set  $\{e^{i\xi jh}\}$  forms a

linearly independent basis, and therefore, we only need to compare the Fourier coefficients. We can abbreviate the stability analysis a bit by formally substituting  $U_j^n$  with  $A e^{i \xi j h}$ . A method is stable if and only if  $|A| \leq 1$ . We call this *von Neumann stability analysis*.

Let's use von Neumann analysis to determine the stability condition of the leapfrog method

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = \alpha \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

Replace  $U_j^n$  with  $A^n e^{i \xi j h}$ :

$$\frac{A^{n+1} e^{i \xi j h} - A^{n-1} e^{i \xi j h}}{2k} = \alpha A^n \frac{e^{i \xi h} - 2 + e^{-i \xi h}}{k^2} e^{i \xi j h}.$$

Then

$$\frac{A^2 - 1}{2k} = -\alpha \frac{4}{h^2} \sin^2 \frac{\xi h}{2} A,$$

from which we have

$$A^2 + 8\alpha \frac{k}{h^2} \sin^2 \frac{\xi h}{2} A - 1 = 0.$$

From the constant term of the quadratic, we know that the product of the roots of this equation is  $-1$ . Hence, both roots are real, because complex roots would appear in conjugate pairs, the product of which is positive. Since  $\{+1, -1\}$  are not the roots, the absolute value of one root must be greater than one. Therefore, the leapfrog method is unconditionally unstable! That the leapfrog method is unconditionally unstable should be quite clear by from Figure 13.1 on the facing page. The eigenvalues  $\lambda_\xi$  of the second-order central difference approximation are negative real numbers, but the region of absolute stability of the leapfrog method is a line segment along the imaginary axis. So, no eigenvalue other than the zero eigenvalue is ever in that region regardless of how small we take  $k$ . We can make a slight modification to the leapfrog method to get the Dufort–Frankel method. As we'll see below, the Dufort–Frankel method is an *explicit* method that is unconditionally *stable*!



Dufort–Frankel (*inconsistent!*)       $O(k^2 + h^2 + \frac{k^2}{h^2})$

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = \alpha \frac{U_{j+1}^n - (U_j^{n+1} + U_j^{n-1}) + U_{j-1}^n}{h^2} \quad (13.10)$$

Let's determine the stability condition of the Dufort–Frankel method using von Neumann analysis. Replacing  $U_j^n$  with  $A^n e^{i j \xi h}$  in (13.10) and dividing by

$A^{n-1}e^{ij\xi h}$  gives us

$$\frac{A^2 - 1}{2k} = \alpha \frac{Ae^{i\xi h} - (A^2 + 1) + Ae^{-i\xi h}}{h^2} = \alpha \frac{2A \cos \xi h - (A^2 + 1)}{h^2}.$$

Let  $\nu = \alpha k / h^2$ , then

$$(1 + 2\nu)A^2 - (4\nu \cos \xi h)A - (1 - 2\nu) = 0.$$

The roots to this quadratic are

$$A_{\pm} = \frac{2\nu \cos \xi h \pm \sqrt{1 - 4\nu^2 \sin^2 \xi h}}{1 + 2\nu}.$$

Consider the two cases. If  $1 - 4\nu^2 \sin^2 \xi h \geq 0$ , then

$$|A_{\pm}| \leq \frac{2\nu + 1}{1 + 2\nu} = 1.$$

On the other hand, if  $1 - 4\nu^2 \sin^2 \xi h \leq 0$ , then

$$|A_{\pm}|^2 = \frac{(2\nu \cos \xi h)^2 + 4\nu^2 \sin^2 \xi h - 1}{(1 + 2\nu)^2} = \frac{4\nu^2 - 1}{4\nu^2 + 4\nu + 1} \leq 1.$$

So, the Dufort–Frankel scheme is unconditionally stable.

A method that is both explicit and unconditionally stable? That makes the Dufort–Frankel method a unicorn among numerical schemes.<sup>2</sup> But, as the saying goes “there ain’t no such thing as a free lunch.” Let’s compute the truncation error of the Dufort–Frankel method by substituting the Taylor expansion about  $(x_j, t_n)$  for each term. First note that we can rewrite

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = \alpha \frac{U_{j+1}^n - (U_j^{n+1} + U_j^{n-1}) + U_{j-1}^n}{h^2}$$

as

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = \alpha \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} - \alpha \frac{U_j^{n+1} - 2U_j^n + U_j^{n-1}}{h^2}. \quad (13.11)$$

Then Taylor expansion simply gives us

$$u_t + \frac{k^2}{6}u_{ttt} + \dots = \alpha \left( u_{xx} + \frac{h^2}{12}u_{xxxx} + \dots \right) - \alpha \left( \frac{k^2}{h^2}u_{tt} + \dots \right).$$

The truncation error is  $O(k^2 + h^2 + \frac{k^2}{h^2})$ . If  $k = O(h)$ , then the  $k^2/h^2$  term is  $O(1)$ , and the method is inconsistent. In this case, we are actually finding the solution to the equation  $u_t = \alpha u_{xx} - \alpha u_{tt}$ . So, while the Dufort–Frankel scheme is absolutely stable, it is only consistent when  $k \ll h$ . Furthermore, we don’t get second-order accuracy unless  $k = O(h^2)$ .

---

<sup>2</sup>It also violates the second Dahlquist barrier.

### 13.3 Higher-dimensional methods

The two-dimensional heat equation is  $u_t = u_{xx} + u_{yy}$ . By making the approximation  $U_{ij}^n = u(x_i, y_j, t_n)$  and defining the discrete operators in  $x$  and  $y$  as

$$\delta_x^2 U_{ij} = (U_{i+1,j} - 2U_{i,j} + U_{i-1,j})/h^2 \quad (13.12a)$$

$$\delta_y^2 U_{ij} = (U_{i,j+1} - 2U_{i,j} + U_{i,j-1})/h^2 \quad (13.12b)$$

where  $\Delta x = \Delta y = h$ , the Crank–Nicolson method is

$$\frac{U_{ij}^{n+1} - U_{ij}^n}{k} = \frac{1}{2} \left( \delta_x^2 U_{ij}^{n+1} + \delta_x^2 U_{ij}^n + \delta_y^2 U_{ij}^{n+1} + \delta_y^2 U_{ij}^n \right). \quad (13.13)$$

For von Neumann analysis in two dimensions we substitute  $\hat{U}_{ij}^n = A^n e^{i(i\xi h + j\eta h)}$  and determine the CFL condition such that  $|A| \leq 1$ . The Crank–Nicolson method is second-order in time and space and unconditionally stable.

In moving from one dimension to two dimensions, we no longer have a simple tridiagonal system. If we have 100 grid points in the  $x$ - and  $y$ -directions, then we will need to invert a  $10^4 \times 10^4$  block tridiagonal matrix. In three-dimensions, we will need to invert a  $10^6 \times 10^6$  matrix. Such matrices are unnecessarily large, and a more efficient method is to use operator splitting. Let's examine two ways of splitting the right-hand operator of (13.13)

$$\frac{1}{2} (\overset{\circledcirc}{\delta_x^2} U_{ij}^{n+1} + \overset{\circledcirc}{\delta_x^2} U_{ij}^n + \overset{\circledcirc}{\delta_y^2} U_{ij}^{n+1} + \overset{\circledcirc}{\delta_y^2} U_{ij}^n)$$

that will ensure that the method remains implicit. The fractional step method splits  $\overset{\circledcirc}{\delta}_x^2 + \overset{\circledcirc}{\delta}_y^2$  from  $\overset{\circledcirc}{\delta}_x^2 + \overset{\circledcirc}{\delta}_y^2$

$$\frac{U_{ij}^* - U_{ij}^n}{k} = \frac{1}{2} (\overset{\circledcirc}{\delta}_x^2 U_{ij}^* + \overset{\circledcirc}{\delta}_x^2 U_{ij}^n) \quad \text{and} \quad \frac{U_{ij}^{n+1} - U_{ij}^*}{k} = \frac{1}{2} (\overset{\circledcirc}{\delta}_y^2 U_{ij}^* + \overset{\circledcirc}{\delta}_y^2 U_{ij}^{n+1}),$$

and the alternate direction implicit (ADI) method splits  $\overset{\circledcirc}{\delta}_x^2 + \overset{\circledcirc}{\delta}_y^2$  from  $\overset{\circledcirc}{\delta}_x^2 + \overset{\circledcirc}{\delta}_y^2$

$$\frac{U_{ij}^* - U_{ij}^n}{k} = \frac{1}{2} (\overset{\circledcirc}{\delta}_x^2 U_{ij}^* + \overset{\circledcirc}{\delta}_y^2 U_{ij}^n) \quad \text{and} \quad \frac{U_{ij}^{n+1} - U_{ij}^*}{k} = \frac{1}{2} (\overset{\circledcirc}{\delta}_x^2 U_{ij}^* + \overset{\circledcirc}{\delta}_y^2 U_{ij}^{n+1}).$$

Let's examine the stability and accuracy of both splitting methods.

#### ► Fractional step method

Consider the splitting method  $u_t = D u$  where  $D = D_1 + D_2 + \dots + D_p$  and  $D_i = \partial^2 / \partial^2 x_i$ . We solve  $u_t = D_i u$  successively for each dimension  $i$ , making a

multidimensional problem a succession of one-dimensional problems. We can discretize  $D_i$  with  $\delta_{x_i}^2$  and use the Crank–Nicolson method at each stage

$$\frac{U^{n+i/p} - U^{n+(i-1)/p}}{k} = D_i \frac{U^{n+i/p} + U^{n+(i-1)/p}}{2} \quad \text{for } i = 1, 2, \dots, p.$$

Because the Crank–Nicolson method is unconditionally stable at each stage, the overall method is unconditionally stable. What about the accuracy? At each fractional step

$$(I - \frac{1}{2}k D_i) U^{n+i/p} = (I + \frac{1}{2}k D_i) U^{n+(i-1)/p}$$

where  $I$  is the identity operator. If  $\|\frac{1}{2}k D_i\| < 1$ , we can expand

$$(I - \frac{1}{2}k D_i)^{-1} = I + \frac{1}{2}k D_i + \frac{1}{4}k^2 D_i^2 + O(k^3)$$

and so

$$\begin{aligned} U^{n+i/p} &= (I + \frac{1}{2}k D_i + \frac{1}{4}k^2 D_i^2 + O(k^3))(I + \frac{1}{2}k D_i) U^{n+(i-1)/p} \\ &= (I + k D_i + \frac{1}{2}k^2 D_i^2 + O(k^3)) U^{n+(i-1)/p}. \end{aligned}$$

Continuing this expansion for each stage

$$U^{n+1} = \prod_{i=1}^p (I + k D_i + \frac{1}{2}k^2 D_i^2 + O(k^3)) U^n,$$

from which it follows that

$$U^{n+1} - U^n = \underbrace{\frac{1}{2}k \sum_{i=1}^p D_i (U^{n+1} + U^n)}_{\text{Crank–Nicolson}} + \underbrace{\frac{1}{4}k^2 \sum_{i,j=1}^p D_i D_j (U^{n+1} + U^n)}_{O(k^2)} + O(k^3).$$

So, the fractional step method is second order.

### ► Alternating direction implicit method

The ADI method was proposed in Donald Peaceman and Henry Rachford in 1955, while working at Humble Oil and Refining Company, in a short-lived effort to simulate petroleum reserves. Applied to the heat equation, the ADI method is

$$U^{n+1/2} - U^n = \frac{1}{2}\nu(\delta_x^2 U^{n+1/2} + \delta_y^2 U^n), \quad (13.14a)$$

$$U^{n+1} - U^{n+1/2} = \frac{1}{2}\nu(\delta_x^2 U^{n+1/2} + \delta_y^2 U^{n+1}). \quad (13.14b)$$

where  $\nu = k/h^2$ . We need two tridiagonal solvers for this two-stage method.

To examine stability take the Fourier transforms ( $x \mapsto \xi$  and  $y \mapsto \eta$ )

$$\begin{aligned}\hat{U}^{n+1/2} &= \hat{U}^n + \frac{1}{2}\nu \left( -4 \sin^2 \frac{1}{2}\xi h \right) \hat{U}^{n+1/2} + \frac{1}{2}\nu \left( -4 \sin^2 \frac{1}{2}\eta h \right) \hat{U}^n \\ \hat{U}^{n+1} &= \hat{U}^{n+1/2} + \frac{1}{2}\nu \left( -4 \sin^2 \frac{1}{2}\xi h \right) \hat{U}^{n+1/2} + \frac{1}{2}\nu \left( -4 \sin^2 \frac{1}{2}\eta h \right) \hat{U}^{n+1}\end{aligned}$$

from which

$$\hat{U}^{n+1} = \frac{1 - 2\nu \sin^2 \frac{1}{2}\eta h}{1 + 2\nu \sin^2 \frac{1}{2}\xi h} \hat{U}^{n+1/2} \quad \text{and} \quad \hat{U}^{n+1/2} = \frac{1 - 2\nu \sin^2 \frac{1}{2}\eta h}{1 + 2\nu \sin^2 \frac{1}{2}\xi h} \hat{U}^n.$$

So,

$$\hat{U}^{n+1} = \underbrace{\frac{(1 - 2\nu \sin^2 \frac{1}{2}\eta h)(1 - 2\nu \sin^2 \frac{1}{2}\eta h)}{(1 + 2\nu \sin^2 \frac{1}{2}\eta h)(1 + 2\nu \sin^2 \frac{1}{2}\eta h)} \hat{U}^n}_{\lambda}.$$

The denominator of  $\lambda$  is always larger than the numerator, so  $|\lambda| \leq 1$  from which it follows that the method is unconditionally stable.

To determine the order of the ADI method, we take the difference and sum of (13.14)

$$(a) - (b) : \quad 2U^{n+1/2} = U^{n+1} + \frac{1}{2}\nu \delta_y^2(U^n - U^{n+1}) \quad (13.15a)$$

$$(a) + (b) : \quad U^{n+1} - U^n = \nu \delta_x^2 U^{n+1/2} + \frac{1}{2}\nu \delta_y^2(U^n + U^{n+1}) \quad (13.15b)$$

Substituting (13.15a) into (13.15b)

$$\begin{aligned}U^{n+1} - U^n &= \frac{1}{2}\nu \delta_x^2 \left( U^n + U^{n+1} + \frac{1}{2}\nu \delta_y^2(U^n - U^{n+1}) \right) + \frac{1}{2}\nu \delta_y^2(U^n + U^{n+1}) \\ &= \underbrace{\nu \left( \delta_x^2 \frac{U^n + U^{n+1}}{2} + \delta_y^2 \frac{U^n + U^{n+1}}{2} \right)}_{\text{Crank--Nicolson}} + \underbrace{\frac{1}{4}\nu^2 \delta_x^2 \delta_y^2(U^n - U^{n+1})}_{O(\nu^2 h^4)}.\end{aligned}$$

The second term  $O(\nu^2 h^4) = O(h^2)$  because  $\nu = k/h^2$ . So, the method is the same order as the Crank–Nicolson scheme  $O(h^2 + k^2)$ . The fractional step method is easier to implement than ADI method, but in general the ADI method is more accurate.

A three-dimensional problem  $u_t = u_{xx} + u_{yy} + u_{zz}$  is treated similarly. Take  $\Delta x = \Delta y = \Delta z = h$  and  $\nu = k/h^2$ . Then the ADI method is

$$\begin{aligned}U^{n+1/3} - U^n &= \frac{1}{6}\nu(3)\delta_x^2(U^{n+1/3} + U^n) + \delta_y^2(2U^n) + \delta_z^2(2U^n) \\ U^{n+2/3} - U^{n+1/3} &= \frac{1}{6}\nu(3)\delta_x^2(U^{n+1/3} + U^n) + \delta_y^2(U^{n+2/3} + U^n) + \delta_z^2(2U^n) \\ U^{n+1} - U^{n+2/3} &= \frac{1}{6}\nu(3)\delta_x^2(U^{n+1/3} + U^n) + \delta_y^2(U^{n+2/3} + U^n) + \delta_z^2(U^{n+1} + U^n)\end{aligned}$$

### 13.4 Nonlinear diffusion equations

Consider the heat equation along a rod for which the heat conductivity  $\alpha(u)$  changes as a positive function of the temperature. In this case,

$$\frac{\partial}{\partial t} u = \frac{\partial}{\partial x} \left( \alpha(u) \frac{\partial}{\partial x} u \right) \quad (13.16a)$$

$$u(0, x) = u_0(x) \quad (13.16b)$$

$$u(t, 0) = u(t, 1) = 0. \quad (13.16c)$$

Let's solve the problem by using the method of lines

$$\begin{aligned} \frac{\partial}{\partial t} u &= \frac{\partial}{\partial x} \left( \alpha(u) \frac{\partial}{\partial x} u \right) \\ &\approx \frac{1}{h} \left( \alpha(U_{j+1/2}) \frac{\partial}{\partial x} U_{j+1/2} - \alpha(U_{j-1/2}) \frac{\partial}{\partial x} U_{j-1/2} \right) \\ &\approx \alpha(U_{j+1/2}) \frac{U_{j+1} - U_j}{h^2} - \alpha(U_{j-1/2}) \frac{U_j - U_{j-1}}{h^2} \\ &\approx \alpha \left( \frac{U_{j+1} + U_j}{2} \right) \frac{U_{j+1} - U_j}{h^2} - \alpha \left( \frac{U_j + U_{j-1}}{2} \right) \frac{U_j - U_{j-1}}{h^2} \end{aligned} \quad (13.17)$$

Because the diffusion equation is stiff, the ODE solver should be a stiff solver. Otherwise, we will be forced to take many tiny time steps to ensure stability. Because the problem is nonlinear, we will need to solve a system of nonlinear equations, typically using Newton's method. In practice, this means that Julia (or whatever language we might use) must numerically compute and invert a Jacobian matrix. Luckily, our system of nonlinear equations is sparse, so the Jacobian matrix is also sparse. To help our solver, we can explicitly tell it that the Jacobian is sparse by passing it the sparsity pattern for the Jacobian, otherwise our solver may foolishly build a nonsparse Jacobian matrix. In this case, the sparsity pattern is a tridiagonal matrix of ones.

**Example.** The nonlinear diffusion equation  $u_t = (u^p u_x)_x$  for some  $p$  is known as the porous medium equation and has been used to model the dispersion of grasshoppers and the flow of groundwater. The following Julia code implements (13.17) with  $\alpha(u) = u^2$ . We use the Sundials.jl package, which has a BDF routine for stiff nonlinear problems. The routine defaults to using a Newton's solver, and we can help it by specifying that the Jacobian matrix is tridiagonal.

```
using Sundials
m = 400; L = 2; x = LinRange(-L,L,m); Δx = x[2]-x[1]
α = (u -> u.^2)
Du(u,Δx,t) = [0;diff(α((u[1:end-1]+u[2:end])/2).*diff(u))/Δx^2;0]
```

| Method                | name         | stiff | time steps | runtime (s) |
|-----------------------|--------------|-------|------------|-------------|
| BDF                   | CVODE_BDF    | ●     | 997        | 0.1         |
| Bogacki–Shampine 2(3) | BS3          | ○     | 19410      | 1.3         |
| Dormand–Prince 4(5)   | DP5          | ○     | 14749      | 2.2         |
| Verner 7(6)           | Vern7        | ○     | 10513      | 2.9         |
| Rosenbrock 2(3)       | Rosenbrock23 | ●     | 554        | 39.6        |
| Trapezoidal ESDIRK    | Trapezoid    | ●     | 948        | 57.6        |

○ low ● medium ● high

Figure 13.2: Solving the porous medium equation with 400 mesh points.

```
u₀ = (abs.(x).<1)
problem = ODEProblem(Du,u₀,(0,2),Δx)
method = CVODE_BDF(linear_solver=:Band, jac_lower=1, jac_upper=1)
solution = solve(problem, method);
```

Because  $u^2 u_x = \left(\frac{1}{3}u^3\right)_x$ , we could solve  $u_t = \left(\frac{1}{3}u^3\right)_{xx}$  instead. In this case, we would simply replace the appropriate lines of code with the following:

```
α = (u -> u.^3/3)
Du(u,Δx,t) = [0;diff(diff(α(u)))/Δx^2;0]
```

There are different ways to visualize time-dependent data in Julia. One is with the `@animate` macro in `Plots.jl`, which combines snapshots together as a gif

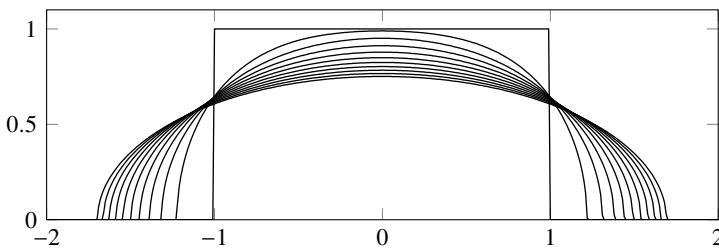
```
anim = @animate for t in LinRange(0,2,200)
 plot(x,solution(t),legend=:none,fill=(0,0.4,:red),ylims=(0,1))
end
gif(anim, "porous.gif", fps = 15)
```

or as an mp4 video with `mp4(anim, "porous.mp4", fps = 15)`. Alternatively, we might save the figures as a set of images in the loop and then make an external call to `ffmpeg` afterwards:

```
savefig("tmp" * lpad(i,3,"0") * ".png")
run(`ffmpeg -y -i "tmp%03d.png" -pix_fmt yuv420p porous.mp4`)
```

Another way is with the `@manipulate` macro in `Interact.jl`, which creates a slider widget allowing a user to move the solution forwards and backwards in time.

```
using Interact
@manipulate for t∈slider(0:0.01:2; value=0, label="time")
 plot(x,solution(t), fill = (0, 0.4, :red))
 plot!(ylims=(0,1),legend=:none)
end
```



The figure above and the QR link below shows the solution to the porous medium equation with an initial distribution given by a rectangular function. The snapshots are taken at equal intervals from  $t \in [0, 2]$ . Compare this solution with that of the one-dimensional heat equation on page 374.

It's worthwhile to benchmark the performance of the BDF method against other methods. While using an implicit method may reduce the number of steps in time, each step may require significant computation to invert a Jacobian. The runtimes using different stiff and non-stiff methods are shown in Figure 13.2 on the previous page. ◀

## ► Stability

Linear von Neumann stability analysis cannot be applied to variable-coefficient and nonlinear problems. For such problems, the *energy method* is an important tool for checking stability. We define the “energy” of a system as the  $L^2$ -norm of the variable  $u$ . A system is stable if the energy is non-increasing. Multiplying equation (13.16a) by  $u$  and integrating over the domain, we have

$$\int_0^1 u \frac{\partial}{\partial t} u \, dx = \int_0^1 u \frac{\partial}{\partial x} \left( \alpha(u) \frac{\partial}{\partial x} u \right) \, dx.$$

After integrating the right-hand side by parts and applying boundary terms, we have

$$\frac{1}{2} \frac{\partial}{\partial t} \int_0^1 u^2 \, dx = - \int_0^1 \alpha(u) \left( \frac{\partial}{\partial x} u \right)^2 \, dx.$$

If  $u$  is not a constant function of  $x$ , then the right-hand side is strictly negative and the  $L^2$ -norm of  $u$  is decreasing in time.

We can apply the same idea to the numerical scheme (13.17) by using the discrete  $\ell^2$ -norm. Now,

$$\sum_{j=1}^{m-1} U_j \frac{\partial}{\partial t} U_j = \sum_{j=1}^{m-1} \frac{1}{h^2} [\alpha_{j+1/2}(U_{j+1} - U_j)U_j - \alpha_{j-1/2}(U_j - U_{j-1})U_j]$$



from which it follows that

$$\begin{aligned} \frac{1}{2} \frac{\partial}{\partial t} \sum_{j=1}^{m-1} (U_j)^2 &= \frac{1}{h^2} \sum_{j=1}^{m-1} \alpha_{j+1/2} (U_{j+1} - U_j) U_j - \alpha_{j-1/2} (U_j - U_{j-1}) U_j \\ &= -\frac{1}{h^2} \sum_{j=1}^{m-1} \alpha_{j-1/2} (U_j - U_{j-1})^2 \leq 0. \end{aligned}$$

This tells us that the  $\ell^2$ -norm is nonincreasing.

Because discrete norms are equivalent, we can use other norms besides the  $\ell^2$  norm to confirm stability. As an example, let's use the  $\ell^\infty$ -norm to confirm the CFL condition of the forward Euler method (13.3). Take  $U_j^0$  to be non-negative for all  $j$ . Then letting  $\nu = k/h^2$ ,

$$U_j^{n+1} - U_j^n = \nu \alpha_{j+1/2} (U_{j+1} - U_j) - \nu \alpha_{j-1/2} (U_j - U_{j-1}),$$

from which

$$U_j^{n+1} = \nu \alpha_{j+1/2} U_{j+1}^n + (1 - \nu \alpha_{j+1/2} - \nu \alpha_{j-1/2}) U_j^n + \nu \alpha_{j-1/2} U_{j-1}^n.$$

If  $\nu \alpha_{j+1/2} \leq \frac{1}{2}$  for all  $j$ , then

$$\begin{aligned} |U_j^{n+1}| &= \nu \alpha_{j+1/2} |U_{j+1}^n| + (1 - \nu(\alpha_{j+1/2} + \alpha_{j-1/2})) |U_j^n| + \nu \alpha_{j-1/2} |U_{j-1}^n| \\ &\leq (\nu \alpha_{j+1/2} + 1 - \nu \alpha_{j+1/2} - \nu \alpha_{j-1/2} - \nu \alpha_{j-1/2}) \|U^n\|_\infty = \|U^n\|_\infty \end{aligned}$$

for all  $j$ . It follows that  $\|U^{n+1}\|_\infty \leq \|U^n\|_\infty \leq \dots \leq \|U^0\|_\infty$  for all  $n$  whenever the CFL condition  $k \leq \frac{1}{2} h^2 / \max_j \alpha(U_j^0)$  is met.

### 13.5 Exercises

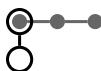
13.1. Show that the heat equation  $u_t = \alpha u_{xx}$  with a bounded initial distribution  $u(0, x) = u_0(x)$  has the solution

$$u(t, x) = \int_{-\infty}^{\infty} G(t, x - s) u_0(s) \, ds \quad \text{with} \quad G(t, s) = \frac{1}{\sqrt{4\pi\alpha t}} e^{-s^2/4\alpha t}.$$

This expression is known as the fundamental solution to the heat equation.

13.2. Use Taylor series expansion to determine the truncation error of the Crank–Nicolson scheme for  $u_t = u_{xx}$ .

13.3. Suppose that we want to solve the heat equation. Instead of using the values  $\{U_{j-1}, U_j, U_{j+1}\}$  to approximate  $u_{xx}$  at  $x_j$ , one could use the values  $\{U_j, U_{j+1}, U_{j+2}\}$ . For example, the stencil for the backward Euler method is



This alternative approach is bad for a number of reasons. First, it provides only a first-order approximation whereas the central difference approximation is second-order. More importantly, it affects stability. Discuss the stability conditions of using such a scheme with various time-stepping methods.

13.4. Prove that the Dufort–Frankel scheme is unconditionally stable by showing that the eigenvalues of the space discretization all lie within the region of absolute stability for the time-marching scheme. Demonstrate that although the scheme is unconditionally stable, it gets the wrong solution when  $k = O(h)$ .

13.5. Solve the heat equation  $u_t = u_{xx}$  over the domain  $[0, 1]$  with initial conditions  $u(0, x) = \sin \pi x$  and boundary conditions  $u(t, 0) = u(t, 1) = 0$  using the forward Euler scheme. Use 20 grid points in space and use the CFL condition to determine the stability requirement on the time step size  $k$ .

1. Examine the behavior of the numerical solution if  $k$  is slightly above or below the stability threshold.
2. Use the slope of the log-log plot of the error at  $t = 1$  to verify the order of convergence. Use several values for the grid spacing  $h$  while keeping the time step  $k \ll h$  constant. Then, use several values for the time step  $k$  keeping the mesh  $h \ll k$ .

13.6. The Schrödinger equation

$$i\varepsilon \frac{\partial \psi}{\partial t} = -\frac{\varepsilon^2}{2} \frac{\partial^2 \psi}{\partial x^2} + V(x)\psi$$

models the quantum behavior of a particle in a potential field  $V(x)$ . The parameter  $\varepsilon$  is the scaled Plank constant, which gives the relative length and time scales. The probability of finding a particle at a position  $x$  is given by  $\rho(t, x) = |\psi(t, x)|^2$  for a normalized wave function  $\psi(t, x)$ . While the Schrödinger equation is a parabolic PDE, it exhibits behaviors closely related to the wave equation, namely the  $L^2$ -norm of the solution is constant in time and the equation has no maximum principle.

1. For constant potential  $V(x) \equiv V$ , describe the stability conditions using a second-order space discretization with different time-marching schemes, such as forward Euler, leapfrog, Crank–Nicolson, and Runge–Kutta.
2. The Crank–Nicolson method does a good job conserving the  $L^2$ -norm of the solution. Use it to solve the initial value problem for the harmonic oscillator

with  $V(x) = \frac{1}{2}x^2$  with initial conditions  $\psi(0, x) = (\pi\varepsilon)^{-1/4}e^{-(x-x_0)^2/2\varepsilon}$ . The solution for this problem

$$\psi(t, x) = (\pi\varepsilon)^{-1/4}e^{-(x-x_0\exp(-it))^2/2\varepsilon}e^{-(1-\exp(-2it))x_0^2/4\varepsilon}e^{-it/2}$$

is called the coherent state,<sup>3</sup> and it is one of the rare examples of an exact closed-form solution to the time-dependent Schrödinger equation. In particular, at time  $t = 2\pi n$ , the solution  $\psi(2\pi n, x) = (-1)^n\psi(0, x)$ . Take  $\varepsilon = 0.3$ . We need to choose a domain that is wide enough to ensure that interactions with the boundaries are negligible. For this problem  $[-4, 4]$  should be sufficient. Use the slope of the log-log plot of the error with the analytic solution at  $t = 2\pi$  to confirm that the method is  $O(k^2 + h^2)$ . To do this, you should use several values for the grid spacing  $h$  while keeping the time step  $k \ll h$  constant. Then use several values for the time step  $k$  keeping the mesh  $h \ll k$ .

3. What happens when the mesh spacing  $h$  or the time step  $k$  are about equal to or larger than  $\varepsilon$ ?

13.7. In polar coordinates, the heat equation is

$$\frac{\partial u}{\partial t} = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial u}{\partial r} \right) + \frac{1}{r^2} \frac{\partial^2 u}{\partial \theta^2}.$$

For a radially symmetric geometry, this equation simplifies to

$$\frac{\partial u}{\partial t} = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial u}{\partial r} \right).$$

Develop a numerical method for this problem and implement it using the regularized step function  $\tanh 30(1 - r)$  as initial conditions. Use reflecting boundary conditions at  $r = 2$  and symmetry at  $r = 0$  to determine an appropriate boundary condition. Implement the method and plot the solution at several steps in time.

13.8. Suppose that we want to model the heat equation  $u_t = u_{xx}$  with open boundary conditions. Over time a function will gradually decrease in height and broaden in width while conserving area. The interesting dynamics may all occur near the origin, but we'll still need to solve the equation far from the origin to get an accurate solution. Rather than using equally-spaced gridpoints, we can choose points that dense near the origin and spread out the farther they get from the origin.

---

<sup>3</sup>Coherent states refer to Gaussian wave packet solutions of the harmonic oscillator and were derived by Schrödinger in 1926 in response to criticism that the wave function  $\psi(t, x)$  did not display classical motion and satisfy the correspondence principle.

- Derive a central difference approximation to the Laplacian operator for arbitrarily-spaced gridpoints. Discuss the error of the approximation.
- Find the numerical solution to the heat equation by approximating open boundary conditions using a non-uniform grid by using an inverse sigmoid function, such as  $x = \tanh^{-1} \xi$ . Use the Gaussian function  $u(x, 0) = e^{-sx^2}$  as the initial distribution with  $s = 10$ . The exact solution over an infinite domain is  $u(x, t) = (1 + 4st)^{-1/2} e^{-sx^2/(1+4st)}$ .

 13.9. The Allen–Cahn equation

$$u_t = \Delta u + \varepsilon^{-1} u(1 - u^2)$$

is a reaction-diffusion equation that models a nonconservative phase transition. The solution  $u(t, x, y)$  has stable equilibria at  $u = \pm 1$  with a thin phase interface given by  $-1 < u < 1$ . As a loose analogy, think of ice melting and freezing in a pool of water with  $u = 1$  representing the ice and  $u = -1$  representing the water.

Determine the numerical solution to the two-dimensional Allen–Cahn equation using a numerical method that is second order in time and space with stability condition independent of  $\varepsilon$ . Consider the domain  $[-8, 8] \times [-8, 8]$  with reflecting boundary conditions. Take  $\varepsilon = 1/50$  and take the initial conditions

$$u(0, x, y) = \begin{cases} 1, & \text{if } x^4 < 32x^2 - 16y^2 \\ -1, & \text{otherwise} \end{cases}$$

Also, try initial conditions consisting of an array of normally-distributed random numbers. Observe the behavior of the solution over time.

## Chapter 14

---

# Hyperbolic Equations



Nonlinear hyperbolic equations are synonymous with shock wave formation from the hydraulic lift of a tsunami as it approaches the shore to the sonic boom of a jet aircraft to the pressure wave of an atomic blast. Hyperbolic equations also model plasmas, crowds, combustion, earthquakes, and traffic flow.

A principal difference between hyperbolic and parabolic equations is that while solutions to linear parabolic equations are always smooth and become increasingly smoother over time, solutions to hyperbolic equations often are not. In fact, the solutions of nonlinear hyperbolic equations with smooth initial conditions may become discontinuous in finite time resulting in shock waves. These discontinuities introduce challenges when we try to approximate them using high-order polynomial interpolants.

Another difference is that the stability condition for explicit schemes for hyperbolic equations are typically  $k = O(h)$  rather than the  $k = O(h^2)$  that we had for parabolic equations. While there was a significant advantage to using an implicit solver for parabolic equations, there is no significant advantage for hyperbolic equations. Therefore, we only consider explicit schemes.

The subject of numerical methods for hyperbolic equations has had considerable development in the last fifty years. This chapter is an introduction—see Randall LeVeque’s textbook *Finite Volume Methods for Hyperbolic Problems* for a complete coverage.

### 14.1 Linear hyperbolic equations

Let  $u(t, x)$  be a physical quantity such as density, pressure, or velocity. If the flux of  $u$  is given by  $f(u)$ , then the time evolution of the quantity  $u$  is given by

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0.$$

In the previous chapter, the flux was given by Fick's Law  $f(u) = -\alpha(x) \frac{\partial}{\partial x} u$ , which led to diffusion that characterized parabolic equations. Hyperbolic equations are characterized by advection. Let's start with the simplest flux  $f(u) = cu$  and look at the more general nonlinear case later. Information propagating with velocity  $c$  is described by the linear advection equation

$$\frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0 \quad \text{with} \quad u(x, 0) = u_0(x). \quad (14.1)$$

This problem can be solved by the *method of characteristics*. Compare the total derivative of  $u(t, x)$  and the advection equation:

$$\frac{d}{dt} u(t, x(t)) = \frac{\partial u}{\partial t} + \frac{dx}{dt} \frac{\partial u}{\partial x} \quad \text{and} \quad \frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0.$$

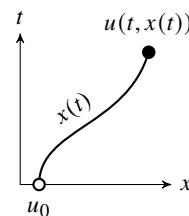
By taking  $dx/dt = c$ , the total derivative

$$\frac{d}{dt} u(t, x(t)) = 0,$$

which says that the solution  $u(t, x(t))$  is constant along the *characteristic curves*  $x(t)$  for which  $dx/dt = c$ . The characteristic curves are exactly those for which  $x(t) = ct + x_0$  for some  $x_0$ . The solution to (14.1) is then given by

$$u(t, x(t)) = u(0, x(0)) = u(0, x_0) = u_0(x_0) = u_0(x - ct).$$

We can find the solution at time  $t$  by tracing backwards along the characteristic curve to the initial conditions. The value  $c$  is called the *characteristic speed*, and  $u$ —which is constant along the characteristic curve—is called the *Riemann invariant*. It's not necessary that  $c$  be a constant—the method of characteristics also works for general  $f(u)$ . We'll come back to the nonlinear case later in the chapter.



## 14.2 Methods for linear hyperbolic equations

Let's first develop numerical schemes to solve for the simple linear problem  $u_t + cu_x = 0$ . The schemes developed for this linear problem will be prototypic methods that we can later extend to solve nonlinear hyperbolic problems.

### ► Upwind method

The simplest numerical scheme for solving  $u_t + cu_x = 0$  is one that uses a forward Euler approximation in time and a first-order difference in space. Such a scheme is called an *upwind method*.

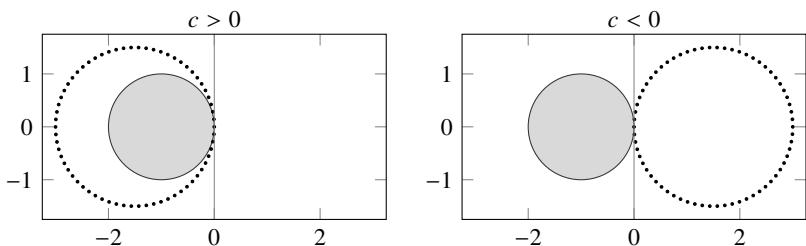
A natural first question about such a scheme might be “what is its stability condition?” We can answer this question using von Neumann analysis that we developed in the previous chapter. Consider the semidiscrete system of equations

$$\frac{\partial}{\partial t} u(t, x_j) = c \frac{u(t, x_{j-1}) - u(t, x_j)}{h}.$$

We find the corresponding system in the Fourier domain by formally substituting  $u(t, x_j)$  with  $\hat{u}(t, \xi) e^{i\xi j h}$ . In this case, we have the decoupled system

$$\frac{\partial}{\partial t} \hat{u}(t, \xi) = c \frac{e^{-i\xi h} - 1}{h} \hat{u}(t, \xi),$$

whose eigenvalues are  $(c/h) \cdot (e^{-i\xi h} - 1)$ . The locus of  $e^{-i\xi h} - 1$  is a circle of radius one centered at  $-1$ . The region of absolute stability for the forward Euler method is also a circle of radius one centered at  $-1$ . If the characteristic speed  $c > 0$ , then  $(c/h) \cdot (e^{-i\xi h} - 1)$  is a simply a circle of radius  $c/h$  centered at  $-c/h$ . Therefore, the upwind method (14.3) is stable if the *CFL condition*  $k < h/c$  holds. We call  $|c|k/h$  the *CFL number* for the upwind scheme. A useful method of visualizing the CFL condition of numerical method is by overlaying the eigenvalues of operator on the right-hand side over the region of absolute stability of the ODE solver:



When the characteristic speed  $c$  is positive, the upwind method

$$U_j^{n+1} - U_j^n = -\frac{ck}{h} (U_j^n - U_{j-1}^n) \quad (14.2)$$

is made more stable by taking smaller time steps  $k$  so that  $k < h/c$ . But when  $c$  is negative, all of the eigenvalues are in the right half-plane. There's nothing we can do to make the method absolutely stable by scaling  $k$ , so the method is unconditionally unstable. We can, however, modify (14.2) specifically for the case when the characteristic speed  $c$  is negative. The two varieties of upwind methods are listed in the following boxes:

Upwind Method ( $c > 0$ ) $O(k + h)$ 

$$\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_j^n - U_{j-1}^n}{h} = 0 \quad (14.3)$$

Upwind Method ( $c < 0$ ) $O(k + h)$ 

$$\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_{j+1}^n - U_j^n}{h} = 0 \quad (14.4)$$

To implement the schemes, we must first determine the sign of  $c$ . We can combine the two varieties to get a scheme that does this automatically

$$\frac{U_j^{n+1} - U_j^n}{k} + \frac{c + |c|}{2} \frac{U_j^n - U_{j-1}^n}{h} + \frac{c - |c|}{2} \frac{U_{j+1}^n - U_j^n}{h} = 0.$$

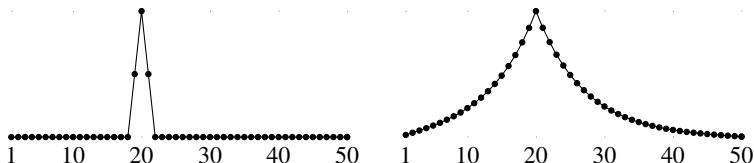
To make intuitive sense of why we should need two separate upwind methods depending on the characteristic speed (or one combined automatic-switching scheme), just think about the weather. To predict the weather we need to look in the upwind direction from which it is coming, not the downwind direction in which it is going. Information propagates at a finite speed  $c$  along any characteristic of the advection equation (14.1). So, in a finite period of time the solution at any specific point in space can only be affected by the initial conditions in some bounded region of space. This region is called the *domain of dependence*. Similarly, over a finite period of time that same point in space can only influence a bounded region of space. This region is called the *domain of influence*. In short, a domain of dependence looks into the past and domain of influence looks into the future. Consider the analytic solution to the advection equation:  $u(t_n, x_j) = u_0(x_j - ct_n)$ . Take a uniform step size  $k$  in time and a uniform grid spacing  $h$ . The domain of dependence for the analytic solution is  $[x_j, x_j + cnk]$ . The domain of dependence for the numerical solution is  $[x_j, x_j + nh]$ . By keeping  $ck/h < 1$ , we ensure that the domain of dependence of the exact solution is contained in the domain of dependence of the numerical solution.

For parabolic equations, like the heat equation, information propagates instantaneously throughout the domain, although its effect may be infinitesimal. If we put a flame to one end of a heat conducting bar, the temperature at the other end immediately begins to rise, albeit almost undetectably.<sup>1</sup> When we solved the heat equation using an explicit scheme like the forward Euler method, we

<sup>1</sup>Such a model is clearly not physical because heat is conducted at a finite velocity and definitely less than the speed of light. But, what mathematical models are perfect?

found a rather restrictive CFL condition. With such a scheme, information at one grid point can only propagate to its nearest neighbor in one time step. The more grid points we have, the greater the number of time steps needed to send the information from one side of the domain to the other. The matrix  $\mathbf{I} + k/h^2\mathbf{D}$  associated with the forward Euler method for the heat equation is tridiagonal.

On the other hand, the matrix  $(\mathbf{I} - k/h^2\mathbf{D})^{-1}$  associated with the backward Euler method is completely filled in. With such a scheme, information at one gridpoint is propagated to every other grid point in one time step. The normalized plots below show a values of a column taken from  $\mathbf{I} + k/h^2\mathbf{D}$  with  $h = \frac{1}{2}k^2$  and  $(\mathbf{I} - k/h^2\mathbf{D})^{-1}$  with  $h = k$ :



#### ► Lax–Friedrichs method

The upwind method is  $O(h + k)$ . It seems reasonable that we could build a more accurate,  $O(k + h^2)$  method by using a central difference approximation for the space derivative while keeping the forward Euler approximation in time. But, what about the stability? By formally replacing  $u(t, x_j)$  with  $\hat{u}(t, \xi)e^{i\xi j h}$ , the decoupled central difference system is

$$\frac{\partial}{\partial t} \hat{u}(t, \xi) = c \frac{e^{i\xi h} - e^{-i\xi h}}{2h} \hat{u}(t, \xi) = -\frac{c}{h} (i \sin \xi h) \hat{u}(t, \xi)$$

in the Fourier domain. The eigenvalues of this system are purely imaginary, but the region of stability for the forward Euler scheme is a unit circle in the left half plane, so the method is unconditionally unstable.

|  |                                                                                                     |
|--|-----------------------------------------------------------------------------------------------------|
|  | Centered-difference ( <i>unstable!</i> ) <span style="float: right;"><math>O(k + h^2)</math></span> |
|  | $\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_{j+1}^n + U_{j-1}^n}{2h} = 0 \quad (14.5)$                 |

We can modify the central difference scheme to make it stable. One way to do this is by choosing an ODE solver whose region of absolute stability includes part of imaginary axis, like the leapfrog scheme. Another way is to somehow push the eigenspectrum into the left half-plane, where they can then be shrunk down to fit into the region of absolute stability of the forward Euler scheme. The Lax–Friedrichs method approximates the  $U_j^n$  term in the time derivative of (14.5) as  $\frac{1}{2}(U_{j+1}^n + U_{j-1}^n)$ .

|  |                                                                                                     |                |
|--|-----------------------------------------------------------------------------------------------------|----------------|
|  | Lax–Friedrichs                                                                                      | $O(k + h^2/k)$ |
|  | $\frac{U_j^{n+1} - \frac{1}{2}(U_{j+1}^n + U_{j-1}^n)}{k} + c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = 0$ | (14.6)         |

The method is accurate to order  $O(k + h^2/k)$ . What about stability? We can do von Neumann analysis by formally substituting  $A^n e^{ij\xi h}$  for  $U_j^n$  in (14.6) and simplifying to get

$$A - \frac{e^{i\xi h} + e^{-i\xi h}}{2} = -c \frac{k}{h} i \sin \xi h.$$

We separate out  $A$  to get  $A = \cos \xi h - i(ck/h) \sin \xi h$ , from which we see that  $|A|^2 = \cos^2 \xi h + (ck/h)^2 \sin^2 \xi h$ . So, the Lax–Friedrichs scheme is stable if  $|c|k/h \leq 1$ . Therefore, the CFL number for the Lax–Friedrichs scheme is  $|c|k/h$ .

### ► Lax–Wendroff method

Let's look at another way to make the unstable central difference scheme (14.5) stable. Using the Taylor series expansion about  $U_j^n = u(t_n, x_j)$ , we have

$$\begin{aligned} U_j^{n+1} &= u + ku_t + \frac{1}{2}k^2u_{tt} + O(k^3) \\ U_{j+1}^n &= u + hu_x + \frac{1}{2}h^2u_{xx} + \frac{1}{6}h^3u_{xxx} + O(k^4) \\ U_{j-1}^n &= u - hu_x + \frac{1}{2}h^2u_{xx} - \frac{1}{6}h^3u_{xxx} + O(k^4). \end{aligned}$$

we see that the central difference method

$$\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = 0$$

is consistent with

$$u_t + cu_x = -\frac{1}{2}ku_{tt} + O(k^2 + h^2). \quad (14.7)$$

Because  $u_t + cu_x = 0$ , we have  $u_t = -cu_x$  from which  $u_{tt} = -cu_{tx} = c^2u_{xx}$ . Therefore, (14.7) is the same as

$$u_t + cu_x = -\frac{1}{2}c^2ku_{xx} + O(k^2 + h^2).$$

Note that  $u_t = -\frac{1}{2}c^2ku_{xx}$  is a backward heat equation, which is unstable. This term appears to be the source of the instability in the central difference method. We can fix the scheme by counteracting  $-\frac{1}{2}c^2ku_{xx}$  with  $+\frac{1}{2}c^2ku_{xx}$ . Not only will this improve stability, but it also increases the accuracy from  $O(k + h^2)$  to  $O(k^2 + h^2)$ . The new stabilized scheme is

$$\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = \frac{c^2}{2} k \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

The original central difference in space, forward-Euler scheme (14.5) failed because the eigenvalues were along the imaginary axis and fell entirely outside of the region of stability of the forward-Euler scheme no matter the size of  $k$ . By adding a viscosity term  $u_{xx}$  to the problem, we push the eigenvalues into the left-half plane and by taking  $k$  sufficiently small, we can ensure stability.

|                                                                                   |                     |                |
|-----------------------------------------------------------------------------------|---------------------|----------------|
|  | Lax–Wendroff Method | $O(k^2 + h^2)$ |
|                                                                                   |                     | (14.8)         |

$$\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = \frac{c^2}{2} k \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}$$

Note that by rearranging the terms of the Lax–Friedrichs scheme we get a form which looks very similar to the Lax–Wendroff scheme:

$$\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = \frac{h^2}{2k} \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}$$

Like the Lax–Wendroff scheme, the Lax–Friedrichs scheme achieves stability by adding a viscosity term onto the unstable central difference scheme. Whereas the Lax–Wendroff scheme adds just enough viscosity ( $c^2 k / 2$ ) to counteract the unstable second-order term, the Lax–Friedrichs scheme aggressively adds more viscosity ( $h^2 / 2k$ ). We can generalize the Lax–Wendroff and Lax–Friedrichs methods by using a diffusion coefficient  $\alpha$ :

$$\frac{\partial}{\partial t} U_j - c \frac{U_{j+1} - U_{j-1}}{2h} = \alpha \frac{U_{j+1} - 2U_j + U_{j-1}}{h^2}. \quad (14.9)$$

From earlier in this chapter and the previous chapter, we know that the Fourier transform of this equation is

$$\frac{\partial}{\partial t} \hat{u}(t, \xi) = -4 \frac{\alpha}{h^2} \sin^2 \frac{\xi h}{2} \hat{u}(t, \xi) - i \frac{c}{h} \sin \xi h \hat{u}(t, \xi)$$

with eigenvalues

$$-4 \frac{\alpha}{h^2} \sin^2 \frac{\xi h}{2} - i \frac{c}{h} \sin \xi h.$$

The eigenvalues lie on an ellipse in the negative half plane bounded by  $-4\alpha/h^2$  along the real axis and  $\pm c/h$  along the imaginary axis. For a forward Euler method, this ellipse must be scaled by  $k$  to lie entirely within the unit circle centered at  $-1$ . For the Lax–Friedrichs method,  $\alpha = h^2/2k$  and the  $k$ -scaled ellipse is

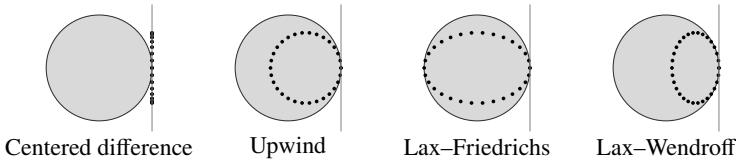
$$-2 \sin^2 \frac{\xi h}{2} - i \frac{ck}{h} \sin \xi h.$$

The ellipse always goes out to  $-2$  along the real axis no matter the value of  $k$ . So, the CFL condition for the Lax–Friedrichs method  $k < h/|c|$  is due entirely

to the imaginary components of the eigenvalues. For the Lax–Wendroff method,  $\alpha = c^2/2k$  and the  $k$ -scaled ellipse is

$$-2 \left( \frac{ck}{h} \right)^2 \sin^2 \frac{\xi h}{2} - i \frac{ck}{h} \sin \xi h.$$

When  $k < h/|c|$  the  $k$ -scaled ellipse is completely inside the unit circle and when  $k > h/|c|$  the  $k$ -scaled ellipse is completely outside the unit circle. The CFL numbers for the Lax–Wendroff method and the Lax–Friedrichs method are both  $|c|k/h$ . The following figure shows the eigenvalues for the different methods overlaid on the region of absolute stability of the forward Euler method:



The eigenvalues of the central difference scheme are outside of the region of absolute stability, so that scheme is unconditionally unstable. The eigenvalues for the Lax–Friedrichs scheme extend farther left than those of the upwind scheme, which themselves extend farther left than those of the Lax–Wendroff method. Consequently, the Lax–Friedrichs method is more dissipative than the upwind method, which itself is more dissipative than the Lax–Wendroff method.

### 14.3 Numerical diffusion and dispersion

By examining the truncation error for these numerical schemes, we will get a more complete understanding of the nature of the numerical solution. Let's start by looking at the upwind scheme. From Taylor series expansion, we find that the upwind method is a numerical approximation of

$$u_t + cu_x = \frac{1}{2} ch \left( 1 - \frac{ck}{h} \right) u_{xx} + O(h^2). \quad (14.10)$$

That is, the upwind scheme is a first order approximation to

$$u_t + cu_x = 0, \quad (14.11)$$

but it is a second order approximation to

$$u_t + cu_x = \frac{1}{2} ch \left( 1 - \frac{ck}{h} \right) u_{xx}. \quad (14.12)$$

The right-hand side of (14.12) contributes *numerical viscosity* (also called numerical diffusion or numerical dissipation) to the computed solution. You can

think of it this way: while the upwind scheme does a decent job approximating the original problem (14.11), it does a better job approximating the parabolic equation (14.12). Note what happens if we take smaller step sizes in time (keeping the mesh size in space fixed). As  $ck/h \rightarrow 0$ , we get  $u_t + cu_x = \frac{1}{2}chu_{xx}$  and the solution is overly dissipative. On the other hand, note that if we take  $k$  equal to the CFL number ( $k = h/c$ ) then not only does the  $u_{xx}$  term of (14.10) disappear, but the whole right side disappears. That is, the upwind scheme exactly solves  $u_t + cu_x = 0$  when  $k = h/c$ .

The Lax–Wendroff scheme is a second-order approximation to (14.11), but it is a third-order approximation to

$$u_t + cu_x = -\frac{1}{6}ch^2 \left(1 - \left(\frac{ck}{h}\right)^2\right) u_{xxx}. \quad (14.13)$$

The right-hand side of (14.13) contributes *numerical dispersion* to the computed solution.

There is an important distinction between dissipation and dispersion. If we take a Fourier component

$$u(t, x) = e^{i(\omega t - \xi x)}$$

as an *ansatz* to the modified upwind equation (14.12)  $u_t + cu_x = \alpha u_{xx}$ , we have the relation

$$i\omega e^{i(\omega t - \xi x)} - ic\xi e^{i(\omega t - \xi x)} = -\alpha\xi^2 e^{i(\omega t - \xi x)}$$

from which we have  $i\omega - ic\xi = -\alpha\xi^2$ . From this equation, we can determine the *dispersion relation*

$$\omega = c\xi + i\alpha\xi^2.$$

Plugging this  $\omega$  back into our ansatz gives us

$$u(t, x) = e^{i(\omega t - \xi x)} = e^{ic\xi t} e^{-\alpha\xi^2 t} e^{-i\xi x} = \underbrace{e^{i\xi(ct-x)}}_{\text{advection}} \cdot \underbrace{e^{-\alpha\xi^2 t}}_{\text{dissipation}}.$$

Dissipation damps out terms with high wave numbers  $\xi$ . Using the same ansatz in the modified Lax–Wendroff equation (14.13)  $u_t + cu_x = \beta u_{xxx}$  yields

$$i\omega e^{i(\omega t - \xi x)} - ic\xi e^{i(\omega t - \xi x)} = -i\beta\xi^3 e^{i(\omega t - \xi x)},$$

which we can simplify to derive the dispersion relation

$$\omega = c\xi + \beta\xi^3.$$

For the Lax–Wendroff scheme, we have

$$u(t, x) = e^{i(c\xi + \beta\xi^3)t} e^{-i\xi x} = \underbrace{e^{i\xi(ct-x)}}_{\text{advection}} \cdot \underbrace{e^{i\beta\xi^3 t}}_{\text{dispersion}}.$$

The *group velocity* is given by the first derivative of the dispersion relation

$$v_{\text{group}} = \frac{d\omega}{d\xi} = c + 3\beta\xi^2.$$

If the group velocity is a function of  $\xi$ , we say that the wave is *dispersive*. The *phase velocity* is given by the ratio of the dispersion relation and  $\xi$ :

$$v_{\text{phase}} = \frac{\omega}{\xi} = c + \beta\xi^2.$$

See the QR link at the bottom of the page. Because the group velocity of a dispersive wave is a function of wave number, waves all travel at different velocities. So, numerical dispersion causes oscillations in the solution. On the other hand, numerical viscosity damps out high wave numbers. High wave numbers are associated with large derivatives, for example, where there are discontinuities. In short, a dispersive scheme oscillates around a discontinuity, and a dissipative scheme smooths out the discontinuity. Figure 14.1 on the facing page shows solution to the advection equation  $u_t + u_x = 0$  using the upwind, Lax–Friedrichs, and Lax–Wendroff methods for a rectangle function when  $k$  is slightly smaller than  $h$  and when  $k$  is two-thirds  $h$ . Solutions are unstable for  $k$  is greater than  $h$ . When  $k$  is much smaller than  $h$ , the solutions exhibit significant diffusion in the upwind and Lax–Friedrichs methods or dispersion in the Lax–Wendroff method.

## 14.4 Linear hyperbolic systems

Let's turn our attention to the one-dimensional linear wave equation

$$u_{tt} - c^2 u_{xx} = 0. \quad (14.14)$$

We can rewrite this equation as the system of partial differential equations by setting  $u_t = v$  and  $u_x = w$ . Then

$$\begin{aligned} v_t - c^2 w_x &= 0 \\ w_t - v_x &= 0 \end{aligned}$$

which is equivalent to

$$\begin{bmatrix} v \\ w \end{bmatrix}_t + \begin{bmatrix} 0 & -c^2 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix}_x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

In general, one has  $\mathbf{u}_t + \mathbf{A}\mathbf{u}_x = \mathbf{0}$  with  $\mathbf{u}(t) \in \mathbb{R}^n$  and  $\mathbf{A} \in \mathbb{R}^{n \times n}$ . This system is called *strictly hyperbolic* if  $\mathbf{A}$  has  $n$  real eigenvalues. In this case, the matrix  $\mathbf{A}$  has a complete set of linearly independent eigenvectors and can be diagonalized



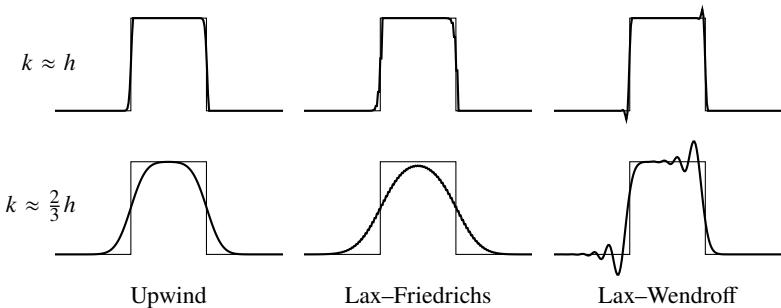


Figure 14.1: Solution to the advection equation for an initial rectangle function. Solutions move from left to right. See the QR link at the bottom of the page.

in real space as  $\mathbf{T}^{-1}\mathbf{A}\mathbf{T} = \mathbf{\Lambda} = \text{diag}(\nu_1, \nu_2, \dots, \nu_n)$ , where  $\mathbf{T}$  is a matrix of eigenvectors  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ . That is,  $\mathbf{A}\mathbf{v}_i = \nu_i \mathbf{v}_i$ . The eigenvalues  $\nu_i$  are the *characteristic speeds*. For the wave equation (14.14), the characteristic speeds are  $\nu_{\pm} = \pm c$ .

By diagonalizing the matrix  $\mathbf{A}$ , the system is completely decoupled into a system of independent advection equations. Consider

$$\mathbf{u}_t + \mathbf{A}\mathbf{u}_x = \mathbf{0}.$$

Multiply this equation by  $\mathbf{T}^{-1}$  to get  $\mathbf{T}^{-1}\mathbf{u}_t + \mathbf{T}^{-1}\mathbf{A}\mathbf{u}_x = \mathbf{0}$ , which is simply  $\mathbf{T}^{-1}\mathbf{u}_t + \mathbf{\Lambda}\mathbf{T}^{-1}\mathbf{u}_x = \mathbf{0}$ . By letting  $\mathbf{v} = \mathbf{T}^{-1}\mathbf{u}$  we have the equivalent system

$$\mathbf{v}_t + \mathbf{\Lambda}\mathbf{v}_x = \mathbf{0},$$

in which each variable is now uncoupled:

$$\frac{\partial v_i}{\partial t} + \nu_i \frac{\partial v_i}{\partial x} = 0 \quad (14.15)$$

for  $i = 1, 2, \dots, n$ . By using the method of characteristics, we get the solution  $v_i(t, x) = v_i(0, x - \nu_i t)$ . The initial conditions are given by  $\mathbf{v}(0) = \mathbf{T}\mathbf{u}(0)$ . By setting  $\mathbf{u}(t) = \mathbf{T}\mathbf{v}(t)$ , the problem is solved.

Now, let's solve the problem numerically. We can discretize the uncoupled system (14.15) using the upwind scheme

$$\frac{(V_i)_j^{n+1} - (V_i)_j^n}{k} + \frac{\nu_i + |\nu_i|}{2} \frac{(V_i)_j^n - (V_i)_{j-1}^n}{h} + \frac{\nu_i - |\nu_i|}{2} \frac{(V_i)_{j+1}^n - (V_i)_j^n}{h} = 0.$$

If we let  $\mathbf{\Lambda}^+$  denote the diagonal matrix whose elements are  $\frac{1}{2}(\nu_i + |\nu_i|)$  and  $\mathbf{\Lambda}^-$  denote the diagonal matrix whose elements are  $\frac{1}{2}(\nu_i - |\nu_i|)$ , then uncoupled



upwind scheme is the same as

$$\frac{\mathbf{V}_j^{n+1} - \mathbf{V}_j^n}{k} + \Lambda^+ \frac{\mathbf{V}_j^n - \mathbf{V}_{j-1}^n}{h} + \Lambda^- \frac{\mathbf{V}_{j+1}^n - \mathbf{V}_j^n}{h} = 0.$$

By making the substitution  $\mathbf{U}_j^n = \mathbf{T}\mathbf{V}_j^n$  and defining  $\mathbf{A}^\pm = \mathbf{T}\Lambda^\pm\mathbf{T}^{-1}$ , we can rewrite the above equation as

$$\frac{\mathbf{U}_j^{n+1} - \mathbf{U}_j^n}{k} + \mathbf{A}^+ \frac{\mathbf{U}_j^n - \mathbf{U}_{j-1}^n}{h} + \mathbf{A}^- \frac{\mathbf{U}_{j+1}^n - \mathbf{U}_j^n}{h} = 0$$

We call  $\mathbf{A}^\pm = \mathbf{T}\Lambda^\pm\mathbf{T}^{-1}$  the *characteristic decomposition*.

Let's extend the ideas developed for linear hyperbolic equations to nonlinear hyperbolic equations. Consider the equation

$$\frac{\partial}{\partial t} u + \frac{\partial}{\partial x} f(u) = 0 \quad (14.16)$$

for a function  $u(t, x)$  with  $u(0, x) = u_0(x)$  and a function  $f(u)$  called the *flux*. Such an equation is called a *conservation law*. If we integrate it with respect to  $x$ , we have

$$\frac{d}{dt} \int_a^b u \, dx + \int_a^b \frac{\partial}{\partial x} f(u) \, dx = \frac{d}{dt} \int_a^b u \, dx + (f(b) - f(a)) = 0.$$

If there is no net flux through the boundary, then  $\frac{d}{dt} \int_a^b u \, dx = 0$ , from which it follows that  $\int_a^b u \, dx$  is constant. So,  $u$  is a conserved quantity.

If the flux  $f(u)$  is differentiable, then we can apply the chain rule and (14.16) becomes

$$\frac{\partial u}{\partial t} + f'(u) \frac{\partial u}{\partial x} = 0,$$

which can be solved using the method of characteristics

$$\frac{d}{dt} u(t, x(t)) = \frac{\partial u}{\partial t} + \frac{dx}{dt} \frac{\partial u}{\partial x} = \frac{\partial u}{\partial t} + f'(u) \frac{\partial u}{\partial x} = 0.$$

This says that the solution  $u(t, x)$  is constant (a Riemann invariant) along the characteristics given by

$$\frac{dx}{dt} = f'(u).$$

Because  $u$  is invariant along  $x(t)$ , it follows that

$$\frac{dx}{dt} = f'(u(t, x(t))) = f'(u(0, x(0))) = f'(u_0(x_0)).$$

Integrating this equation, we see that the characteristics are straight lines

$$x(t) = f'(u_0(x_0))t + x_0,$$

and the solution is given by

$$u(t, x) = u_0(x - f'(u_0)t).$$

However, if  $u(t, x)$  is not differentiable (or at least not Lipschitz continuous) at some point, the chain rule is no longer valid. Even if  $u_0(x)$  is initially smooth,  $u(t, x)$  may not necessarily be differentiable for all time if the flux is nonlinear. To see what happens, we consider the inviscid Burgers equation, a toy model for the Euler equations of gas dynamics.

### ► Burgers' equation

Burgers' equation  $u_t + uu_x = \varepsilon u_{xx}$  is a simplification of Navier–Stokes equation.<sup>2</sup> By neglecting the viscosity, we have the inviscid Burgers equation  $u_t + uu_x = 0$ . Furthermore, if the solution  $u(t, x)$  remains differentiable, we can write Burgers' equation as a conservation law  $u_t + f(u)_x = 0$  for a flux  $\frac{1}{2}u^2$ . We'll take the initial conditions  $u(0, x) = u_0(x)$ .

Formally, the variable wave speed  $u$  in the inviscid Burgers' equation plays the same role as the constant wave speed  $c$  in the linear hyperbolic equation  $u_t + cu_x = 0$ . The wave speed  $u$  is a Riemann invariant—the characteristics  $x_0(t)$  are straight line paths with slopes  $dx/dt = u(0, x) = u_0(x)$  along which  $u(t, x(t)) = u(0, x(0))$  is constant. The larger the value of  $u_0(x)$ , the faster the solution  $u(t, x(t))$  moves; the smaller the value of  $u_0(x)$ , the slower the solution  $u(t, x(t))$  moves.

It seems quite plausible that given the right initial conditions a fast moving characteristic might eventually catch up with and overtake a slow moving characteristic. Specifically, if a characteristic curve of higher slope is to the left of one of lower slope, then the two curves will intersect at some point. See Figure 14.2 on the next page and the QR link at the bottom of this page. The solution at this point would then be multivalued because information comes from two (or possibly more) characteristic curves. Furthermore, at the moment when a multivalued solution in  $u(t, x)$  appears, its derivative blows up and the solution overturns.

We can find the point when solution overturns by computing when the derivative first becomes infinite.

$$\frac{du}{dx} = \frac{du}{dx_0} \frac{dx_0}{dx} = u'_0(x_0) \left( \frac{dx}{dx_0} \right)^{-1} = \frac{u'_0(x)}{u'_0(x)t + 1} \quad (14.17)$$

If we set the right-hand-side to infinity and solve for  $t$ , we have  $t = -(u'_0(x_0))^{-1}$ . So, the time at which the solution first becomes multivalued is the infimum of

---

<sup>2</sup>Jan Burgers: “I have no objection to the use of the term ‘Burgers equation’ for the nonlinear heat flow equation (provided it is not written ‘Burger’s equation’).”



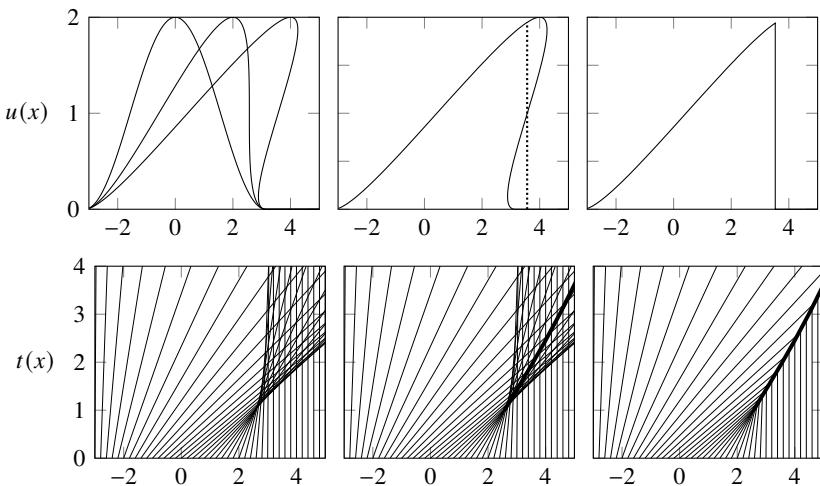


Figure 14.2: Top left: solutions  $u$  to Burgers' equation at  $t = 0$  (initial shock formation),  $t_c = 1$  (initial shock formation), and  $t = 2$  (unphysical multivalued solution). Top right: weak solution at  $t = 2$ . Bottom: characteristic curves  $x(t)$ .

$-(u'_0(x_0))^{-1}$ . The solution to Burgers' equation becomes singular if and only if the gradient of the initial distribution  $u_0(x)$  is ever negative.

Can a faster moving part really overtake a slower moving part? No. A multivalued solution is both unphysical and mathematically ill-posed. So, it seems that something is wrong with either our formulation or our reasoning. We relied on the fact that  $u(t, x)$  is smooth in order to apply the method of characteristics. But as soon as the derivative of  $u(t, x)$  blows up, the method is clearly invalidated. In the next section, we will look at how we can solve the ill-posed problem.<sup>3</sup>

### ► Weak solutions

The solution to Burgers' equation by the method of characteristics is in general ill-posed, because it provides a multivalued solution. So, we need to choose a solution with the most *physical* relevance. We do this mathematically by extending the solution as a “generalized weak solution.” A weak solution allows us to solve a differential equation even when that solution is not differentiable. For something like Burgers' equation, this weak solution incorporates a type of

<sup>3</sup>An alternative method of solution starts with the viscous Burgers equation  $u_t + uu_x = \varepsilon u_{xx}$  and applies the Hopf–Cole transformation  $u = -2\varepsilon \frac{d}{dx}(\log \phi)$ , which changes the equation into a heat equation  $\phi_t = \varepsilon \phi_{xx}$ . The heat equation is solved as a fundamental solution to the Cauchy problem. Finally, the inviscid solution is expressed as the limit as  $\varepsilon \rightarrow 0$ .

discontinuity called a *shock*. To construct a weak solution of  $u_t + f(u)_x = 0$ , multiply the equation by a sufficiently smooth *test function*  $\varphi$  that vanishes at the boundaries; integrate over space and time; and finally, use integration by parts to pass the derivatives over to  $\varphi$ . Since  $\varphi$  vanishes at the boundaries, there will be no boundary terms. Essentially, we are using a well-behaved function  $\varphi$  to take the derivatives of  $f(u)$ , so that we can avoid differentiating  $f(u)$ , particularly when  $f(u)$  becomes non-differentiable. In this chapter, we'll use weak solutions as a means of determining an analytical solution. In the next chapter, we'll apply weak solutions numerically as the finite element method.

Suppose that we have a test function  $\varphi$ , which is differentiable and has compact support, meaning that the closure of the set over which  $\varphi$  is nonzero is bounded. Notably,  $\varphi$  is zero at the boundaries, where we need it to be zero. We denote this by  $\varphi \in C_0^1(\Omega)$ . The  $C^1$  denotes that the function and its first derivative are continuous, and the subscript 0 denotes that the function has compact support over the region  $\Omega = \{(x, t)\} \subset \mathbb{R}^2$ . In this case,  $u_t + f(u)_x = 0$  implies

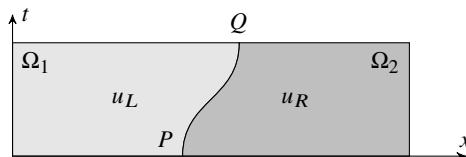
$$\iint_{\Omega} (\varphi u_t + \varphi f(u)_x) dx dt = 0.$$

We integrate by parts to get

$$\iint_{\Omega} (\varphi_t u + \varphi_x f(u)) dx dt = 0.$$

If this equation holds for all  $\varphi \in C_0^1(\Omega)$ , then we say that  $u(t, x)$  is a *weak solution* of  $u_t + f(u)_x = 0$ . If  $u(t, x)$  is smooth, the weak solution is the classical or *strong solution*.

Consider an arbitrary curve  $PQ$  dividing  $\Omega$  into subdomains  $\Omega_1$  and  $\Omega_2$ .



If  $u(t, x)$  is a weak solution to  $u_t + f(u)_x = 0$ , then

$$\begin{aligned} 0 &= - \iint_{\Omega} \varphi u_t + \varphi f(u)_x dx dt + \iint_{\Omega} \varphi u_t + \varphi f(u)_x dx dt \\ &= \iint_{\Omega} \varphi_t u + \varphi_x f(u) dx dt + \iint_{\Omega} \varphi u_t + \varphi f(u)_x dx dt \end{aligned}$$

Because  $\varphi$  vanishes on the boundary:

$$\begin{aligned} &= \iint_{\Omega} (\varphi u)_t + (\varphi f(u))_x \, dx \, dt \\ &= \iint_{\Omega_1} (\varphi u)_t + (\varphi f(u))_x \, dx \, dt + \iint_{\Omega_2} (\varphi u)_t + (\varphi f(u))_x \, dx \, dt. \end{aligned}$$

By Green's theorem:

$$= \int_{\partial\Omega_1} -\varphi u \, dx + \varphi f(u) \, dt + \int_{\partial\Omega_2} -\varphi u \, dx + \varphi f(u) \, dt.$$

Because  $\varphi(\partial\Omega) = 0$ :

$$\begin{aligned} &= \int_P^Q -\varphi u_L \, dx + \varphi f(u_L) \, dt + \int_Q^P -\varphi u_R \, dx + \varphi f(u_R) \, dt \\ &= \int_P^Q \varphi(u_R - u_L) \, dx - \varphi(f(u_R) - f(u_L)) \, dt, \end{aligned}$$

where  $u_L$  is the value of  $u$  along  $PQ$  from the left and  $u_R$  is the value of  $u$  along  $PQ$  from the right. So, for all test functions  $\varphi \in C_0^1(\Omega)$ ,

$$(u_R - u_L) \, dx - (f(u_R) - f(u_L)) \, dt = 0$$

along  $PQ$ . Therefore,

$$s = \frac{dx}{dt} = \frac{f(u_R) - f(u_L)}{u_R - u_L}.$$

This expression is called the *Rankine–Hugoniot jump condition* for shocks, and  $s$  is called the *shock speed*. Note that in the limit as  $u_L \rightarrow u_R$ , the shock speed  $s \rightarrow f'(u)$ . To simplify notation, one often uses brackets to indicate difference:

$$s = \frac{f(u_R) - f(u_L)}{u_R - u_L} \equiv \frac{[f]}{[u]}.$$

We can now continue to use the PDE with the weak solution.

## ► The Riemann problem

A *Riemann problem* is an initial value problem with the initial data is given by two constant states separated by a discontinuity. For the advection equation, it is

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{with} \quad u(0, x) = \begin{cases} u_L & x < 0 \\ u_R & x > 0 \end{cases}. \quad (14.18)$$

Notice that if we scale  $t$  and  $x$  both by a positive constant  $c$ , i.e., if we take  $\hat{t} = ct$  and  $\hat{x} = cx$ , then the scaled problem is identical to the original problem

$$\frac{\partial u}{\partial \hat{t}} + \frac{\partial}{\partial \hat{x}} f(u) = 0 \quad \text{with} \quad u(0, \hat{x}) = \begin{cases} u_L & \hat{x} < 0 \\ u_R & \hat{x} > 0 \end{cases}$$

This suggests an ansatz for the problem. We'll look for a *self-similar solution* to Burgers' equation to simplify the two-variable PDE into a one-variable ODE. Let  $\xi = x/t = \hat{x}/\hat{t}$ . Taking  $u(t, x) = u(\xi)$  in Burgers' equation:

$$0 = u_t + \left(\frac{1}{2}u^2\right)_x = \xi_t u' + \xi_x uu' = \left(-\frac{x}{t^2}\right)u' + \left(\frac{1}{t}\right)uu'.$$

Multiplying by  $t$  and replacing  $x/t$  by  $\xi$ , we have

$$-\xi u' + uu' = u'(u - \xi) = 0.$$

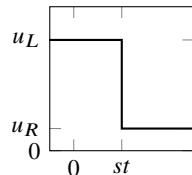
So, either  $u' = 0$ , in which case  $u(x)$  is constant; or  $u = \xi$ , in which case  $u(t, x) = x/t$ . The solution may also be a piecewise combination of these solutions. Let's consider the two cases: when  $u_L > u_R$  and when  $u_L < u_R$ .

1. If  $u_L > u_R$ , then the Rankine–Hugoniot condition says

$$s = \frac{\frac{1}{2}u_R^2 - \frac{1}{2}u_L^2}{u_R - u_L} = \frac{1}{2}(u_R + u_L)$$

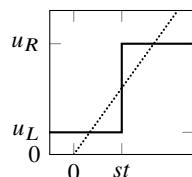
and the exact solution to the Riemann problem is

$$u(t, x) = \begin{cases} u_L, & \text{if } x/t < s \\ u_R, & \text{if } x/t > s \end{cases}$$



2.  $u_L < u_R$ , then one exact solution is

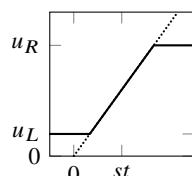
$$u(t, x) = \begin{cases} u_L, & \text{if } x/t < s \\ u_R, & \text{if } x/t > s \end{cases}$$



where  $s = (u_R + u_L)/2$  is given by the Rankine–Hugoniot condition.

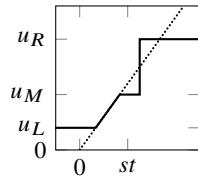
Another solution is

$$u(t, x) = \begin{cases} u_L, & \text{if } x/t < u_L \\ x/t, & \text{if } u_L < x/t < u_R \\ u_R, & \text{if } x/t > u_R \end{cases}$$



A third solution is

$$u(t, x) = \begin{cases} u_L, & \text{if } x/t < u_L \\ x/t, & \text{if } u_L < x/t < u_M \\ u_M, & \text{if } u_M < x/t < s \\ u_R, & \text{if } x/t > s \end{cases}$$



where  $s = (u_R + u_M)/2$  is given by the Rankine–Hugoniot condition.

In fact, there are *infinitely* many weak solutions. What is the *physically* permissible weak solution? To find it, we need to impose another condition. Physically, there is a quantity called “entropy” which is a constant along smooth particle trajectories but jumps to a higher value across a discontinuity. The mathematical definition of entropy  $\varphi(u)$  is the negative of physical entropy. For the problem  $u_t + f(u)_x = 0$  where  $f(u)$  is the convex flux, a solution satisfies the *Lax entropy condition*

$$f'(u_L) > s \equiv \frac{f(u_R) - f(u_L)}{u_R - u_L} > f'(u_R).$$

This says that the characteristics, which have speed  $f'(u)$ , must impinge from both sides on the shock, which has speed  $s$ . For Burgers’ equation  $f(u) = \frac{1}{2}u^2$ , so  $f'(u) = u$ .

1. If  $u_L > u_R$ , then  $f'(u_L) = u_L > \frac{1}{2}(u_R + u_L) > u_R = f'(u_R)$ . Therefore, the solution is an entropy shock.
2. If  $u_L < u_R$ , then the shock does not satisfy the entropy condition. So, the only possible weak solution is a continuous, rarefaction wave.

**Example.** We can model traffic using a nonlinear hyperbolic equation. Suppose that we have a single-lane road without any on or off ramps (no sources or sinks). Let  $\rho(t, x)$  be the density of cars measured in cars per car length. So,  $\rho = 0$  says that the road is completely empty and  $\rho = 1$  says that there is bumper-to-bumper traffic. Then

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x} f(\rho) = 0$$

models the traffic flow, where the flux  $f(\rho)$  tells us the density of cars passing a given point in a given time interval (number of cars per second). We can define flux  $f(\rho) = u(\rho)\rho$ , where  $u(\rho)$  simply tells us the speed of each car as a function of traffic density. Let’s make a couple reasonable assumptions to model  $u(\rho)$ :

1. Everyone travels as fast as possible while still obeying the speed limit  $u_{\max}$ .
2. Everyone keeps a safe distance behind the car ahead of them and adjusts their speed accordingly. Drive as fast as possible if there are no other cars on the road. Stop moving if traffic is bumper-to-bumper.

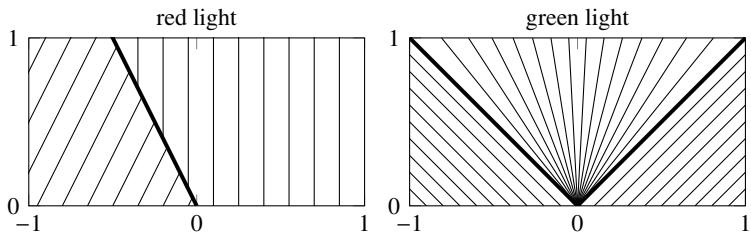


Figure 14.3: Characteristics for traffic flow with  $\rho_L = \frac{1}{2}$ .

In this case, the simplest model for  $u(\rho)$  is  $u(\rho) = u_{\max}(1 - \rho)$ . Notice the that flux  $f(\rho) = u(\rho)\rho = u_{\max}(\rho - \rho^2)$  is zero when the roads are completely empty (there are no cars to move) and when the roads are completely full (no one is moving). The flux  $f(\rho)$  is maximum at  $f'(\rho) = u_{\max}(1 - 2\rho) = 0$  when  $\rho = \frac{1}{2}$  (the roads are half full with the cars traveling at  $\frac{1}{2}u_{\max}$ , half the posted speed limit). So, our traffic equation is now

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(u_{\max}(1 - \rho)\rho) = 0.$$

This equation is very similar in form to the Burgers equation. Let's consider two simple examples: cars approaching a red stop light and cars leaving after the stop light turns green. Both of these examples are Riemann problems.

**Red light.** We imagine several cars traveling with density  $\rho_L$  approaching a queue of stopped cars with density  $\rho_R = 1$ . In this case

$$\rho(x, 0) = \begin{cases} \rho_L, & x < 0 \\ \rho_R = 1, & x \geq 0. \end{cases}$$

The Rankine–Hugoniot relationship says

$$s = \frac{f(\rho_L) - f(\rho_R)}{\rho_L - \rho_R} = \frac{u_{\max}\rho_L(1 - \rho_L) - 0}{\rho_L - 1} = -u_{\max}\rho_L.$$

There is a shock wave moving backwards as the cars queue up.

**Green light.** We imagine several cars in queue with  $\rho = 1$  and speed  $u_L = 0$ . This time the Lax entropy condition tells us that there is a rarefaction wave. The leading car moves forwards with a speed  $u_{\max}$  and a rarefaction wave moves backwards with a speed  $f'(\rho) = u_{\max}(1 - 2\rho)$  with  $\rho = 1$ , i.e., speed  $f'(1) = -u_{\max}$ . See Figure 14.3 above. ◀

## 14.5 Hyperbolic systems of conservation laws

Let's extend the discussion of one-dimensional nonlinear equations to a system of  $n$  one-dimensional nonlinear conservation laws

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial}{\partial x} \mathbf{f}(\mathbf{u}) = 0, \quad (14.19)$$

with  $\mathbf{u}(0, x) = \mathbf{u}_0(x)$ , where  $\mathbf{u} = (u_1, u_2, \dots, u_n)$  and  $\mathbf{f}(\mathbf{u}) = (f_1, f_2, \dots, f_n)$ . We can write this system in quasilinear form

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{f}'(\mathbf{u}) \frac{\partial \mathbf{u}}{\partial x} = 0$$

where the Jacobian  $\mathbf{f}'(\mathbf{u})$  has elements  $\partial f_i / \partial u_j$ . If the Jacobian  $\mathbf{f}'(\mathbf{u})$  has  $n$  real eigenvalues and hence a complete set of linearly independent eigenvectors, then the system (14.19) is called *hyperbolic*. In this case, there is an invertible map  $\mathbf{T}(\mathbf{u})$  such that  $\mathbf{T}^{-1} \mathbf{f}' \mathbf{T} = \mathbf{\Lambda} = \text{diag}(\nu_1, \nu_2, \dots, \nu_n)$ . The eigenvalues  $\nu_i$  are the characteristic speeds and  $\mathbf{T}$  is the matrix of eigenvectors.

**Example.** Let's examine the one-dimensional shallow water equations and show that they form a hyperbolic system. The shallow water equations are

$$h_t + (hu)_x = 0 \quad (14.20a)$$

$$(hu)_t + (hu^2 + \frac{1}{2}gh^2)_x = 0 \quad (14.20b)$$

where  $g$  is the gravitational acceleration,  $h$  is the height of the water, and  $u$  is the velocity of the water. The mass of a column of water is proportional to its height, and we can let  $m = hu$  denote the momentum. Then (14.20) becomes

$$\begin{aligned} h_t + m_x &= 0 \\ m_t + \frac{2m}{h}m_x - \frac{m^2}{h^2}h_x + gh h_x &= 0, \end{aligned}$$

which can be written as

$$\begin{bmatrix} h \\ m \end{bmatrix}_t + \begin{bmatrix} 0 & 1 \\ gh - u^2 & 2u \end{bmatrix} \begin{bmatrix} h \\ m \end{bmatrix}_x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The eigenvalues of the Jacobian matrix  $\mathbf{f}'$  are given by the zeros of

$$\begin{vmatrix} -\nu & 1 \\ gh - u^2 & 2u - \nu \end{vmatrix} = \nu^2 - 2uv + u^2 - gh.$$

So  $\nu_{\pm} = u \pm \sqrt{gh}$ . If the height of the water  $h$  is positive, then there are two real eigenvalues and the system is strictly hyperbolic.

The two eigenvalues give us the characteristic speeds for the shallow water equation. Notably the speed of water waves are functions of the depth of the water  $h$  along with the velocity of the current  $u$ . Tsunamis, for example, occur when a disturbance creates a wave in the ocean with a wavelength of a hundred kilometers or more and a displacement of a meter or less. The wave travels quickly in the ocean where the depth may be several kilometers—the speed is  $\sqrt{gh}$ . Once it reaches the continental shelf, where the depth decreases dramatically, the wave slows considerably, and the height of the wave grows. A shock forms as the faster characteristics intersect with the slower characteristics.

Refraction of water waves is also caused by the  $\sqrt{gh}$  speed dependence. Ocean waves turn to hit the beach perpendicularly even when the wind is not blowing straight into shore. Waves are produced by a Kelvin–Helmholtz instability resulting from the wind blowing over the surface of the water. In the deeper water, waves are driven in the direction of the wind. When they reach shallow water, the wave moves at different speeds. The part of the wave in shallower water moves slower than the part in deeper, turning the wave into shore

◀

## ► Riemann invariants

Let's recap what we've discussed so far about Riemann invariants. Information is advected along characteristics, and the quantity that is constant along a characteristic curve is called the *Riemann invariant*. For the advection equation  $u_t + cu_x = 0$ , the characteristics are given by  $dx/dt = c$  and the Riemann invariant is  $u$ . For a general scalar conservation law  $u_t + f(u)_x = 0$ , we have  $u_t + f'(u)u_x = 0$  when  $u$  is differentiable. The characteristics are given by  $dx/dt = f'(u)$  and it follows that

$$\frac{d}{dt}u(t, x(t)) = \frac{\partial u}{\partial t} + \frac{dx}{dt}\frac{\partial u}{\partial x} = \frac{\partial u}{\partial t} + f'(u)\frac{\partial u}{\partial x} = 0.$$

So,  $u(t, x)$  is the Riemann invariant.

For systems of conservation laws, it's a little more complicated. Let's once again consider the system

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial}{\partial x} \mathbf{f}(\mathbf{u}) = \mathbf{0}$$

with  $\mathbf{u} \in \mathbb{R}^n$ . If  $\mathbf{u}$  is differentiable,

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{f}'(\mathbf{u}) \frac{\partial \mathbf{u}}{\partial x} = \mathbf{0} \tag{14.22}$$

where  $\mathbf{f}'$  is the Jacobian matrix of  $\mathbf{f}$  with real eigenvalues  $\{\nu_1, \nu_2, \dots, \nu_n\}$ . Let  $\nabla_{\mathbf{u}} w_i$ —for some  $w(\mathbf{u})$ —be the *left eigenvector* of  $\mathbf{f}'(\mathbf{u})$  corresponding to the eigenvalues  $\nu_i$ . Then left multiplying (14.22) by  $\nabla_{\mathbf{u}} w_i$  yields

$$\nabla_{\mathbf{u}} w_i \frac{\partial \mathbf{u}}{\partial t} + \nabla_{\mathbf{u}} w_i \mathbf{f}'(\mathbf{u}) \frac{\partial \mathbf{u}}{\partial x} = \mathbf{0}.$$

Because  $\nabla_{\mathbf{u}} w_i$  is the left eigenvector of  $\mathbf{f}'(\mathbf{u})$ , it follows that

$$\nabla_{\mathbf{u}} w_i \frac{\partial \mathbf{u}}{\partial t} + v_i \nabla_{\mathbf{u}} w_i \frac{\partial \mathbf{u}}{\partial x} = \mathbf{0}.$$

And by the chain rule, we have

$$\frac{\partial w_i}{\partial t} + v_i \frac{\partial w_i}{\partial x} = \mathbf{0} \quad (14.23)$$

which is simply the scalar advection equation. So,  $w_i$  is a Riemann invariant, because it is constant along the characteristic curve given by  $dx/dt = v_i$ . Notice that (14.23) is the diagonalization of the original system.

**Example.** Let's compute the Riemann invariants for the shallow water equations. The Jacobian matrix

$$\mathbf{f}' = \begin{bmatrix} 0 & 1 \\ gh - u^2 & 2u \end{bmatrix}$$

has eigenvalues  $v_{\pm} = u \pm \sqrt{gh}$ . We compute the left eigenvectors to be  $(u \mp \sqrt{gh}, 1)$ . Recall that we defined  $\mathbf{u} = (h, m)$  with  $m = hu$ . In terms of  $h$  and  $m$ , the eigenvectors are  $(mh^{-1} \mp \sqrt{gh}, 1)$ . We need these vectors to be exact differentials, so let's rescale them by  $1/h$ . We now have

$$\nabla_{\mathbf{u}} w_{\pm} = \left( \frac{m}{h^2} \mp \sqrt{\frac{g}{h}}, \frac{1}{h} \right).$$

So, the Riemann invariants are

$$w_{\pm}(h, m) = \frac{m}{h} \mp 2\sqrt{gh} = u \mp 2\sqrt{gh},$$

which are constant along the characteristics  $dx/dt = u \pm \sqrt{gh}$ . ◀

## ► Open boundary conditions

The advection equation  $u_t + cu_x = 0$  has one space derivative, so it needs one influx boundary condition for uniqueness. A hyperbolic system of conservation laws (14.19) with  $n$  equations needs  $n$  boundary conditions.

Often we want to solve the shallow water equations or the Euler equations using open boundary conditions. For example, in the ocean we may want to prescribe inflow boundary conditions to model incoming water waves. Or we may want to model airflow over an airfoil by injecting a flow upwind of the airfoil. Downwind of the airfoil, we want the gas to escape the domain without reflecting back into our calculations. Or maybe we may want to model a bomb blast in the semi-infinite domain above the ground. Without an open boundary, information

will likely be reflected back into our domain of interest, polluting our solution. It is typically a bad idea to force the boundary to be equal to the external data, because the external data does not perfectly match the information leaving the domain.

One simple fix is to use a larger domain and stop computation before the reflected information pollutes the solution—effectively, putting the boundaries outside of the domain of dependence. We could also use a thin sponge layer with added viscosity to absorb energy at the boundary. Another approach, developed by Björn Enquist and Andrew Majda in 1977, creates an artificial boundary that tries to match the reflected waves with waves of opposite amplitude.

In the next several paragraphs, we'll examine a characteristics-based approach to implementing open boundary conditions. Recall that the Riemann invariant is constant along a characteristic. For the shallow water equation, we have the Riemann invariant  $w_+ = u - 2\sqrt{gh}$  along the characteristic of slope  $u + \sqrt{gh}$  and the Riemann invariant  $w_- = u + 2\sqrt{gh}$  along the characteristic of slope  $u - \sqrt{gh}$ . We'll match the Riemann invariants of the two characteristics intersecting at each boundary. We can then determine  $u$  and  $h$  using the values of the Riemann invariants:

$$u = (w_- + w_+)/2 \quad \text{and} \quad h = (w_-^2 - w_+^2)/16g.$$

Take the boundaries at  $x = x_L$  and  $x = x_R$ , and suppose that the external values are given by  $h(t, x_L) = h_L$  and  $u(t, x_L) = u_L$  on the left and  $h(t, x_R) = h_R$  and  $u(t, x_R) = u_R$  on the right. Let's partition the domain into  $m$  gridpoints at  $\{x_1, x_2, \dots, x_m\}$ . If  $|u| < \sqrt{gh}$ , then the characteristics travel in opposite directions. We'll use extrapolated values to compute the Riemann invariant for the characteristic leaving the domain, and we'll use the boundary values to compute the Riemann invariant for the characteristic entering the domain. So, on the left boundary of the domain

$$w_- = \text{extrapolation of } u + 2\sqrt{gh} \quad \text{and} \quad w_+ = u_L - 2\sqrt{gh_L}.$$

To determine  $w_-$  at  $x_0$  with second-order accuracy, we extrapolate

$$w_- = (2U_1 - U_2) + 2\sqrt{g(2H_1 - H_2)},$$

where  $H_i$  and  $U_i$  are the numerical solutions for  $h$  and  $u$  at  $x_i$ . The value  $w_+ = u_L - 2\sqrt{gh_L}$  is set according to the external values  $u_L$  and  $h_L$ . Similarly, on the right boundary of the domain

$$w_- = u_R + 2\sqrt{gh_R} \quad \text{and} \quad w_+ = \text{extrapolation of } u - 2\sqrt{gh}.$$

To determine  $w_+$  at  $x_{m+1}$  with second-order accuracy, we extrapolate

$$w_+ = (2U_m - U_{m-1}) - 2\sqrt{g(2H_m - H_{m-1})}.$$

The value  $w_- = u_R + 2\sqrt{gh_R}$  is set according to the external values  $u_R$  and  $h_R$ .

If  $|u| > \sqrt{gh}$ , then both characteristics travel in the same direction, and we match both Riemann invariants on the influx boundary and extrapolate both Riemann invariants on the outflux boundary.

### ► The Riemann problem

First-order Godonov schemes approximate a problem using a series of Riemann problems—one for every mesh point in space. These Riemann problems are solved analytically and the resulting solutions are numerically remeshed. The solution to the Riemann problem is good as long as characteristics don't cross—which happens to determine the CFL condition.

The Riemann problem for a system of hyperbolic equations

$$\mathbf{u}_t + \mathbf{f}(\mathbf{u})_x = \mathbf{0} \quad \text{with} \quad \mathbf{u}(x, 0) = \begin{cases} \mathbf{u}_L, & x < 0 \\ \mathbf{u}_R, & x > 0 \end{cases}$$

can be solved analytically using a self-similar solution by taking the ansatz  $\mathbf{u}(t, x) = \mathbf{u}(x/t) = \mathbf{u}(\xi)$ . Then

$$\left(-\frac{x}{t^2}\right)\mathbf{u}' + \frac{1}{t}\mathbf{f}'(\mathbf{u})\mathbf{u}' = \mathbf{0},$$

from which we have

$$(\mathbf{f}'(\mathbf{u}) - \xi \mathbf{I})\mathbf{u}' = \mathbf{0}.$$

So either  $\mathbf{u}' = \mathbf{0}$  in which case  $\mathbf{u}$  is constant in  $\xi$  or  $\det(\mathbf{f}'(\mathbf{u}) - \xi \mathbf{I}) = \mathbf{0}$  in which case  $\xi$  is an eigenvalue of  $\mathbf{f}'(\mathbf{u})$ . The solution is piecewise constant connected by shocks or rarefactions when  $\mathbf{u}' = \mathbf{0}$ . The solution associated with  $\det(\mathbf{f}'(\mathbf{u}) - \xi \mathbf{I}) = \mathbf{0}$  is a *contact discontinuity* that travels with characteristic speed  $\xi$ . Unlike shock discontinuities, across which there is mass flow, contact discontinuities are merely discontinuities that flow with the fluid. Think of the surface separating two different types of fluids that moving together.

## 14.6 Methods for nonlinear hyperbolic systems

A discretization  $U_j$  is *conservative* if  $\sum_{j=1}^m U_j$  is constant in time. Equivalently, a scheme is conservative if it can be written as

$$\frac{\partial}{\partial t} U_j + \frac{F_{j+1/2} - F_{j-1/2}}{h} = 0 \tag{14.24}$$

where  $F_{j+1/2}$  is the numerical flux. In this case, by summing (14.24) over all gridpoints it follows that  $\frac{\partial}{\partial t} \sum_{j=1}^m U_j = 0$  if there is no net numerical flux

through the boundaries. In a foundational 1959 paper, Peter Lax and Burton Wendroff proved their Lax–Wendroff theorem, which states that if a consistent, conservative scheme converges, then it converges to the weak solution of a conservation law.

In particular, we can rewrite the nonlinear Lax–Friedrichs method

$$\frac{U_j^{n+1} - \frac{1}{2}(U_{j+1}^n + U_{j-1}^n)}{k} + \frac{f(U_{j+1}^n) - f(U_{j-1}^n)}{2h} = 0$$

in conservative form

$$\frac{U_j^{n+1} - U_j^n}{k} + \frac{F_{j+1/2}^n - F_{j-1/2}^n}{h} = 0$$

where the numerical flux

$$F_{j+1/2}^n = \frac{f(U_{j+1}^n) + f(U_j^n)}{2} - \frac{h^2}{2k} \frac{U_{j+1}^n - U_j^n}{h}.$$

Recall from the discussion on page 401 that the second term adds artificial viscosity to an otherwise unstable central difference scheme. The diffusion coefficient  $\alpha = h^2/2k$ , however, tends to make the Lax–Friedrichs scheme overly diffusive. The coefficient of the diffusion term is bounded below by the CFL condition for the Lax–Friedrichs method, which states that  $h/k$  must be greater than the characteristic speed. In other words, the diffusion coefficient is bounded below by the norm of the Jacobian matrix  $\frac{1}{2}h\|\mathbf{f}'(\mathbf{u})\|$ . We can take the magnitude of the largest eigenvalue as a suitable matrix norm. Because we need just enough viscosity to counter the effect of  $f'(U_{j+1/2})$ , we can lessen the artificial viscosity in the Lax–Friedrichs method by choosing a local diffusion coefficient  $\alpha_{j+1/2}$  to be maximum of  $\|\mathbf{f}'(\mathbf{u})\|$  for  $\mathbf{u}$  in the interval  $(U_j, U_{j+1})$ . Such an approach to defining the numerical flux

$$F_{j+1/2}^n = \frac{f(U_{j+1}^n) + f(U_j^n)}{2} - \alpha_{j+1/2} \frac{U_{j+1}^n - U_j^n}{h}$$

where

$$\alpha_{j+1/2} = \frac{1}{2}h \max_{\mathbf{u} \in (U_j, U_{j+1})} \|\mathbf{f}'(\mathbf{u})\|$$

is called the *local Lax–Friedrichs method*. The Lax–Wendroff method defines the diffusion coefficient of the artificial viscosity term as

$$\alpha_{j+1/2} = \frac{1}{2}k \|\mathbf{f}'(U_{j+1/2})\|^2$$

where the Jacobian matrix is evaluated at  $U_{j+1/2} = \frac{1}{2}(U_{j+1} + U_j)$ .

### ► Finite volume methods

Rather than solving the problem using Taylor series expansion in the form of a finite difference scheme, we can instead use an integral formulation called a *finite volume method*. In such a method we consider breaking the domain into cells centered at  $x_{j+1/2} = \frac{1}{2}(x_{j+1} + x_j)$  with boundaries at  $x_j$  and  $x_{j+1}$ . We then compute the flux through the cell boundaries to determine the change inside the cell. Finite volume methods are especially useful in two and three dimensions when the mesh can be irregularly shaped.

The problem  $u_t + f(u)_x = 0$  where  $u \equiv u(t, x)$  is equivalent to

$$\frac{1}{kh} \int_{t_n}^{t_{n+1}} \int_{x_{j-1/2}}^{x_{j+1/2}} (u_t + f(u)_x) dx dt = 0$$

which in turn can be integrated to be

$$\frac{1}{kh} \int_{x_{j-1/2}}^{x_{j+1/2}} u(t, x) \Big|_{t_n}^{t_{n+1}} dx + \frac{1}{kh} \int_{t_n}^{t_{n+1}} f(u(t, x)) \Big|_{x_{j-1/2}}^{x_{j+1/2}} dt = 0$$

or simply

$$\begin{aligned} & \int_{x_{j-1/2}}^{x_{j+1/2}} u(t_{n+1}, x) dx - \int_{x_{j-1/2}}^{x_{j+1/2}} u(t_n, x) dx \\ & + \int_{t_n}^{t_{n+1}} f(u(t, x_{j+1/2})) dt - \int_{t_n}^{t_{n+1}} f(u(t, x_{j-1/2})) dt = 0. \end{aligned} \quad (14.25)$$

Physically this says that the change in mass of a cell from time  $t_n$  to time  $t_{n+1}$  equals the flux through the boundaries at  $x_{j+1/2}$  and  $x_{j-1/2}$ . Let's define

$$U_j^n = \frac{1}{h} \int_{x_{j-1/2}}^{x_{j+1/2}} u(t_n, x) dx$$

and the flux at  $x_{j+1/2}$  over the time interval  $[t_n, t_{n+1}]$  to be

$$F_{j+1/2}^{n+1/2} = \frac{1}{k} \int_{t_n}^{t_{n+1}} f(u(t, x_{j+1/2})) dt.$$

Think of  $u$  as density and  $U$  as mass, and think of  $f$  as flux density and  $F$  as flux. Then we have

$$\frac{U_j^{n+1} - U_j^n}{k} + \frac{F_{j+1/2}^{n+1/2} - F_{j-1/2}^{n+1/2}}{h} = 0. \quad (14.26)$$

Now, we just need appropriate numerical approximations for these integrals.

For a first-order approximation, we simply take piecewise-constant approximations  $u(t, x)$  in the cells. Then

$$U_j^n = \frac{1}{2} (U_{j-1/2}^n + U_{j+1/2}^n)$$

by the midpoint rule and

$$F_{j+1/2}^{n+1/2} = f(U_{j+1/2}^n).$$

This gives us a staggered version of the Lax–Friedrichs scheme

$$\frac{U_{j+1/2}^{n+1} - \frac{1}{2}(U_j^n + U_{j+1}^n)}{k} + \frac{f(U_{j+1}^n) - f(U_j^n)}{h} = 0.$$

The CFL condition is given by  $k \leq h|\nu_{\max}|$  where  $\nu_{\max}$  is the largest eigenvalue of the Jacobian  $f'(u)$ . Recall that approximating  $U_{j+1/2}^n$  using  $\frac{1}{2}(U_j^n + U_{j+1}^n)$  introduces numerical viscosity into the solution.

To extend the method to second order, we need to use piecewise linear approximations (Nessyahu and Tadmor [1990]). Define

$$P_j(x) = U_j^n + \frac{\partial}{\partial x} U_j^n \cdot (x - x_j)$$

where  $\frac{\partial}{\partial x} U_j^n$  is a slope and  $x \in [x_j, x_{j+1}]$ . We can approximate

$$\begin{aligned} U_{j+1/2}^n &= \frac{1}{h} \int_{x_j}^{x_{j+1/2}} P_j(x) \, dx + \frac{1}{h} \int_{x_j}^{x_{j+1/2}} P_{j+1}(x) \, dx \\ &= \frac{1}{2}(U_j^n + U_{j+1}^n) + \frac{1}{8}h\left(\frac{\partial}{\partial x} U_j^n - \frac{\partial}{\partial x} U_{j+1}^n\right). \end{aligned}$$

From (14.26), we have

$$U_{j+1/2}^{n+1} = \frac{1}{2}(U_j^n + U_{j+1}^n) + \frac{1}{8}h\left(\frac{\partial}{\partial x} U_j^n - \frac{\partial}{\partial x} U_{j+1}^n\right) - \frac{k}{h}(F_{j+1}^{n+1/2} - F_j^{n+1/2}).$$

We still need a second-order approximation for the flux  $F_j^{n+1/2}$ . Using the midpoint rule:

$$F_j^{n+1/2} = \frac{1}{k} \int_{t_n}^{t_{n+1}} f(u(t, x_j)) \, dt \approx f(u(t_{n+1/2}, x_j))$$

with

$$u(t_{n+1/2}, x_j) \approx u(t_n, x_j) + \frac{k}{2} \frac{\partial}{\partial t} u(t_n, x_j) = u(t_n, x_j) - \frac{dt}{2} \frac{\partial}{\partial x} f(u(t_n, x_j))$$

where we use  $u_t + f(u)_x = 0$  for the equality above. This give us the staggered, two-step predictor-corrector method:

$$\begin{aligned} U_j^{n+1/2} &= U_j^n - \frac{k}{2} \frac{\partial}{\partial x} f(U_j^n) \\ U_{j+1/2}^{n+1} &= \frac{1}{2}(U_j^n + U_{j+1}^n) + \frac{h}{8}\left(\frac{\partial}{\partial x} U_j^n - \frac{\partial}{\partial x} U_{j+1}^n\right) - \frac{k}{h}\left(f(U_{j+1}^{n+1/2}) - f(U_j^{n+1/2})\right) \end{aligned}$$

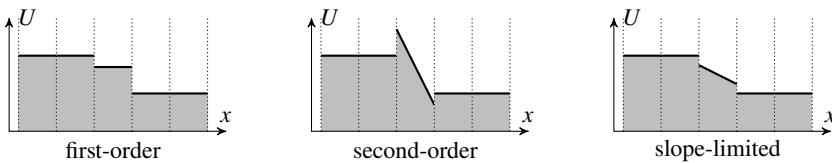


Figure 14.4: A second-order method may introduce oscillations near a discontinuity. A slope limiter is used to minimize oscillations.

Fitting a higher-order polynomial to a discontinuity introduces oscillations into the solution. Because of this, we need to use slope limiters to approximate the slopes  $\frac{\partial}{\partial x} U_j$  and  $\frac{\partial}{\partial x} f(U_j)$ . For  $\frac{\partial}{\partial x} U_j$ , we define the slope  $\sigma_j$  to be

$$\sigma_j = \frac{U_{j+1} - U_j}{h} \phi(\theta_j) \quad \text{with} \quad \theta_j = \frac{U_j - U_{j-1}}{U_{j+1} - U_j}$$

where two common limiting functions are the minmod and the van Leer:

|          |                                                         |
|----------|---------------------------------------------------------|
| minmod   | $\phi(\theta) = \max(0, \min(1, \theta))$               |
| van Leer | $\phi(\theta) = \frac{ \theta  + \theta}{1 +  \theta }$ |

The van Leer slope limiter gives a sharper shock than the minmod slope limiter. A slope limiter works by reducing a second-order method (which is typically dispersive) to a first-order method (which is typically dissipative) wherever there appears to be an oscillation, e.g., whenever  $U_j - U_{j-1}$  and  $U_{j+1} - U_j$  have opposite signs. While the slope limiter removes oscillations and stabilizes the solution, it reduces the method to first-order approximation near an oscillation. Hence, near discontinuities we can get at best first-order and more typically half-order convergence. To get higher orders, one may use essentially non-oscillatory (ENO) interpolation schemes introduced by Chi-Wang Shu and Stanley Osher in 1988.

Notice that  $\sigma_j$  is a symmetric function of  $U_{j-1}$ ,  $U_j$  and  $U_{j+1}$ . That is,

$$\frac{U_{j+1} - U_j}{h} \phi\left(\frac{U_j - U_{j-1}}{U_{j+1} - U_j}\right) = \frac{U_j - U_{j-1}}{h} \phi\left(\frac{U_{j+1} - U_j}{U_j - U_{j-1}}\right).$$

The slope limiter as written above is undefined whenever  $U_{j+1} = U_j$ . Because this happens precisely when we want the slope to be zero, we can fix the problem by adding a conditional expression (a promoted BitArray) as in the following code

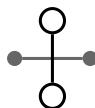
```
ΔU = [0;diff(U)]
s = @. ΔU.*ϕ([ΔU[2:end];0]/(ΔU + (ΔU==0)))
```

Notice that  $\Delta U$  is padded with a zero on the left and then on the right, so that  $s$  has the same number of elements as  $U$ .

• Julia has several packages for modeling and solving hyperbolic problems. Trixi.jl is a numerical simulation framework for hyperbolic conservation laws. Kinetic.jl is a toolbox for the study of computational fluid dynamics and scientific machine learning. Oceananigans.jl is a fluid flow solver for modeling ocean dynamics. ViscousFlow.jl models viscous incompressible flows around bodies of various shapes.

## 14.7 Exercises

- ✍ 14.1. Show that the order of convergence for the Lax–Friedrichs scheme is  $O(k + h^2/k)$  and compute the dispersion relation. Taking the CFL condition and numerical viscosity into consideration, how should we best choose  $h$  and  $k$  to minimize error.
- 14.2. Solve the advection equation  $u_t + u_x = 0$ , with initial conditions  $u(0, x) = 1$  if  $|x - \frac{1}{2}| < \frac{1}{4}$  and  $u(0, x) = 0$  otherwise, using the upwind scheme and the Lax–Wendroff scheme. Use the domain  $x \in [0, 1]$  with periodic boundary conditions. Compare the solutions at  $t = 1$ .
- ✍ 14.3. Consider a finite difference scheme for the linear hyperbolic equation  $u_t + cu_x = 0$  that uses the following stencil (leap-frog in time and central difference in space):



Determine the order of the scheme and the stability conditions for a consistent scheme. Discuss whether the scheme is dispersive or dissipative. Comment on how dispersion or dissipation affects the stability, i.e., is this a good scheme? Show that you can derive this scheme starting with (14.5) and counteracting the unstable term.

14.4. Consider a finite difference scheme for  $u_t + u_x = 0$  that uses a fourth-order central difference approximation for  $u_x$  and the fourth-order Runge–Kutta method in time. Determine the order of the scheme and discuss the stability. Determine whether the scheme is dispersive or dissipative.

14.5. Derive the conservative form of the Lax–Wendroff scheme discussed on page 419.

4 14.6. The compressible Euler equations of gas dynamics are

$$\begin{aligned}\rho_t + (\rho u)_x &= 0 && \text{conservation of mass} \\ (\rho u)_t + (\rho u^2 + p)_x &= 0 && \text{conservation of momentum} \\ E_t + ((E + p)u)_x &= 0 && \text{conservation of energy}\end{aligned}$$

where  $\rho$  is the density,  $u$  is the velocity,  $E$  is the total energy, and  $p$  is the pressure. This system is not a closed system—there are more unknowns than equations. To close the system we add the equation of state for an ideal gases:

$$E = \frac{1}{2}\rho u^2 + \frac{p}{\gamma - 1}$$

where the heat capacity ratio (adiabatic index)  $\gamma$  is constant—for air,  $\gamma$  is around 1.4. Sound speed is given by  $c = \sqrt{\gamma p / \rho}$ . Show that this system forms a hyperbolic system and find the eigenvalues of the Jacobian.

4 14.7. Solve Burgers' equation  $u_t + \frac{1}{2}(u^2)_x = 0$  over the domain  $[-1, 3]$ , with initial conditions  $u(0, x) = 1$  if  $0 < x < 1$  and  $u(0, x) = 0$  otherwise, both analytically and numerically using the local Lax–Friedrichs method. Use the analytic solution to confirm that the numerical solution is correct by plotting together at several time steps.

4 14.8. The “dam break” problem. Consider the shallow water equations

$$h_t + (hu)_x = 0 \tag{14.28a}$$

$$(hu)_t + (hu^2 + \frac{1}{2}gh^2)_x = 0 \tag{14.28b}$$

where  $g$  is the gravitational acceleration,  $h$  is the height of the water, and  $u$  is the velocity of the water. Set  $g = 1$ . Solve the problem with initial conditions  $h = 1$  if  $x < 0$ ,  $h = 0.2$  if  $x > 0$  and  $u = 0$  over the domain  $[-1, 1]$  using the first and second order central schemes with constant boundary conditions. (For the second order, you will need to use a slope limiter.) Use 100 and 200 points for space discretization and the time step according to the CFL condition. Plot the numerical result at  $t = 0.25$  against the “exact” solution obtained using a very fine mesh in space and step size in time.

## Chapter 15

---

# Elliptic Equations



In this chapter, we introduce the finite element method (FEM) and applications of it to elliptic equations. Typical elliptic equations are the Laplace equation and the Poisson equation, which model the steady-state distribution of electrical charge or temperature either with a source term (the Poisson equation) or without (the Laplace equation), and the steady-state Schrödinger equation. Finite element analysis is also frequently applied to modeling strain on structures such as beams. Finite element methods, pioneered in the 1960s, are widely used for engineering problems because the flexible geometry allows one to use irregular elements and the variational formulation makes it easy to deal with boundary conditions and compute error estimates.

### 15.1 A one-dimensional example

Consider a heat conducting bar with uniform heat conductivity, an unknown temperature distribution  $u(x)$ , a known heat source  $f(x)$ , and constant temperature  $u(0) = u(1) = 0$  at the ends of the bar. Fourier's law tells us that the heat flux  $q(x) = -u'(x)$  and the conservation of energy says that  $q'(x) = f(x)$ , from which we have the equation  $-u'' = f$ . The steady-state solution can be found by formulating the problem in three different ways—as a *boundary value problem* (D), as a *minimization problem* (M), and as a *variational problem* (V) (also called the weak formulation).

(D) *Boundary value:* Find  $u$  such that  $-u'' = f$  with  $u(0) = u(1) = 0$ .

Let  $V$  be the vector space of piecewise-differentiable functions over the interval  $[0, 1]$  that vanish on the endpoints. That is,  $v(0) = v(1) = 0$ . Define the *total potential energy*

$$F(v) = \frac{1}{2} \langle v', v' \rangle - \langle f, v \rangle$$

where the inner product  $\langle u, v \rangle = \int_0^1 u(x)v(x) dx$ . In the next section, we'll introduce the notation  $a(\cdot, \cdot)$  and  $L(\cdot)$  in the place of these inner products. For now, let's keep things simple.

(M) *Minimization:* Find  $u \in V$  such that  $F(u) \leq F(v)$  for all  $v \in V$ .

(V) *Variational:* Find  $u \in V$  such that  $\langle u', v' \rangle = \langle f, v \rangle$  for all  $v \in V$ .

**Theorem 49.** *Under suitable conditions the boundary value problem (D), minimization problem (M) and variational problem (V) are all equivalent.*

*Proof.* First, we show (D) is equivalent to (V). We start by showing (D) implies (V). If  $-u'' = f$ , then for all  $v \in V$  we have  $-\langle u'', v \rangle = \langle f, v \rangle$ . After integrating by parts  $-u'v|_0^1 + \langle u', v' \rangle = \langle f, v \rangle$ . Because all elements of  $V$  vanish on the boundary, it follows that  $\langle u', v' \rangle = \langle f, v \rangle$ . To show that (V) implies (D) given that  $u'$  is differentiable, we reverse the steps. For all  $v \in V$  we have  $\langle u', v' \rangle = \langle f, v \rangle$ . Integrating by parts  $u'v'|_0^1 - \langle u'', v \rangle = \langle f, v \rangle$ . So,  $\int_0^1 (u'' + f)v dx = 0$ . Because this expression is true for all  $v \in V$ , it follows that  $-u'' = f$ .

Now, we show (M) is equivalent to (V). First, we'll show that (V) follows from (M). Define the perturbation  $g(\varepsilon) = F(u + \varepsilon w)$  for an arbitrary function  $w$  with  $\varepsilon > 0$ . The *variational derivative* of  $F$  is defined as  $\delta F = dg/d\varepsilon|_{\varepsilon=0}$ . The energy  $F(u)$  is at a minimum  $u$  when its variation derivative equals zero. Let's find when this occurs. The perturbation

$$\begin{aligned} g(\varepsilon) &= \frac{1}{2} \langle u' + \varepsilon w', u' + \varepsilon w' \rangle - \langle f, u + \varepsilon w \rangle \\ &= \frac{1}{2} \langle u', u' \rangle + \varepsilon \langle u', w' \rangle + \frac{1}{2} \varepsilon^2 \langle w', w' \rangle - \langle f, u \rangle - \varepsilon \langle f, w \rangle. \end{aligned}$$

From this, the variational derivative  $\delta F$  is

$$\left. \frac{dg}{d\varepsilon} \right|_{\varepsilon=0} = \langle u', w' \rangle + \varepsilon \langle w', w' \rangle - \langle f, w \rangle \Big|_{\varepsilon=0} = \langle u', w' \rangle - \langle f, w \rangle,$$

which is zero when  $\langle u', w' \rangle = \langle f, w \rangle$  for all  $w \in V$ . Finally, we show that (M) follows from (V). Let  $w = v - u$ . Then  $w \in V$  and

$$F(v) = F(u + w) = \frac{1}{2} \langle u' + w', u' + w' \rangle - \langle f, u + w \rangle.$$

Expanding the right hand side:

$$F(v) = \frac{1}{2} \langle u', u' \rangle + \langle u', w' \rangle + \frac{1}{2} \langle w', w' \rangle - \langle f, u \rangle - \langle f, w \rangle.$$

Because  $\langle u', w' \rangle = \langle f, w \rangle$ , it follows that

$$F(v) = F(u) + \frac{1}{2} \langle w', w' \rangle \geq F(u). \quad \square$$

The basic idea of the finite element method is to find a solution to the variational problem ⑤ or the minimization problem ⑥ in a finite-dimensional subspace  $V_h$  of  $V$ . The finite element solution  $u_h$  is a projection of  $u$  onto  $V_h$ . The minimization problem is called the *Rayleigh–Ritz* formulation and the variational problem is called the *Galerkin* formulation:

⑥ *Rayleigh–Ritz*: Find  $u_h \in V_h$  such that  $F(u_h) \leq F(v_h)$  for all  $v_h \in V_h$ .

⑤ *Galerkin*: Find  $u_h \in V_h$  such that  $\langle u'_h, v'_h \rangle = \langle f, v_h \rangle$  for all  $v_h \in V_h$ .

The function  $v_h$  is called the *test function* and  $u_h$  is called the *trial function*.

Let's apply the Galerkin formulation to the original boundary value problem. Consider  $V_h$  to be the subspace of continuous piecewise-linear polynomials with nodes at  $\{x_i\}$ . We might alternatively consider other higher-order splines as basis elements. We'll restrict ourselves to a uniform partition with nodes at  $x_j = jh$  and with  $x_1 = 0$  and  $x_m = 1$  to simplify the derivations. The finite element solution for a nonuniform partition can similarly be derived. A basis element  $\varphi_j(x) \in V_h$  is

$$\varphi_j(x) = \begin{cases} 1 + t_j(x), & t_j(x) \in (-1, 0] \\ 1 - t_j(x), & t_j(x) \in (0, 1] \\ 0, & \text{otherwise} \end{cases} \quad \begin{array}{c} \text{---} \\ x_{j-1} \bullet x_j \bullet x_{j+1} \\ \text{---} \end{array} \quad (15.1)$$

with  $t_j(x) = (x - x_j)/h$ . Note that  $\{\varphi_j(x)\}$  forms a *partition of unity*, i.e.,  $\sum_{j=0}^m \varphi_j(x) = 1$  for all  $x$ . Then  $v_h \in V_h$  is defined by

$$v_h(x) = \sum_{j=1}^m v_j \varphi_j(x)$$

with  $v_j = v(x_j)$  and  $v \in V$ . The finite element solution is given by

$$u_h(x) = \sum_{i=1}^m \xi_i \varphi_i(x)$$

where  $\xi_i = u(x_i)$  are unknowns. The Galerkin problem to find  $u_h$  such that  $\langle u'_h, v'_h \rangle = \langle f, v_h \rangle$  now becomes the problem to find  $\xi_i$  such that

$$\left\langle \sum_{i=1}^m \xi_i \varphi'_i(x), \sum_{j=1}^m v_j \varphi'_j(x) \right\rangle = \left\langle f, \sum_{j=1}^m v_j \varphi_j(x) \right\rangle.$$

By bilinearity of the inner product, this is the same as

$$\sum_{j=1}^m v_j \left\langle \sum_{i=1}^m \xi_i \varphi'_i(x), \varphi'_j(x) \right\rangle = \sum_{j=1}^m v_j \langle f, \varphi_j(x) \rangle.$$

In order that this equality holds for all  $v_j$  it must follow that

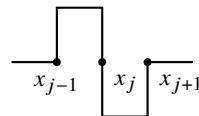
$$\left\langle \sum_{i=1}^m \xi_i \varphi'_i(x), \varphi'_j(x) \right\rangle = \langle f, \varphi_j(x) \rangle$$

for  $j = 1, 2, \dots, m$ . Again, by bilinearity

$$\sum_{i=1}^m \langle \varphi'_i(x), \varphi'_j(x) \rangle \xi_i = \langle f, \varphi_j(x) \rangle.$$

So we have the linear system of equation  $\mathbf{A}\xi = \mathbf{b}$  where  $\mathbf{A}$  is called the *stiffness matrix* with

$$a_{ij} = \langle \varphi'_i, \varphi'_j \rangle = \begin{cases} \int_0^1 (\varphi'_i)^2 dx = \frac{2}{h}, & i = j \\ \int_0^1 \varphi'_i \varphi'_j dx = -\frac{1}{h}, & i = j \pm 1 \\ 0, & \text{otherwise.} \end{cases}$$



The vector  $\mathbf{b} = (f_1, f_2, \dots, f_m)$  where  $f_j = \langle f, \varphi_j \rangle$  is called the *load vector*. We see that by using regular, piecewise-linear basis functions, the system  $\mathbf{A}\xi = \mathbf{b}$  is simply

$$\frac{1}{h^2} \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & \ddots & & \\ & \ddots & \ddots & -1 & \\ & & -1 & 2 & \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_m \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}.$$

## 15.2 A two-dimensional example

Let's extend the framework we developed in the previous section to the two-dimensional Poisson equation

$$-\Delta u = f \quad \text{where} \quad u|_{\partial\Omega} = 0 \tag{15.2}$$

for a region  $\Omega \in \mathbb{R}^2$ . Before writing the Poisson equation in variational form, we'll need a few more concepts. A mapping  $L : V \rightarrow \mathbb{R}$  is a *linear functional* or linear form if

$$L(\alpha u + \beta v) = \alpha L(u) + \beta L(v)$$

for all real numbers  $\alpha$  and  $\beta$  and  $u, v \in V$ . The mapping  $a : V \times V \rightarrow \mathbb{R}$  is a *bilinear form* or bilinear functional if

$$\begin{aligned} a(\alpha u + \beta w, v) &= \alpha a(u, v) + \beta a(w, v), \quad \text{and} \\ a(u, \alpha v + \beta w) &= \alpha a(u, v) + \beta a(u, w) \end{aligned}$$

for all real numbers  $\alpha$  and  $\beta$  and  $u, v, w \in V$ .

The space of  $L^2$ -functions, also called square-integrable functions, is

$$L^2(\Omega) = \left\{ v \mid \iint_{\Omega} |v|^2 d\mathbf{x} < \infty \right\}.$$

In this case, we define the  $L^2$ -inner product as  $\langle v, w \rangle_{L^2} = \iint_{\Omega} vw d\mathbf{x}$  and the  $L^2$ -norm as  $\|v\|_{L^2} = \sqrt{\langle v, v \rangle}$ . A *Sobolev space* is the space of  $L^2$ -functions whose derivatives are also  $L^2$ -functions:

$$H^1(\Omega) = \{v \in L^2(\Omega) \mid \nabla v \in L^2(\Omega)\}.$$

We can further define Sobolev spaces  $H^k(\Omega)$  as the space of functions whose  $k$ th derivatives are also  $L^2$ -functions. We define the Sobolev inner product as

$$\langle v, w \rangle_{H^1} = \iint_{\Omega} vw + \nabla v \cdot \nabla w d\mathbf{x}.$$

and the Sobolev norm as

$$\|v\|_{H^1} = \sqrt{\iint_{\Omega} v^2 + |\nabla v|^2 d\mathbf{x}}.$$

Finally, define  $H_0^1(\Omega)$  to be the set of all Sobolev functions which vanish on the boundary of  $\Omega$ . The Sobolev norms are extension the  $L^2$ -norms and give a measure of the smoothness or regularity of a function.

**Theorem 50** (Green's first identity).

$$\iint_{\Omega} v \Delta u d\mathbf{x} = \int_{\partial\Omega} v \frac{\partial u}{\partial n} d\mathbf{s} - \iint_{\Omega} \nabla v \cdot \nabla u d\mathbf{x}$$

where  $\frac{\partial u}{\partial n} = \nabla u \cdot \hat{\mathbf{n}}$  is a directional derivative with unit outer normal  $\hat{\mathbf{n}}$ .

*Proof.* Start by noting that

$$\nabla \cdot (v \nabla u) = \nabla v \cdot \nabla u + v \Delta u.$$

Integrating both sides over  $\Omega$  gives us

$$\iint_{\Omega} \nabla \cdot (v \nabla u) \, d\mathbf{x} = \iint_{\Omega} \nabla v \cdot \nabla u + v \Delta u \, d\mathbf{x}.$$

From this it follows that

$$\begin{aligned} \iint_{\Omega} v \Delta u \, d\mathbf{x} &= \iint_{\Omega} \nabla \cdot (v \nabla u) \, d\mathbf{x} - \iint_{\Omega} \nabla v \cdot \nabla u \, d\mathbf{x} \\ &= \int_{\partial\Omega} (v \nabla u) \cdot \hat{\mathbf{n}} \, ds - \iint_{\Omega} \nabla v \cdot \nabla u \, d\mathbf{x} \\ &= \int_{\partial\Omega} v \frac{\partial u}{\partial n} \, ds - \iint_{\Omega} \nabla v \cdot \nabla u \, d\mathbf{x}. \end{aligned} \quad \square$$

Now, let's get back to the two-dimensional Poisson equation. Let  $V = H_0^1(\Omega)$ , the space of Sobolev functions over  $\Omega$  that vanish on the boundary  $\partial\Omega$ . Then from (15.2), for all  $v \in V$

$$-\iint_{\Omega} v \Delta u \, dx \, dy = \iint_{\Omega} f v \, dx \, dy. \quad (15.3)$$

Using Green's identity and noting that  $v$  vanishes on  $\partial\Omega$ , we have

$$\iint_{\Omega} \nabla u \cdot \nabla v \, dx \, dy = \iint_{\Omega} f v \, dx \, dy.$$

Define the bilinear form  $a(\cdot, \cdot)$  as

$$a(u, v) = \iint_{\Omega} \nabla u \cdot \nabla v \, dx \, dy$$

and the linear functional  $L(\cdot)$

$$L(v) = \iint_{\Omega} f v \, dx \, dy.$$

Then the variational formulation is

(V) Find  $u \in V$  such that  $a(u, v) = L(v)$  for all  $v \in V$ .

Let's now look at the implementation using FEM. The finite element method starts with triangulation of the domain, subdividing it into triangles in two-dimensional space and simplices such as tetrahedron for higher-dimensional spaces. Consider some triangulation  $T_h$  of  $\Omega$ . That is, let  $T_h = \cup_j K_j$  where  $K_j$  are triangular elements and  $n_j = (x_j, y_j)$  are nodes associated with the vertices of the triangular elements. We'll take the finite element space  $V_h \subset H_0^1(T_h)$  as

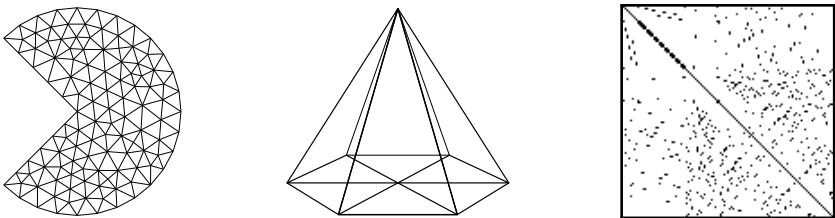


Figure 15.1: Triangulation of a domain  $\Omega$  (left) using a simple pyramidal basis functions (middle) and the resultant sparse stiffness matrix (right).

the space of functions that are linear over each element  $K_j$ . For all  $v_h \in V_h$  we can express

$$v_h(x, y) = \sum_{i=1}^m v(n_i) \varphi_i(x, y)$$

where  $\{\varphi_i\}$  is a set of pyramidal basis functions with  $\varphi_i(n_j) = \delta_{ij}$ . Such basis functions are often called *Lagrange elements*. The finite element formulation is

**(v)** Find  $u_h \in V_h$  such that  $a(u_h, v) = L(v)$  for all  $v \in V_h$ .

Let  $u_h(x, y) = \sum_{i=1}^m \xi_i \varphi_i(x, y)$  where  $\xi_i = u_h(n_i)$ , then for all  $j = 1, 2, \dots, m$

$$a\left(\sum_{i=1}^m \xi_i \varphi_i(x), \varphi_j\right) = L(\varphi_j)$$

from which it follows by linearity that

$$\sum_{i=1}^m \xi_i a(\varphi_i, \varphi_j) = L(\varphi_j),$$

which we can write as  $\mathbf{A}\boldsymbol{\xi} = \mathbf{b}$ , where elements of the stiffness matrix are

$$a_{ij} = a(\varphi_i, \varphi_j) = \iint_{T_h} \nabla \varphi_i \cdot \nabla \varphi_j \, dx \, dy$$

and elements of the load vector are

$$b_j = L(\varphi_j) = \iint_{T_h} \varphi_j(x) f(x) \, dx \, dy.$$

The stiffness matrix  $\mathbf{A}$  is sparse because  $n_i$  can only be a vertex for a small number of triangles.

**Example.** Let's write the following problem with Neumann boundary conditions as an FEM problem:

$$-\Delta u + u = f \quad \text{with} \quad \left. \frac{\partial u}{\partial n} \right|_{\partial\Omega} = g.$$

Let  $V = H^1(\Omega)$  and take  $v \in V$ . This time  $v$  is allowed to be non-zero on the boundary  $\partial\Omega$ . Then

$$\iint_{\Omega} -v\Delta u + vu \, dx \, dy = \iint_{\Omega} fv \, dx \, dy.$$

Using Green's identity, we have

$$\iint_{\Omega} \nabla v \cdot \nabla u \, dx \, dy - \int_{\partial\Omega} v \frac{\partial u}{\partial n} \, ds + \iint_{\Omega} vu \, dx \, dy = \iint_{\Omega} fv \, dx \, dy.$$

On the boundary  $\frac{\partial u}{\partial n} = g$ , so

$$\iint_{\Omega} \nabla v \cdot \nabla u \, dx \, dy + \iint_{\Omega} vu \, dx \, dy = \iint_{\Omega} fv \, dx \, dy + \int_{\partial\Omega} gv \, ds.$$

Let

$$a(u, v) = \iint_{\Omega} \nabla u \cdot \nabla v + uv \, dx \, dy$$

and let

$$L(v) = \langle f, v \rangle + \langle g, v \rangle \equiv \iint_{\Omega} fv \, dx \, dy + \int_{\partial\Omega} gv \, ds.$$

The variational form of the Neumann problem is

(V) Find  $u \in V$  such that  $a(u, v) = L(v)$  for all  $v \in V$ .

Let  $V_h$  be the space of continuous, piecewise-linear functions on  $\Omega$  with a triangulation  $T_h$ , and let  $\{\varphi\}$  be the set of basis functions. Recasting the problem as a finite element approximation

(V) Find  $u_h \in V_h$  such that  $a(u_h, v) = L(v)$  for all  $v \in V_h$ .

By taking the finite elements  $u_h(x, y) = \sum_{i=1}^m \xi_i \varphi_i(x, y)$ , then

$$\sum_{i=1}^m \xi_i a(\varphi_i, \varphi_j) = L(\varphi_j)$$

for  $j = 1, 2, \dots, m$ . In other words, we solve the system  $\mathbf{A}\xi = \mathbf{b}$  where

$$a_{ij} = a(\varphi_i, \varphi_j) = \iint_{T_h} \nabla \varphi_i \cdot \nabla \varphi_j + \varphi_i \varphi_j \, dx \, dy$$

and

$$b_j = \iint_{T_h} f \varphi_j \, dx \, dy + \int_{\partial T_h} g \varphi_j \, ds.$$

Note that this time our stiffness matrix  $\mathbf{A}$  and load vector  $\mathbf{b}$  include the boundary elements.  $\blacktriangleleft$

### 15.3 Stability and convergence

Now that we know how a finite element method works, let's examine when it works and how well it works. To do this, we'll use the Lax–Milgram lemma and Céa's lemma. We start with a few definitions. A bilinear form  $a$  is *symmetric* if  $a(u, v) = a(v, u)$ . A bilinear form  $a$  is *continuous* or *bounded* if there exists a positive  $\gamma$  such that  $|a(u, v)| \leq \gamma \|u\|_V \|v\|_V$  for all  $u, v \in V$ . Similarly, a linear functional  $L$  is continuous if there exists a positive  $\Lambda$  such that  $|L(u)| \leq \Lambda \|u\|_V$  for all  $u \in V$ . A bilinear form  $a$  is *coercive* or *V-elliptic* if there exists a positive  $\alpha$  such that  $|a(u, u)| \geq \alpha \|u\|_V^2$  for all  $u \in V$ .

**Theorem 51** (Lax–Milgram lemma). *If  $a$  is a bounded, coercive bilinear linear form on  $V$ , then for every functional  $L$ , there exists a unique  $u$  such that  $a(u, v) = L(v)$  for all  $v \in V$ . And if  $L$  is bounded, then so is  $\|u\|_V$ .*

*Proof.* Suppose that there are two solutions  $u_1$  and  $u_2$ . So  $a(u_1, v) = L(v)$  and  $a(u_2, v) = L(v)$  for all  $v \in V$ . Then  $|a(u_1 - u_2, v)| = 0$ . By coercivity

$$\alpha \|u_1 - u_2\|_V^2 \leq a(u_1 - u_2, u_1 - u_2) = 0.$$

It follows that  $u_1 = u_2$ . Furthermore, if  $L(u) \leq \Lambda \|u\|_V$ , then it follows that  $\alpha \|u\|_V^2 \leq a(u, u) = L(u) \leq \Lambda \|u\|_V$ . So  $\|u\|_V \leq \Lambda/\alpha$ .  $\square$

**Theorem 52** (Cauchy–Schwarz inequality).  $|\langle v, w \rangle| \leq \|v\| \|w\|$

*Proof.* Note that for any  $\alpha$ ,  $0 \leq \|v - \alpha w\|^2 = \|v\|^2 - 2\alpha \langle v, w \rangle + |\alpha|^2 \|w\|^2$ . By taking  $\alpha = |\langle v, w \rangle| / \|w\|^2$ , we have  $0 \leq \|v\|^2 \|w\|^2 - |\langle v, w \rangle|^2$ .  $\square$

**Theorem 53** (Poincaré inequality). *For  $u \in H_0^1$  and  $\Omega$  bounded, then*

$$\int_{\Omega} u^2 \, dx \leq \beta \int_{\Omega} (u')^2 \, dx \quad \text{for some } \beta.$$

*Proof.* We'll prove an easier one-dimensional case. Without loss of generality, take  $u(0) = 0$  and  $x \in [0, 1]$ . By the Cauchy–Schwarz inequality

$$u(x) = \int_0^x u'(x) \, dx \leq \sqrt{\int_0^x 1 \, dx} \sqrt{\int_0^x (u')^2 \, dx} \leq \sqrt{\int_0^1 (u')^2 \, dx}.$$

Squaring and integrating both sides give us

$$\int_0^1 u^2(x) dx \leq \int_0^1 \left( \int_0^1 (u')^2 dx \right) dx = \int_0^1 (u')^2 dx. \quad \square$$

**Example.** Let's show that the finite element formulation of the Poisson equation  $-u'' = f$  with  $u(0) = 0$  and  $u(1) = 0$  is well-posed (unique and stable). By setting

$$a(u, v) = \int_0^1 u'v' dx \text{ and } L(v) = \int_0^1 fv dx,$$

the finite element formulation for the Poisson equation is

(V) Find  $u \in H_0^1([0, 1])$  such that  $a(u, v) = L(v)$  for all  $v \in H_0^1([0, 1])$

To show well-posedness, we'll just need to check that the conditions of continuity and coercivity of the Lax–Milgram lemma hold. By the Cauchy–Schwarz inequality

$$|a(u, v)| = \left| \int_0^1 u'v' dx \right| \leq \sqrt{\int_0^1 |u'|^2 dx} \sqrt{\int_0^1 |v'|^2 dx} \leq \|u\|_{H_0^1} \|v\|_{H_0^1}$$

where the Sobolev norm  $\|u\|_{H_0^1} = \sqrt{\int_0^1 u^2 + |u'|^2 dx}$ . So  $a$  is continuous. Similarly,

$$|L(v)| = \left| \int_0^1 fv dx \right| \leq \|f\|_{L^2} \|v\|_{L^2} \leq \|f\|_{L^2} \|v\|_{H_0^1}.$$

It follows that  $L$  is continuous. Finally, from the Poincaré inequality, it follows that  $a(u, u) = \int_0^1 (u')^2 dx \geq \alpha \int_0^1 u^2 + (u')^2 dx$ . So  $a$  is coercive.  $\blacktriangleleft$

**Theorem 54** (Céa's lemma). *If  $a$  is continuous and coercive with respective bounding constants  $\gamma$  and  $\alpha$ , then  $\|u - u_h\| \leq (\gamma/\alpha) \|u - v\|$  for all  $v \in V_h$ .*

*Proof.* Let  $V$  be a Hilbert space and consider the problem of finding the element  $u \in V$  such that  $a(u, v) = L(v)$  for all  $v \in V$ . And consider similar problem of finding the element  $u_h \in V_h$  such that  $a(u_h, v) = L(v)$  for all  $v \in V_h$  for the finite-dimensional subspace  $V_h$  of  $V$ . Then,  $a(u - u_h, v) = 0$  for all  $v \in V$ . This says, that  $u_h$  is a projection of  $u$  in the inner-product  $a(\cdot, \cdot)$ . Then, by using the Lax–Milgram lemma,

$$\begin{aligned} \alpha \|u - u_h\|^2 &\leq a(u - u_h, u - u_h) = a(u - u_h, u - v) + a(u - u_h, v - u_h) \\ &= a(u - u_h, u - v) \leq \gamma \|u - u_h\| \|u - v\| \end{aligned}$$

So,  $\|u - u_h\| \leq (\gamma/\alpha) \|u - v\|$  for all  $v \in V_h$ .  $\square$

Céa's lemma says that the finite element solution  $u_h$  is the best solution up to the constant  $\gamma/\alpha$ . We can use Céa's lemma to estimate the approximation error. Suppose that  $u \in H_0^2$  is a solution to the Galerkin problem. Let  $p_h$  be a piecewise-linear polynomial interpolant of the solution  $u \in H_0^2$ . Then by Céa's lemma

$$\|u - u_h\|_{H_0^1} \leq \frac{\gamma}{\alpha} \|u - p_h\|_{H_0^1}.$$

By Taylor's theorem, there is a constant  $C$  such that  $|u' - p'_h| \leq Ch\|u''\|_{L^2}$ . So,

$$\|u - u_h\|_{H_0^1} \leq \frac{\gamma Ch}{\alpha} \|u\|_{L^2} \leq \frac{\gamma Ch}{\alpha} \|u\|_{H_0^2}.$$

For degree- $k$  piecewise-polynomial basis functions, the  $H_0^1$ -error is  $O(h^k)$  if  $u \in H_0^{k+1}$ .

## 15.4 Time-dependent problems

The finite element method can also be applied to time-dependent problems. Consider the one-dimensional heat equation with a source term  $u_t - u_{xx} = f(x)$ . Let the initial condition  $u(x, 0) = u_0(x)$  and boundary values  $u(0, t) = u(1, t) = 0$ . Then for all  $v \in H_0^1([0, 1])$ ,

$$\langle u_t, v \rangle - \langle u_{xx}, v \rangle = \langle f, v \rangle$$

where  $\langle u, v \rangle = \int_0^1 uv \, dx$ . From this we have the Galerkin formulation:

**(V)** Find  $u_h \in V_h$  such that  $\langle u_t, v \rangle + \langle u_{xx}, v_x \rangle = \langle f, v \rangle$  for all  $v \in V_h$ .

Let  $\{\varphi_0, \varphi_1, \dots, \varphi_m\}$  be a basis of  $V_h \subset V$  and let  $u_h(x, t) = \sum_{i=1}^m \xi_i(t) \varphi_i(x)$  and take  $v = \varphi_j$ . Then we have the system

$$\sum_{i=1}^m \xi'_i(t) \langle \varphi_i, \varphi_j \rangle + \sum_{i=1}^m \xi_i(t) \langle \varphi'_i, \varphi'_j \rangle = \langle f, \varphi_j \rangle$$

for  $j = 1, 2, \dots, m$ . More concisely, we have the system of differential equations  $\mathbf{A}\xi'(t) + \mathbf{B}\xi(t) = \mathbf{c}$  where the elements of  $\mathbf{A}$ ,  $\mathbf{B}$ , and  $\mathbf{c}$  are  $a_{ij} = \langle \varphi_i, \varphi_j \rangle$ ,  $b_{ij} = \langle \varphi'_i, \varphi'_j \rangle$ , and  $c = \langle f, \varphi_j \rangle$ . We can rewrite the system of equations as  $\xi'(t) + \mathbf{A}^{-1}\mathbf{B}\xi(t) = \mathbf{A}^{-1}\mathbf{c}$ , which can then be solved using a method such as the backward Euler method

$$\frac{\xi^{n+1} - \xi^n}{\Delta t} + \mathbf{A}^{-1}\mathbf{B}\xi^{n+1} = \mathbf{A}^{-1}\mathbf{c}$$

or the Crank–Nicolson method

$$\frac{\xi^{n+1} - \xi^n}{\Delta t} + \frac{1}{2}\mathbf{A}^{-1}\mathbf{B}\xi^{n+1} + \frac{1}{2}\mathbf{A}^{-1}\mathbf{B}\xi^n = \mathbf{A}^{-1}\mathbf{c}.$$

Numerical stability can be determined by confirming that the energy decreases over time. Consider the heat equation  $u_t = u_{xx}$  with boundary conditions  $u(0, t) = u(1, t) = 0$ . By multiplying by  $u$  and integrating over  $[0, 1]$  we have  $\frac{1}{2} \frac{d}{dt} \|u\|^2 = -\|u_x\|^2 \leq 0$ . So,  $\|u(\cdot, t)\|_{L^2} \leq \|u(\cdot, 0)\|_{L^2}$ .

The backward Euler scheme for the heat equation is

$$\frac{1}{\Delta t} \langle u_h^{n+1} - u_h^n, v \rangle + a(u_h^{n+1}, v) = 0$$

for all  $v \in V_h$  where  $a(u, v) = \int_0^1 u'v' dx$ . Take  $v = u_h^{n+1}$ . Then

$$\langle u_h^{n+1}, u_h^{n+1} \rangle - \langle u_h^n, u_h^{n+1} \rangle = -\Delta t a(u_h^{n+1}, u_h^{n+1}) \leq 0.$$

So,  $\|u_h^{n+1}\|_{L^2}^2 \leq \langle u_h^n, u_h^{n+1} \rangle \leq \|u_h^n\|_{L^2} \|u_h^{n+1}\|_{L^2}$  by the Cauchy–Schwarz inequality. Therefore,  $\|u_h^{n+1}\|_{L^2} \leq \|u_h^n\|_{L^2}$  for all  $n$  and all stepsizes  $\Delta t$ . Hence, the scheme is unconditionally stable.

## 15.5 Practical implementation

In practice, one will typically use a computational software package to manage the tedious, low-level steps of the finite element method. Once we have formulated the weak form of the problem, we can implement a solution by breaking implementation into the following steps:

1. Preprocessing. Define the geometry and generate the mesh.
2. Assembling. Construct the stiffness matrices, boundary conditions, and shape functions.
3. Solving. Use either a direct or iterative sparse linear solver.
4. Postprocessing. Visualize the solution, analyze the results, and estimate the error.

In this section, we'll use FEniCS.jl, a wrapper to the popular, open-source FEniCS<sup>1</sup> platform. FEniCS is written in C++ and designed to run on hardware ranging from laptops to high-performance clusters, with wrappers in Python, Julia, and Matlab. The FEniCS.jl library lacks some of the functionality of the FEniCS Python library. Furthermore, because FEniCS is an external program, you'll need to download and install it in addition to the wrapper.<sup>2</sup> If you are looking for a finite-element library without any external dependencies, try the relatively new Gridap.jl library.

---

<sup>1</sup>The “FE” represents “finite element,” the “CS” represents “computational software,” and “ni” is simply used to string the two together into a homophone of “phoenix,” the mascot of University of Chicago, where the project originated.

<sup>2</sup><https://fenicsproject.org/>

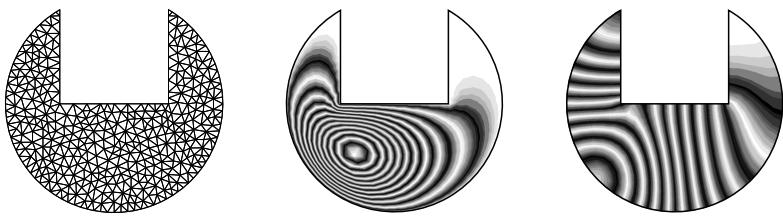


Figure 15.2: FEM mesh (left), solution of the Poisson equation with a heat source centered at  $(-0.5, -0.5)$  (middle), and solution of the heat equation with initial conditions centered at  $(-0.5, -0.5)$  (right). Each contour depicts a 10 percent change in value.

• Julia's FEM packages include Ferrite.jl, Gridap.jl, and FEniCS.jl.

**Example.** Let's solve the Poisson equation  $-\Delta u = f$  over the domain constructed by subtracting the unit square centered at  $(0, 0.5)$  from the unit circle centered at  $(0, 0)$ . Take the source term  $f(x, y) = e^{-(2x+1)^2-(2y+1)^2}$ , which models a heat source near  $(-0.5, -0.5)$  with Dirichlet boundary conditions  $u(x, y) = 0$ . The variational form is  $a(u_h, v) = L(v)$  where

$$a(u, v) = \iint_{\Omega} \nabla u \cdot \nabla v \, dx \, dy \quad \text{and} \quad L(v) = \iint_{\Omega} v f \, dx \, dy.$$

We'll first define the geometry, the finite element space consisting of Lagrange polynomials, and the trial and test functions. Given a function space  $V$ , FEniCS uses `TestFunction(V)` for a test function, `TrialFunction(V)` to define the trial function, and `FeFunction(V)` to define the solution function where the result will be stored.

```
using FEniCS
square = Rectangle(Point([-0.5, 0]), Point([0.5, 1]))
circle = Circle(Point([0.0, 0.0]), 1)
mesh = generate_mesh(circle - square, 16)
V = FunctionSpace(mesh, "Lagrange", 1)
u, v, w = TrialFunction(V), TestFunction(V), FeFunction(V)
```

The mesh is shown in the figure above. Now, define the boundary conditions and the source term. Note that FEniCS requires functions to be entered as C++ expressions.

```
bc = DirichletBC(V, Constant(0), "on_boundary")
f = Expression("exp(-pow(2*x[0]+1,2)-pow(2*x[1]+1,2))", degree=2)
```

Finally, define  $a(u_h, v)$  and  $L(v)$  and solve the problem.

```
a = dot(grad(u), grad(v))*dx
L = f*v*dx
lvsolve(a,L,w,bc)
```

The solution  $w$  can be exported as a visualization toolkit file (VTK) to be visualized using ParaView:<sup>3</sup>

```
File("poisson_solution.pvd",w)
```

The solution is shown in the figure on the preceding page. Notice that the peak temperature is to the right of the heat source peak. ◀

**Example.** Let's modify the example above to solve the heat equation  $u_t = \Delta u$  over the same domain with Neumann boundary conditions. We'll take the initial conditions  $u(0, x, y) = f(x, y) = e^{-(4x+2)^2-(4y+2)^2}$ . The variational form of the heat equation is  $\langle u_t, v \rangle + \langle \nabla u, \nabla v \rangle = 0$ . We define the geometry, the finite element space, and the test functions as before, adding an expression for the initial condition.

```
using FEniCS, OrdinaryDiffEq
square = Rectangle(Point([-0.5, 0]), Point([0.5, 1]))
circle = Circle(Point([0.0, 0.0]), 1)
mesh = generate_mesh(circle - square, 16)
V = FunctionSpace(mesh, "Lagrange", 1)
f = Expression("exp(-pow(2*x[0]+1,2)-pow(2*x[1]+1,2))", degree=2)
```

We define variables for the initial conditions, trial function, test function, and solution function.

```
u, v = interpolate(f,V), TestFunction(V)
u_t, w_t = TrialFunction(V), FeFunction(V)
```

We define the variation form of the heat equation and pass it along with the variables to an in-place function using a named tuple  $p$ . The function `get_array` maps a variable in function space to a one-dimensional array and the function `assign` maps the one-dimensional array to a variable in function space. Neumann boundary conditions are the default boundary conditions in FEniCS, and we prescribe it by setting `bc = []`.

---

<sup>3</sup>ParaView is an open-source application for visualizing scientific data sets such as VTK.

```

F = dot(u_t, v)*dx + dot(grad(u), grad(v))*dx
p = (u=u, w_t=w_t, F=F)
function heat_equation!(u_t_vec, u_vec, p, t)
 assign(p.u, u_vec)
 lvsolve(lhs(p.F), rhs(p.F), p.w_t, [])
 copy!(u_t_vec, get_array(p.w_t))
end

```

Finally, let's solve the problem for  $t \in [0, 0.5]$  using a fourth-order semi-implicit ODE solver and export the solution.

```

problem = ODEProblem(heat_equation!, get_array(u), (0,0.0), p)
method = KenCarp4(autodiff=false)
solution = OrdinaryDiffEq.solve(problem, method)
assign(p.u, solution(0.5))
File("heat_solution.pvd", p.u)

```

The solution is shown in the figure on page 437. 

## 15.6 Exercises

 15.1. Solve the differential equation  $u'' + u - 8x^2 = 0$  with Neumann boundary conditions  $u'(0) = 0$  and  $u'(1) = 1$  using the finite element method with piecewise-linear elements.

 15.2. Consider the boundary value problem

$$u'''' = f \quad \text{with} \quad u(0) = u'(0) = u(1) = u'(1) = 0$$

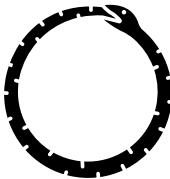
for the deflection  $u(x)$  of a uniform beam under the load  $f(x)$ . Formulate a finite element method and find the corresponding linear system of equations in the case of a uniform partition. Determine the solution for  $f(x) = 384$ . You may wish to use the basis elements  $\phi_j(t) = H_{00}(|t|)$  and  $\psi_j(t) = H_{10}(|t|)$  for  $t \in [-1, 1]$  and equal zero otherwise. The Hermite polynomials are  $H_{00}(t) = 2t^3 - 3t^2 + 1$  and  $H_{10}(t) = t^3 - 2t^2 + t$ .



## Chapter 16

---

# Fourier Spectral Methods



We improved the accuracy of finite difference methods by using wider stencils to better approximate the derivatives. With each additional grid point added to the stencil, we were able to increase the order of accuracy by one. One might reason that by using all available grid points to approximate the derivatives we could build a highly accurate method. This is precisely what spectral methods do. If the solution is smooth, the error shrinks faster than any power of grid spacing. This is sometimes called *spectral accuracy*. If only the first  $p$  derivatives of the function exist, convergence is at most  $O(h^{p+1})$ . In this chapter, we examine an important class of spectral methods—the Fourier spectral method—which uses trigonometric interpolation over a periodic domain to approximate a solution.

Similar spectral methods, such as the Chebyshev spectral method discussed in Chapter 10, can be used on problems with non-periodic solutions. There are also tricks that we can sometimes use to be able to use Fourier spectral methods if the problem isn't periodic. For example, we can extend the domain so that the solution does not have appreciable boundary interaction. Or we might add absorbing “sponge” layers or reflecting barriers to the problem to prevent the solution from wrapping around the boundaries.

### 16.1 Discrete Fourier transform

The Fourier transform has already been discussed in sections 6.1 in reference to the fast Fourier transform and 10.3 in reference to approximating functions. In this section, we'll re-examine the Fourier transform by focusing on trigono-

metric interpolation and spectral differentiation. Specifically, we'll compare the continuous Fourier transform, discrete Fourier transform, and Fourier series.

### ► Trigonometric interpolation

Let the function  $u(x)$  be periodic with period  $2\pi$ , i.e.,  $u(x + 2\pi) = u(x)$ . We can later modify the period of  $u(x)$  by rescaling  $x$ . We can approximate  $u(x)$  by a Fourier polynomial  $p(x) = \sum_{\xi=-m}^{m-1} c_\xi e^{-i\xi x}$  by choosing coefficients  $c_\xi$  such that  $u(x_j) = p(x_j)$  at equally spaced nodes  $x_j = j\pi/m$  for  $j = 0, 1, \dots, 2m - 1$ . To find the coefficients  $c_\xi$ , we'll need the help of an identity.

**Theorem 55.**  $\sum_{j=0}^{2m-1} e^{i\xi x_j}$  equals zero for nonzero integers  $\xi$  and  $2m$  for  $\xi = 0$ .

*Proof.*

$$\sum_{j=0}^{2m-1} e^{i\xi x_j} = \sum_{j=0}^{2m-1} e^{i\xi j\pi/m} = \sum_{j=0}^{2m-1} \omega^{\xi j} = \begin{cases} \frac{\omega^{2m} - 1}{\omega - 1}, & \text{if } \xi \neq 0 \\ 2m, & \text{if } \xi = 0. \end{cases}$$

where  $\omega = e^{i\pi/m}$ . Note that  $\omega^{2m} = (e^{i\xi\pi/m})^{2m} = e^{i\xi 2\pi} = 1$ .  $\square$

It follows that

$$\sum_{j=0}^{2m-1} u(x_j) e^{-i\xi x_j} = \sum_{j=0}^{2m-1} \sum_{l=-m}^{m-1} c_\xi e^{i(l-\xi)x_j} = \sum_{l=-m}^{m-1} c_\xi \sum_{j=0}^{2m-1} e^{i(l-\xi)x_j} = 2mc_\xi.$$

From this we have the discrete Fourier transform and the discrete inverse Fourier transform

$$c_\xi = \frac{1}{2m} \sum_{j=0}^{2m-1} u(x_j) e^{-i\xi x_j} \quad \text{and} \quad u(x_j) = \sum_{\xi=-m}^{m-1} c_\xi e^{i\xi x_j} \quad (16.1)$$

where  $\xi = -m, -m+1, \dots, m-1$  and  $j = 0, 1, \dots, 2m-1$ . Compare these expressions with the (continuous) Fourier transform and the (continuous) inverse Fourier transform

$$\hat{u}(\xi) = F[u] = \frac{1}{2\pi} \int_{-\infty}^{\infty} u(x) e^{-i\xi x} dx \quad \text{and} \quad u(x) = \int_{-\infty}^{\infty} \hat{u}(\xi) e^{i\xi x} d\xi.$$

We can think of the discrete Fourier transform as a Riemann sum and the continuous Fourier transform as a Riemann integral.

The  $2\pi$ -periodization of a function  $u(x)$  is the function

$$u_p(x) = \sum_{m=-\infty}^{\infty} u(x + 2\pi m)$$

if the series converges. Note that if the function  $u_p(x)$  exists, then it is periodic  $u_p(x) = u_p(x + 2\pi m)$ . Also, if  $u(x)$  vanishes outside of  $[0, 2\pi]$ , then  $u_p(x) = u(x)$  for  $x \in [0, 2\pi]$ . Because  $u_p(x)$  is periodic, we can write its Fourier series

$$u_p(x) = \sum_{\xi=-\infty}^{\infty} c_{\xi} e^{i\xi x} \quad (16.2)$$

with

$$c_{\xi} = \frac{1}{2\pi} \int_0^{2\pi} u_p(x) e^{-i\xi x} dx = \frac{1}{2\pi} \int_{-\infty}^{\infty} u(x) e^{-i\xi x} dx$$

which is simply the Fourier transform  $\hat{u}(\xi)$ . This says that

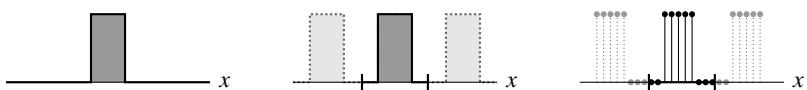
$$\sum_{m=-\infty}^{\infty} u(x + 2\pi m) = \sum_{m=-\infty}^{\infty} \hat{u}(m) e^{imx},$$

which is known as the *Poisson summation formula*. If  $u(x)$  vanishes outside of  $[0, 2\pi]$ , then

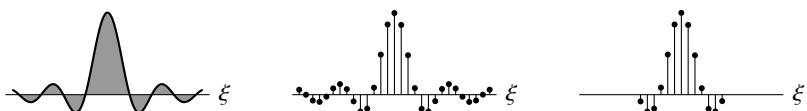
$$u(x) = \sum_{m=-\infty}^{\infty} \hat{u}(m) e^{imx}.$$

Furthermore, by truncating the Fourier series (16.2) by restricting the sum to run from  $-n$  to  $n - 1$ , i.e., restricting the frequency domain, we have the discrete Fourier transform.

Let's summarize the Fourier transform, the Fourier series, and the discrete Fourier transform by comparing them in the spatial domain  $x$  and the frequency domain  $\xi$ .



In the Fourier transform the spatial domain  $x$  extends over the entire real line  $(-\infty, \infty)$ . In the Fourier series, the spatial domain is periodic. In the discrete Fourier transform, the spatial domain is both periodic and discrete.



In the Fourier transform, the frequency domain  $\xi$  also extends over the entire real line  $(-\infty, \infty)$ . In the Fourier series, the frequency domain also extends over the real line  $(-\infty, \infty)$  but is discrete. In the discrete Fourier transform, the frequency domain is both discrete and bounded.

## ► Spectral differentiation

Let's find the derivative of the Fourier polynomial approximation of  $u(x)$ . As before, we take  $u(x)$  with period  $2\pi$ . In this case,

$$\frac{d}{dx} u(x) = \sum_{\xi=-m}^{m-1} c_\xi \frac{d}{dx} e^{i\xi x} = \sum_{\xi=-m}^{m-1} (i\xi c_\xi) e^{i\xi x}.$$

That is to say, the discrete Fourier transform

$$F \left[ \frac{d}{dx} u \right] = i\xi F[u].$$

From this equation, we can see the origin of the term “spectral method.” When a solution  $u(t, x)$  is expanded into a series of orthogonal eigenfunctions of a linear operator, that solution is related to the spectrum (the set of eigenvalues) of that operator. A *pseudospectral* method solves the problem on a discrete mesh.

Similarly, to compute the Fourier transform of an antiderivative we divide by the factor  $i\xi$ . When  $u(x)$  has period  $\ell$ , we make the transform  $x \mapsto (2\pi/\ell)x$  and  $dx \mapsto (2\pi/\ell) dx$ , and the discrete Fourier transform of  $\frac{d}{dx} u$  is  $i(2\pi/\ell)\xi \hat{u}$ . An analogous statement for the Fourier transform holds by integration by parts. Of course, as long as  $u(x)$  is smooth, we can differentiate any number of times. So, the Fourier transform of the  $p$ th derivative of  $u(x)$  is

$$F \left[ \frac{d^p}{dx^p} u \right] = \left( i\xi \frac{2\pi}{\ell} \right)^p F[u]. \quad (16.3)$$

Consider the heat equation  $u_t = u_{xx}$  with initial conditions  $u(0, x)$  and periodic boundary conditions over a domain of length  $2\pi$ . The Fourier transform  $\hat{u}(t, \xi) = F u(t, x)$  of the heat equation is  $\hat{u}_t = -\xi^2 \hat{u}$ , which has the solution

$$\hat{u}(t, \xi) = e^{-\xi^2 t} \hat{u}(0, \xi).$$

It follows that the formal solution to the heat equation is

$$u(t, x) = F^{-1} [e^{-\xi^2 t} F u(0, x)]. \quad (16.4)$$

In practice, we'll use a fast Fourier transform (FFT) to compute the discrete transforms. Let's examine the numerical implementation of the discrete solution in Julia. Some care is required in ordering the indices of  $\xi$ . Most scientific computing languages including Julia, Matlab, and Python define the coefficients of the discrete Fourier transform  $c_\xi$  by starting with the zero frequency term  $c_0$ . For  $n$  nodes, we'll take  $\xi = 0, 1, 2, \dots, \frac{1}{2}m - 1, -\frac{1}{2}m, -\frac{1}{2}m + 1, \dots, -2, -1$ . This means that for a domain of length  $\ell$ , the derivative operator  $i\xi \cdot (2\pi/\ell)$  is

```
iξ = im*[0:(m-1)÷2;-m÷2:-1]*(2π/ℓ)
```

Alternatively, we can use the FFTW.jl function `ifftshift`

```
iξ = im*ifftshift(-m÷2:(m-1)÷2)*(2π/ℓ)
```

or the AbstractFFTs.jl function `fftfreq`

```
iξ = im*fftfreq(m,m)*(2π/ℓ)
```

to compute the zero-frequency shift.

We'll use the FFTW.jl package to implement the discrete Fourier transform. The Fastest Fourier Transform in the West (FFTW) library uses heuristics and trial to select from among several FFT algorithms. The first time that the library is called it may try several different algorithms for the same problem and thereafter use the fastest. The Julia implementation of (16.4) using 256 points over a domain of length 4 is

```
using FFTW
m = 256; ℓ = 4
ξ² = (fftshift(-(m-1)÷2:m÷2)*(2π/ℓ)).^2
u = (t,u₀) -> real(ifft(exp.(-ξ²*t).*fft(u₀)))
```

If a problem uses Fourier transforms repeatedly on arrays with the same size as an array  $u$ , as is the case in time-marching schemes, then it can be advantageous to have FFTW try out all of the possible algorithms initially to see which is fastest. In this case, we can define  $F = \text{plan\_fft}(u)$  and then use  $F*u$  or  $F\backslash u$  where the multiplication and inverse operators are overloaded. For example, we replace the last line in the code above with

```
F = plan_fft(u₀)
u = (t,u₀) -> real(F\exp.(-ξ²*t).*F*u₀))
```

The `fft` and `ifft` functions in Julia are all multi-dimensional transforms unless an additional parameter is provided. For example, `fft(x)` performs an FFT overall dimensions of  $x$ , whereas `fft(x, 2)` only performs an FFT along the second dimension of  $x$ .

## ► Smoothness and spectral accuracy

Smooth functions have rapidly decaying Fourier transforms. The Fourier transform of the smoothest functions like the constant function or sine function are Dirac delta distributions. And Gaussian distributions are transformed to Gaussian distributions. On the other hand, functions with discontinuities have relatively slowly decaying Fourier transforms.

Let's use splines to examine smoothness, starting with a piecewise constant function. The rectangular function

$$B_0(x) = \begin{cases} 1, & x \in [-\frac{1}{2}h, \frac{1}{2}h] \\ 0, & \text{otherwise} \end{cases}$$

is a zeroth-order B-spline with the Fourier transform  $\hat{B}_0(\xi) = F[B_0(x)]$ :

$$\int_{-\infty}^{\infty} B_0(x) e^{-i\xi x} dx = \int_{-\frac{1}{2}h}^{\frac{1}{2}h} e^{-i\xi x} dx = \frac{e^{i\xi/2h} - e^{-i\xi/2h}}{i\xi} = \frac{\sin \xi/2h}{\xi/2} = \frac{1}{h} \operatorname{sinc} \frac{\xi}{2h}.$$

Here are the rectangular function and its Fourier transform, the sinc function<sup>1</sup>:



The rectangular function is not differentiable at  $x = \pm \frac{1}{2}h$ . In fact, it is not even continuous at those points. Its Fourier transform decays as  $O(|\xi|^{-1})$  as  $|\xi| \rightarrow \infty$ . We can use a linear combination of scaled and translated basis functions  $B_0(x)$  to build piecewise constant functions, the Fourier transform of which are a linear combination of sinc functions. The Fourier transform of the rectangular function that equals one over  $x \in [-\frac{1}{2}h + a, \frac{1}{2}h + a]$  and zero otherwise is

$$\int_{-\frac{1}{2}h}^{\frac{1}{2}h} e^{-i\xi(x+a)} dx = e^{ia\xi} \frac{\sin \xi/2h}{\xi/2} = \frac{1}{h} e^{ia\xi} \operatorname{sinc} \frac{\xi}{2h}.$$

We can use the  $B_0$ -spline basis to generate higher-order B-spline bases, starting with a piecewise linear basis  $B_1(x)$ . One way to do this is to take the self-convolution of  $B_0(x)$ . Another way is to use de Boor's algorithm, discussed on page 236. A third way is to take the antiderivative

$$B_1(x) = \int_{-\infty}^x f(x-h/2) - f(x+h/2) dx = \begin{cases} 1+x/h, & x \in [-h, 0] \\ 1-x/h, & x \in [0, h] \\ 0, & \text{otherwise} \end{cases}$$

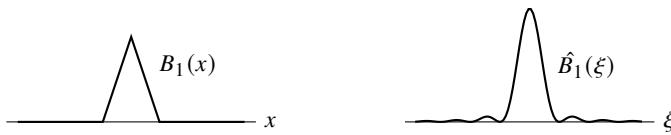
The Fourier transform of the triangular function  $B_1(x)$  is

$$\hat{B}_1(\xi) = (i\xi)^{-1} \left( e^{i\xi/2h} - e^{-i\xi/2h} \right) \frac{1}{h} \operatorname{sinc} \frac{\xi}{2h} = \frac{1}{h^2} \operatorname{sinc}^2 \frac{\xi}{2}.$$

And here are the triangular function and the sinc-squared function:

---

<sup>1</sup>The sinc function (pronounced “sink”) was introduced by Phillip Woodward in his 1952 paper “Information theory and inverse probability in telecommunication” saying that the function  $(\sin x)/x$  “occurs so often in Fourier analysis and its applications that it does seem to merit some notation of its own.” It’s sometimes called “cardinal sine,” not to be confused with “cardinal sin.”



The triangular function is not differentiable at  $x = -1, 0, 1$  but it is weakly differentiable everywhere. The sinc-squared function decays as  $O(|\xi|^{-2})$  as  $|\xi| \rightarrow \infty$ . We can use a linear combination of scaled and translated basis functions  $B_1(x)$  to build piecewise linear functions, whose Fourier transforms happen to be linear combination of sinc-squared functions. In general, a degree- $p$  piecewise polynomial can be constructed using B-splines of order  $p$ . The Fourier transform of a B-spline with  $h$ -separated knots is given by  $\hat{B}_p(\xi) = h^{-(p+1)} \operatorname{sinc}^{p+1}(\xi/2h)$ . In practice, we can't take infinitely many frequencies and must be satisfied with a Fourier polynomial  $P_n f$  as our approximation to the Fourier series  $f$ :

$$f(x) = \sum_{\xi=-\infty}^{\infty} c_{\xi} e^{-i\xi x} = \sum_{\xi=-m}^{m-1} c_{\xi} e^{-i\xi x} + \tau_m(x) = P_m f(x) + \tau_m(x).$$

Because the Fourier series is a complete, orthonormal system, we can use Parseval's theorem on page 249 to compute the  $l^2$ -norm of the truncation error

$$\|\tau_m\|^2 = \|f - P_m f\|^2 = \|P_{\infty} f - P_m f\|^2 = \sum_{\xi>|m|} |c_{\xi}|^2.$$

Let  $f$  be approximated using a spline with knots separated by distance  $h$ . Then the Fourier transform of this approximation is a linear combination of terms  $h^{-(p+1)} \operatorname{sinc}^{p+1}(\xi/2h)$ . It follows that if  $f$  has  $p-1$  continuous derivatives in  $L^2$  and a  $p$ th derivative of bounded variation, then

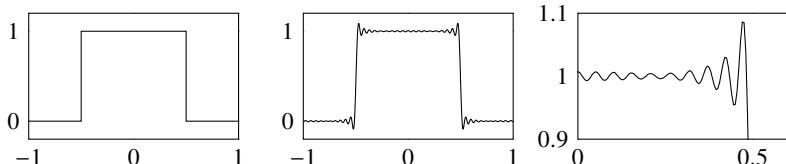
$$|c_{\xi}| = O(h^{-(p+1)} \operatorname{sinc}^{p+1}(\xi/2h)) = O(|\xi|^{-(p+1)})$$

and

$$\|\tau_n\|^2 = \sum_{\xi>|m|} |c_{\xi}|^2 = \sum_{\xi>|m|} O(|\xi|^{-2p-2}) = O(m^{-2p-1}).$$

Therefore, the truncation error  $\|\tau_m\| = O(m^{-p-1/2})$ .

For a rectangular function,  $p=0$  and  $|\tau_n| = O(n^{-1/2})$ . We should expect slow convergence, as we see in the following figure. The original rectangular function  $f$  is on the left, the frequency-limited projection  $P_n f$  is in the middle, and a close-up of  $P_n f$  near a discontinuity is on the right.



The trigonometric polynomial approximation results in a *Gibbs phenomenon*, where the band-limited discrete Fourier transform results in oscillations in the neighborhood of a discontinuity.

## 16.2 Nonlinear stiff equations

As we have seen in previous chapters, some partial differential equations such as the Navier–Stokes equation combine a stiff linear term with a non-stiff nonlinear term. Three techniques to solving such equations include time splitting, integrating factors, and implicit-explicit (IMEX) methods. Time splitting allows us to completely avoid the stability issues caused by the stiff linear term, but we are typically unable to get beyond second-order-in-time accuracy using them. Integrating factors can get us much better accuracy, but we may not fully avoid the stability issues caused by the stiffness. And with IMEX methods, we can get highly accurate solutions with some reduced stability.

### ► Time splitting

Time splitting methods are familiar from chapters 12 and 13. Consider  $u_t = L u + N u$ , where  $L$  is a linear operator and  $N$  is a nonlinear operator. Take the splitting

1.  $v_t = L v$  where  $v(0, x) = u(0, x)$ ,
2.  $w_t = N w$  where  $w(0, x) = v(\Delta t, x)$ .

Recall from the discussion on page 361 that after one time step  $u(\Delta t, x) = w(\Delta t, x) + O(\Delta t^2)$ , which means that we have a second-order splitting error with each time step and a first-order global splitting error. We can increase the order of accuracy by using Strang splitting.

Let's apply time splitting to a spectral method. Let  $\hat{u} \equiv F u$  and  $\hat{v} \equiv F v$  be the Fourier transforms of  $u$  and  $v$ . Then Fourier transform of the equation  $v_t = L v$  is  $\hat{v}_t = \hat{L} \hat{v}$ , where  $\hat{L}$  is the Fourier transform of the operator  $L$ , assuming it exists. This differential equation has the formal solution  $\hat{v}(\Delta t, \xi) = e^{\Delta t \hat{L}} \hat{u}(0, \xi)$ . Applying the inverse Fourier transform gives us

$$v(\Delta t, x) = F^{-1} \left[ e^{\Delta t \hat{L}} F u(0, x) \right].$$

We now solve  $w_t = N w$  with initial conditions  $w(0, x) = v(\Delta t, x)$  to get the solution  $u(\Delta t, x) = w(\Delta t, x) + O(\Delta t^2)$ . This solution is subsequently used as the initial condition  $v(0, x)$  for the next time step.

**Example.** Consider the Allen–Cahn equation

$$u_t = u_{xx} + \varepsilon^{-1} u(1 - u^2)$$

with periodic boundary conditions. We can write the problem as  $u_t = L u + N u$  where  $L u = u_{xx}$  and  $N u = \varepsilon^{-1} u(1 - u^2)$ . The Fourier transform of  $u_{xx}$  is  $-\xi^2 F u$ , and the differential equation  $u_t = \varepsilon^{-1} u(1 - u^2)$  has the analytical solution

$$u(\Delta t, x) = u_0 \left( u_0^2 - (u_0^2 - 1)e^{-2\varepsilon\Delta t} \right)^{-1/2}$$

where the initial conditions  $u_0(x) = u(0, x)$ . While having an analytic solution is convenient, we could have also solved the differential equation numerically if one didn't exist.

For each time step  $t_n$  we have

1.  $v(x) = F^{-1} \left[ e^{-\xi^2 \Delta t} F[u(t_n, x)] \right]$
2.  $u(t_{n+1}, x) = u_0 \left( u_0^2 - (u_0^2 - 1)e^{-2\varepsilon\Delta t} \right)^{-1/2}$  where  $u_0 = v(x)$ .

Each time step has an  $O(\Delta t^2)$  splitting error, so we have  $O(\Delta t)$  error after  $n$  iterations. By using Strang splitting we can achieve  $O(\Delta t^2)$  error. ◀

**Example.** A electron in an external potential  $V(x)$  is modeled in quantum mechanics using the Schrödinger equation

$$u_t = \frac{1}{2} i\varepsilon u_{xx} - i\varepsilon^{-1} V(x) u.$$

with scaled Planck constant  $\varepsilon$ . We can separate the spatial operator into a kinetic term  $\frac{1}{2} i\varepsilon u_{xx}$  and a potential energy term  $-i\varepsilon^{-1} V(x) u$ . Both of these terms are linear in  $u$ , and we can solve the problem using Strang splitting

$$u(x, t_{n+1}) = e^{-iV(x)\Delta t/2\varepsilon} F^{-1} e^{-i\varepsilon\xi^2\Delta t/2} F e^{-iV(x)\Delta t/2\varepsilon} u(x, t_n)$$

for a spectral-order-in-space and second-order-in-time solution. ◀

## ► Integrating factors

We are unable to get better than second-order accuracy by using time-splitting. We can achieve arbitrary-order accuracy if we don't split the operators. Of course, we must use an implicit A-stable or L-stable method to completely avoid stability issues if the problem is stiff. High-order implicit schemes can be difficult to implement if we have nonlinear terms. So, another approach is to use integrating factors to lessen the effect of the stiffness in a high-order explicit scheme.

Suppose that we want to solve  $u_t = L u + N u$ , where  $L$  is a linear operator,  $N$  is a nonlinear operator, and the initial conditions are given by  $u(0, x)$ . We can rewrite the equation as  $u_t - L u = N u$ . Then, taking the Fourier transform, we get

$$\hat{u}_t - \hat{L}\hat{u} = F[N u].$$

Multiple both sides of the equation by  $e^{-t\hat{L}}$  and simplify to get

$$\left( e^{-t\hat{L}} \hat{u} \right)_t = e^{-t\hat{L}} F[N u]. \quad (16.5)$$

Let  $\hat{w} = e^{-t\hat{L}} \hat{u}$ . Then the solution is

$$u(t, x) = F^{-1} \left[ e^{t\hat{L}} \hat{w}(t, \xi) \right]$$

where

$$\frac{\partial}{\partial t} \hat{w}(t, \xi) = e^{-t\hat{L}} F \left[ N \left( F^{-1} \left[ e^{t\hat{L}} \hat{w}(t, \xi) \right] \right) \right] \quad (16.6)$$

with initial conditions  $\hat{w}(0, \xi) = e^{0\hat{L}} F[u(0, x)] = F[u(0, x)]$ .

We can break the time interval into several increments  $\Delta t = t_{n+1} - t_n$  and implement integrating factors over each interval. In this case, we have for each interval starting at  $t_n$  and ending at  $t_{n+1}$  the following set-solve-set procedure

$$\begin{array}{ll} \text{set} & \hat{w}(t_n, \xi) = F u(t_n, x) \\ \text{solve} & \frac{\partial}{\partial t} \hat{w}(t, \xi) = e^{-\Delta t \hat{L}} F \left[ N \left( F^{-1} \left[ e^{\Delta t \hat{L}} \hat{w}(t, \xi) \right] \right) \right] \\ \text{set} & u(t_{n+1}, x) = F^{-1} \left( e^{\Delta t \hat{L}} \hat{w}(t_{n+1}, \xi) \right) \end{array}$$

It's not necessary to switch back to the variable  $u(t, x)$  with each iteration. Instead, we can solve the problem in the variable  $\hat{u}(t, \xi)$  and only switch back when we want to observe the solution.

The benefit of such integrating factors is that there is no splitting error. But, integrating factors only to lessen the effect of stiffness. Cox and Matthews developed a class of exponential time differencing Runge–Kutta (ETDRK) methods that solve (16.5) using matrix exponentials

$$u(t_{n+1}) = e^{\Delta t \hat{L}} u(t_n) + \int_0^{\Delta t} e^{(\Delta t - s)\hat{L}} \hat{N} u(t_n + s) ds$$

and evaluate the integral numerically using Runge–Kutta methods. Later, Hochbruck and Ostermann extended such a method to Rosenbrock methods.

## ► IMEX methods

Implicit-explicit (IMEX) methods solve  $u_t = L u + N u$  by using implicit integration on  $L u$  and explicit integration on  $N u$  while using the same approximation for the left-hand side. For example, a common second-order IMEX method combines a Crank–Nicolson method with an Adams–Bashforth method. Ascher, Ruuth, and Wetton extended this method to create a class of semi-implicit BDF (SBDF) methods. Later, Kennedy and Carpenter developed a class of high-order

additive Runge–Kutta (ARK) methods combining an explicit singly diagonal implicit Runge–Kutta (ESDIRK) scheme for the stiff term and a traditional explicit Runge–Kutta scheme (ERK) for the nonstiff term.

• The `DifferentialEquations.jl` function `SplitODEProblem` defines a problem to be solved using an IMEX or exponential method, such as `KenCarp4()` or `ETDRK4()`.

• The `DiffEqOperators.jl` function `DiffEqArrayOperator` can be used to define a spectral operator as a diagonal matrix.

**Example.** The Kuramoto–Sivashinsky equation (KSE) combines a nonlinear convection term, a fourth-order diffusion term, and a second-order backward-diffusion term

$$u_t + uu_x + u_{xxxx} + u_{xx} = 0 \quad (16.7)$$

to model small thermal instabilities in a laminar flame front. Imagine a flame fluttering above a gas burner of a kitchen stove. The backward-diffusion term adds instability into the system at large scales, the fourth-order term brings stability back into the system at small scales, and the nonlinear convection term transfers stability between scales resulting in the formation of fronts. Eventually, the KSE exhibits chaotic behavior.

The KSE is both quite stiff and also quite sensitive to perturbations, so we should use a high-order IMEX method. Take  $\hat{u} = F u$ , then the KSE

$$u_t = -u_{xxxx} - u_{xx} - \frac{1}{2}(u^2)_x$$

can be written as

$$\hat{u}_t = (-\xi^4 + \xi^2)\hat{u} - \frac{1}{2}i\xi F(F^{-1}\hat{u})^2.$$

Let's solve the KSE using an initial condition  $u(0, x) = e^{-x^2}$  over a periodic domain  $[-50, 50]$  and a timespan of 200.

```
using FFTW, DifferentialEquations, DiffEqOperators
ℓ = 100; T = 200.0; n = 512
x = LinRange(-ℓ/2, ℓ/2, n+1)[2:end]
u₀ = exp.(-x.^2)
iξ = im*[0:(n÷2); (-n÷2+1):-1]*2π/ℓ
F = plan_fft(u₀)
L = DiffEqArrayOperator(Diagonal(-iξ.^2-iξ.^4))
N = (u, _, _) -> -0.5iξ.*((F*(F\u)).^2)
problem = SplitODEProblem(L, N, F*u₀, (0, T))
solution = solve(problem, KenCarp4(), reltol=1e-12)
u = t -> real(F\solution(t))
```

We can visualize the time evolution of the solution as a grayscale image with time along the horizontal axis and space along the vertical axis.

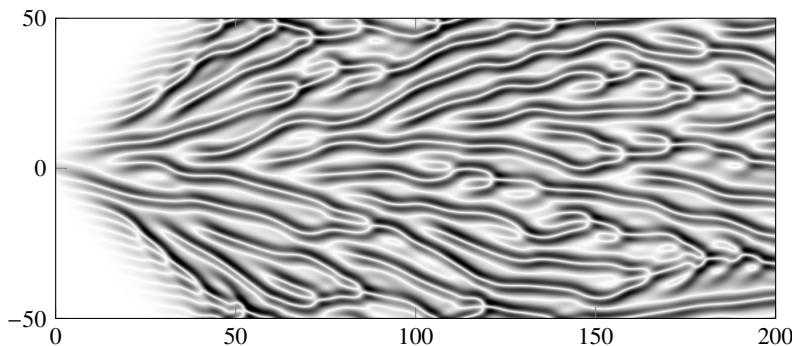


Figure 16.1: Solution to the Kuramoto–Sivashinsky equation for a Gaussian bump. Space is along the vertical axis and time is along the horizontal axis.

```
using ColorSchemes
s = hcat(u.(LinRange(0,T,500))...)
get(colorschemes[:binary],abs.(s),:extrema)
```

Black pixels depict magnitudes close to the maximum and white pixels depict values close to zero. See the figure above and the animation at the QR link at the bottom of this page. ▶

### 16.3 Incompressible Navier–Stokes equation

In this section, we examine the two-dimensional incompressible Navier–Stokes equation. The numerical solution follows a similar treatment discussed in Deuflhard and Hohmann [2003]. One-dimensional Fourier spectral methods can be easily modified to handle two- and three-dimensional problems by replacing the vector  $\xi$  with the tensor  $\xi_x + \xi_y$  or  $\xi_x + \xi_y + \xi_z$ .

Consider the two-dimensional incompressible Navier–Stokes equation

$$\frac{\partial}{\partial t} \mathbf{u} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) = -\nabla p + \varepsilon \Delta \mathbf{u} \quad (16.8)$$

$$\nabla \cdot \mathbf{u} = 0 \quad (16.9)$$

which models fluid flow. The vector  $\mathbf{u} = (u(t, x, y), v(t, x, y))$  is the velocity field,  $p(t, x, y)$  is the pressure, and  $\varepsilon$  is the inverse of the Reynold's number. The symbol  $\otimes$  denotes a tensor product— $\nabla \cdot (\mathbf{u} \otimes \mathbf{u})$  is  $\sum_j \partial_j u_i u_j$  in index notation.



Explicitly, the Navier–Stokes equation says

$$\begin{aligned} u_t + (u^2)_x + (uv)_y &= -p_x + \varepsilon(u_{xx} + u_{yy}) \\ v_t + (uv)_y + (v^2)_x &= -p_y + \varepsilon(v_{xx} + v_{yy}) \\ u_x + u_y &= 0. \end{aligned}$$

To solve the problem, we will find  $\mathbf{u}$  using the conservation of momentum equation (16.8) and use the conservation of mass (16.9) as a constraint by forcing  $\mathbf{u}$  to be divergence-free. Because (16.8) has stiff linear terms and nonlinear terms, we will use a second-order IMEX Adams–Bashforth/Crank–Nicolson method

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = \underbrace{-\frac{1}{2}(\nabla p^{n+1} + \nabla p^n)}_{\text{Crank–Nicolson}} + \underbrace{\frac{3}{2}\mathbf{H}^n - \frac{1}{2}\mathbf{H}^{n-1}}_{\text{Adams–Bashforth}} + \underbrace{\frac{1}{2}\varepsilon(\Delta\mathbf{u}^{n+1} + \Delta\mathbf{u}^n)}_{\text{Crank–Nicolson}}$$

where  $\mathbf{H}^n = \nabla \cdot (\mathbf{u}^n \otimes \mathbf{u}^n)$ . There are a couple problems with this set-up. First, we don't explicitly know  $p^{n+1}$ . Second, we haven't enforced the constraint (16.8). So, let's modify the approach by introducing an intermediate solution  $\mathbf{u}^*$ :

$$\frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} = -\frac{1}{2}\nabla p^n + \frac{3}{2}\mathbf{H}^n - \frac{1}{2}\mathbf{H}^{n-1} + \frac{1}{2}\varepsilon(\Delta\mathbf{u}^* + \Delta\mathbf{u}^n) \quad (16.10a)$$

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^*}{\Delta t} = -\frac{1}{2}\nabla p^{n+1} + \frac{1}{2}\varepsilon(\Delta\mathbf{u}^{n+1} - \Delta\mathbf{u}^*) \quad (16.10b)$$

We'll be able to use the intermediate solution so that we don't explicitly need to compute the pressure  $p$ . First, solve (16.10a). Formally, we have

$$\mathbf{M}_- \mathbf{u}^* = -\frac{1}{2}\nabla p^n + \frac{3}{2}\mathbf{H}^n - \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+ \mathbf{u}^n$$

where the operators

$$\mathbf{M}_- = \left( \frac{1}{\Delta t} - \frac{1}{2}\varepsilon\Delta \right) \quad \text{and} \quad \mathbf{M}_+ = \left( \frac{1}{\Delta t} + \frac{1}{2}\varepsilon\Delta \right).$$

From this we have

$$\mathbf{u}^* = \mathbf{M}_- \left( -\frac{1}{2}\nabla p^n + \frac{3}{2}\mathbf{H}^n - \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+ \mathbf{u}^n \right). \quad (16.11)$$

We need to determine  $\nabla p^{n+1}$  in such a way as to implicitly enforce the IMEX scheme by using (16.10b):

$$-\frac{1}{2}\nabla p^{n+1} = \mathbf{M}_- (\mathbf{u}^{n+1} - \mathbf{u}^*).$$

Equivalently,

$$-\frac{1}{2}\nabla p^n = \mathbf{M}_- (\mathbf{u}^n - \mathbf{u}^{*-1}) \quad (16.12)$$

where  $\mathbf{u}^{*-1}$  is the intermediate solution at the previous timestep. Substituting  $-\frac{1}{2}\nabla p^n$  back into the first equation we have

$$\begin{aligned}\mathbf{u}^* &= \mathbf{M}_-^{-1}(\mathbf{M}_-(\mathbf{u}^n - \mathbf{u}^{*-1}) + \frac{3}{2}\mathbf{H}^n - \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+ \mathbf{u}^n) \\ &= \mathbf{u}^n - \mathbf{u}^{*-1} + \mathbf{M}_-^{-1}\left(\frac{3}{2}\mathbf{H}^n - \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+ \mathbf{u}^n\right).\end{aligned}$$

Finally, we enforce (16.9) by projecting the solution  $\mathbf{u}^*$  onto a divergence-free vector field. Note that if we define

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \nabla \Delta^{-1} \nabla \cdot \mathbf{u}^*,$$

then<sup>2</sup>

$$\nabla \cdot \mathbf{u}^{n+1} = \nabla \cdot \mathbf{u}^* - \nabla \cdot \nabla \Delta^{-1} \nabla \cdot \mathbf{u}^* = \nabla \cdot \mathbf{u}^* - \Delta \Delta^{-1} \nabla \cdot \mathbf{u}^* = 0.$$

We now have

$$\begin{aligned}\mathbf{u}^* &= \mathbf{u}^n - \mathbf{u}^{*-1} + \mathbf{M}_-^{-1}\left(\frac{3}{2}\mathbf{H}^n - \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+ \mathbf{u}^n\right) \\ \mathbf{u}^{n+1} &= \mathbf{u}^* - \nabla \Delta^{-1} \nabla \cdot \mathbf{u}^*.\end{aligned}$$

To initialize the multistep method, we'll take

$$\mathbf{u}^{-1} = \mathbf{u}^{*-1} = \mathbf{u}^0.$$

In the Fourier domain, we replace

$$\nabla \rightarrow i(\xi_x, \xi_y) \quad \text{and} \quad \Delta = \nabla \cdot \nabla \rightarrow -\xi_x^2 - \xi_y^2 = -|\xi|^2.$$

And, from (16.13) we get (using the hat notation  $\hat{\square}$  for the Fourier transform  $F \square$ )

$$\hat{\mathbf{u}}^* = \hat{\mathbf{u}}^n - \hat{\mathbf{u}}^{*-1} + \hat{\mathbf{M}}_+^{-1}\left(\frac{3}{2}\hat{\mathbf{H}}^n - \frac{1}{2}\hat{\mathbf{H}}^{n-1} + \hat{\mathbf{M}}_- \hat{\mathbf{u}}^n\right) \quad (16.14a)$$

$$\hat{\mathbf{u}}^{n+1} = \hat{\mathbf{u}}^* - \frac{(\xi \cdot \hat{\mathbf{u}}^*)}{|\xi|^2} \xi \quad (16.14b)$$

where

$$\hat{\mathbf{M}}_- = \frac{1}{At} - \frac{1}{2}\varepsilon|\xi|^2 \quad \text{and} \quad \hat{\mathbf{M}}_+ = \frac{1}{At} + \frac{1}{2}\varepsilon|\xi|^2.$$

We must take some care when we divide by  $|\xi|^2$  in (16.14b) because one of the components is zero. In this case we modify the term by replacing the zeros by ones. This fix is itself fixed when we subsequently multiply by  $\xi$ .

We haven't discussed implementation of the nonlinear term  $\hat{\mathbf{H}}$  yet. If  $\mathbf{u}$  is given by  $(u, v)$ , the  $x$ -component of  $\mathbf{H}$  is

$$H(u, v) = (uv)_y + (u^2)_x,$$

---

<sup>2</sup>Do you recognize the projection operators  $\mathbf{P}_A = A(A^T A)^{-1} A^T$  and  $\mathbf{I} - \mathbf{P}_A$ ?

with a similar form for a  $y$ -component. The Fourier transform of a product of two functions  $uv$  is the convolution  $\hat{u} * \hat{v}$ —a very inefficient operator to implement numerically. Therefore, we'll evaluate  $\hat{\mathbf{H}}$  in spatial domain rather than the Fourier domain. Hence, we take

$$\hat{H}(\hat{u}, \hat{v}) = i\xi_y F(F^{-1} \hat{u} F^{-1} \hat{v}) + i\xi_x F(F^{-1} \hat{u})^2.$$

To visualize the solution, we use a tracer. Imagine dropping ink (or smoke) into a moving fluid and then following the motion of the ink. Numerically, we can simulate a tracer by solving the advection equation

$$\frac{\partial Q}{\partial t} + \mathbf{u} \cdot \nabla Q = 0 \quad (16.15)$$

using, for example, the Lax–Wendroff method where  $\mathbf{u} = (u, v)$  is the solution to the Navier–Stokes equation at each time step.

Consider a stratified fluid moving along the  $x$ -direction separated from a stationary fluid by a narrow interface. Suppose that speed of the moving fluid is also modulated along the  $x$ -direction. Think of wind blowing over water. The pressure is reduced in regions in which the moving fluid moves faster, pulling on the stationary fluid and causing it to bulge. Such a situation leads to Kelvin–Helmholtz instability, producing surface waves in water and leading to turbulence in otherwise laminar fluids. Kelvin–Helmholtz instability is seen in the cloud bands of Jupiter, the sun's corona, and billow clouds here on Earth. Some think that the Kelvin–Helmholtz instability in billow clouds may have inspired the swirls in van Gogh's painting "The Starry Night." See the figure on the following page. To model an initial stratified flow, we can take  $(u, v)$  with

$$u = \frac{1}{4}Q(x, y) \cdot (2 + \sin \ell \pi x) \text{ where } Q(x, y) = 1 + \tanh\left(10 \cdot \left(1 - |y - \frac{1}{2}\ell|\right)\right)$$

with  $\ell = 2$  and  $v = 0$ .

The Julia code to solve the Navier–Stokes equation has three parts: defining the functions, initializing the variables, and iterating over time. We start by defining functions for  $\hat{\mathbf{H}}$  and for the flux used in the Lax–Wendroff method.

```
 $\Delta^\circ(Q, \text{step}=1) = Q - \text{circshift}(Q, (\text{step}, 0))$
 $\text{flux}(Q, c) = c.*\Delta^\circ(Q) - 0.5c.*(1 - c).*(\Delta^\circ(Q, 1)+\Delta^\circ(Q, -1))$
 $H_0(u, v, i\xi_1, i\xi_2) = -i\xi_1.*\text{fft}(\text{ifft}(u).^2) - i\xi_2.*\text{fft}(\text{ifft}(u).*\text{ifft}(v))$
```

Then we initialize the variables.

```
using FFTW
 $\ell = 2; n = 128; \epsilon = 0.001; \Delta t = 0.001; \Delta x = \ell/n$
 $x = \text{LinRange}(\Delta x, \ell, n)'; y = x'$
 $Q = (@. 0.5(1+\tanh(10(1-abs(\ell/2 - y)/(\ell/4))))).*\text{ones}(1, n)$
```

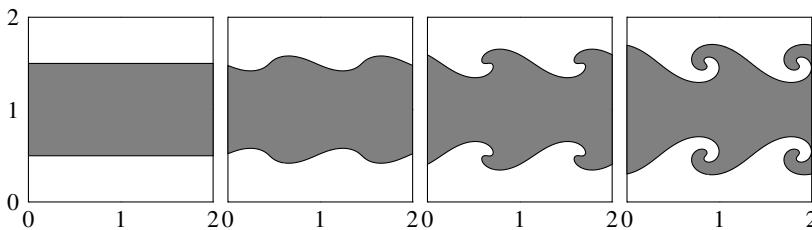


Figure 16.2: The solution to the Navier–Stokes equation at time 0, 0.4, 0.8, and 1.2. Notice the development of Kelvin–Helmholtz instability over time. Also, see the QR link at the bottom of this page.

```

un = Q .* (1 .+ 0.5sin.(ℓ*π*x))
vn = zeros(n,n)
(un,vn) = (fft(un),fft(vn))
(u°,v°) = (un,vn)
iξ1 = im*fftfreq(n,n)'*(2π/ℓ)
iξ2 = transpose(iξ1)
ξ2 = iξ1.^2 .+ iξ2.^2
(H1n, H2n) = (H0(un,vn,iξ1,iξ2), H0(vn,un,iξ2,iξ1))
M+ = (1/Δt .+ (ε/2)*ξ2)
M- = (1/Δt .- (ε/2)*ξ2);

```

Finally, we loop in time.

```

for i = 1:1200
 Q := flux(Q,(Δt/Δx).*real(ifft(vn))) +
 flux(Q',(Δt/Δx).*real(ifft(un))')'
 (H1n-1, H2n-1) = (H1n, H2n)
 (H1n, H2n) = (H0(un,vn,iξ1,iξ2), H0(vn,un,iξ2,iξ1))
 u° = un - u° + (1.5H1n - 0.5H1n-1 + M+.*un)./M-
 v° = vn - v° + (1.5H2n - 0.5H2n-1 + M+.*vn)./M-
 φ = (iξ1.*u° + iξ2.*v°)./(ξ2 .+ (ξ2.≈ 0))
 (un, vn) = (u° - iξ1.*φ, v° - iξ2.*φ)
end
contour(x',y, Q, fill=true, levels=1, legend=:none)

```

## 16.4 Exercises

- 16.1. Solve the Schrödinger equation discussed on page 449 using Strang splitting and using integrating factors. Take a harmonic potential  $V(x) = x^2$  with initial conditions  $u(0, x) = e^{-(x-3)^2/\varepsilon}$  with  $\varepsilon = 0.1$  over a domain  $[-8, 8]$ .



16.2. Suppose that you use a fourth-order Runge–Kutta scheme to solve the Burgers equation  $u_t + \frac{1}{2} (u^2)_x = 0$  discretized with a Fourier spectral method. What order can you expect? What else can you say about the numerical solution?

16.3. The Kortweg–deVries (KdV) equation

$$u_t + 3(u^2)_x + u_{xxx} = 0 \quad (16.16)$$

is a nonlinear equation used to model wave motion in a dispersive medium such as shallow water or optic fiber. The purpose of this exercise is to explore the behavior the KdV equation by first developing a high-order numerical scheme to solve the equation and then using the scheme to simulate soliton interaction.

Dispersion means that different frequencies travel at different speeds causing a solution to break up and spread out over time. For certain class of initial conditions, the nonlinearity of the KdV equation counteracts the dispersion and the solution maintains its shape. We call such a solution a solitary wave or a soliton. It can be easily shown by substitution that

$$\phi(x, t; x_0, c) = \frac{1}{2}c \operatorname{sech}^2\left(\frac{\sqrt{c}}{2}(x - x_0 - ct)\right)$$

is a soliton solution to the KdV equation. This solution is a traveling wave solution which simply moves with speed  $c$  and does not change shape.

Solve the KdV equation (16.16) on the interval  $[-15, 15]$  for  $t \in [0, 2]$  using a fourth-order Runge–Kutta method with integrating factors. Observe the solutions to (16.16) with initial conditions given by  $\phi(x, 0; -4, 4)$ ,  $\phi(x, 0; -9, 9)$ , and  $\phi(x, 0; -4, 4) + \phi(x, 0; -9, 9)$ . These initial conditions produce solitons with speeds 4 and 9 centered at  $x = -4$  and  $x = -9$ , respectively. In the two soliton case, the two solitons should combine nonlinearly as the faster one overtakes the slower one. You may also want to observe a three-soliton collision by adding another soliton (say  $\phi(x, 0; -12, 12)$ ).

16.4. The two-dimensional Swift–Hohenberg equation

$$\frac{\partial u}{\partial t} = \varepsilon u - u^3 - (\Delta + 1)^2 u$$

with  $u \equiv u(t, x, y)$  models Rayleigh–Bénard convection, which results when a shallow pan of water is heated from below. Compute the numerical solution for  $\varepsilon = 1$  over a square with sides of length 100. Assume periodic boundary conditions with 256 grid points in  $x$  and  $y$ . Choose  $u(0, x, y)$  randomly from the uniform distribution  $[-1, 1]$ . Output the solution at time  $t = 100$  and several intermediate times.

¶ 16.5. If a homogeneous molten alloy of two metals is rapidly cooled, a sudden phase separation can set in. Alternatively, imagine vigorously shaking a bottle of vinaigrette and then watching the oil and vinegar separate, first into tiny globules and then coarsening into larger and larger ones. The Cahn–Hilliard equation

$$\frac{\partial \phi}{\partial t} = \Delta \left( f'(\phi) - \varepsilon^2 \Delta \phi \right) \quad \text{with} \quad f'(\phi) = \phi^3 - \phi$$

models such phase separation. The dynamics of the Cahn–Hilliard equation are driven by surface tension flow, in which interfaces move with velocities proportional to mean curvature and the total volumes of the binary regions are constant over time. A convexity-splitting method

$$\frac{\partial \phi}{\partial t} = L \phi + N \phi = \left( -\varepsilon^2 \Delta^2 + \alpha \Delta \right) \phi^{(n+1)} + \Delta \left( f'(\phi^{(n)}) - \alpha \phi^{(n)} \right)$$

balances the terms to ensure energy stability—see Wu et al. [2014] or Lee et al. [2014]. Solve the two-dimensional Cahn–Hilliard equation with  $\varepsilon = 0.1$  starting with random initial conditions over a square periodic domain of length 8 over a time  $[0, 8]$ . Take  $\alpha = 1$ .

¶ 16.6. The two-dimensional Kuramoto–Sivashinsky equation can be written as

$$\frac{\partial v}{\partial t} + \Delta v + \Delta^2 v + \frac{1}{2} |\nabla v|^2 = 0.$$

where  $\mathbf{u} = \nabla v$  is the two-dimensional vector-field extension of the variable  $u$  in the one-dimensional KSE (16.7). To solve this equation, we'll further impose a mean-zero restriction  $\int_{\Omega} |\nabla v|^2 d\Omega = 0$ . Solve the two-dimensional KSE using a Gaussian bump or normal random initial conditions over a periodic domain of length 50 and a timespan  $t = 150$ .

The mean-zero restriction can be enforced by zeroing out the zero-frequency component in the Fourier domain. Kalogirou [2014] develops a second-order IMEX BDF method using the splitting

$$\frac{\partial v}{\partial t} = L v + N v = -(\Delta^2 + \Delta + \alpha)v - (\frac{1}{2} |\nabla v|^2 - \alpha v),$$

where  $\alpha$  is a positive splitting-stabilization constant added to ensure that operator on the implicit term is positive-definite and thus invertible.

# Back Matter



## Appendix A

---

# Solutions

### A.1 Numerical methods for linear algebra

1.3. For the Krylov subspace:

- (a) If  $\mathbf{x}$  is an eigenvector of  $\mathbf{A}$ , then  $\mathbf{Ax} = \lambda\mathbf{x}$ ,  $\mathbf{A}^2\mathbf{x} = \lambda^2\mathbf{x}$  and so on. The subspace is spanned entirely by  $\mathbf{x}$ , so its dimension is 1.
- (b) If  $\mathbf{x} = a\mathbf{v}_1 + b\mathbf{v}_2$  for two linearly independent eigenvectors of  $\mathbf{A}$ , then

$$\begin{aligned}\mathbf{Ax} &= a\mathbf{Av}_1 + b\mathbf{Av}_2 = a\lambda_1\mathbf{v}_1 + b\lambda_2\mathbf{v}_2 \\ \mathbf{A}^2\mathbf{x} &= a\mathbf{A}^2\mathbf{v}_1 + b\mathbf{A}^2\mathbf{v}_2 = a\lambda_1^2\mathbf{v}_1 + b\lambda_2^2\mathbf{v}_2 \\ &\text{and so on...}\end{aligned}$$

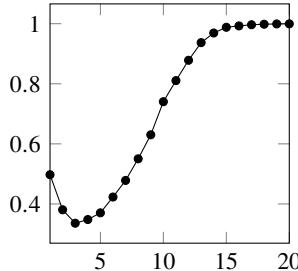
So, the Krylov subspace is spanned by linear combinations of the eigenvectors  $\mathbf{v}_1$  and  $\mathbf{v}_2$  and the dimension is 2.

- (c) If  $\mathbf{A}$  is a projection operator, then  $\mathbf{A}^2 = \mathbf{A}$ . It follows that the Krylov subspace  $\mathcal{K}_r(\mathbf{A}, \mathbf{x}) = \text{span}\{\mathbf{x}, \mathbf{Ax}\}$ . If  $\mathbf{x}$  is in the null space of  $\mathbf{A}$  then  $\mathbf{Ax} = \mathbf{0}$ . So, the maximum dimension is 2.
- (d) For any vector  $\mathbf{x}$ , the vector  $\mathbf{Ax}$  is in the column space of  $\mathbf{A}$ , which has dimension  $n - m$ . So, the Krylov subspace is spanned by  $\mathbf{x}$  and the column space of  $\mathbf{A}$ . So, the maximum dimension of the Krylov subspace is  $n - m + 1$  (and  $n - m$  if  $\mathbf{x}$  is in the column space of  $\mathbf{A}$ ).

1.4. The number of invertible (0,1)-matrices has been the subject of some research and is a yet unsolved problem. There is an entry in the Online Encyclopedia of Integer Sequences (often simply called Sloane in reference to its curator Neil Sloane) at <http://oeis.org/A055165>. We can estimate the number by generating

random (0,1)-matrices and testing them for singularity. The following Julia code generates the accompanying plot:

```
using LinearAlgebra, Plots
N = 10000
n = [sum([!iszero(det(rand(Bool,d,d))) for i in 1:N]) for d in 1:20]
plot(n/N,marker=:o)
```



1.5. (a) Let  $\mathbf{M}_n = \mathbf{D}_n + 2\mathbf{I}$ . If  $\{\lambda, \mathbf{x}\}$  is an eigenpair of  $\mathbf{M}_n$ , then

$$\mathbf{D}_n \mathbf{x} = \frac{1}{2}(\mathbf{M}_n - \mathbf{I})\mathbf{x} = \frac{1}{2}(\lambda - 1)\mathbf{x}.$$

So,  $\left\{\frac{1}{2}(\lambda - 1), \mathbf{x}\right\}$  is an eigenpair of  $\mathbf{D}_n$ . The eigenvalues  $\lambda$  of  $\mathbf{M}_n$  are zeros of the characteristic polynomial  $p_n(\lambda) = \det(\mathbf{M}_n - \lambda\mathbf{I})$ , which we can evaluate using the method of cofactors. In this case we have,

$$\begin{aligned} p_n(\lambda) &= \det(\mathbf{M}_n - \lambda\mathbf{I}) \\ &= -\lambda \det(\mathbf{M}_{n-1} - \lambda\mathbf{I}) - \det(\mathbf{M}_{n-2} - \lambda\mathbf{I}) \\ &= -\lambda p_{n-1}(\lambda) - p_{n-2}(\lambda) \end{aligned}$$

By making the change of variables  $\lambda = -2\gamma$  and  $p_n(\lambda) = T_n(\gamma)$ , we have

$$T_n(\gamma) = 2\gamma T_{n-1}(\gamma) - T_{n-2}(\gamma),$$

which is the three-term recurrence relation for the Chebyshev polynomial  $T_n(\gamma)$ . The Chebyshev nodes (the roots of the Chebyshev polynomial) are given by

$$\gamma_j = \cos\left(\frac{2j-1}{2n}\pi\right), \quad j = 1, 2, \dots, n$$

So,

$$\lambda_j = -2 \cos\left(\frac{2j-1}{2n}\pi\right), \quad j = 1, 2, \dots, n$$

Therefore, the eigenvalues of  $\mathbf{D}_n$  are given by

$$\frac{1}{2}(\lambda_j - 1) = -1 - \cos\left(\frac{2j-1}{2n}\pi\right), \quad j = 1, 2, \dots, n$$

- (b) The spectral radius of  $\mathbf{D}_n + 2\mathbf{I}$  is

$$\rho(\mathbf{D}_n + 2\mathbf{I}) = \max_{j=1,\dots,n} \left| -2 \cos\left(\frac{2j-1}{2n}\pi\right) \right| = 2 \cos\left(\frac{\pi}{2n}\right)$$

- (c) The 2-condition number for a symmetric matrix is the ratio of the largest and smallest eigenvalues.

$$\kappa_2(\mathbf{D}_n) = \frac{1 + \cos(\pi/2n)}{1 - \cos(\pi/2n)} \approx \frac{2 - \frac{1}{2}(\pi/2n)^2}{\frac{1}{2}(\pi/2n)^2} \approx \frac{16}{\pi^2} n^2$$

when  $n$  is large. So the condition number of  $D_n$  is  $O(n^2)$ .

- (d) An easy way to input  $\mathbf{D}_n$  in Julia is with

```
D = SymTridiagonal(-2ones(n), ones(n-1))
```

Then we can compute the eigenvalues and eigenvectors using `eigen(D)`. Alternatively, we can use the pipe operator `D |> eigen`.

### 1.9. We'll start with (e)

- (e) The trace of a matrix equals the sum of its diagonal elements. The  $i$ th diagonal element of  $\mathbf{A}^\top \mathbf{A}$  equals  $\sum_{j=1}^n a_{ij}^2$ . So  $\text{tr}(\mathbf{A}^\top \mathbf{A}) = \sum_{i,j=1}^n a_{ij}^2$ , which equals  $\|\mathbf{A}\|_F^2$ .
- (b) The matrix  $\mathbf{Q}^\top \mathbf{Q}$  is  $n \times n$  identity matrix  $\mathbf{I}$ . The trace of  $\mathbf{I}$  is  $n$ . Therefore, using the result from (e),  $\|\mathbf{Q}\|_F = \sqrt{n}$ .
- (c) Using the result from (e),

$$\|\mathbf{Q}\mathbf{A}\|_F^2 = \text{tr}((\mathbf{Q}\mathbf{A})^\top \mathbf{Q}\mathbf{A}) = \text{tr}(\mathbf{A}^\top \mathbf{Q}^\top \mathbf{Q}\mathbf{A}) = \text{tr}(\mathbf{A}^\top \mathbf{A}) = \|\mathbf{A}\|_F^2.$$

- (d) Suppose that  $\mathbf{A}$  has the singular value decomposition  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top$ . Then using the result of (c),

$$\|\mathbf{A}\|_F^2 = \|\mathbf{U}\Sigma\mathbf{V}^\top\|_F^2 = \|\Sigma\mathbf{V}^\top\|_F^2 = \|\mathbf{V}\Sigma^\top\|_F^2 = \|\Sigma^\top\|^2 = \sum_{i=1}^n \sigma_i^2.$$

- (a) By reshaping the  $m \times n$  vector  $\mathbf{A}$  as an  $mn \times 1$  array  $\mathbf{a}$ , we can interpret the Frobenius norm as simply the 2-norm of  $\mathbf{a}$ . The 2-norm of a vector satisfies positivity, homogeneity and the triangle inequality, so the Frobenius norm does, too.

(g) To show submultiplicativity  $\|\mathbf{AB}\|_F \leq \|\mathbf{A}\|_F \|\mathbf{B}\|_F$ , we compute

$$\begin{aligned}\|\mathbf{AB}\|_F^2 &= \sum_{i,j} \sum_k (a_{ik} b_{kj})^2 = \sum_{i,j} \sum_k a_{ik}^2 b_{kj}^2 = \sum_k \left( \sum_i a_{ik}^2 \right) \left( \sum_j b_{kj}^2 \right) \\ &\leq \sum_{i,k} a_{ik}^2 \sum_{k,j} b_{kj}^2 = \|\mathbf{A}\|_F^2 \|\mathbf{B}\|_F^2.\end{aligned}$$

(f)  $\|\mathbf{Ax}\|_F \leq \|\mathbf{Ax}\|_F \|\mathbf{x}\|$  follows from submultiplicativity proved in (g).

2.3. The determinant of a product is given by the product of the determinants. We can use this identity along with LU factorization to efficiently determine the compute of a matrix. Using partial pivoting, the LU factorization of a matrix  $\mathbf{A}$  is given by  $\mathbf{PA} = \mathbf{LU}$  where  $\mathbf{P}$  is a permutation matrix,  $\mathbf{L}$  is a lower triangular matrix with ones along its diagonal, and  $\mathbf{U}$  is an upper triangular matrix. The determinant of a triangular matrix is the product of the diagonal elements. So,  $\det \mathbf{L} = 1$ . The determinant of a permutation matrix  $\mathbf{P}$  is the sign of the permutation  $\text{sign}(\mathbf{P})$ , which equals  $+1$  if  $\mathbf{P}$  was built from  $\mathbf{I}$  using an even number of row exchanges (even parity) and  $-1$  if  $\mathbf{P}$  was built using an odd number of row exchanges (odd parity). So,

$$\det \mathbf{A} = \det \mathbf{P} \det \mathbf{L} \det \mathbf{U} = \text{sign}(\mathbf{P}) \prod_{i=1}^N \text{diag } \mathbf{U}.$$

The `LinearAlgebra.jl` function `lu` will give us  $\mathbf{U}$  and  $\mathbf{P}$ , so we just need to compute the sign of the permutation.

```
function detx(A)
 L,U,P = lu(A)
 s = 1
 for i in 1:length(P)
 m = findfirst(P.==i)
 i!=m && (s *= -1; P[[m,i]] = P[[i,m]])
 end
 s * prod(diag(U))
end
```

2.4. Let's fill in the steps of the Cuthill–McKee algorithm outlined on page 51. Start with a  $(0,1)$ -adjacency matrix  $\mathbf{A}$ , and create a list  $r$  of vertices ordered from lowest to highest degree. Create another initially empty list  $q$  that will serve as a temporary first-in-first-out queue for the vertices before finally adding them to the permutation list  $p$ , also initially empty. Start by moving the first element

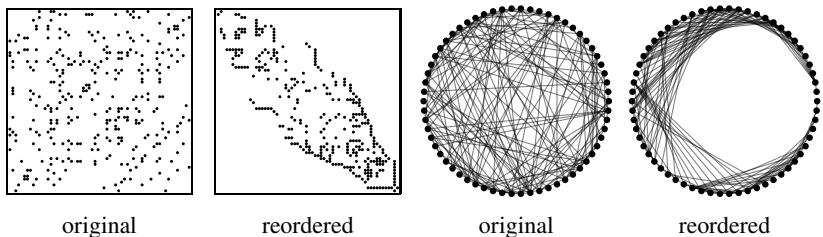


Figure A.1: Sparsity plots and graph drawings of the dolphins of Doubtful Sound.

from  $r$  to  $q$ . As long as the queue  $q$  is not empty, move the first element of  $q$  to the end of  $p$  and move any vertex in  $r$  adjacent to this element to end of  $q$ . If  $q$  is ever empty, go back and move the first element of  $r$  to  $q$ . Continue until all the elements from  $r$  have been moved through  $q$  to  $p$ . Finally, we reverse the order of  $p$ . The following Julia function produces a permutation vector  $p$  for a sparse matrix  $A$  using the reverse Cuthill–McKee algorithm.

```
function rcuthillmckee(A)
 r = sortperm(vec(sum(A .!= 0, dims=2)))
 p = Int64[]
 while ~isempty(r)
 q = [popfirst!(r)]
 while ~isempty(q)
 q1 = popfirst!(q)
 append!(p, q1)
 k = findall(!iszero, A[q1, r])
 append!(q, splice!(r, k))
 end
 end
 reverse(p)
end
```

We can test the solution using the following code:

```
using SparseArrays, LinearAlgebra, Plots
A = Symmetric(sprand(1000, 1000, 0.003))
p = rcuthillmckee(A)
plot(spy(A), spy(A[p,p]), colorbar = false)
```

2.5. Let's plot the graph of the Doubtful Sound dolphins reusing the code on page 48. The sparsity plots and graphs before and after Cuthill–McKee resorting are shown in the figure above.

```
p = rcuthillmckee(A)
gplot(Graph(A[p,p]), layout=circular_layout)
```

2.7. Let's load the CSV file as a dataframe. This will allow us to inspect and interpret the data.

```
using CSV, DataFrames
diet = DataFrame(CSV.File(download(bucket*"diet.csv")))
```

We can solve the diet problem as a dual LP problem: “Find the minimum of the objective function  $z = \mathbf{b}^T \mathbf{y}$  subject to constraint  $\mathbf{A}^T \mathbf{y} \geq \mathbf{c}$  and non-negativity restriction  $\mathbf{y}, \mathbf{c} \geq 0$ .” The nutritional value for each of the commodities is given by  $\mathbf{A}$ , the nutrient minimums is given by  $\mathbf{b}$ , and because the nutritional values of foods are normalized per dollar of expenditure,  $\mathbf{c}$  is a vector of ones.

```
A = Array(diet[2:end,4:end])'
c = ones.(size(A,2))
b = Array(diet[1,4:end])
food = diet[2:end,1]
solution = simplex(b,A',c)
print("foods: ", food[solution.y .!= 0], "\n")
print("daily cost: ", solution.z)
```

We can alternatively use JuMP.jl along with GLPK.jl. The JuMP function `value`, which returns the solution’s values, has a name common to other libraries. To avoid confusion or conflicts, we’ll explicitly call it using `JuMP.value`.

```
using JuMP, GLPK
model = Model(with_optimizer(GLPK.Optimizer))
@variable(model, x[1:size(A,2)] ≥ 0)
@objective(model, Min, c' * x)
@constraint(model, A * x .≥ b)
optimize!(model)
print("foods: ", food[JuMP.value.(x) .!= 0], "\n")
print("daily cost: ", objective_value(model))
```

A combination of the following foods meet the nutritional requirements at the lowest cost: enriched wheat flour, beef liver, cabbage, spinach, and navy beans. The cost is roughly 11 cents per day in 1939 dollars—about \$2.12 today.

2.8. Let's write a breath-first search algorithm—we can modify the code for the Cuthill–McKee algorithm in exercise 2.4. We'll keep track of three arrays: an array  $q$  that records the unique indices of the visited nodes and their neighboring nodes, an array  $p$  of pointers to the parent nodes recorded in this first array, and

an array  $r$  of nodes that haven't yet been added to array  $q$ . Start with a source node  $a$  and find all nearest neighbors using the adjacency matrix  $A$  restricted to rows  $r$ . Then with each iteration, march along  $q$  looking for new neighbors until we've either found the destination node  $b$  or we've run out of new nodes to visit, indicating that there is no path. Once we've found  $b$ , we backtrack along  $p$  to determine the path.

```
function findpath(A,a,b)
 r = collect(1:size(A,2))
 q = [a]; p = [-1]; i = 0
 splice!(r,a)
 while i<length(q)
 k = findall(!iszero, A[q[i+1],r])
 any(r[k].==b) && return(append!(q[backtrack(p,i)],b))
 append!(q,splice!(r,k))
 append!(p,fill(i,length(k)))
 end
 display("No path.")
end
```

```
function backtrack(p,i)
 s = []; while (i!= -1); append!(s,i); i = p[i]; end
 return(reverse(s))
end
```

Let's import the data and build an adjacency matrix. We'll use the function `get_adjacency_matrix` defined on page 48 to build the adjacency matrix, and we'll use `get_names` to build a list of the names.

```
get_names(filename) = readdlm(download(bucket*filename*.txt'), '\n')

actors = get_names("actors")
movies = get_names("movies")
B = get_adjacency_matrix("actor-movie");
```

The biadjacency matrix  $B$  indicates which actors (columns) appeared in which movies (rows). The matrix  $BB^T$  gives us the movie-movie adjacency matrix, and the matrix  $B^TB$  gives us the actor-actor adjacency matrix. The adjacency matrix of a bipartite graph can be written as a block matrix

$$A = \begin{bmatrix} \mathbf{0} & B^T \\ B & \mathbf{0} \end{bmatrix}.$$

Using  $A$  will tell us both the actors and their connecting movies.

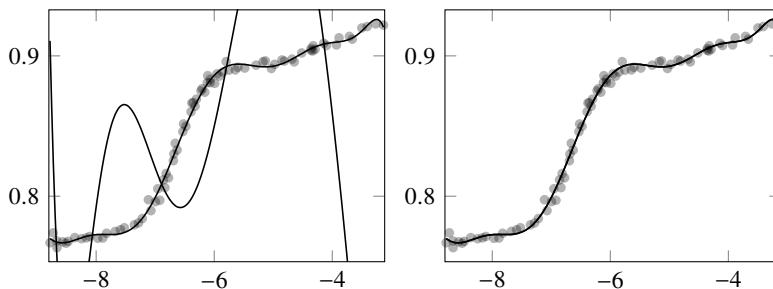


Figure A.2: Solutions to Exercise 3.4. The left plot shows solutions using the Vandermonde matrix built using the original  $x$ -data and the right plot shows solutions using the Vandermonde matrix built using the mean-centered  $x$ -data.

```
(m,n) = size(B)
A = [spzeros(n,n) B' ; B spzeros(m,m)];
actormovie = [actors ; movies];
```

Let's find the link between actors Emma Watson and Kevin Bacon.

```
index = x -> findfirst(actors.==x)[1]
actormovie[findpath(A, index("Emma Watson"), index("Kevin Bacon"))]
```

Emma Watson appeared in *Harry Potter and the Chamber of Secrets* with John Cleese, who appeared in *The Big Picture* with Kevin Bacon. Let's try another:

```
actormovie[findpath(A, index("Bruce Lee"), index("Tommy Wiseau"))]
```

Bruce Lee appeared in *Enter the Dragon* with Jackie Chan, who appeared in *Rush Hour* with Barry Shabaka Henley, who was in *Patch Adams* with Greg Sestero, who appeared in *The Room* with the great Tommy Wiseau. One might wonder which actor is the center of Hollywood. We'll come back to answer this question in exercise 4.6.

**3.4.** We'll start by defining a function to construct a Vandermonde matrix and evaluate a polynomial using that matrix:

```
vandermonde(t,n) = vec(t).^(0:n-1)'
build_poly(c,X) = vandermonde(X,length(c))*c
```

Now, we'll make a function that determines the coefficients and residuals using three different methods. The command `Matrix(Q)` returns the thin version of the `QRCompactWYQ` object returned by the `qr` function.

```

function solve_filip(x,y,n)
 V = vandermonde(x, n)
 c = Array{Float64}(undef, 3, n)
 c[1,:] = (V'*V)\(V'*y)
 Q,R = qr(V)
 c[2,:] = R\(Matrix(Q)'*y)
 c[3,:] = pinv(V,1e-9)*y
 r = [norm(V*c[i,:]-y) for i in 1:3]
 return(c,r)
end

```

Let's download the NIST Filip dataset, solve the problem, and plot the solutions:

```

using DelimitedFiles
data = readdlm(download(bucket*"filip.csv"), ',', Float64)
coef = readdlm(download(bucket*"filip-coeffs.csv"), ',')
(x,y) = (data[:,2],data[:,1])
β,r = solve_filip(x, y, 11)
X = LinRange(minimum(x),maximum(x),200)
Y = [build_poly(β[i,:],X) for i in 1:3]
plot(X,Y); scatter!(x,y,opacity=0.5)
[coef β']

```

Let's also solve the problem and plot the solutions for the standardized data:

```

using Statistics
zscore(X,x=X) = (X .- mean(x))/std(x)
c,r = solve_filip(zscore(x), zscore(y), 11)
Y = [build_poly(c[i,:],zscore(X,x))*std(y).+mean(y) for i in 1:3]
plot(X,Y); scatter!(x,y,opacity=0.5)

```

The 2-condition number of the Vandermonde matrix is  $1.7 \times 10^{15}$ , which makes it very ill-conditioned. The residuals are  $\|\mathbf{r}\|_2 = 0.97998$  using the normal equations and  $\|\mathbf{r}\|_2 = 0.028211$  using QR decomposition, which matches the residual for the coefficients provided by NIST. The relative errors are roughly 11 and  $4 \times 10^{-7}$ , respectively. The normal equation solution clearly fails, which is evident in Figure A.2 on the facing page.

After mean centering the  $x$ -data, the 2-condition number of the new Vandermonde matrix is reduced to  $1.3 \times 10^5$ . By further scaling the  $x$ -data between  $-1$  and  $1$ , the 2-condition number is reduced to  $\kappa_2 = 4610$ . So, the matrix is well conditioned even when using the normal equation method. The residuals of both methods are  $\|\mathbf{r}\|_2 = 0.028211$  and both methods perform well.

**3.6.** Let's start with the model  $u(t) = a_1 \sin(a_2 t + a_3) + a_4$ . This formulation is nonlinear, but we can work with it to make it linear. First, we know that the

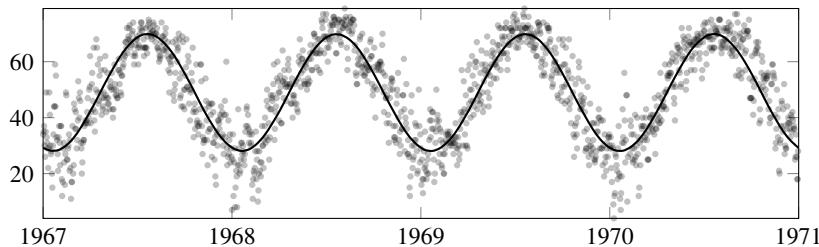


Figure A.3: Daily temperatures of Washington, DC between 1967 and 1971

period is one year. Second, we can write  $\sin(x + y)$  as  $\sin x \cos y + \sin y \cos x$ . Doing so, we have a linear model

$$u(t) = c_1 \sin(2\pi t) + c_2 \cos(2\pi t) + c_3,$$

where  $t$  is the time in units of one year,  $a_1 = \sqrt{c_1^2 + c_2^2}$ , and  $a_3 = \tan^{-1} c_2/c_1$ . We can solve this problem using:

```
using CSV, DataFrames, Dates
data = download(bucket*"dailytemps.csv")
df = DataFrame(CSV.File(data))
day = Dates.value.(df[:, :date] .- df[1, :date])/365
u = df[:, :temperature]
tempsmodel(t) = [sin.(2π*t) cos.(2π*t) one.(t)]
c = tempsmodel(day)\u
scatter(df[:, :date], u, alpha=0.3)
plot!(df[:, :date], tempsmodel(day)*c)
```

The solution is plotted in the figure above.

**3.6.** The following Julia code reads the MNIST training data and computes the economy SVD of the training set to get singular vectors:

```
using MLDatasets, Arpack, Images
image_train, label_train = MNIST.traindata()
image_train = reshape(permutedims(image_train,[3,2,1]),60000,:)
V = [(D = image_train[label_train.==i,:];
 svds(D,nsv=12)[1].V) for i ∈ 0:9];
```

Let's display the principal components of the “digit 3” subspace.

```
pix = -abs.(reshape(V[3+1],28,:))
rescale = scaleminmax(minimum(pix), maximum(pix))
pix .|> rescale .|> Gray
```

|                 |   | actual class |     |     |     |     |     |     |     |     |     |
|-----------------|---|--------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|                 |   | 0            | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
| predicted class | 0 | 984          | 8   | 0   | 1   | 6   | 0   | 1   | 0   | 0   | 0   |
|                 | 1 | 0            | 983 | 2   | 1   | 5   | 0   | 0   | 3   | 6   | 0   |
|                 | 2 | 12           | 3   | 945 | 10  | 3   | 0   | 7   | 7   | 12  | 1   |
|                 | 3 | 2            | 1   | 9   | 952 | 0   | 8   | 0   | 5   | 19  | 4   |
|                 | 4 | 3            | 8   | 5   | 0   | 944 | 1   | 3   | 7   | 4   | 25  |
|                 | 5 | 2            | 3   | 0   | 46  | 3   | 922 | 7   | 1   | 14  | 2   |
|                 | 6 | 5            | 3   | 0   | 0   | 1   | 5   | 986 | 0   | 0   | 0   |
|                 | 7 | 4            | 7   | 6   | 0   | 4   | 1   | 0   | 933 | 5   | 40  |
|                 | 8 | 6            | 22  | 6   | 16  | 5   | 13  | 5   | 6   | 910 | 11  |
|                 | 9 | 11           | 4   | 5   | 8   | 12  | 3   | 0   | 20  | 12  | 925 |

Figure A.4: Confusion matrix for PCA identification of digits in MNIST dataset.



The image of any 3 in the training data is largely a linear combinations of these basis vectors. We can now predict the best digit associated with each test image using these ten subspaces  $V$ . We'll finally build a confusion matrix to check the accuracy of the method. Each row of the confusion matrix represents the predicted class and each column represents the actual class. See Figure A.4. For example, element (7, 9) is 40 meaning that 40 of test images that were identified as a 7 were actually a 9. Overall, the method is about 95 percent accurate.

```
image_test, label_test = MNISTtestdata()
image_test = reshape(permutedims(image_test,[3,2,1]),10000,:)
r = [(q = image_test*V[i]*V[i]' .- image_test;
 sum(q.^2,dims=2)) for i in 1:10]
r = hcat(r...)
prediction = [argmin(r[i,:]) for i in 1:10000] .- 1
[sum(prediction[label_test== i].== j) for j in 0:9, i in 0:9]
```

3.8. Let  $\mathbf{A}$  be the actor-movie adjacency matrix and  $\mathbf{B}$  be the movie-genre adjacency matrix. We'll compute a truncated singular value decomposition  $\mathbf{U}_k \Sigma_k \mathbf{V}_k^T$  of  $\mathbf{BA}$  to get the projection to a lower-dimensional latent space. Let  $\{\mathbf{q}_i\}$  be the columns of  $\mathbf{V}_k^T$ . Each column represents the projection corresponding to an actor's genre signature in the latent space. We'll find the closest vectors  $\{\mathbf{q}_i\}$  to  $\mathbf{q}_j$ , i.e., we look for the column  $i$  that maximizes  $\cos \theta_i = \hat{\mathbf{q}}_j^T \mathbf{q}_i / \| \hat{\mathbf{q}}_j \| \| \mathbf{q}_i \|$ . Let's start by importing the data and building the adjacency matrices using the functions `get_names` and `get_adjacency_matrix` defined on page 467

```

actors = get_names("actors")
genres = get_names("genres")
A = get_adjacency_matrix("movie-genre")
A = (A*Diagonal(1 ./ sum(A,dims=1)[:,]))
B = get_adjacency_matrix("actor-movie");

```

We'll use the Arpack.jl package to compute the SVD for a sparse matrix, with a suitable small dimensional latent space:

```

using Arpack
X,_ = svds(A*B, nsv=10)
U = X.U; Σ = Diagonal(X.S); Vᵀ = X.Vt
Q = Vᵀ./sqrt.(sum(Vᵀ.^2, dims=1));

```

Now, let's find the actors most similar to Steve Martin:

```

index = x -> findfirst(actors.==x)[1]
q = Q[:,index("Steve Martin")][:,]
z = Q'*q
r = sortperm(-z)
actors[r[1:10]]

```

The ten actors most similar to Steve Martin are Steve Martin (of course), Lisa Kudrow, Rob Reiner, Christine Baranski, Patrick Cranshaw, George Gaynes, Jay Chandrasekhar, Alexandra Holden, Luke Wilson, and David Spade. Using the following code, we determine that the Steve Martin's signature is comedy (40%), drama (10%), romance (9%), crime (5%), romantic comedy (4%), . . . :

```

p = U*Σ*q
r = sortperm(-p)
[genres[r] p[r]/sum(p)]

```

Statistician George Box once quipped: “All models are wrong, some are useful.” How is this actor similarity model wrong? It doesn’t differentiate the type or size of roles an actor has in a movie, e.g., such as a character actor appearing in an serious movie for comic relief. Classification of movies into genres is overly simplistic and sometimes arbitrary. Perhaps using more keywords would add more fidelity. Actors change over their careers and the model. The information was crowdsourced through Wikipedia editors and contains errors. The genres may not be consistent across editors. The data set is limited to the years 1972–2012. How does the dimension of the latent space change the model? By choosing a full latent space, we keep all the information about the movies and actors. But we are unable to make generalizations and comparisons. Such a model is overfit. On the other hand, by choosing a very small dimension, the underrepresented genres are effectively removed and we are simply left with the major genres. Such a model is underfit.

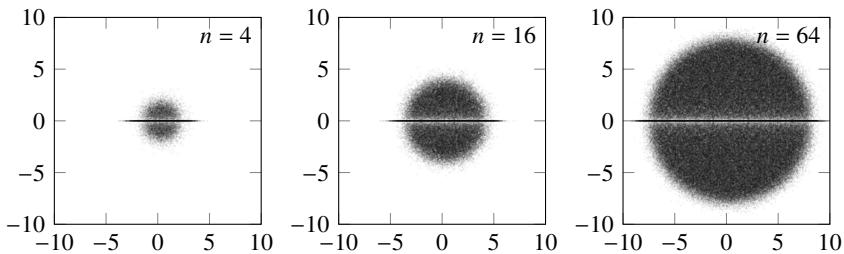


Figure A.5: Distribution of eigenvalues of normal random  $n \times n$  real matrices.

3.9. To simplify the calculations, we'll pick station one (although any station would do) as a reference station and measure all other stations relative to it  $(x_i, y_i, t_i) \leftarrow (x_i, y_i, t_i) - (x_1, y_1, t_1)$ . For this station, the reference circle is now  $x^2 + y^2 = (ct)^2$ . We can remove the quadratic terms from the other equations  $(x - x_i)^2 + (y - y_i)^2 = (ct - ct_i)^2$  by subtracting the reference equation from each of them to get a system of  $n - 1$  linear equations

$$2x_i x + 2y_i y - 2c^2 t_i t = x_i^2 + y_i^2 - c^2 t_i^2.$$

Geometrically, this new equation represents the *radical axis* of the two circles—the common chord if the circles intersect. The system in matrix form is  $\mathbf{Ax} = \mathbf{b}$ , which we can solve using either ordinary least squares or total least squares. We'll use the function `tls` defined on page 77.

```

xyt = [3 3 12; 1 15 14; 10 2 13; 12 15 14; 0 11 12]
reference = xyt[1,:]; xyt .-= reference'
A = [2 2 -2].*xyt; b = (xyt.^2)*[1; 1; -1]
x0ls = A\b + reference
xtls = tls(A,b) + reference

```

The solution is about  $(6.53, 8.80, 7.20)$ . Both methods are within a percent of one another.

4.1. Let's plot the eigenvalues of five thousand random  $n \times n$  real matrices:

```

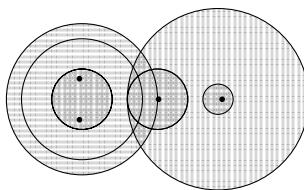
E = collect(Iterators.flatten([eigvals(randn(n,n)) for i∈1:2000]))
scatter(E, mc=:black, ma=0.05, legend=nothing, aspect_ratio=:equal)

```

The distribution is shown in the figure above for  $n = 4, 16$ , and  $64$  follows Girko's circular law, which says that as  $n \rightarrow \infty$ , eigenvalues are uniformly distributed in a disk of radius  $\sqrt{n}$ .

4.3. The Gershgorin circles are plotted below.

$$\mathbf{A} = \begin{bmatrix} 9 & 0 & 0 & 1 \\ 2 & 5 & 0 & 0 \\ 1 & 2 & 0 & 1 \\ 3 & 0 & -2 & 0 \end{bmatrix}$$



Row circles are shaded  $\blacksquare$ , column circles are shaded  $\blacksquare$ , and the eigenvalues  $\blacksquare$  lie in their intersections.

4.4. The following Julia code finds an eigenpair using Rayleigh iteration starting with a random initial vector. By running it repeatedly for different initial guesses, we can find all four eigenpairs. The method converges rapidly and  $\mathbf{A} - \rho\mathbf{I}$  becomes badly conditioned as  $\rho$  approaches one of the eigenvalues. So we break out of the iteration before committing the cardinal sin of using an ill-conditioned operator.

```
function rayleigh(A)
 x = randn(size(A,1),1)
 while true
 x = x/norm(x)
 ρ = (x'*A*x)[1]
 M = A - ρ*I
 abs(cond(M, 1)) < 1e12 ? x = M\x : return(ρ,x)
 end
end
```

4.5. The implicit QR method is summarized as

1. Compute the Hessenberg matrix  $\mathbf{H}$  that is unitarily similar to  $\mathbf{A}$ .
2. One QR-cycle is
  - a) Take  $\rho = h_{nn}$  (Rayleigh shifting)
  - b) Compute  $\mathbf{Q}\mathbf{H}\mathbf{Q}^T$  for Givens rotation  $\mathbf{Q} : \begin{bmatrix} h_{11} - \rho \\ h_{21} \end{bmatrix} \rightarrow \begin{bmatrix} * \\ 0 \end{bmatrix}$ .
  - c) “Chase the bulge” using Givens rotations  $\mathbf{Q} : \begin{bmatrix} h_{i+1,i} \\ h_{i+2,i} \end{bmatrix} \rightarrow \begin{bmatrix} * \\ 0 \end{bmatrix}$ .
  - d) Deflate using the top-left  $(n-1) \times (n-1)$  matrix if  $|h_{n,n-1}| < \varepsilon$ .

The LinearAlgebra.jl function `hessenberg(A)` returns an upper Hessenberg matrix that is unitarily similar to  $\mathbf{A}$ . The function `givens(a,b,1,2)` returns the Givens rotation matrix for the vector  $(a, b)$ . The following code implements the implicit QR method:

```

function implicitqr(A)
n = size(A,1)
tolerance = 1E-12
H = Matrix(hessenberg(A).H)
while true
 if abs(H[n,n-1])<tolerance
 if (n-1)<2; return(diag(H)); end
 end
 Q = givens([H[1,1]-H[n,n];H[2,1]],1,2)[1]
 H[1:2,1:n] = Q*H[1:2,1:n]
 H[1:n,1:2] = H[1:n,1:2]*Q'
 for i = 2:n-1
 Q = givens([H[i,i-1];H[i+1,i-1]],1,2)[1]
 H[i:i+1,1:n] = Q*H[i:i+1,1:n]
 H[1:n,i:i+1] = H[1:n,i:i+1]*Q'
 end
end
end

```

**4.6.** Let's compute the eigenvector centrality of the actor-actor adjacency matrix. We'll start by importing the data and building the adjacency matrix using the functions `get_names` and `get_adjacency_matrix` defined on page 467.

```

actors = get_names("actors")
B = get_adjacency_matrix("actor-movie")
M = (B'*B .!= 0) - I
v = ones(size(M,1))
for k = 1:8
 v = M*v; v /= norm(v);
end
r = sortperm(-v); actors[r][1:10]

```

Eigenvector centrality tells us those actors who appeared in movies with a lot of other actors, who themselves appeared in a lot of movies with other actors. The ten most central actors are Samuel L. Jackson, M. Emmet Walsh, Gene Hackman, Bruce Willis, Christopher Walken, Robert De Niro, Steve Buscemi, Dan Hedaya, Whoopi Goldberg, and Frank Welker. Arguably, Samuel L. Jackson is the center of Hollywood.

```
findfirst(actors[r] .== "Kevin Bacon")
```

Kevin Bacon is the 92nd most connected actor in Hollywood. Let's also look at degree centrality, which ranks each node by its number of edges, i.e., the number of actors with whom an actor has appeared.

```
actors[-sum(M,dims=1)[:,] > sortperm][1:10]
```

The ten most central actors are Frank Welker, Samuel L. Jackson, M. Emmet Walsh, Whoopi Goldberg, Bruce Willis, Robert De Niro, Steve Buscemi, Gene Hackman, Christopher Walken, and Dan Aykroyd. The noticeable change is that voice actor Frank Welker moves from tenth to first position.

4.7. The following Julia code implements the randomized SVD algorithm:

```
function randomizedsvd(A,k)
 Z = rand(size(A,2),k);
 Q = Matrix(qr(A*Z).Q)
 for i in 1:3
 Q = Matrix(qr(A'*Q).Q)
 Q = Matrix(qr(A*Q).Q)
 end
 W = svd(Q'*A)
 return((Q*W.U,W.S,W.V))
end
```

Let's import a high resolution photograph of a fox into Julia as a grayscale image, converted it to an array:

```
using FileIO, Images
img = load(download(bucket*"red-fox.jpg"))
A = Float64.(Gray.(img))
U,S,V = randomizedsvd(A,10)
Gray.([A U*Diagonal(S)*V'])
```

Let's also compare the elapsed time for the randomized SVD, the sparse SVD in Arpack.jl, and the full SVD in LinearAlgebra.jl.

```
using Arpack
@time randomizedsvd(A,10)
@time svds(A, nsv=10)
@time svd(A);
```

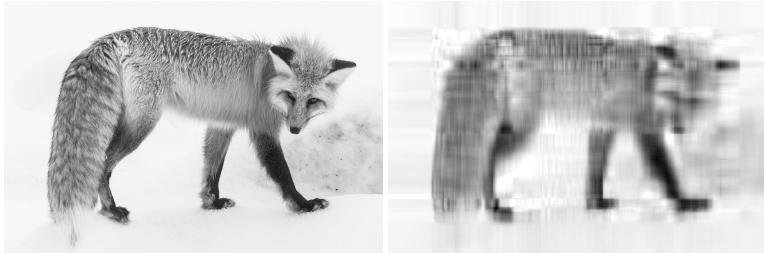
With rank  $k = 10$ , the elapsed time for the randomized SVD is about 0.05 seconds compared to 0.1 seconds for a sparse SVD and 3.4 seconds for a full SVD svd(A). We can compute the relative error using

$$\varepsilon_k = \frac{\sqrt{\sum_{i=k+1}^n \sigma_i^2}}{\sqrt{\sum_{i=1}^n \sigma_i^2}} = 1 - \frac{\sqrt{\sum_{i=1}^k \sigma_i^2}}{\|A\|_F}$$

which can be implemented as

```
ε = 1 .- sqrt.(cumsum(S.^2))/norm(A)
plot(ε,marker=:circle)
```

The relative error in the first ten singular values ranges from 2 percent down to 0.5 percent. The image on the left is the original image  $A$  and the image on the right is the projected image  $\text{Gray} \cdot (U * \text{Diagonal}(S) * V')$ .



4.8. We'll approximate  $A$  with a sufficiently high-dimensional matrix and compute its largest singular value.

```
using Arpack
m = 5000
A = [1/(1 + (i+j-1)*(i+j)/2 - j) for i ∈ 1:m, j ∈ 1:m]
svds(A,nsv=1)[1].S[1]
```

We find  $\|A\|_2 = 1.2742241528\dots$ , which agrees to within 11 digits of the answer given by the Sloane number A117233.

5.1. The Gauss–Seidel method converges if and only if the spectral radius of the matrix  $(L + D)^{-1}U$  is strictly less than one. The eigenvalues of

$$(L + D)^{-1}U = \begin{bmatrix} 1 & 0 \\ -\sigma & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 & \sigma \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \sigma \\ 0 & \sigma^2 \end{bmatrix}$$

are 0 and  $\sigma^2$ . So, the Gauss–Seidel method converges when  $|\sigma| < 1$ .

5.3. Let's start by setting up the problem and defining the exact solution  $u_e$ . To improve readability, we can define `kron` as an infix operator using the Unicode character  $\otimes$  within Julia.

```
using SparseArrays, LinearAlgebra
⊗(x,y) = kron(x,y); φ = x -> x-x^2
n = 50 ; x = (1:n)/(n+1); Δx = 1/(n+1)
J = sparse(I, n, n)
D = spdiagm(-1 => ones(n-1), 0 => -2ones(n), 1 => ones(n-1))
```

```
A = (D⊗J⊗J + J⊗D⊗J + J⊗J⊗D) / Δx^2
f = [ϕ(x)*ϕ(y) + ϕ(x)*ϕ(z) + ϕ(y)*ϕ(z) for x∈x, y∈x, z∈x][:,]
ue = [ϕ(x)*ϕ(y)*ϕ(z)/2 for x∈x, y∈x, z∈x][:,];
```

In practice, we could simply use routines from `IterativeSolvers.jl`, but because we want to evaluate the error at intermediate steps, we'll write our own routines. We define a function that implements the stationary methods—Jacobi ( $\omega = 0$ ), Gauss–Seidel ( $\omega = 1$ ), and SSOR ( $\omega = 1.9$ )—and returns the error.

```
function stationary(A,b,ω=0,n=400)
 ε = []; u = zero.(b)
 P = (1-ω)*sparse(Diagonal(A)) + ω*sparse(LowerTriangular(A))
 for i=1:n
 u += P\ (b-A*u)
 append!(ε, norm(u - ue, 1))
 end
 return(ε)
end
```

And, we define a function for the conjugate gradient method.

```
function conjugategradient(A,b,n=400)
 ε = []; u = zero.(b)
 p = r = b-A*u
 for i=1:n
 Ap = A*p
 α = (r'*p)/(Ap'*p)
 u += α.*p; r -= α.*Ap
 β = (r'*Ap)/(Ap'*p)
 p = r - β.*p
 append!(ε, norm(u - ue, 1))
 end
 return(ε)
end
```

Finally, we run each method and plot the errors.

```
ε = zeros(400,4)
@time ε[:,1] = stationary(A,-f,0)
@time ε[:,2] = stationary(A,-f,1)
@time ε[:,3] = stationary(A,-f,1.9)
@time ε[:,4] = conjugategradient(A,-f)
method = ["Jacobi" "Gauss-Seidel" "SOR" "Conjugate Gradient"]
plot(ε,yaxis=:log,labels=method,bglegend=RGBA(1,1,1,0.7))
```

Each method takes approximately two seconds to complete 400 iterations. The errors are plotted in the figure on the next page. Notice that the conjugate

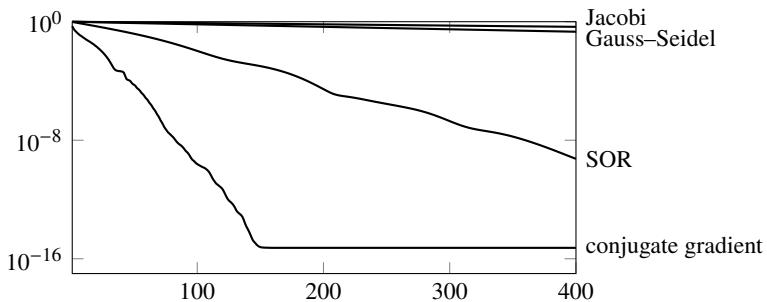


Figure A.6: Relative error as a function of the number of iterations.

gradient converges quite quickly relative to the stationary methods—the error is at machine precision after about 150 iterations, or about 0.7 seconds. For comparison, using the conjugate gradient method `cg` from `IterativeSolvers.jl` takes roughly 0.4 seconds. We can compute the convergence rate by taking the slopes of the lines

```
k = 1:120; ([one.(k) k]\log10.(ε[k,:]))[2,:]
```

The slopes are 0.008, 0.0016, 0.019 and 0.1. The inverse of these slopes tells us the number of iterations required to gain one digit of accuracy—roughly 1200 using a Jacobi method and 10 iterations using the conjugate gradient method. Convergence of the Jacobi method is quite slow, requiring around 18 thousand iterations, or 90 seconds, to reach machine precision. Still, it is faster than the direct solver `A\(-f)`, which takes 130 seconds.

5.4. The  $(1, 1)$ -entry of  $\mathbf{A}^{-1}$  is the first element of the solution  $\mathbf{x}$  to  $\mathbf{Ax} = \xi$  for  $\xi = (1, 0, 0, \dots, 0)$ . We'll use a preconditioned conjugate gradient method to solve the problem.

```
using Primes, LinearAlgebra, SparseArrays, IterativeSolvers
n = 20000
d = 2 .^ (0:14); d = [-d;d]
P = Diagonal(primes(224737))
B = [ones(n - abs(d)) for d ∈ d]
A = sparse(P) + spdiagm(n , n, (d .=> B)...)
b = zeros(n); b[1] = 1
cg(A, b; Pl=P)[1]
```

The computed solution  $0.7250783\dots$  agrees to with the answer provided by the Sloane number A117237 to machine precision. Using the `BenchmarkTools.jl` function `@btime`, we find that the conjugate gradient method takes 11 milliseconds

on a typical laptop. If we don't specify the preconditioner, the solver takes 1500 times longer.

6.1. Let  $m = n/3$ . Then a DFT can be written as a sum of three DFTs

$$y_j = \sum_{k=0}^{n-1} \omega_n^{kj} c_k = \sum_{k=0}^{m-1} \omega_m^{kj} c'_k + \omega_n^j \sum_{k=0}^{m-1} \omega_m^{kj} c''_k + \omega_n^{2j} \sum_{k=0}^{m-1} \omega_m^{kj} c'''_k$$

with  $c'_k = c_{3k}$ ,  $c''_k = c_{3k+1}$ , and  $c'''_k = c_{3k+2}$ . So,  $y_j = y'_j + \omega_n^j y''_j + \omega_n^{2j} y'''_j$ . By computing over  $j = 0, \dots, m-1$  and noting

$$\begin{aligned} \omega_m^{k(m+j)} &= \omega_m^{km} \omega_m^{kj} = \omega_m^{kj}, & \omega_n^{m+j} &= \omega_n^m \omega_n^j = \omega_3 \omega_n^j, \\ \omega_m^{k(2m+j)} &= \omega_m^{2km} \omega_m^{kj} = \omega_m^{kj}, & \omega_n^{2m+j} &= \omega_n^{2m} \omega_n^j = \omega_3^2 \omega_n^j, \end{aligned}$$

where  $\omega_3 = e^{-2i\pi/3}$ ,  $\omega_3^2 = e^{2i\pi/3}$  and  $\omega_3^4 = e^{-2i\pi/3}$ , we have the system

$$\begin{aligned} y_j &= y'_j + \omega_n^j y''_{m+j} + \omega_n^{2j} y'''_{2m+j} \\ y_{m+j} &= y'_j + \omega_3 \omega_n^j y''_{m+j} + \omega_3^2 \omega_n^{2j} y'''_{2m+j} \\ y_{2m+j} &= y'_j + \omega_3^2 \omega_n^j y''_{m+j} + \omega_3^4 \omega_n^{2j} y'''_{2m+j}. \end{aligned}$$

Note that when  $n = 3$ , this system is simply  $\mathbf{F}_3$ . In matrix notation

$$\mathbf{F}_n \mathbf{c} = (\mathbf{F}_3 \otimes \mathbf{I}_{n/3}) \boldsymbol{\Omega}_n (\mathbf{I}_3 \otimes \mathbf{F}_{n/3}) \mathbf{P} \mathbf{c}$$

were  $\boldsymbol{\Omega} = \text{diag}(1, \omega_n, \omega_n^2, \dots, \omega_n^{n-1})$ .

```
function fftx3(c)
 n = length(c)
 ω = exp(-2im*pi/n)
 if mod(n,3) == 0
 k = collect(0:n/3-1)
 u = [transpose(fftx3(c[1:3:n-2]));
 transpose((ω.^k).*fftx3(c[2:3:n-1]));
 transpose((ω.^2k).*fftx3(c[3:3:n]))]
 F = exp(-2im*pi/3).^([0;1;2]*[0;1;2]')
 return(reshape(transpose(F*u), :, 1))
 else
 F = ω.^ (collect(0:n-1)*collect(0:n-1)')
 return(F*c)
 end
end
```

6.2. The following function takes two integers as strings, converts them to padded arrays, computes a convolution, and then carries the digits:

```
using FFTW
function multiply(p_,q_)
 p = [parse(Int,split(reverse(p_),""));zeros(length(q_),1)]
 q = [parse(Int,split(reverse(q_),""));zeros(length(p_),1)]
 pq = Int.(round.(real.(ifft(fft(p).*fft(q)))))

 carry = pq .÷ 10
 while any(carry.>0)
 pq -= carry*10 - [0;carry[1:end-1]]
 carry = pq .÷ 10
 end
 n = findlast(x->x>0, pq)
 return(reverse(join(pq[1:n[1]])))
end
```

Arbitrary-precision arithmetic, which also uses the Schönhage–Strassen algorithm, is significantly faster:

```
using Random
p = randstring('0':'9', 100000)
q = randstring('0':'9', 100000)
@time multiply(p,q)
@time parse(BigInt, p)*parse(BigInt, q);
```

6.3. We'll split the DCT into the sum of two series, one with even indices running forwards  $\{0, 2, 4, \dots\}$  and the other with odd indices running backwards  $\{\dots, 5, 3, 1\}$ . Doing so makes the DFT naturally pop out after we express the cosine as the real part of the complex exponential. If we were computing a DST instead of a DCT, we would subtract the second series because sine is an odd function at the boundaries.

$$\begin{aligned}\hat{f}_\xi &= \sum_{k=0}^{n-1} f_k \cos\left(\frac{(k + \frac{1}{2})\xi\pi}{n}\right) \\ &= \sum_{k=0}^{n/2-1} f_{2k} \cos\left(\frac{(2k + \frac{1}{2})\xi\pi}{n}\right) + \sum_{k=n/2}^{n-1} f_{2(n-k)-1} \cos\left(\frac{(2n - 2k - \frac{1}{2})\xi\pi}{n}\right) \\ &= \operatorname{Re} \left( e^{-i\xi\pi/2n} \sum_{k=0}^{n/2-1} f_{2k} e^{-i2k\xi\pi/n} + e^{-i\xi\pi/2n} \sum_{k=n/2}^{n-1} f_{2(n-k)-1} e^{-i2k\xi\pi/n} \right)\end{aligned}$$

Because cosine is an even function, it doesn't matter which sign we choose as long as we are consistent between the two series. We'll choose negative so

that the DCT is in terms of a DFT rather than an inverse DFT. We then have  $DCT(f_k) = \text{Re}(e^{-i\xi\pi/2n} \cdot DFT(f_{P(k)}))$  where  $P(k)$  is a permutation of the index  $k$ . For example,  $P(\{0, 1, 2, 3, 4, 5, 6, 7, 8\})$  is  $\{0, 7, 2, 5, 4, 3, 6, 1, 8\}$ . We can build an inverse DCT by running the steps backwards.

While `fft` and `ifft` are both one-dimensional functions in Matlab and Python, they are multi-dimensional in Julia. So, first we'll need to define one-dimensional functions:

```
fft1(f) = hcat([fft(f[:,i]) for i in 1:size(f,2)]...)
ifft1(f) = hcat([ifft(f[:,i]) for i in 1:size(f,2)]...)
```

Now, we can define the functions `dctII` and `idctII`:

```
function dctII(f)
 n = size(f,1)
 ω = exp.(-0.5im*π*(0:n-1)/n)
 return(real(ω.*fft1(f[[1:2:n; n-mod(n,2):-2:2],:])))
end
```

```
function idctII(f)
 n = size(f,1)
 ω = exp.(-0.5im*π*(0:n-1)/n)
 f[1,:] = f[1,:]/2
 f[[1:2:n; n-mod(n,2):-2:2],:] = 2*real(ifft1(f./ω))
 return(f)
end
```

We can verify our code by computing the DCT directly:

```
y = [f[1:n]'*cos.(π/n*k*(0.5.+([0:n-1]...))) for k ∈ 0:n-1]
```

Alternatively, we can use the `FFTW.jl` function `dct`, which includes the additional scaling:

```
y *=sqrt(2/n); y[1] /=sqrt(2)
```

The two-dimensional DCT or IDCT simply applies the one-dimensional DCT or IDCT in each direction:

```
dct2(f) = dctII(dctII(f'))
idct2(f) = idctII(idctII(f'))
```

Let's compress an image by zeroing out the higher-frequency terms:

```
using FileIO, FFTW, Images
img = load(download(bucket*"red-fox.jpg"))
f = dct2(Float64.(Gray.(img)))
```

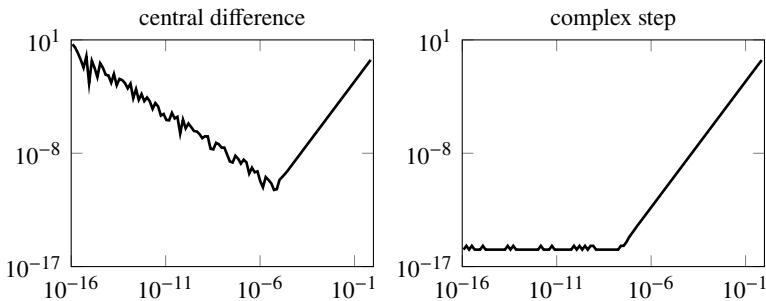


Figure A.7: The total error as a function of the stepsize  $h$  in the central difference and complex step approximations of  $f'(x)$ .

```
f[51:end,:] = 0; f[:,51:end] = 0
f = idct2(f)
[Gray.(img) Gray.(f)]
```

## A.2 Numerical methods for analysis

7.1. The truncation error of the central difference approximation is bounded by  $\varepsilon_{\text{trunc}} = \frac{1}{3}h^2m$  where  $m = \sup_{\xi} |f''(\xi)|$ . The value of  $m$  is approximately one for  $f(x) = e^x$ . The round-off error is bounded by  $\varepsilon_{\text{round}} = 2 \text{eps}/(2h)$ . The total error  $\varepsilon(h) = \varepsilon_{\text{trunc}} + \varepsilon_{\text{round}}$  is minimized when  $\varepsilon'(h) = 0$ , i.e., when

$$\varepsilon'(h) = \frac{2}{3}mh - 2 \text{eps}/h^2 = 0.$$

So, the error is minimum at  $h = (3 \text{eps}/m)^{1/3}$ . For  $f(x) = e^x$  at  $x = 0$ , the error reaches a minimum of about  $\text{eps}^{2/3} \approx 10^{-11}$  when  $h = (3 \text{eps})^{1/3} \approx 8 \times 10^{-6}$ .

By taking the Taylor series expansion of  $f(x + ih)$ , we have

$$f(x + ih) = f(x) + ih f'(x) - \frac{1}{2}h^2 f''(x) - \frac{1}{2}ih^3 f'''(\xi)$$

where  $|\xi - x| < h$ . From the imaginary part,  $f'(x) = \text{Im } f(x + ih)/h + \varepsilon_{\text{trunc}}$  with truncation error  $\varepsilon_{\text{trunc}} \approx \frac{1}{2}h^2|f'''(x)|$ . The round-off error  $\varepsilon_{\text{round}} \approx \text{eps}|f(x)|$ . For  $f(x) = e^x$ , the total error  $\varepsilon(h) = \frac{1}{2}h^2 + \text{eps}$ , which is smallest when  $h < \sqrt{2 \text{eps}} \approx 10^{-8}$ . See Figure A.7 above.

7.2. Let's write the  $7/3$ ,  $4/3$ , and  $1$  in binary format. For double-precision floating-point numbers, we keep 52 significant bits not including the leading 1 and round off the trailing bits. For  $7/3$  the trailing bits  $0101\dots$  round up to  $1000\dots$ , and for  $4/3$  the trailing bits  $0010\dots$  round down to  $0000\dots$ . Therefore, we have

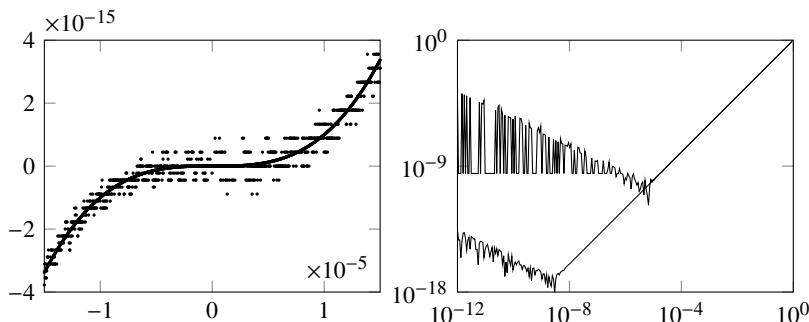


Figure A.8: Left:  $(x - 1)^3$  (solid line) and  $x^3 - 3x^2 + 3x - 1$  (dots). The  $x$ -axis is offset from  $x = 1$ . Right: The total error in the central difference approximation of  $f'(x)$  as a function of stepsize  $h$ .

Because binary  $10 - 1 = 1$ , the double-precision representation of  $7/3 - 4/3 - 1$  equals

which is machine epsilon.

7.3. See Figure A.8 above. The round-off error dominates the total error when  $h < 10^{-5}$  in  $x^3 - 3x^2 + 3x - 1$  and when  $h < 10^{-9}$  in  $(x - 1)^3$

8.4. To speed up convergence for multiple roots, we can try taking a larger step

$$x^{(k+1)} = x^{(k)} - m \frac{f(x^{(k)})}{f'(x^{(k)})}$$

for some  $m > 1$ . If  $x^*$  is a root with multiplicity  $n$ , then  $f^{(j)}(x^*) = 0$  for  $j = 0, 1, \dots, n-1$  and  $f^{(n)}(x^*) \neq 0$ . The error computed using (8.2) is

$$e^{(k+1)} = \frac{\frac{n-m}{n!} f^{(n)}(x^*) (e^{(k)})^n + \frac{n+1-m}{(n+1)!} f^{(n+1)}(x^*) (e^{(k)})^{n+1} + o((e^{(k)})^{n+1})}{\frac{1}{(n-1)!} f^{(n)}(x^*) (e^{(k)})^{n-1} + o((e^{(k)})^{n-1})}.$$

By taking  $m = n$ , we once again get quadratic convergence.

$$e^{(k+1)} \approx \frac{f^{(n)}(x^*)}{n(n+1)} (e^{(k)})^2.$$

### 8.3. When $p = 1$ ,

$$p \frac{(1/f)^{(p-1)}}{(1/f)^{(p)}} = \frac{(1/f)}{(1/f)'} = \frac{1/f}{-f'/f^2} = -\frac{f}{f'}$$

which yields Newton's method  $x^{(k+1)} = x^{(k)} - f(x^{(k)})/f'(x^{(k)})$ . Take  $p = 2$  and let  $f(x) = x^2 - a$ . Then

$$2 \frac{(1/f)'}{(1/f)''} = 2 \frac{\frac{2x}{(x^2-a)^2}}{\frac{2(3x^2+a)}{(x^2-a)^3}} = 2 \frac{-x(x^2-a)}{3x^2+a}$$

from which

$$x^{(k+1)} = \frac{x^{(k)}((x^{(k)})^2 + 3a)}{3(x^{(k)})^2 + a}.$$

Starting with  $x^{(0)} = 1$ , we get

where correct number of decimals (1, 2, 7, 19, and 62, respectfully) roughly triples with each iteration.

8.7. We want to find the  $p$  such that the error  $|e^{(k+1)}| \approx C|e^{(k)}|^p$  for some positive  $C$ . If we express the secant method

$$x^{(k+1)} = x^{(k)} - f(x^{(k)}) \frac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})}$$

in terms of the error  $e^{(k)} = x^{(k)} - x^*$ , we have

$$e^{(k+1)} = e^{(k)} - \frac{f(x^* + e^{(k)}) - f(x^* + e^{(k-1)})}{f'(x^* + e^{(k)}) - f'(x^* + e^{(k-1)})}.$$

Substituting the Taylor expansion

$$f(x^* + e^{(k)}) = f'(x^*)e^{(k)} + \frac{1}{2}f''(x^*)(e^{(k)})^2 + o(e^{(k)})$$

into this expression gives us

$$\begin{aligned} e^{(k+1)} &= e^{(k)} - \frac{e^{(k)} f'(x^*) (1 + M e^{(k)}) (e^{(k)} - e^{(k-1)})}{f'(x^*) (e^{(k)} - e^{(k-1)}) (1 + M (e^{(k)} + e^{(k-1)}))} + o((e^{(k)})^2) \\ &= e^{(k)} - \frac{e^{(k)} (1 + M e^{(k)})}{1 + M (e^{(k)} + e^{(k-1)})} + o((e^{(k)})^2) \\ &= \frac{M e^{(k)} e^{(k-1)}}{(1 + M (e^{(k)} + e^{(k-1)}))} + o((e^{(k)})^2) \approx M e^{(k)} e^{(k-1)} \end{aligned}$$

where  $M = f''(x^*)/2f'(x^*)$ . For Newton's method  $e^{(k+1)} \approx M e^{(k)} e^{(k)}$ . By substituting  $|e^{(k+1)}| \approx C |e^{(k)}|^p$  and  $|e^{(k)}| \approx C |e^{(k-1)}|^p$  into the expression  $|e^{(k+1)}| \approx |M| |e^{(k)}| |e^{(k-1)}|$ , we have  $C |e^{(k)}|^p \approx |M/C| |e^{(k)}| |e^{(k)}|^{1/p}$ , from which it follows that  $C^2 = |M|$  and  $p = 1 + p^{-1}$ . Therefore  $p = (\sqrt{5} + 1)/2 \approx 1.618$ .

8.9. For the fixed-point method to converge, we need  $|\phi'(x)| < 1$  in a neighborhood of the fixed point  $x^*$ . If  $\phi(x) = x - \alpha f(x)$ , then  $|\phi'(x)| = |1 - \alpha f'(x)| < 1$  when  $0 < \alpha f'(x) < 2$ . Taking  $\alpha < 2/f'(x^*)$  should ensure convergence. The asymptotic convergence factor is given by  $|\phi'(x)|$ , so it's best to choose  $\alpha$  close to  $1/f'(x^*)$ . The solution  $x^*$  is unknown, so we instead could use a known  $f'(x^{(k)})$  at each iteration, which is Newton's method  $x^{(k+1)} = x^{(k)} - f(x^{(k)})/f'(x^{(k)})$ .

8.11. Define the homotopy as  $h(t, \mathbf{x}) = f(\mathbf{x}) + (t - 1)f(\mathbf{x}_0)$  where  $\mathbf{x} = (x, y)$ . The Jacobian matrix is

$$\frac{\partial h}{\partial \mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} 4x(x^2 + y^2 - 1) & 4y(x^2 + y^2 + 1) \\ 6x(x^2 + y^2 - 1)^2 - 2xy^3 & 6y(x^2 + y^2 - 1)^2 - 3x^2y^2 \end{bmatrix}.$$

We need to solve the ODE

$$\frac{d}{dt} \mathbf{x}(t) = - \left( \frac{\partial f}{\partial \mathbf{x}} \right)^{-1} f(\mathbf{x}_0)$$

with initial conditions  $(x(0), y(0)) = \mathbf{x}_0$ . Newton's method, which takes

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left( \frac{\partial f}{\partial \mathbf{x}^{(k)}} \right)^{-1} f(\mathbf{x}^{(k)}),$$

is quite similar to the homotopy continuation. We'll define the two solvers:

```
using DifferentialEquations
function homotopy(f,df,x)
 dxdt(x,p,t) = -df(x)\p
```

```

sol = solve(ODEProblem(dxdt,x,(0.0,1.0),f(x)))
sol.u[end]
end

```

```

function newton(f,df,x)
for i in 1:100
 Δx = -df(x)\f(x)
 norm(Δx) > 1e-8 ? x += Δx : return(x)
end
end

```

The routines take a function  $f$ , its Jacobian matrix  $df$ , and an initial guess  $x$ , and they return one of the zeroes. For our problem, we'll define  $f$  and  $df$  as

```

f = z -> ((x,y)=tuple(z...));
[(x^2+y^2)^2-2(x^2-y^2); (x^2+y^2-1)^3-x^2*y^3]
df = z -> ((x,y)=tuple(z...));
[4x*(x^2+y^2-1) 4y*(x^2+y^2+1);
 6x*(x^2+y^2-1)^2-2x*y^3 6y*(x^2+y^2-1)^2-3x^2*y^2]

```

The solutions are

```

display(homotopy(f,df,[1,1]))
display(newton(f,df,[1,1]))

```

*8.12.* Let's implement the ECDH algorithm for  $y^2 = x^3 + 7 \pmod{r}$ . We'll first define the group operator  $\oplus$ . Given points  $P = (x_0, y_0)$  and  $Q = (x_1, y_1)$ , the new point is  $P \oplus Q$  is  $(x, y)$  with

$$\begin{aligned} x &= \lambda^2 - x_0 - x_1 \\ y &= -\lambda(x - x_0) - y_0, \end{aligned}$$

where  $\lambda = (3x_0^2 + a)/2y_0$  when  $P = Q$  and  $\lambda = (y_1 - y_0)/(x_1 - x_0)$  otherwise.

```

function ⊕(P,Q)
 a = 0
 r = BigInt(2)^256 - 2^32 - 977
 if P[1]==Q[1]
 d = invmod(2*P[2], r)
 λ = mod((3*powermod(P[1],2,r)+a)*d, r)
 else
 d = invmod(Q[1]-P[1], r)
 λ = mod((Q[2]-P[2])*d, r)
 end
end

```

```

x = mod(powermod(λ, 2, r) - P[1] - Q[1], r)
y = mod(-λ*(x-P[1])-P[2], r)
[x;y]
end

```

We'll use the double-and-add method to compute the product  $mp$ . The double-and-add method is the Horner form applied to a polynomial ring. Consider the a polynomial

$$x^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0.$$

Note that by taking  $x = 2$  and  $a_i \in \{0, 1\}$ , we have a binary representation of a number  $m = 1a_{n-1}a_{n-2} \dots a_2a_1a_0$ . We can write the polynomial in Horner form

$$x(x(\dots(x + a_{n-1}) + \cdots + a_2) + a_1) + a_0.$$

Similarly, the representation of the number  $m$  in Horner form is

$$m = 2(2(2 \cdots 2(2 + a_{n-1}) + \cdots + a_2) + a_1) + a_0.$$

Furthermore, the product of the numbers  $m$  and  $p$  is

$$mp = 2(2(2 \cdots 2(2p + a_{n-1}p) + \cdots + a_2p) + a_1p) + a_0p.$$

Formally substituting the group operator  $\oplus$  and the point  $P$  into this expression gives us

$$mP = 2(2(2 \cdots 2(2P \oplus a_{n-1}P) \oplus \cdots \oplus a_2P) \oplus a_1P) \oplus a_0P.$$

We can evaluate this expression from the inside out using an iterative function or from the outside in using a recursive one. Let's do the later. This allows us to right shift through  $m = 1a_{n-1}a_{n-2} \dots a_2a_1a_0$  and inspect the least significant bit with each function call.

```

Base.isodd(m::BigInt) = ((m&1)==1)
function dbl_add(m,P)
 if m > 1
 Q = dbl_add(m>>1,P)
 return isodd(m) ? (Q⊕Q)⊕P : Q⊕Q
 else
 return P
 end
end

```

Alice chooses a private key  $m$  (say 1829442) and sends Bob her public key  $mp$ . Similarly, Bob chooses a private key  $n$  (say 3727472) and sends Alice his public key  $nP$ . Now, both can generate the same cipher using a shared secret key  $nmp$ .

```

P1 = big"0x79BE667EF9DCBBAC55A06295CE87
 0B07029BFCDB2DCE28D959F2815B16F81798"
P2 = big"0x483ADA7726A3C4655DA4FBFC0E11
 08A8FD17B448A68554199C47D08FFB10D4B8"
P = [P1; P2]
m, n = 1829442, 3727472
mP = dbl_add(m, P)
nmP = dbl_add(n, mP)

```

8.13. The derivative of  $f(x) = \frac{1}{2}mx^2$  is  $f'(x) = mx$ . The gradient descent method applied to this function is

$$x^{(k+1)} = x^{(k)} - \alpha f'(x^{(k)}) = (1 - \alpha m)x^{(k)}.$$

The convergence factor for this fixed-point method is  $|1 - \alpha m|$ . The method converges if and only if the learning rate  $0 < \alpha < 2/m$ . Convergence is monotonic when the learning rate is less than  $1/m$  and zigzagging when it is greater than  $1/m$ . Convergence is fastest when the learning rate is close to  $1/m$ , the reciprocal of the second-derivative of  $f(x)$ —i.e., the inverse of the Hessian.

The momentum method

$$x^{(k+1)} = x^{(k)} + \alpha p^{(k)} \quad \text{with} \quad p^{(k)} = -f'(x^{(k)}) + \beta p^{(k-1)}$$

can be rewritten to explicitly remove  $p^{(k)}$  and  $p^{(k-1)}$  as

$$x^{(k+1)} = x^{(k)} - \alpha f'(x^{(k)}) + \beta(x^{(k)} - x^{(k-1)}).$$

Hence, we have the system

$$\begin{bmatrix} x^{(k+1)} \\ x^{(k)} \end{bmatrix} = \begin{bmatrix} 1 - \alpha m + \beta & -\beta \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x^{(k)} \\ x^{(k-1)} \end{bmatrix}.$$

The fixed-point system  $\mathbf{x}^{(k+1)} = \mathbf{Ax}^{(k)}$  converges if the spectral radius of  $\mathbf{A}$  is less than one. The eigenvalues of the system are

$$\lambda_{\pm} = \frac{1}{2}(1 - \alpha m + \beta) \pm \frac{1}{2}\sqrt{(1 - \alpha m + \beta)^2 - 4\beta}.$$

By taking  $\alpha m = 2 + 2\beta$ , we have

$$\lambda_{\pm} = \frac{1}{2}(-1 - \beta) \pm \frac{1}{2}(1 - \beta) = \{-\beta, 1\}.$$

So the method converges if  $0 < \alpha < (2 + 2\beta)/m$  and  $0 < \beta < 1$ . Furthermore, note that if  $(1 - \alpha m + \beta)^2 < 4\beta$ , then the discriminant is negative and the eigenvalues are complex. From this,

$$|\lambda_{\pm}|^2 = \frac{1}{4}(1 - \alpha m + \beta) - \frac{1}{4}(1 - \alpha m + \beta)^2 + \beta = \beta.$$

It follows that if  $(1 - \sqrt{\beta})^2/m < \alpha < (1 + \sqrt{\beta})^2/m$ , then the spectral radius  $\rho(\mathbf{A}) < \sqrt{\beta}$ . To dig deeper, see Gabriel Goh's interactive article "Why Momentum Really Works."

**9.1.** We first show that the Vandermonde matrix is non-singular if the nodes are distinct. Let  $\mathbf{V}$  be the Vandermonde matrix using nodes  $x_0, x_1, \dots, x_n$ . Consider the relation

$$\det(\mathbf{V}) = \prod_{j=0}^n \prod_{\substack{i \neq j \\ j \neq k}} (x_i - x_j) \quad (\text{A.1})$$

The relation clearly holds for  $n = 1$ . Assume that (A.1) is true for  $n$  nodes, we will show that it also holds for  $n + 1$  nodes by using cofactor expansion with respect to the last column. Let  $\mathbf{V}[x_k]$  be the Vandermonde matrix formed using all the nodes  $\{x_0, x_1, \dots, x_n\}$  except for  $x_k$ . Then

$$\det(\mathbf{V}[x_k]) = \prod_{\substack{j=0 \\ j \neq k}}^n \prod_{\substack{i \neq j \\ i \neq k}} (x_i - x_j) \quad (\text{A.2})$$

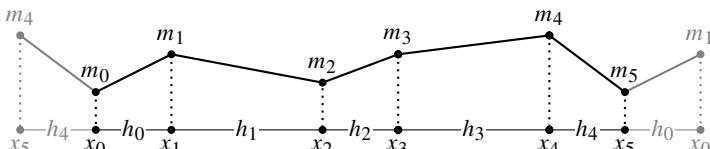
By cofactor expansion  $\det(\mathbf{V})$  equals

$$x_0^n \det(\mathbf{V}[x_0]) + \dots + (-1)^k x_k^n \det(\mathbf{V}[x_k]) + \dots + (-1)^n x_n^n \det(\mathbf{V}[x_n])$$

By (A.2),

$$\det(\mathbf{V}) = \sum_{k=0}^n (-1)^k x_k^n \prod_{\substack{j=0 \\ j \neq k}}^n \prod_{\substack{i \neq j \\ i \neq k}} (x_i - x_j).$$

**9.2.** To help visualize periodic boundary conditions we can imagine repeating the leftmost and rightmost knots onto the right and left sides of our domain:



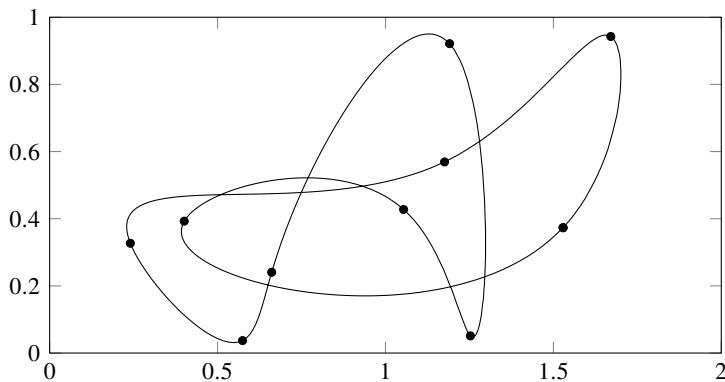


Figure A.9: A closed parametric spline passing through 10 knots.

The following code modifies `spline_natural` on page 234 to determine the coefficients  $\{m_0, m_1, \dots, m_{n-1}\}$  of a spline with periodic boundary conditions. It is assumed that  $y_0 = y_n$ .

```
function spline_periodic(x,y)
 h = diff(x)
 d = 6*diff(diff([y[end-1];y])./[h[end];h])
 alpha = h[1:end-1]
 beta = 2*(h+circshift(h,1))
 C = Matrix(SymTridiagonal(beta,alpha))
 C[1,end]=h[end]; C[end,1]=h[end]
 m = C\d
 return([m;m[1]])
end
```

Now, we can compute a parametric spline interpolant with  $nx \times n$  points through a set of  $n$  random points using the function `evaluate_spline` defined on page 235. One solution is shown in Figure A.9 above.

```
n = 5; nx = 30
x = rand(n); y = rand(n)
x = [x;x[1]]; y = [y;y[1]]
t = [0;cumsum(sqrt.(diff(x).^2 + diff(y).^2))]
X = evaluate_spline(t,x,spline_periodic(t,x),nx*n)
Y = evaluate_spline(t,y,spline_periodic(t,y),nx*n)
scatter(x,y); plot!(X[2],Y[2],legend=false)
```

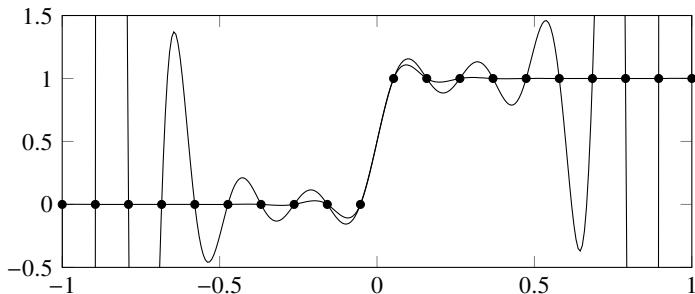


Figure A.10: Interpolation using polynomial and  $|x|^3$  radial basis function .

9.3. First, let's set up the domain  $x$  and the function  $y$  which we are interpolating.

```
n = 20; N = 200
x = LinRange(-1,1,n)
y = float(x .> 0);
```

Next, define the radial basis functions and the polynomials bases.

```
phi1(x,a) = abs.(x.-a).^3
phi2(x,a) = exp.(-20(x.-a).^2)
phi3(x,a) = x.^a
```

Finally, we construct and plot the interpolating functions.

```
X = LinRange(-1,1,N)
interp(phi,a) = phi(X,a')*(phi(x,a')\y)
Y1 = interp(phi1,x)
Y2 = interp(phi2,x)
Y3 = interp(phi3,(0:n-1))
scatter(x,y,seriestype = :scatter, marker = :dot, legend = :none)
plot!(X,[Y1,Y2,Y3],ylim=[-0.5,1.5])
```

As can be seen in Figure A.10 above, the polynomial interpolant suffers from the Runge phenomenon with a maximum error of about 500. The radial basis function on the other hand gives us a good approximation. It should be noted that the radial basis function with  $\phi(x) = |x|^3$  simply produces a cubic spline because the resulting function is cubic on the subintervals and has continuous second derivatives at the nodes.

9.4. Let  $v_j(x) = B(h^{-1}x - j)$  be B-splines with nodes equally spaced at  $x_i = ih$ :

$$B(x) = \begin{cases} \frac{2}{3} - \frac{1}{2}(2 - |x|)x^2, & |x| \in [0, 1) \\ \frac{1}{6}(2 - |x|)^3, & |x| \in [1, 2) \\ 0, & \text{otherwise.} \end{cases}$$

Every node in the domain needs to be covered by three B-splines, so we'll need to have an additional B-spline centered just outside either boundary to cover each of the boundary nodes. Take the approximation  $\sum_{i=-1}^{n+1} c_i v_i(x)$  for  $u(x)$ . Bessel's equation  $xu'' + u' + xu = 0$  at the collocation points  $\{x_i\}$  is now

$$\sum_{j=0}^n \left( x_i v_j''(x_i) + v_j'(x_i) + x_i v_j(x_i) \right) c_j = 0$$

where

|                    | $v_j''(x_i)$ | $v_j'(x_i)$             | $v_j(x_i)$    |
|--------------------|--------------|-------------------------|---------------|
| when $j = i$       | $-2h^{-2}$   | 0                       | $\frac{2}{3}$ |
| when $j = i \pm 1$ | $h^{-2}$     | $\mp \frac{1}{2}h^{-1}$ | $\frac{1}{6}$ |
| otherwise          | 0            | 0                       | 0             |

This gives us the system  $\mathbf{Ac} = \mathbf{d}$  where

$$a_{i,i} = -2x_i h^{-2} + \frac{2}{3}x_i \quad \text{and} \quad a_{i,i\pm 1} = x_i h^{-2} \mp \frac{1}{2}h^{-1} + \frac{1}{6}$$

and  $a_{ij} = 0$  otherwise, and where  $d_i = 0$  for  $i = 0, 1, \dots, n$ . At the endpoints, we have the boundary conditions

$$\frac{1}{6}c_{-1} + \frac{2}{3}c_0 + \frac{1}{6}c_0 = u_a = 1 \quad \text{and} \quad \frac{1}{6}c_{n-1} + \frac{2}{3}c_n + \frac{1}{6}c_{n+1} = u_b = 0,$$

from which

$$a_{0,-1} = \frac{1}{6}, \quad a_{0,0} = \frac{2}{3}, \quad a_{0,1} = \frac{1}{6}, \quad \text{and} \quad a_{n,n-1} = \frac{1}{6}, \quad a_{n,n} = \frac{2}{3}, \quad a_{0,n+1} = \frac{1}{6}$$

and  $d_0 = 1$ .

Once we have the coefficients  $c_j$ , we construct the solution  $\sum_{j=-1}^{n+1} c_j v_j(x)$ . The Julia code is given below. We start by defining a general collocation solver for  $L u(x) = f(x)$  that takes an array  $x$  for the equally spaced collocation points and boundary conditions  $bc$ . The solver returns an array of coefficients  $c$  for each node, including two elements for the two B-splines just outside the domain.

```
function collocation_solve(L,f,bc,x)
 h = x[2]-x[1]
 S = L(x)*([1 -1/2 1/6; -2 0 2/3; 1 1/2 1/6]./[h^2 h 1])'
 d = [bc[1]; f(x); bc[2]]
```

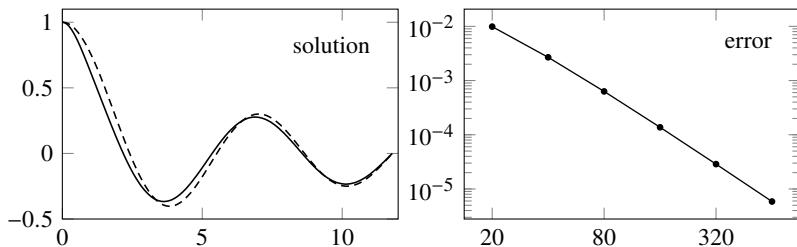


Figure A.11: Numerical solution (solid) and exact solution (dashed) to Bessel's equation for 15 collocation points (left). The log-log plot of the error as a function of number of points (right).

```

A = Matrix(Tridiagonal([S[:,1];0], [0;S[:,2];0], [0;S[:,3]]))
A[1,1:3], A[end,end-2:end] = [1 4 1]/6, [1 4 1]/6
return(A\d)
end

```

We could have kept matrix  $A$  as a Tridiagonal matrix by first using the second row to zero out  $A[1,3]$  and the second to last row to zero out  $A[\text{end},\text{end}-2]$ . Next, we define a function that will interpolate between collocation points:

```

function collocation_build(c,x,N)
 X = LinRange(x[1],x[end],N)
 h = x[2] - x[1]
 i = Int32.(X .÷ h .+ 1); i[N] = i[N-1]
 C = [c[i] c[i.+1] c[i.+2] c[i.+3]]'
 B = (x->[(1-x).^3;4-3*(2-x).*x.^2;4-3*(1+x).*x.^2;x.^3]/6)
 Y = sum(C.*hcat(B.((X.-x[i])/h)...),dims=1)
 return(Array(X),reshape(Y,(:,1)))
end

```

Now, we can solve Bessel's equation.

```

using Roots, SpecialFunctions
n = 20; N = 141
L = (x -> [x one.(x) x])
f = (x -> zero.(x))
b = fzero(besselj0, 11)
x = range(0,b,length=n)
c = collocation_solve(L,f,[1,0],x)
X,Y = collocation_build(c,x,70)
plot(X,[Y besselj0.(X)])

```

The solution is plotted in Figure A.11 above using 15 collocation points. Even with only 15 points, the numerical solution is fairly accurate. Finally, let's

measure the convergence rate as we increase the number of collocation points. A method is order  $p$  if the error is  $\varepsilon = O(n^{-p})$ , where  $n$  is the number of points. This means that  $\log \varepsilon \approx -p \log n + \log m$  for some  $m$ . We'll plot the error as a function of the number of collocation points.

```
N = 10*2 .^(1:7); ε = []
for n in N
 x = LinRange(0,b,n)
 c = collocation_solve(L,f,[1,0],x)
 X,Y = collocation_build(c,x,n)
 append!(ε, norm(Y-besselj0.(X)) / n)
end
plot(N, ε, xaxis=:log, yaxis=:log, marker=:o)
```

The log-log slope will give us the order of convergence. The Julia code

```
([log.(N) one.(N)]\log.(ε))[1]
```

returns approximately  $-2.18$ , confirming that the collocation method is second-order accurate.

**9.5.** The Bernstein polynomial representation of the Bézier curve is

$$\begin{bmatrix} x \\ y \end{bmatrix} = (1-t)^3 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 3t(1-t)^2 \begin{bmatrix} 1 \\ c \end{bmatrix} + 3t^2(1-t) \begin{bmatrix} c \\ 1 \end{bmatrix} + t^3 \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

We want to find the parameter  $c$  that minimizes the maximum deviation from the circle  $R(t) = x^2 + y^2 - 1$ . To do this, we'll find the  $c$  for which the magnitude of a local maximum at  $t = t_c$  equals the magnitude of the local minimum at  $t = \frac{1}{2}$ . Before leaping into the problem, there are a few things we can do to make it less messy. Scaling and shifting  $t \mapsto \frac{1}{2}(t+1)$  will make  $R$  a symmetric function over  $(-1, 1)$ . Because  $R$  is now a symmetric function (with only even powers of  $t$ ), we can make a second change of variables  $t \mapsto \sqrt{t}$  to reduce the function from a sextic polynomial to a cubic polynomial. To find the location  $t_c$ , we need to solve  $R'(t) = 0$ . The function  $R$  has another critical point at  $t = 1$ , so we can factor  $(t-1)$  out of the derivative, leaving us with a linear equation. (The critical point at  $t = 0$  naturally falls out because we are differentiating with respect to  $\sqrt{t}$ .) We now just need to solve  $R(t_c) = -R(0)$ , which happens to be a sextic equation.

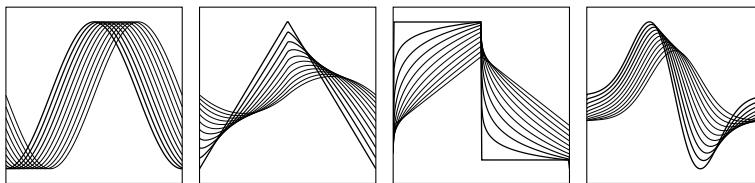
We'll use Wolfram Cloud (<https://www.wolframcloud.com>) to implement the solution. Wolfram Cloud has a free basic plan that provides complete access to Mathematica with a few limitations, perfect for a problem like this one.

```
x = (1-t)^3 + 3t*(1-t)^2 + 3c*t^2*(1-t);
y = 3c*t*(1-t)^2 + 3t^2*(1-t) + t^3;
R = Collect[(x^2 + y^2 - 1) /. {t->(t+1)/2} /. {t->Sqrt[t]}, t];
R1 = Simplify[R /. Solve[D[R,t]/(t-1) == 0, t]];
```

```
R2 = Simplify[R /. t -> 0];
FindRoot[R1==R2, {c,0.5}]
Sqrt[(R1 + 1) /. %] - 1
```

We find that  $c \approx 0.551915$  is the optimal solution, resulting in less than 0.02 percent maximum deviation from the unit circle—not too bad!

10.3. The  $p$ th derivative of the sine, piecewise-quadratic, piecewise-linear, and Gaussian functions are plotted below for several values of  $p \in [0, 1]$ :



Note that the  $p$ th derivative for  $\sin x$  is  $\sin(x+p\pi/2)$ , which is simply a translation of the original function. A Julia solution is as follows

```
using FFTW
n = 256; ℓ = 2
x = (0:n-1)/n*ℓ .- ℓ/2
ξ = [0:(n/2-1); (-n/2):-1]*(2π/ℓ)
f₁ = exp.(-16*x.^2)
f₂ = sin.(π*x)
f₃ = x.* (1 .- abs.(x))
deriv(f,p) = real(ifft((im*ξ).^p.*fft(f)))
```

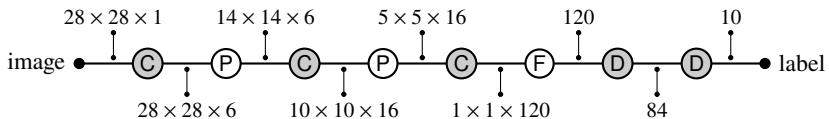
We can use `Interact.jl` to create an interactive widget.

```
using Interact
func = Dict("Gaussian"=>f₁, "sine"=>f₂, "spline"=>f₃)
@manipulate for f in togglebuttons(func; label="function"),
 p in slider(0:0.01:2; value=0, label="derivative")
 plot(x, deriv(f,p), legend=:none)
end
```

Alternatively, we could use the `Plots.jl @animate` macro to produce a gif. See the QR link at the bottom of this page.

10.4. The LeNet-5 model consists of several hidden layers—three sets of convolutional layers combined by pooling layers, which are then flattened and followed by two fully-connected layers. The diagram on the next page shows the LeNet-5 architecture:





In this diagram, (C) is a trainable convolutional layer, (P) is a pooling layer, (F) is a flattening layer, and (D) is a trainable fully-connected layer. The  $28 \times 28$  pixel greyscale images are initially padded with two pixels all around the boundary before the first  $5 \times 5$  convolution layer, creating a set of six  $28 \times 28$  feature maps. Each of these feature maps are then subsampled to  $14 \times 14$  maps by an average pooling layer ( $2 \times 2$  subarrays are averaged to  $1 \times 1$  subarrays). These maps are fed, without padding, through a second convolutional layer with sixteen  $5 \times 5 \times 6$  filters. The resultant  $10 \times 10 \times 16$  feature map is again halved in size to  $5 \times 5 \times 16$ . The last (unpadded) convolutional layer consists on 120 filters, producing a  $1 \times 1 \times 120$  feature map, which is subsequently flattened to 120 values. These values are fed through two fully-connected layers, one with 84 neurons and one with 10 neurons, to an output layer with each of the ten labels.

Let's use the Flux.jl package to develop and train the model. We'll start by defining the model architecture:

```

using MLDatasets, Flux, Flux.Data
model = Chain(
 Conv((5,5), 1=>6, tanh, pad=SamePad()),
 MeanPool((2,2)),
 Conv((5,5), 6=>16, tanh),
 MeanPool((2,2)),
 Conv((5,5), 16=>120, tanh),
 Flux.flatten,
 Dense(120, 84, tanh),
 Dense(84, 10))

```

Let's load the MNIST training and test data as single-precision floating-point numbers. Using single-precision numbers can significantly speed performance over double-precision numbers, because the memory usage is halved. We'll convert each  $28 \times 28$ -pixel image into a  $28 \times 28 \times 1$ -element array. We'll also convert the each of the labels into one hot arrays—arrays whose elements equal one in positions matching the respective labels and equal zero otherwise.

```

image_train, label_train = MNIST.traindata(Float32)
image_test, label_test = MNISTtestdata(Float32)
image_train = Flux.unsqueeze(image_train, 3)
image_test = Flux.unsqueeze(image_test, 3)
label_train = Flux.onehotbatch(label_train, 0:9)
label_test = Flux.onehotbatch(label_test, 0:9)

```

The categorical cross entropy, or softmax loss,

$$L(\mathbf{y}) = -\log \frac{e^{y_i}}{\sum_{i=0}^9 e^{y_i}}$$

is used for multi-label classification. Large datasets can consume significant memory. The Flux.Data.DataLoader function breaks up the dataset and handles iteration over mini-batches of data to reduce memory consumption. Diederik Kingma and Jimmy Ba introduced the Adam optimizer in 2014—well after LeNet’s 1998 debut—so its use in LeNet is anachronistic. The Adam optimizer improves the stochastic gradient descent available at the time.

```
loss(x,y) = Flux.Losses.logitcrossentropy(model(x),y)
parameters = Flux.params(model)
data = DataLoader((image_train, label_train), batchsize=100)
optimizer = ADAM()
```

Now, we can train the model. Let’s loop over 5 epochs. It might take several minutes to train—adding the ProgressMeter.jl macro `@showprogress` before the for loop will display a progress bar.

```
using ProgressMeter
@showprogress for epochs = 1:5
 Flux.train!(loss, parameters, data, optimizer)
end
```

We use the test data to see how well the model performed. The `onecold` function is the inverse of the `onehot` function, returning to position of the largest element.

```
accuracy(x,y) = sum(Flux.onecold(x) .== Flux.onecold(y))/size(y,2)
accuracy(model(image_test),label_test)
```

The LeNet-5 model achieves about 2 percent error rate—better than the 5 percent error rate obtained by using principal component analysis in exercise 3.6. The intermediate layer inputs are shown in the figure on the facing page. Notice how the first convolutional layer detects edges of the image and the second convolutional layer uses this edge detection to identify gross structure in the image.

*11.1.* The third-order approximation to  $f''(x)$  is of the form

$$h^{-1} (c_{10}f(x) + c_{11}f(x+h) + c_{12}f(x+2h) + c_{13}f(x+3h)) + m_1 h^3 f^{(4)}(\xi)$$

where  $\xi$  is some point in the interval  $[x, x+3h]$ . To find the coefficients  $c_{10}$ ,  $c_{11}$ ,  $c_{12}$ ,  $c_{13}$ , and  $m_1$ , we define  $d = [0, 1, 2, 3]$  from nodes at  $x, x+h, x+2h$

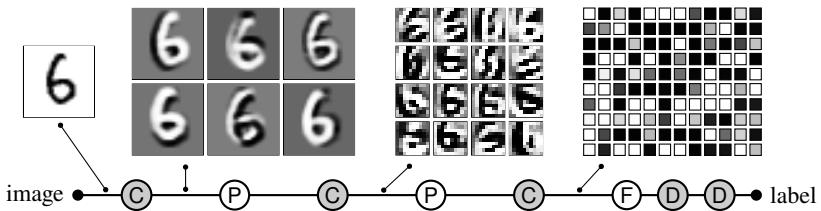


Figure A.12: An input image and the intermediate transformations of that image following the three convolutional matrices of LeNet-5.

and  $x + 3h$  and invert the scaled Vandermonde matrix  $C_{ij} = [d_i^j / i!]$  $^{-1}$ . The coefficients of the truncation error are given by  $C \cdot d.^n / \text{factorial}(n)$ :

```
d = [0,1,2,3]; n = length(d)
C = inv(d.^{0:n-1}' .// factorial.(0:n-1)')
[C C*d.^n/factorial(n)]
```

$$\begin{matrix} 1 & 0 & 0 & 0 & 0 \\ -11/6 & 3 & -3/2 & 1/3 & 1/4 \\ 2 & -5 & 4 & -1 & -11/12 \\ -1 & 3 & -3 & 1 & 3/2 \end{matrix}$$

From the second row, we have

$$f'(x) \approx \frac{1}{h} \left( -\frac{11}{6} f(x) + 3f(x+h) - \frac{3}{2}f(x+2h) + \frac{1}{3}f(x+3h) \right)$$

with a truncation error  $\frac{1}{4}f^{(4)}(\xi)h^3$  for some  $\xi \in [x, x+3h]$ . The value  $|f^{(4)}(\xi)|$  is bounded by 1 for  $f(x) = \sin x$ .

The round-off error is bounded by  $h^{-1}(|-\frac{11}{6}| + |3| + |-\frac{3}{2}| + |\frac{1}{3}|)\text{eps}$  or approximately  $7h^{-1}\text{eps}$  where  $\text{eps}$  is machine epsilon. Truncation error is a decreasing function as  $h$  gets smaller and round-off error is an increasing error as  $h$  gets smaller. Following the discussion on page 185, the total error is minimum when the two are equal, i.e., when  $h = (28\text{eps})^{1/4} \approx 3 \times 10^{-4}$ .

From the third row we have

$$f''(x) \approx \frac{1}{h^2} (2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h))$$

with a truncation error  $\frac{11}{12}h^2f^{(4)}(\xi)$ .

11.2. In practice we don't need to save every intermediate terms. Instead by taking  $i = m$ ,  $j = m - n$ , and  $\bar{D}_j = D_{m,n}$ , we have the update

$$\bar{D}_j \leftarrow \frac{\bar{D}_{j+1} - \delta^{p(i-j)} \bar{D}_j}{1 - \delta^{p(i-j)}} \quad \text{where } j = i - 1, \dots, 1 \quad \text{and} \quad \bar{D}_i \leftarrow \phi(\delta^i h).$$

The Julia code for Richardson extrapolation taking  $\delta = \frac{1}{2}$  is

```
function richardson(f,x,m)
 D = []
 for i in 1:m
 append!(D, φ(f,x,2^i))
 for j in i-1:-1:1
 D[j] = (4^(i-j)*D[j+1] - D[j])/(4^(i-j) - 1)
 end
 end
 D[1]
end
```

This reformulated implementation is about 20 times faster when  $n = 8$  and about 100 times faster when  $n = 12$ .

11.3. We'll extend the dual class on page 297 by adding methods for the square root, division, and cosine.

```
Base.:sqrt(u) = Dual(sqrt(value(u)), deriv(u) / 2sqrt(value(u)))
Base.:(u, v) = Dual(value(u)/value(v),
 (value(u)*deriv(v)-value(v)*deriv(u))/(value(v)*value(v)))
Base.:cos(u) = Dual(cos(value(u)), -sin(value(u))*deriv(u))
```

We can define a function that implements Newton's method.

```
function get_zero(f,x)
 ε = 1e-12; δ = 1
 while abs(δ) > ε
 fx = f(Dual(x))
 δ = value(fx)/deriv(fx)
 x -= δ
 end
 return(x)
end
```

The call `get_zero(x->4sin(x)+sqrt(x), 4)` returns  $3.6386\dots$  as expected. To find a minimum or maximum, we replace two lines in Newton's method to get

```
function get_extremum(f,x)
 ε = 1e-12; δ = 1
```

```

while abs(δ)>ε
 fx = f(Dual(Dual(x)))
 δ = deriv(value(fx))/deriv(deriv(fx))
 x -= δ
end
return(x)
end

```

The call `get_extremum(x->4sin(x)+sqrt(x), 4)` of this modified function returns  $4.6544\dots$  as expected.

**11.4.** Let's take  $\gamma$  to be a circle centered at  $z = a$ . That is, take  $z = a + \varepsilon e^{i\theta}$  with  $dz = i\varepsilon e^{i\theta} d\theta$ . Then

$$f^{(p)}(a) = \frac{p!}{2\pi\varepsilon^p} \int_0^{2\pi} f(a + \varepsilon e^{i\theta}) e^{-ip\theta} d\theta.$$

The composite trapezoidal method applied to a smooth, periodic function has spectral convergence:

$$f^{(p)}(a) = \frac{p!}{n\varepsilon^p} \sum_{k=0}^{n-1} f(a + \varepsilon e^{2i\pi k/n}) e^{-2i\pi pk/n} + O((\varepsilon/n)^n)$$

We can implement the method as

```

function cauchyderivative(f, a, p, n = 20, ε = 0.1)
 ω = exp.(2π*im*(0:(n-1))/n)
 factorial(p)/(n*ε^p)*sum(@. f(a+ε*ω)/ω^p)
end

```

```

f = x -> exp(x)/(cos(x)^3 + sin(x)^3)
cauchyderivative(f, 0, 6)

```

Note that by taking  $n = 2$  and  $p = 1$  we have the usual central difference approximation of the derivative:

$$f'(a) = \frac{f(a + \varepsilon) - f(a - \varepsilon)}{2\varepsilon} + O(\varepsilon^2).$$

We don't have to start the contour integral  $\theta = 0$ . Let's start at  $\theta = \pi/2$ . Suppose that  $f(x)$  is a real function and again take  $n = 2$  and  $p = 1$ . Then

$$f'(a) = \frac{f(a + i\varepsilon) - f(a - i\varepsilon)}{2i\varepsilon} + O(\varepsilon^2) = \frac{\operatorname{Im} f(a + i\varepsilon)}{\varepsilon} + O(\varepsilon^2),$$

which is the complex-step derivative.

11.5. The following function computes the nodes and weights for Gauss-Legendre quadrature by using Newton's method to find the roots of  $P_n(x)$ :

```
function gauss_legendre(n)
 x = -cos((4*(1:n).-1)*π/(4n+2))
 Δ = one.(x)
 dPn = 0
 while(maximum(abs.(Δ))>1e-16)
 Pn, Pn-1 = x, one.(x)
 for k ∈ 2:n
 Pn, Pn-1 = ((2k - 1)*x.*Pn-(k-1)*Pn-1)/k, Pn
 end
 dPn = @. n*(x*Pn - Pn-1)/(x^2-1)
 Δ = Pn ./ dPn
 x -= Δ
 end
 return(x, @. 2/((1-x^2)*dPn^2))
end
```

11.7. We'll make a change of variables  $\xi = (x - s)/\sqrt{4t}$  and  $d\xi = -s/\sqrt{4t}$ . Then

$$u(t, x) = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} u_0(x - 2\xi\sqrt{t}) e^{-\xi^2} d\xi.$$

We can implement the solution in Julia as

```
using FastGaussQuadrature
ξ,w = gausshermite(40)
u₀ = x -> sin(x)
u = (t,x) -> w * u₀.(x.-2sqrt(t)*ξ)/sqrt(π)
x = LinRange(-12,12,100); plot(x,u.(1,x))
```

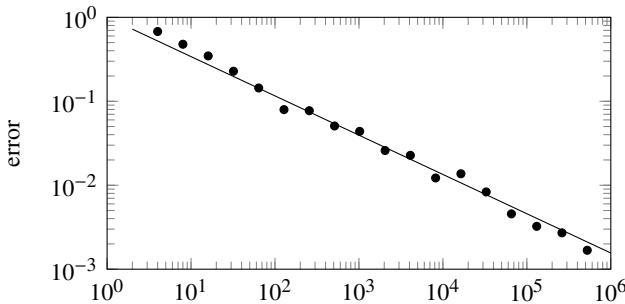
11.8. Using the function

```
mc_π(n) = sum(sum(rand(2,n).^2,dims=1).<1)/n*4
```

we compute  $\pi \approx 3.138$  using a sample size  $n = 10^4$ . To confirm the convergence rate for the Monte Carlo method we further sample over  $m = 20$  runs to smooth out the noise inherent in the Monte Carlo method.

```
m = 20; d = []; N = 2 .^ (1:20)
for n ∈ N
 append!(d,sum([abs(π - mc_π(n)) for i∈1:m])/m)
end
```

```
s = log.(N.^[0 1])\log.(d)
scatter(N,d,xaxis=:log, yaxis=:log)
plot!(N,exp(s[1]).*N.^s[2])
```



The slope of the log-log plot of error  $m = -0.468$  agrees with an  $O(N^{-1/2})$  convergence rate. We can compute the area of an eight-dimensional unit hypersphere by modifying our original code:

```
mc_pi(n,d=2) = sum(sum(rand(d,n).^2,dims=1).<1)/n*2^d
```

By taking  $n = 10^6$ , we get 4.088—close to the actual  $\pi^4 r^8 / 24 \approx 4.059$ .

### A.3 Numerical methods for differential equations

12.1. Note that when  $\theta = 1$ ,  $\theta = \frac{1}{2}$ , and  $\theta = 0$  the  $\theta$ -scheme is simply the forward Euler scheme, the trapezoidal scheme, and the backward Euler scheme, respectively. So, the regions of stability of the  $\theta$ -scheme will correspond with those three schemes when  $\theta$  is any of those three values. Take  $f(U^n) = \lambda U^n$  and take  $r = U^{n+1}/U^n$ . Then

$$\frac{r - 1}{k} = (1 - \theta)\lambda r + \theta\lambda.$$

We want to find the conditions on the complex-valued  $\lambda k$  plane such that  $|r| \leq 1$ . So, let's find the boundary of the region  $\lambda k$  where  $|r| = 1$ . Take  $\lambda k = x + iy$ . Then

$$\begin{aligned} 1 = |r|^2 = r\bar{r} &= \frac{1 + \theta(x + iy)}{1 - (1 - \theta)(x + iy)} \frac{1 + \theta(x - iy)}{1 - (1 - \theta)(x - iy)} \\ &= \frac{1 + 2\theta x + \theta^2(x^2 + y^2)}{1 - 2(1 - \theta)x + (1 - \theta)^2(x^2 + y^2)} \end{aligned}$$

from which  $1 - 2(1 - \theta)x + (1 - \theta)^2(x^2 + y^2) = 1 + 2\theta x + \theta^2(x^2 + y^2)$ . After rearranging and combining terms,  $(1 - 2\theta)x^2 - 2x + (1 - 2\theta)y^2 = 0$  or equivalently

$$x^2 - \frac{2}{1 - 2\theta}x + y^2 = 0.$$

Completing the square give us

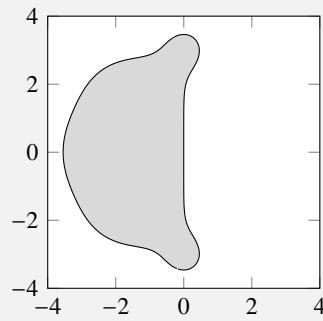
$$\left(x - \frac{1}{1 - 2\theta}\right)^2 + y^2 = \frac{1}{(1 - 2\theta)^2}.$$

So the region of absolute stability is bounded by a circle centered on the real axis at  $\lambda k = (1 - 2\theta)^{-1}$  with radius  $(1 - 2\theta)^{-1}$ , that is, tangential to the imaginary axis. Note that when  $\theta \rightarrow \frac{1}{2}$ , the boundary approaches the imaginary axis.

**12.4.** The following Julia code produces the accompanying plot:

```
A = [0 0 0 0 0;
 1/3 0 0 0 0;
 1/6 1/6 0 0 0;
 1/8 0 3/8 0 0;
 1/2 0 -3/2 2 0];
b = [1/6 0 0 2/3 1/6];

using LinearAlgebra, Plots
function rk_stability_plot(A,b)
 E = ones(length(b),1)
 r(λk) = abs.(1 .+ λk * b*((I - λk*A)\E))
 x,y = LinRange(-4,4,100),LinRange(-4,4,100)
 s = reshape(vcat(r.(x'.+im*y)...),(100,100))
 contour(x,y,s,levels=[1],legend=false)
end
rk_stability_plot(A,b)
```



**12.7.** To solve  $u' = f(u) + g(u)$  with a third-order L-stable IMEX method, we need to choose an L-stable implicit scheme (BDF3) and match it with an explicit scheme that uses the same time discretization. The BDF3 approximates the time derivative at  $U^n$  using  $U^n$ ,  $U^{n-1}$ ,  $U^{n-2}$ , and  $U^{n-3}$  and uses  $f(U^n)$  directly. For the explicit scheme, we'll need to approximate  $f(U^n)$  using  $f(U^{n-1})$ ,  $f(U^{n-2})$ ,  $f(U^{n-3})$ , and  $f(U^{n-4})$ . We are evaluating the equation at time  $t_n$  in both the implicit and explicit schemes to avoid splitting error. The BDF3 scheme is well-known. It can be found with a quick search in books and online, or we can also easily reformulate it using the `multistepcoeffs` algorithm on page 343 with an input `m = [1 2 3]` and `n = [0]`. We can also reformulate it by combining

terms of the Taylor series expansion

$$\begin{aligned} u(t) &= u \\ u(t-k) &= u - ku' + \frac{1}{2}k^2u'' - \frac{1}{6}k^3u''' + O(k^4) \\ u(t-2k) &= u - 2ku' + \frac{4}{2}k^2u'' - \frac{8}{6}k^3u''' + O(k^4) \\ u(t-3k) &= u - 3ku' + \frac{9}{2}k^2u'' - \frac{27}{6}k^3u''' + O(k^4) \\ u(t-4k) &= u - 4ku' + \frac{16}{2}k^2u'' - \frac{64}{6}k^3u''' + O(k^4) \end{aligned}$$

to keep the  $u'$  term while eliminating the  $u$ ,  $u''$ , and  $u'''$  terms. In this case, (taking the first four equations) we solve the system

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & -2 & -3 \\ 0 & 1/2 & 4/2 & 9/2 \\ 0 & -1/6 & -8/6 & -27/6 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

```
i = 0:3; c = ((-i)').^i ./ factorial.(i)) \ [0; 1; 0; 0]
```

to get the coefficients  $\left\{\frac{11}{6}, -3, \frac{3}{2}, -\frac{1}{3}\right\}$ . So BDF3 is

$$\frac{11}{6}U^{n+1} - 3U^n + \frac{3}{2}U^{n-1} - \frac{1}{3}U^{n-2} = kf(U^{n+1}).$$

To approximate  $f(U^{n+1})$  in the explicit scheme, we keep the  $u$  term and eliminate the  $u'$ ,  $u''$  and  $u'''$  terms in the Taylor series expansions. This time (taking the last four equations) we solve the system

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -2 & -3 & -4 \\ 1/2 & 4/2 & 9/2 & 16/2 \\ -1/6 & -8/6 & -27/6 & -64/6 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

```
i = 0:3; c = ((-(i+1)').^i ./ factorial.(i)) \ [1; 0; 0; 0]
```

to get the coefficients  $\{4, -6, 4, -1\}$  giving

$$\tilde{g}(U^n, U^{n-1}, U^{n-2}, U^{n-3}) = g\left(4U^n - 6U^{n-1} + 4U^{n-2} - U^{n-3}\right).$$

Then  $g(U^n) = \tilde{g}(U^n, U^{n-1}, U^{n-2}, U^{n-3}) + O(k^4)$ . Combining the implicit and explicit methods gives us

$$\frac{\frac{11}{6}U^{n+1} - 3U^n + \frac{3}{2}U^{n-1} - \frac{1}{3}U^{n-2}}{k} = f(U^{n+1}) + \tilde{g}(U^n, U^{n-1}, U^{n-2}, U^{n-3}).$$

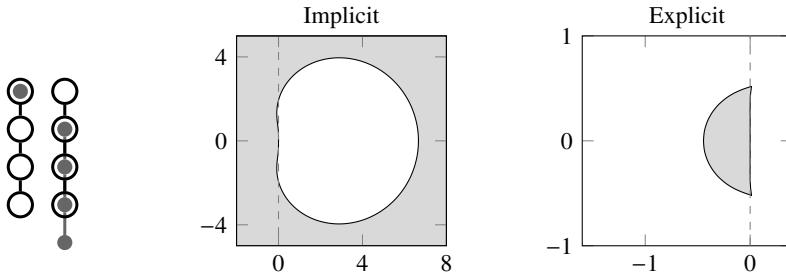


Figure A.13: The stencil and regions of absolute stability (in gray) for the implicit and explicit parts of the L-stable IMEX method.

The region of stability for the BDF3 is known from Figure 12.5. To compute the regions of stability for the explicit scheme we take  $g(u) = \lambda u$  and let  $r = U^{n+1}/U^n$  and find when  $|r| \leq 1$ . Then

$$\frac{11}{6}r^4 - 3r^3 + \frac{3}{2}r^2 - \frac{1}{3}r = \lambda k (4r^3 - 6r^2 + 4r - 1).$$

Finally, we express  $\lambda k(r)$  as a rational function and find the boundary of the domain  $|r| \leq 1$  by taking  $r = e^{i\theta}$ . Regions of stability are shown in Figure A.13 above.

12.8. Take  $f(u) = \lambda u$  and let

$$\alpha = \frac{1}{b_{-1}^*} \sum_{j=0}^{s-1} b_j U^{n-j} \quad \text{and} \quad \beta = \frac{1}{b_{-1}^*} \sum_{j=0}^{s-1} b_j^* U^{n-j}.$$

Then the Adams–Bashforth–Moulton PECE equation (12.7) can be written as

$$\tilde{U}^{n+1} = U^n + \alpha z \tag{A.3a}$$

$$U^{n+1} = U^n + (\tilde{U}^{n+1} + \beta)z. \tag{A.3b}$$

where  $z = \lambda k b_{-1}^*$ . Substitute (A.3a) into (A.3b):

$$U^{n+1} = U^n + (U^{n+1} + \beta)z + \alpha z^2.$$

For an addition corrector iteration, we substitute this new expression in for  $\tilde{U}^{n+1}$  in (A.3b) giving us

$$U^{n+1} = U^n + (U^n + \beta)(z + z^2) + \alpha z^3.$$

Continuing for additional corrector iterations:

$$U^{n+1} = U^n + (z + z^2 + \cdots + z^s)(U^n + \beta) + z^{s+1}\alpha.$$

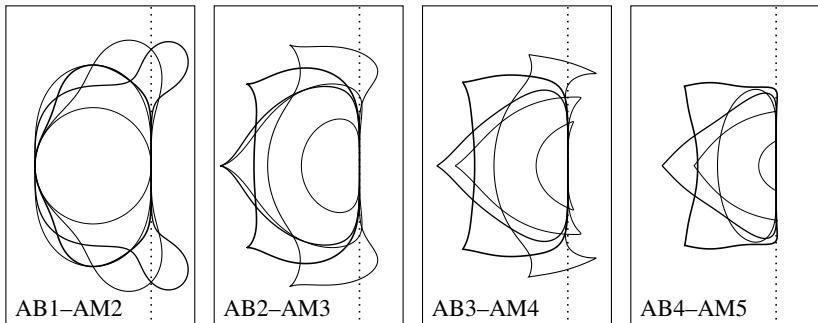


Figure A.14: Regions of stability for Adams–Bashforth–Moulton PE(CE) $^m$  methods for  $m = 0$  (thin curve) to  $m = 4$  (thick curve). The domain is  $[-2.5, .75] \times [-2.75, 2.75]$ .

To find the boundary of the region of absolute stability, let  $r = U^n/U^{n+1}$  and look for the solutions to the following equation when to  $|r| = 1$

$$1 = r + (z + z^2 + \cdots + z^s)(r + \beta(r)) + \alpha(r)z^{s+1} \quad (\text{A.4})$$

where  $\alpha(r) = (b_{-1}^*)^{-1} \sum_{j=0}^{s-1} b_j r^j$  and  $\beta(r) = (b_{-1}^*)^{-1} \sum_{j=0}^{s-1} b_j^* r^j$ . To do this we take  $r = \exp(i\theta)$  and solve (A.4) by finding the roots of the polynomial. We'll use the function `multistepcoeffs` from page 343 to compute the coefficients of the Adams–Bashforth and Adams–Moulton. The following function provides the orbit of points in the complex plane for an  $n$ th order Adams–Bashforth–Moulton PE(CE) $^m$ .

```
using Polynomials
function PECE(n,m)
 _,a = multistepcoeffs(1,hcat(1:n...))
 _,b = multistepcoeffs(1,hcat(0:n...))
 α(r) = a · r.^(1:n)/b[1]
 β(r) = b[2:end] · r.^(1:n)/b[1]
 z = [(c = [r-1; repeat([r + β(r)],m); α(r)];
 Polynomials.roots(Polynomial(c)))
 for r in exp.(im*LinRange(0,2π,200))]
 hcat(z/b[1]...)
end
```

The first-order ABM PE(CE) $^m$  is plotted using

```
z = vcat([PECE(1,i)[:] for i in 0:4]...)
scatter(z,label="",markersize=.5,aspect_ratio=:equal)
```

After splicing the solution together and snipping off unwanted sections of the curves, we get the regions in Figure A.14 on the preceding page.

12.9. Let  $T(x)$  be the Taylor series for some function. To compute the Padé approximation  $P_m(x)/Q_n(x) = T(x) + O(x^{m+n+1})$ , we'll find the coefficients of  $P_m(x) = Q_n(x)T(x) + O(x^{m+n+1})$ :

$$\sum_{i=0}^m p_i x^i = \left(1 + \sum_{i=1}^n q_i x^i\right) \left(\sum_{i=0}^{\infty} a_i x^i\right) + O(x^{m+n+1}). \quad (\text{A.5})$$

We need to solve the linear system obtained by collecting the powers of  $x$

$$\sum_{i=0}^m p_i x^i - \sum_{i=0}^m \sum_{j=1}^n a_i q_j x^{i+j} = \sum_{i=0}^{m+n} a_i x^i.$$

The Julia Polynomials.jl method `PolyCompat.PadeApproximation.Pade` will do just this for us, but it will be instructive to write our own function to solve (A.5).

```
function pade(a,m,n)
 A = Rational(1)*Matrix(I(m+n+1))
 A[:,m+2:end] = [i>j ? -a[i-j] : 0 for i∈1:m+n+1, j∈1:n]
 pq = A\a[1:m+n+1]
 pq[1:m+1], [1; pq[m+2:end]]
end
```

Let's compute the coefficients of  $P_3(x)$  and  $Q_2(x)$  of the Padé approximation of  $\log(x+1)$ . Its Taylor series is  $x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \frac{1}{4}x^4 + \frac{1}{5}x^5 - \dots$ .

```
m = 3; n = 2
a = [0; ((-1).^(0:m+n).//(1:m+n+1))]
(p,q) = pade(a,m,n)
```

Finally, we substitute  $r - 1$  in for  $x$  in the Padé approximant, regrouping by powers of  $r$ . This amounts to multiplying the two coefficient vectors by upper inverse Pascal matrices.

```
S = n -> [(-1).^(i+j)*binomial(j,i) for i∈0:n, j∈0:n]
S(m)*p, S(n)*q
```

12.13. The SIR system of equations can be easily solved using a standard explicit ODE solver. Solution to the SIR problem. While we can write the code that solves this problem quite succinctly, Julia also lets us write code that makes the mathematics quite clear. Note that `SIR!` is an in-place function.

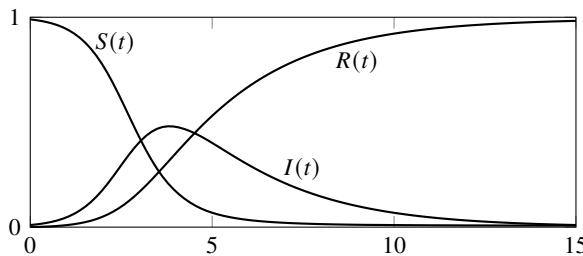


Figure A.15: Solution to the SIR model in problem 12.13.

```
function SIR!(du,u,p,t)
 S,I,R = u
 β,γ = p
 du[1] = dS = -β*S*I
 du[2] = dI = +β*S*I - γ*I
 du[3] = dR = +γ*I
end
```

Now, we set up the problem, solve it, and present the solution:

```
using DifferentialEquations
u₀ = [0.99; 0.01; 0]
tspan = (0.0,15.0)
p = (2.0,0.4)
problem = ODEProblem(SIR!,u₀,tspan,p)
solution = solve(problem)
plot(solution,labels=["susceptible" "infected" "recovered"])
```

The solution is shown in Figure A.15.

**12.14.** We can write the Duffing equation as the system  $x' = v$  and  $v' = -\gamma v - \alpha x - \beta x^3 + \delta \cos \omega t$ , which we can solve using a standard, high-order explicit ODE solver:

```
using DifferentialEquations, Plots
function duff!(dx,x,γ,t)
 dx[1] = x[2]
 dx[2] = -γ*x[2]+x[1]-x[1]^3+0.3*cos(t)
end
problem = ODEProblem(duff!,[1.0,0.0],(0.0,200.0),0.37)
solution = solve(problem,Vern7())
plot(solution,vars=(1,2))
```

See Figure A.16 on the next page.

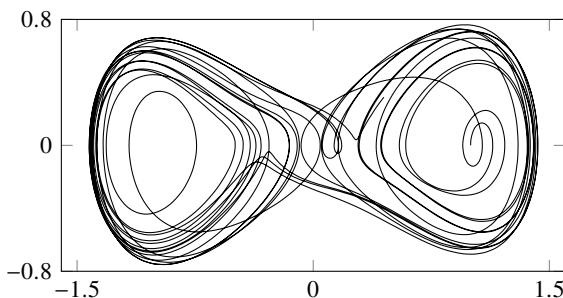


Figure A.16: The phase map of the Duffing equation in problem 12.14.

**12.15.** Implementing the shooting method is straightforward. We define our right-hand side function `airy` along with the domain endpoints `x` and the boundary conditions `bc`. Let's take a starting guess of  $y'(-12)$  as `guess=5`. The following code solves the initial value problem and returns the second boundary point:

```
function solveIVP(f,u0,tspan)
 sol = solve(ODEProblem(f,u0,tspan))
 return(sol.u[end][1])
end
```

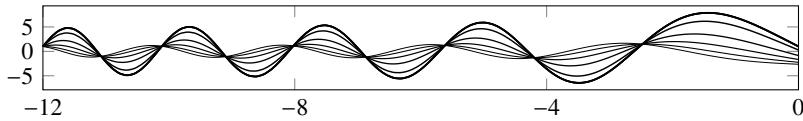
Now, we can solve the boundary value problem using the shooting method with boundary condition `bc` and an initial guess `guess`. Because `find_zero` takes only two arguments, we'll use an anonymous function `shoot_airy` to let us format and pass our additional parameters to `solveIVP`.

```
using DifferentialEquations, Roots
airy(y,p,x) = [y[2];x*y[1]]
domain = (-12.0,0.0); bc = (1.0,1.0); guess = 5
shoot_airy = (guess -> solveIVP(airy,[bc[1];guess],domain)-bc[2])
v = find_zero(shoot_airy, guess)
```

We find that initial condition `v` is approximately 16.149. Let's plot the solution:

```
sol = solve(ODEProblem(airy,[bc[1],v],domain))
plot(sol)
```

By plotting each intermediate solution in `solveIVP`, we can see the shooting method in action. In the following figure, the lines grow thicker with each iteration until we get to the solution—the thickest line.



In practice, the `BoundaryValueDiffEq.jl` package provides a simple interface for solving the boundary problems, using either the shooting method or a collocation method.

### 13.2. Take the Taylor series approximation

$$\begin{aligned} U_j^n &: u(t, x) = u \\ U_j^{n+1} &: u(t + k, x) = u + ku_t + \frac{1}{2}k^2u_{tt} + O(k^3) \\ U_{j+1}^n &: u(t, x + k) = u + hu_x + \frac{1}{2}h^2u_{xx} + \frac{1}{6}h^3u_{xxx} + \frac{1}{24}h^4u_{xxxx} + O(h^5) \\ U_{j-1}^n &: u(t, x - k) = u - hu_x + \frac{1}{2}h^2u_{xx} - \frac{1}{6}h^3u_{xxx} + \frac{1}{24}h^4u_{xxxx} - O(k^5) \end{aligned}$$

Substituting approximations into the Crank–Nicolson scheme

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{2h^2} + \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{2h^2}$$

gives

$$u_t + \frac{1}{2}k + O(k^2) = u_{xx} + \frac{1}{12}h^2u_{xxxx} + O(h^3)$$

which says that the Crank–Nicolson is  $O(k + h^2)$  to  $u_t = u_{xx}$ .

### 13.3. By making the substitution

$$U_{j+2}^n = \hat{u}(t_n, \xi)e^{ijh\xi}$$

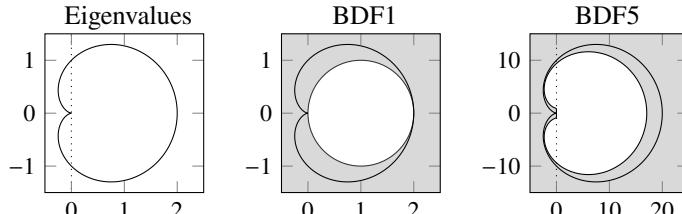
in the expression

$$\frac{U_{j+2}^n - 2U_{j+1}^n + U_{j-1}^n}{h^2},$$

we have

$$\frac{1}{h^2} \left( e^{i2h\xi} - 2e^{ih\xi} + 1 \right) \hat{u} = \frac{1}{h^2} e^{ih\xi} \left( e^{ih\xi} - e^{-ih\xi} \right)^2 \hat{u} = -\frac{2}{h^2} e^{ih\xi} \sin^2 \frac{\xi h}{2} \hat{u}$$

as the right hand side of  $\hat{u}_t = \lambda \hat{u}$ . The eigenvalues form a cardioid.



Examining the regions of stability for the methods in Figures 12.5, 12.6, and 12.8, we see that BDF1–BDF5 are the only time-difference schemes that would be absolutely stable and only if we take a sufficiently *large* timestep. For example, for the backward Euler (BDF1) we need to take  $k > h^2/2$  and for BDF5 methods we need to take  $k > 5h^2$ . Every other method we've examined is unconditionally unstable.

**13.4.** Start with the Dufort–Frankel scheme as (13.11) and move the second term on the right over to the left-hand side:

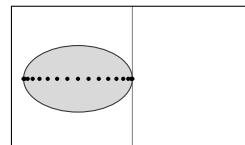
$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} + a \frac{U_j^{n+1} - 2U_j^n + U_j^{n-1}}{h^2} = a \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

Then from (13.9), the eigenvalues from the right-hand side are

$$\lambda = \frac{2a}{h^2} (\cos \xi h - 1).$$

The region of absolute stability determined by the left-hand side is

$$\lambda_\xi = i \frac{1}{k} \sin \theta - \frac{2a}{h^2} (\cos \xi h - 1),$$



which is an ellipse in the negative half-plane that extends from 0 to  $-4a/h^2$ , exactly the size that we need to fit the eigenspectrum.

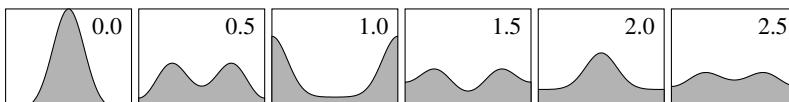
The Dufort–Frankel requires two starting values  $U^0$  and  $U^1$ . In practice, we might use a forward Euler step to generate  $U^1$  from  $U^0$ . But, because we are merely demonstrating consistency of the Dufort–Frankel, we'll simply set  $U^1$  equal to  $U^0$ .

```

 $\Delta x = 0.01; \Delta t = 0.01$
 $\ell = 1; x = -\ell:\Delta x:\ell; m = \text{length}(x)$
 $U^n = \exp(-8*x.^2); U^{n-1} = U^n$
 $\nu = \Delta t/\Delta x^2; \alpha = 0.5 + \nu; \gamma = 0.5 - \nu$
 $B = \nu * \text{Tridiagonal}(\text{ones}(m-1), \text{zeros}(m), \text{ones}(m-1))$
 $B[1, 2] *= 2; B[end, end-1] *= 2$
@gif for i = 1:300
 $U^n, U^{n-1} = (B*U^n + \gamma*U^{n-1})/\alpha, U^n$
 plot(x, U^n, ylim=(0, 1), label=:none, fill=(0, 0.3, :red))
end

```

The Dufort–Frankel solution to the heat equation is shown in the snapshots below and in the QR link at the bottom of the page:



Notice that while the long-term behavior is dissipative, the solution is largely oscillatory and the dynamics are more characteristic of a viscous fluid than heat propagation.

**13.6.** Let's determine the CFL condition. The constant-potential Schrödinger equation can be written as

$$\frac{\partial \psi}{\partial t} = \frac{i\varepsilon}{2} \frac{\partial^2 \psi}{\partial x^2} - i\varepsilon^{-1} V \psi.$$

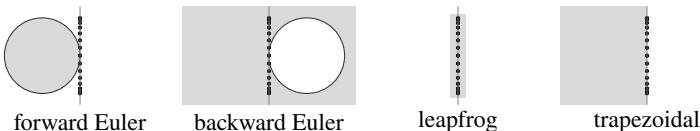
Using (13.9), we have

$$\frac{\partial \hat{u}}{\partial t} = -i \frac{2\varepsilon}{h^2} \sin^2 \left( \frac{\xi h}{2} \right) \hat{u} - \varepsilon^{-1} V \hat{u}. \quad (\text{A.6})$$

So, the eigenvalues  $\lambda_\xi$  of the system  $\hat{u}_t = \lambda_\xi \hat{u}$  are

$$\lambda_\xi = -i \left( \frac{2\varepsilon}{h^2} \sin^2 \frac{\xi h}{2} + \varepsilon^{-1} V \right).$$

These eigenvalues lie along the imaginary axis, bounded by  $|\lambda_\xi| \leq 2\varepsilon/h^2 + \varepsilon^{-1}|V|$ . For practical considerations,  $2\varepsilon/h^2$  is likely to be the dominate term in this expression if we wish to accurately model the dynamics of the equation, therefore we we'll only consider the contribution of this term on stability. Note where the regions of absolute stability intersect the imaginary axis for different methods in the following figures:



forward Euler      backward Euler      leapfrog      trapezoidal

We see that the backward Euler and trapezoidal methods are unconditionally stable and that the forward Euler is unconditionally unstable. The leapfrog method is stable when  $|k\lambda_\xi| < 1$ , meaning that its CFL condition is  $k < h^2/2\varepsilon$ . From Figure 12.8 on page 351 we see that the Runge–Kutta method (RK4) is stable when  $|k\lambda_\xi| < 2.5$ , so its CFL condition is  $k < 1.25h^2/\varepsilon$ . From Figures 12.5 and 12.6, we see that BDF2 is unconditionally stable and that higher-order BDF and Adams methods are conditionally stable.

The following Julia code solves the Schrödinger equation with  $V(x) = \frac{1}{2}x^2$  using a Crank–Nicolson method over the domain  $[-3, 3]$  with initial conditions  $\psi(0, x) = (\pi\varepsilon)^{-1/4} e^{-(x-1)^2/2\varepsilon}$ :

```
function schroedinger(m,n,e)
 x = LinRange(-4,4,m); Δx = x[2]-x[1]; Δt = 2π/n; V = x.^2/2
```

```

ψ = exp.(-(x.-1).^2/2ε)/(π*ε)^(1/4)
diags = 0.5im*ε*[1 -2 1]/Δx^2 .- im/ε*[0 1 0].*V
D = Tridiagonal(diags[2:end,1], diags[:,2], diags[1:end-1,3])
D[1,2] *= 2; D[end,end-1] *= 2
A = I + 0.5Δt*D
B = I - 0.5Δt*D
for i ∈ 1:n
 ψ = B\ (A*ψ)
end
return ψ
end

```

To verify the convergence rate of the method, we'll take a sufficiently small time step  $k$  so that its contribution to the error is negligible. Then we compute the error for several decreasing values of  $h$ . We repeat this process by taking  $h$  sufficiently small and computing the error for decreasing values of  $k$ .

```

ε = 0.3; m = 20000; e_x=[]; e_t=[]
N = floor.(Int,exp10.(LinRange(2,3.7,6)))
x = LinRange(-4,4,m)
ψ_m = -exp.(-(x.-1).^2/2ε)/(π*ε)^(1/4);
for n ∈ N
 x = LinRange(-4,4,n)
 ψ_n = -exp.(-(x.-1).^2/2ε)/(π*ε)^(1/4)
 append!(e_t,norm(ψ_m - schroedinger(m,n,ε))/m)
 append!(e_x,norm(ψ_n - schroedinger(n,m,ε))/n)
end
plot(2π./N,e_t,marker=:dot, xaxis=:log, yaxis=:log)
plot!(8 ./N,e_x,marker=:dot)

```

The error is plotted in Figure A.17 on the next page. We can compute the slopes of the lines by using `polyfit(log(n), log(error_x), 1)`. We find a slope of 2.0 for  $k$  and 2.5 for  $h$ .

Finally, let's examine the effect  $\varepsilon$  has on the solution  $\rho(t,x) = |\psi(t,x)|^2$ . Take  $k$  sufficiently small so that its contribution to the error is negligible, and compute the error as a function of  $h$  for several values of  $\varepsilon$ . We'll find the solution at  $t = \pi$  using  $10^4$  time steps, with four values of  $\varepsilon$  logarithmically spaced between 0.1 and 0.01 and several values of  $h$  logarithmically spaced between 0.2 and 0.002. The error as a function of  $h$  is plotted in Figure A.17 on the facing page for each of the four values of  $\varepsilon$ . Note that we have several similar plots with  $\varepsilon$  decreasing as we move left. The log-log curves are linear until they reach a maximum of 2. How can we explain such behavior? As the mesh spacing  $h$  increases relative to  $\varepsilon$ , the numerical solution doesn't adequately resolve the dynamics of the wave packet, slowing it down until the numerical solution no longer coincides with the true solution. If we plot the error against

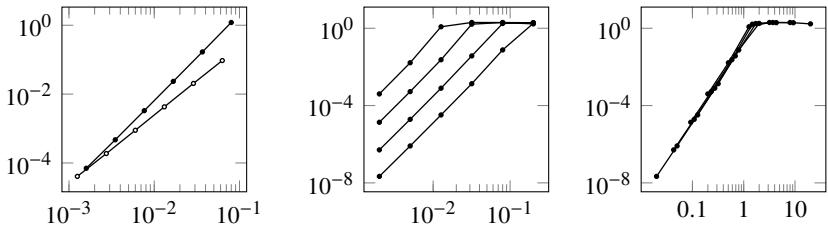


Figure A.17: Left: error in solution  $\psi(t, x)$  as a function of step size  $\bullet$  and time step  $\circ$ . Middle: error versus step size for various  $\varepsilon$ . Right: error versus step size as a multiple of  $\varepsilon$ .

$h/\varepsilon$ , we see that all of the curves line up with a turning point when  $h$  is greater than  $\varepsilon$ . In general, the Crank–Nicolson method is order  $O((k/\varepsilon)^2 + (h/\varepsilon)^2)$ .

13.7. We'll start by expanding the derivative

$$\frac{\partial u}{\partial t} = \frac{1}{r} \frac{\partial}{\partial r} \left( r \frac{\partial u}{\partial r} \right) \quad \text{to get} \quad \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r}.$$

This equation is problematic at  $r = 0$  because of the division by zero. Because  $u(t, r)$  is an even function, odd derivatives are zero at the origin. Therefore, we can apply L'Hopital's rule to get  $u_r/r = r_{rr}$  at  $r = 0$ . At  $r = 0$ , we have

$$\frac{\partial u}{\partial t} = 2 \frac{\partial^2 u}{\partial r^2}.$$

Let's discretize space to get  $\frac{\partial}{\partial t} U_j = D U_j$  where

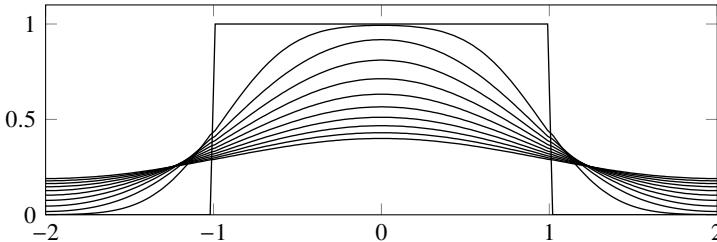
$$\begin{aligned} D U_0 &= 2 \frac{U_1 - 2U_0 + U_{-1}}{h^2}, \text{ and} \\ D U_j &= \frac{U_{j+1} - 2U_j + U_{j-1}}{h^2} + \frac{1}{r_j} \frac{U_{j+1} - U_{j-1}}{2h} \quad \text{for } j = 1, 2, \dots, m. \end{aligned}$$

Now, let's take care of the boundary conditions. An insulating boundary at  $r = 1$  means that  $u_r(1) = 0$ . We can approximate the derivative to second-order at  $x_m$  with  $U_{m+1} - U_{m-1} = 0$  by using a ghost point at  $x_{m+1}$ . This constraint will allow us to remove  $U_{m+1}$  from the system. The derivative at  $x_0$  is also zero—this time because of symmetry. So, we'll replace  $U_{-1}$  with  $U_1$  to close the system. We'll use the Crank–Nicolson method  $U_j^{n+1} = (I - \frac{1}{2}k D)^{-1} (I + \frac{1}{2}k D) U_j^n$ . The following Julia code implements the method:

```
t = 0.5; n=100; m = 100
r = LinRange(0,2,m); Δr = r[2]-r[1]; Δt = t/n;
u₀ = @. tanh(32(1 - r)); u = u₀
d = @. [1 -2 1]/Δr^2 + (1/r)*[-1 0 1]/2Δr
D = Tridiagonal(d[2:end,1],d[:,2],d[1:end-1,3])
D[1,1:2] = [-4 4]/Δr^2; D[end,end-1:end] = [2 -2]/Δr^2
A = I - 0.5Δt*D
B = I + 0.5Δt*D
for i = 1:n
 u = A\ (B*u)
end
```

The following is a slower, but high-order alternative to the Crank–Nicolson method:

```
using Sundials
problem = ODEProblem((u,p,t)->D*u,u₀,(0,t))
method = CVODE_BDF(linear_solver=:Band,jac_upper=1,jac_lower=1)
solution = solve(problem,method)
```



The figure above and the QR link below show the solution with an initial distribution given by a step function. The snapshots are taken at equal intervals from  $t \in [0, \frac{1}{2}]$  of the solution reflected about  $x = 0$ . Compare this solution with that of the one-dimensional heat equation on page 374.

**13.8.** Let's take grid points  $\{x_1, x_2, x_3\}$  and define  $h_1 = x_2 - x_1$  and  $h_2 = x_3 - x_1$ . The Taylor series expansion about  $x_1$  is

$$\begin{aligned} f(x_1) &= f(x_2 - h_1) = f(x_2) - h_1 f'(x_2) + \frac{1}{2} h_1^2 f''(x_2) - \frac{1}{6} h_1^3 f'''(x_2) + \dots \\ f(x_3) &= f(x_2 + h_2) = f(x_2) + h_2 f'(x_2) + \frac{1}{2} h_2^2 f''(x_2) + \frac{1}{6} h_2^3 f'''(x_2) + \dots \end{aligned}$$

We combine  $f(x_1)$ ,  $f(x_2)$ , and  $f(x_3)$  by solving the system of equations to determine  $f''(x_2)$  and eliminate  $f(x_2)$  and  $f'(x_2)$ :

$$f''(x_2) = \frac{2f(x_1)}{h_1(h_1 + h_2)} - \frac{2f(x_2)}{h_1 h_2} + \frac{2f(x_3)}{h_2(h_1 + h_2)} + \text{error}$$



where the error is  $\frac{1}{3}(h_2 - h_1)f'''(x_2) + O(h_1^2 + h_2^2)$ . We see that if  $h_2 = h_1$ , then the method is simply the second-order central difference scheme. As long as  $h_2 \approx h_1$  the error will be close to second-order, otherwise the error will locally be first-order.

We'll start by defining a logit function equivalent of LinRange:

```
logitspace(x,n,p) = x*atanh.(LinRange(-p,p,n))/atanh(p)
```

The parameter  $0 < p \leq 1$  will control how linear the function behaves. In the limit as  $p \rightarrow 0$ , logitspace( $x, n, p$ ) will behave like LinRange( $-x, x, n$ ). Now, we define a one-dimensional Laplacian operator for gridpoints given by  $x$ :

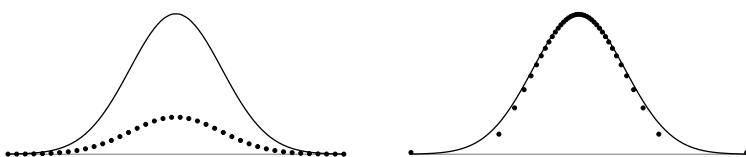
```
function laplacian(x)
 Δx = diff(x); Δx1 = Δx[1:end-1]; Δx2 = Δx[2:end]
 d_- = @. 2/[(Δx1*(Δx1+Δx2); Δx1[1]^2]
 d_0 = @. -2/[(Δx2[end].^2; Δx1*Δx2; Δx1[1].^2]
 d_+ = @. 2/[(Δx2[end].^2; Δx2*(Δx1+Δx2)]
 Tridiagonal(d_-,d_0,d_+)
end
```

We've used Neumann boundary conditions, but we could also have chosen Dirichlet boundary conditions. We can solve the heat equation using a Crank–Nicolson method.

```
function heat_equation(x,t,n,u)
 Δt = t/n
 D² = laplacian(x)
 A = I - 0.5Δt*D²
 B = I + 0.5Δt*D²
 for i ∈ 1:n
 u = A\B*u
 end
 return u
end
```

We'll define an initial condition and compare the equally-spaced grid along with the logit-spaced grid.

```
ϕ = (x,t,s) -> exp.(-s*x.^2/(1+4*s*t))/sqrt(1+4*s*t)
m = 40; t = 15
x = LinRange(-20,20,m)
plot(x,ϕ(x,t,10),label="exact",width=3)
u₁ = heat_equation(x,t,m,ϕ(x,0,10))
plot!(x,u₁,mark=:dot,label="equal")
x = logitspace(20,m,0.999)
u₂ = heat_equation(x,t,m,ϕ(x,0,10))
plot!(x,u₂,mark=:dot,label="logit")
```



The plots above compare solutions using equally-spaced nodes (left) and inverse-sigmoid-spaced nodes (right) to the exact solution.

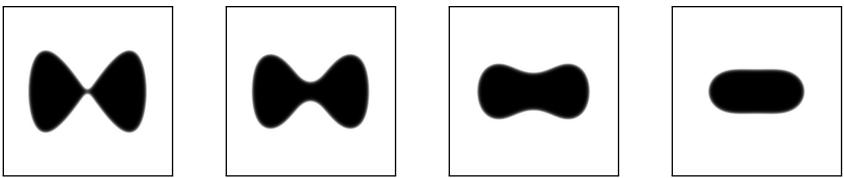
13.9. We'll solve the Allen–Cahn equation using Strang splitting. Note that the differential equation  $u'(t) = \varepsilon^{-1}u(1 - u^2)$  can be solved analytically to get  $u(t) = (u_0^2 - (u_0^2 - 1)e^{-t/\varepsilon})^{-1/2}u_0$ . We'll combine this solution with a Crank–Nicolson method for the heat equation applied first in the  $x$ -direction and then in the  $y$ -direction.

```
L = 16; m = 400; Δx = L/m
T = 4; n = 1600; Δt = T/n
x = LinRange(-L/2,L/2,m)'
u = @. tanh(x^4 - 16*(2*x^2-x.^2))
D = Tridiagonal(ones(m-1),-2*ones(m),ones(m-1))/Δx.^2
D[1,2] *= 2; D[end,end-1] *= 2
A = I + 0.5Δt*D
B = I - 0.5Δt*D
f = (u,Δt) -> @. u/sqrt(u.^2 - (u.^2-1)*exp(-50*Δt))
u = f(u,Δt/2)
for i = 1:n
 u = (B\A*(B\A*u))'
 (i<n) && (u = f(u,Δt))
end
u = f(u,Δt/2); Gray.(u)
```

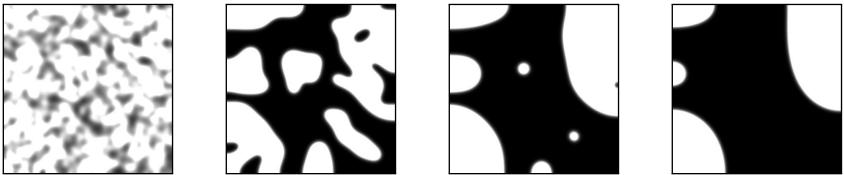
We can watch the evolution of the solution by adding the following three commands before the for loop, inside the for loop, and after the for loop:

```
anim = Animation()
(i%10)==1 && (plot(Gray.(u),border=:none); frame(anim))
gif(anim, "allencahn.gif", fps = 30)
```

Solutions to the Allen–Cahn equation at times  $t = 0.05, 0.5, 2.0$ , and  $4.0$  with initial conditions  $u = \tanh(x.^4 - 16*(x.^2-2*x'.^2))$  are shown on the next page. Also, see the QR link at the bottom of the page.



The solutions with initial conditions  $u = \text{randn}(m, m)$  are



The limiting behavior of the Allen–Cahn equation evident in these figures is called *motion by mean curvature*, in which an interface's normal velocity equals its mean curvature. Regions of high curvature evolve rapidly to regions of lower curvature, smoothing out any bumps, slowing down, and gradually forming circles, which may finally shrink and disappear.

*14.1.* The Lax–Friedrichs method is given by

$$\frac{U_j^{n+1} - \frac{1}{2}(U_{j+1}^n + U_{j-1}^n)}{k} + c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = 0.$$

Let's substitute the Taylor series approximation

$$\begin{aligned} u(t, x) &= u \\ u(t+k, x) &= u + ku_t + \frac{1}{2}k^2u_{tt} + \frac{1}{6}k^3u_{ttt} + O(k^4) \\ u(t, x+h) &= u + hu_x + \frac{1}{2}h^2u_{xx} + \frac{1}{6}h^3u_{xxx} + O(h^4) \\ u(t, x-h) &= u - hu_x + \frac{1}{2}h^2u_{xx} - \frac{1}{6}h^3u_{xxx} + O(h^4) \end{aligned}$$

in for  $U_j^n$ ,  $U_j^{n+1}$ ,  $U_{j+1}^n$ , and  $U_{j-1}^n$  in terms of the Lax–Friedrichs method

$$\frac{U_j^{n+1} - \frac{1}{2}(U_{j+1}^n + U_{j-1}^n)}{k} = u_t + \frac{1}{2}ku_{tt} + O(k^2) - \frac{1}{2}\frac{h^2}{k}u_{xx} + O(h^4)$$

and

$$c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = cu_x + \frac{1}{6}ch^2u_{xxx} + O(k^4).$$

By combining these terms, we have

$$u_t + cu_x + \frac{1}{2}ku_{tt} - \frac{1}{2}\frac{h^2}{k}u_{xx} + \frac{1}{6}ch^2u_{xxx} + \{\text{higher order terms}\} = 0,$$



from which we see that the truncation error is  $O(k + h^2/k)$ .

To find the dispersion relation, we examine the leading order truncation

$$u_t + cu_x + \frac{1}{2}ku_{tt} - \frac{1}{2}\frac{h^2}{k}u_{xx} = 0. \quad (\text{A.7})$$

We will eliminate the  $u_{tt}$  term by first differentiating (A.7) with respect to  $t$  and with respect to  $x$  and then substituting the expression for  $u_{tt}$  back into the original equation, discarding the higher order terms. We have

$$u_{tt} = -cu_{xt} + O(k + h^2/k)$$

$$u_{tx} = -cu_{xx} + O(k + h^2/k)$$

from which

$$u_{tt} = c^2u_{xx} + O(k + h^2/k).$$

After eliminating higher order terms, equation (A.7) becomes

$$u_t + cu_x + \frac{1}{2}k\left(c^2 - \frac{h^2}{k^2}\right)u_{xx} = 0.$$

Substituting the Fourier component  $u(t, x) = e^{i(\omega t - \xi x)}$  into this expression as an ansatz yields

$$\left[i\omega + c(-i\xi) - \frac{1}{2}\left(c^2k - \frac{h^2}{k^2}\right)\xi^2\right]e^{i(\omega t - \xi x)} = 0.$$

So,

$$i\omega + c(-i\xi) - \frac{1}{2}\left(c^2k - \frac{h^2}{k^2}\right)\xi^2 = 0,$$

and the dispersion relation  $\omega(\xi)$  is given by

$$\omega = c\xi - i\frac{1}{2}k\left(c^2 - \frac{h^2}{k^2}\right)\xi^2.$$

Plugging this  $\omega$  back into our ansatz gives us

$$u(t, x) = e^{i(\omega t - \xi x)} = e^{ic\xi t}e^{-\alpha\xi^2}e^{-i\xi x} = \underbrace{e^{i\xi(ct-x)}}_{\text{advection}} \cdot \underbrace{e^{-\alpha\xi^2 t}}_{\text{dissipation}}$$

where

$$\alpha = \frac{1}{2}k\left(\frac{h^2}{k^2} - c^2\right).$$

So, the Lax–Friedrichs scheme is dissipative, especially when  $k \ll h$ .

We can minimize the dissipation (and decrease the error) by choosing  $k$  and  $h$  so that

$$\frac{h^2}{k^2} - c^2 = 0,$$

that is, by taking  $k = h/|c|$ . Note that if  $k > h/|c|$ , then  $\alpha$  is negative and  $e^{-\alpha \xi^2 t}$  grows, causing the solution blows up. This condition is the CFL condition for the Lax–Friedrichs scheme.

*14.3.* The scheme is

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} + c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = 0.$$

By Taylor series expansion

$$\begin{aligned} u(t+k, x) &= u + ku_t + \frac{1}{2}k^2u_{tt} + \frac{1}{6}k^3u_{ttt} + O(k^4) \\ u(t-k, x) &= u - ku_t + \frac{1}{2}k^2u_{tt} - \frac{1}{6}k^3u_{ttt} + O(k^4) \\ u(t, x+h) &= u + hu_x + \frac{1}{2}h^2u_{xx} + \frac{1}{6}h^3u_{xxx} + O(h^4) \\ u(t, x-h) &= u - hu_x + \frac{1}{2}h^2u_{xx} - \frac{1}{6}h^3u_{xxx} + O(h^4) \end{aligned}$$

from which

$$u_t + \frac{1}{2}k^2u_{ttt} + cu_x + \frac{1}{2}h^2u_{ttt} = O(k^2 + h^2).$$

So, the method is  $O(k^2 + h^2)$ .

The scheme is leap frog in time and central difference in space. The Fourier transform of

$$\frac{\partial}{\partial t} U_j = -c \frac{U_{j+1} - U_{j-1}}{2h}$$

is

$$\frac{\partial}{\partial t} \hat{u}(t, \xi) = -c \frac{e^{i\xi h} - e^{-i\xi h}}{h} \hat{u}(t, \xi) = -i \frac{c}{h} \sin(\xi h) \hat{u}(t, \xi).$$

Recall that the leapfrog scheme is absolute stability only on the imaginary axis  $\lambda k \in [-i, +i]$ . The factor  $|\sin(\xi h)| \leq 1$ , so the scheme is stable when  $(|c|/h)k \leq 1$ . That is, when  $k \leq h/|c|$ .

*14.6.* We can express the compressible Euler equations

$$\rho_t + (\rho u)_x = 0 \tag{A.8a}$$

$$(\rho u)_t + (\rho u^2 + p)_x = 0 \tag{A.8b}$$

$$E_t + ((E + p)u)_x = 0 \tag{A.8c}$$

in semilinear form using  $\rho$ ,  $u$  and  $p$  as independent variables. First, rewrite (A.8a) and (A.8b) as

$$\rho_t + u\rho_x + \rho u_x = 0 \quad (\text{A.9})$$

$$u\rho_t + \rho u_t + u^2\rho_x + 2\rho uu_x + p_x = 0 \quad (\text{A.10})$$

We can eliminate the  $u\rho_t$  term in (A.10) by first multiplying (A.9) by  $u$  to get

$$u\rho_t + u^2\rho_x + \rho uu_x = 0$$

and then subtracting this equation from (A.10) to get

$$\rho u_t + \rho uu_x + p_x = 0.$$

$$u_t + uu_x + \frac{1}{\rho}p_x = 0 \quad (\text{A.11})$$

To derive an equation for  $p$ , we will use the equation of state

$$E = \frac{1}{2}\rho u^2 + \frac{p}{\gamma - 1} \quad (\text{A.12})$$

to eliminate  $E$  from (A.8c). Differentiating (A.12) with respect to  $t$  yields

$$E_t = \frac{1}{2}u^2\rho_t + \rho uu_t + \frac{1}{\gamma - 1}p_t.$$

Substituting (A.9) and (A.11) in this expression for  $\rho_t$  and  $u_t$  produces

$$E_t = -\frac{1}{2}u^3\rho_x - \frac{3}{2}\rho u^2u_x - up_x + \frac{1}{\gamma - 1}p_t. \quad (\text{A.13})$$

By expanding  $((E + p)u)_x$  with the equation of state, we have

$$\begin{aligned} ((E + p)u)_x &= \left( \frac{1}{2}u^3\rho + \frac{\gamma}{\gamma - 1}pu \right)_x \\ &= \frac{1}{2}u^3\rho_x + \frac{3}{2}\rho u^2u_x + \frac{\gamma}{\gamma - 1}pu_x + \frac{\gamma}{\gamma - 1}up_x \end{aligned} \quad (\text{A.14})$$

By combining (A.13) and (A.14), we have

$$E_t + ((E + p)u)_x = \frac{1}{\gamma - 1}p_t + \frac{\gamma}{\gamma - 1}pu_x + \frac{1}{\gamma - 1}up_x.$$

So

$$p_t + \gamma pu_x + up_x = 0. \quad (\text{A.15})$$

We now can express (A.9), (A.11) and (A.15) in quasilinear matrix form

$$\begin{bmatrix} \rho \\ u \\ p \end{bmatrix}_t + \begin{bmatrix} u & \rho & 0 \\ 0 & u & 1/\rho \\ 0 & \gamma p & u \end{bmatrix} \begin{bmatrix} \rho \\ u \\ p \end{bmatrix}_x = 0.$$

The system is hyperbolic if the eigenvalues  $\lambda$  of the Jacobian matrix are real:

$$\begin{vmatrix} u - \lambda & \rho & 0 \\ 0 & u - \lambda & 1/\rho \\ 0 & \gamma p & u - \lambda \end{vmatrix} = (u - \lambda)^3 - \gamma p \rho^{-1} (u - \lambda) \\ = (u - \lambda) [(u - \lambda)^2 - c^2] = 0$$

with  $c^2 = \gamma p / \rho$ . The eigenvalues  $\lambda = \{u, u + c, u - c\}$  where  $c$  is the sound speed.

14.7. The solution starts with a shock beginning at  $x = 1$  moving with speed  $\frac{1}{2}$  (half the height, as given by the Rankine–Hugoniot condition). At the same time a rarefaction propagates from  $x = 0$  moving at speed 1 until the rarefaction catches up with the shock at location  $x = 2$  and time  $t = 2$ . We can imagine the trailing edge of a right-angled trapezoid catching up to the leading edge to form a right triangle. Now, the leading edge of the right triangle continues to move forward with speed equal to half the height (again by the Rankine–Hugoniot condition). To compute the position of the leading edge, we only need to know the height  $u(t)$ . The area of the triangle is conserved, so  $A = \frac{1}{2}x(t)u(t, x) = 1$ . That is,  $u = 2/x$ . The speed is given by

$$\frac{dx}{dt} = \frac{1}{2}u(t, x) = \frac{1}{x}$$

The solution to this differential equation is  $x(t) = \sqrt{2t + c}$  for some constant  $c$ . At  $t = 2$ , the leading edge is at  $x = 2$ , and  $c = 0$ . Putting everything together, we have

$$\text{when } t < 2 \quad u(x, t) = \begin{cases} x/t, & 0 < x < t \\ 1, & t < x < 1 + \frac{1}{2}t \\ 0, & \text{otherwise} \end{cases}$$

$$\text{when } t \geq 2 \quad u(x, t) = \begin{cases} x/t, & 0 < x < \sqrt{2t} \\ 0, & \text{otherwise} \end{cases}$$

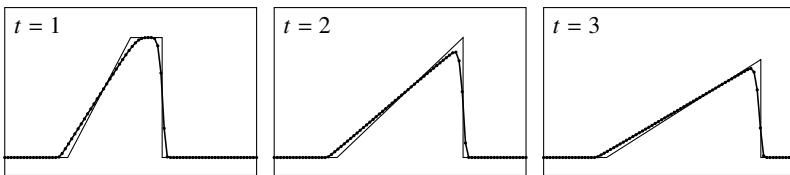
because the area is conserved. We can write the exact solution as the function:

```
U = (x, t) -> @. t<2 ?
 (0<x<t)*(x/t) + (t<x<1+t/2) : (x/t)*(0<x<sqrt(2t))
```

The local Lax–Friedrichs method can be implemented using the following code:

```
m = 80; x = LinRange(-1,3,m); Δx = x[2]-x[1]; j = 1:m-1
n = 100; Lt = 4; Δt = Lt/n
f = u-> u.^2/2; df = u -> u
u = (x.>=0).* (x.<=1)
anim = @animate for i = 1:n
 α = 0.5*max.(abs.(df(u[j])),abs.(df(u[j.+1])))
 F = (f(u[j])+f(u[j.+1]))/2 - α.* (u[j.+1]-u[j])
 u -= Δt/Δx*[0;diff(F);0]
 plot(x,u, fill = (0, 0.3, :blue))
 plot!(x,U(x,i*Δt), fill = (0, 0.3, :red))
 plot!(legend=:none, ylim=[0,1])
end
gif(anim, "burgers.gif", fps = 15)
```

The analytical solution and the numerical solution shown below and in the QR link at the bottom of the page:



#### 14.8. The Nessyahu–Tadmor scheme

$$\begin{aligned} U_j^{n+1/2} &= U_j^n - \frac{1}{2}h\partial_x f(U_j^n) \\ U_{j+1/2}^{n+1} &= \frac{1}{2}(U_j^n + U_{j+1}^n) - \frac{1}{8}h(\partial_x U_{j+1}^n - \partial_x U_j^n) - \frac{k}{h}\left(f(U_{j+1}^{n+1/2}) - f(U_j^{n+1/2})\right) \end{aligned}$$

where we approximate  $\partial_x$  using the slope limiter  $\sigma_j$

$$\sigma_j(U_j) = \frac{U_{j+1} - U_j}{h} \phi(\theta_j) \quad \text{where} \quad \theta_j = \frac{U_j - U_{j-1}}{U_{j+1} - U_j}$$

with the van Leer limiter

$$\phi(\theta) = \frac{|\theta| + \theta}{1 + |\theta|} \cdot s$$

We can implement the solution using Julia as

```
δ = u -> diff(u,dims=1)
ϕ = t -> (abs(t)+t)/(1+abs(t))
fixnan(u) = isnan(u)||isinf(u) ? 0 : u
θ = δu -> fixnan.(δu[1:end-1,:]./δu[2:end,:])
∂x(u) = (δu=δ(u);[[0 0];δu[2:end,:,:]*ϕ.(θ(δu));[0 0]])
F = u -> [u[:,1].*u[:,2] u[:,1].*u[:,2].^2+0.5u[:,1].^2]
```

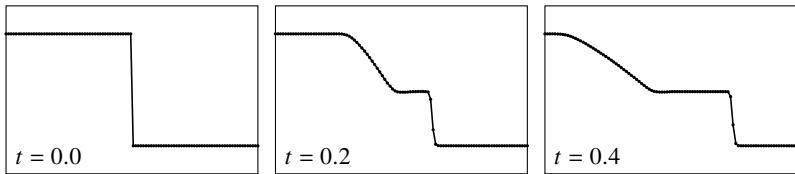


```

m = 100; x = LinRange(-.5,.5,m); Δx = x[2]-x[1]
T = 0.4; n = ceil(T/(Δx/2)); Δt = (T/n)/2;
U = [0.8*(x.<0).+0.2 zero(x)]
Uj = view(U,1:m-1,:); Uj+1 = view(U,2:m,:)
for i = 1:n
 U° = U-0.5*Δt/Δx*∂x(F(U))
 Uj+1 .= (Uj+Uj+1)/2 - Δt/Δx*∂(F(U))/8 - Δt/Δx*δ(F(U°))
 U° = U-0.5*Δt/Δx*∂x(F(U))
 Uj .= (Uj+Uj+1)/2 - Δt/Δx*∂(F(U))/8 - Δt/Δx*δ(F(U°))
end

```

The numerical solution for  $h(t, x)$  is shown below and at the QR link at the bottom of the page:



Notice the shock wave moving to the right and the rarefaction wave moving to the left from the initial discontinuity.

*15.1.* Multiply  $-u'' - u = f(x)$  by  $v$  and integrate:  $\int_0^1 (-u'' - u)v \, dx = \int_0^1 vf \, dx$  where  $f = -8x^2$ . If we integrate by parts once and move the boundary terms to the right-hand side of the equation, then we have the variational form  $a(u_h, v_h) = L(v_h)$  where

$$a(u, v) = \int_0^1 u'v' - uv \, dx \quad \text{and} \quad L(v) = \int_0^1 vf \, dx + vu'|_0^1.$$

The boundary conditions for this problem are  $u'(0) = 0$  and  $u'(1) = 1$ . Take the finite elements  $u_h(x) = \sum_{i=0}^{m+1} \xi_i \varphi_i(x)$  and  $v_h(x) = \sum_{j=0}^{m+1} v_j \varphi_j(x)$  where  $\varphi_j(x)$  are the piecewise basis elements defined in (15.1). Then

$$\sum_{j=0}^{m+1} v_j \sum_{i=0}^{m+1} \xi_i a(\varphi_j, \varphi_i) = \sum_{i=0}^{m+1} v_j L(\varphi_j).$$

Because the expression holds for set  $\{v_j\}$ , it follows that for all  $j$

$$\sum_{i=0}^{m+1} \xi_i a(\varphi_j, \varphi_i) = L(\varphi_j).$$

We can compute the integrals using (15.1):  $a(\varphi_j, \varphi_i) = -h^{-1} - \frac{1}{6}h$  when  $i \neq j$  and  $a(\varphi_i, \varphi_i) = 2h^{-1} - \frac{2}{3}h$  except on the two boundaries where it is



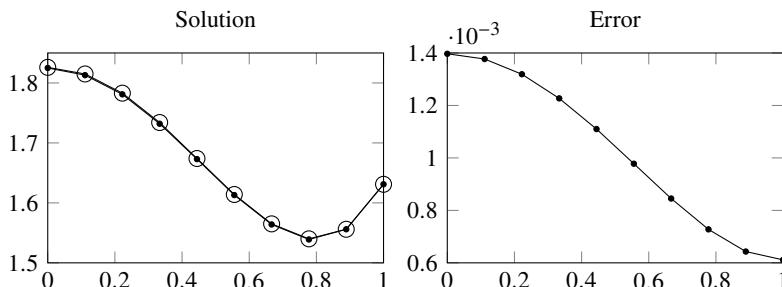


Figure A.18: Solutions to the Neumann problem 15.1. The marker  $\bullet$  is used for the finite element solution and  $\circ$  for the analytical solution.

half that value. Similarly,  $b(\varphi_j) = -\frac{4}{3}h^3 - 8h(jh)^2$ ,  $b(\varphi_0) = -\frac{2}{3}h^3$ , and  $b(\varphi_{m+1}) = -4h + \frac{8}{3}h^2 - \frac{2}{3}h^3 + 1$ , where we've added in the contribution to the boundary term at  $x = 1$ . In Julia

```
m=10; x=LinRange(0,1,m); h=x[2]-x[1]
α = fill(2/h-2h/3,m); α[[1,m]] /= 2; β = fill(-1/h-h/6,m-1)
A = SymTridiagonal(α,β)
b = [-2h^3/3; -4h^3/3 .- 8h*x[2:m-1].^2; -4h+8h^2/3-2h^3/3+1]
u = A\b
s = -16 .+ 8x.^2 .+ 15csc(1).*cos.(x)
plot(x,s,marker=:o,alpha=0.5); plot!(x,u,marker=:o,alpha=0.5)
```

The finite element solution is plotted in Figure A.18 along with the exact solution  $u(x) = -16 + 8x^2 + 15 \csc 1 \cos x$ . Note that we get very little error even with only 10 nodes.

15.2. Let  $V$  be the space of twice differentiable functions that take the same boundary conditions as  $u(x)$ . Multiplying the equation  $u'''' = f$  by  $v \in V$  and integrate by parts twice so that we have a symmetric bilinear form to get  $\int_0^1 u''v'' dx = \int_0^1 fv dx$ . (The boundary terms vanish when we enforce the boundary conditions.) Define  $a(u, v) = \int_0^1 u''v'' dx$  and  $L(v) = \int_0^1 fv dx$ . Then the finite element problem is

(V) Find  $u_h \in V_h$  such that  $a(u_h, v_h) = L(v_h)$  for all  $v_h \in V_h$

The finite elements are

$$u_h(x) = \sum_{i=0}^{m+1} \xi_i \phi_i(x) + \eta_i \psi_i(x) \text{ and } v_h(x) = \sum_{j=0}^{m+1} \alpha_j \phi_j(x) + \beta_j \psi_j(x),$$

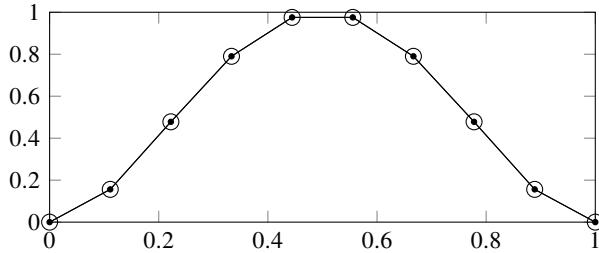


Figure A.19: Solutions to the beam problem. The marker ● is used for the finite element solution and ○ for the analytical solution.

where the basis elements  $\phi_j(x)$  are used to prescribe the value of  $u(x)$  at the nodes and the basis elements  $\psi_j(x)$  are used to prescribe the value of the derivative of  $u(x)$  at the nodes. The coefficients  $\xi_j$  and  $\eta_j$  are unknown and  $\alpha_j$  and  $\beta_j$  are arbitrary. From the boundary conditions, we have  $\xi_0 = \xi_{m+1} = \eta_0 = \eta_{m+1} = 0$ . Substituting  $u_h$  and  $v_h$  into  $a(u_h, v_h) = L(v_h)$  yields

$$a\left(\sum_{i=1}^m \xi_i \phi_i + \eta_i \psi_i, \sum_{j=1}^m \alpha_j \phi_j + \beta_j \psi_j\right) = L\left(\sum_{j=1}^m \alpha_j \phi_j + \beta_j \psi_j\right).$$

Because  $\alpha_j$  and  $\beta_j$  are arbitrary, we must have

$$a\left(\sum_{i=1}^m \xi_i \phi_i + \eta_i \psi_i, \phi_j\right) = L(\phi_j) \quad \text{and} \quad a\left(\sum_{i=1}^m \xi_i \phi_i + \eta_i \psi_i, \psi_j\right) = L(\psi_j)$$

for all  $j = 1, 2, \dots, m$ . Finally, by bilinearity and homogeneity we have

$$\begin{aligned} \sum_{i=1}^m \xi_i a(\phi_i, \phi_j) + \eta_i a(\psi_i, \phi_j) &= L(\phi_j) \\ \sum_{i=1}^m \xi_i a(\phi_i, \psi_j) + \eta_i a(\psi_i, \psi_j) &= L(\psi_j) \end{aligned}$$

We have a system of  $2m$  equations, which we can write as the block matrix

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{C} \end{bmatrix} \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{\eta} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \end{bmatrix},$$

with elements  $a_{ij} = \langle \phi_i'', \phi_j'' \rangle$ ,  $b_{ij} = \langle \psi_i'', \phi_j'' \rangle$ , and  $c_{ij} = \langle \psi_i'', \psi_j'' \rangle$  along with  $f_j^{(1)} = \langle f, \phi_j \rangle = h^{-1}$  and  $f_j^{(2)} = \langle f, \psi_j \rangle = 0$ .

We can compute the solution in Julia using

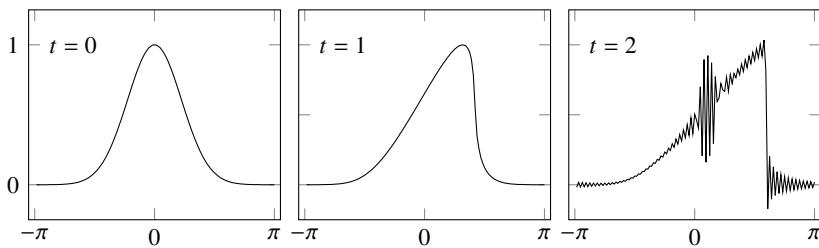


Figure A.20: Solutions to Burgers' equation in exercise 16.2 at  $t = 0, 1$ , and  $2$ . Gibbs oscillations overwhelm the solution after a discontinuity develops.

```
m = 8; x = LinRange(0,1,m+2); h = x[2]-x[1]
σ = (a,b,c) -> Tridiagonal(fill(a,m-1),fill(b,m),fill(c,m-1))/h^3
M = [σ(-12,24,-12) σ(-6,0,6);σ(6,0,-6) σ(2,8,2)];
b = [384h*ones(m);zeros(m)]
u = M\b
s = 16*(x.^4 - 2x.^3 + x.^2)
plot(x,[s [0;u[1:m];0]],marker=:o,alpha=0.5)
```

If we only want the solution at the nodes, we can take  $u(x_j) = \xi_j$ :

```
s = 16*(x.^4 - 2x.^3 + x.^2)
plot(x,[s [0;u[1:n];0]],marker=:o,alpha=0.5)
```

The solution using 8 interior nodes is shown in Figure A.19. Because the problem has a constant load, the finite element solution matches the analytical solution exactly.

16.2. Fourier spectral methods have spectral accuracy in space so long as the solution is a smooth function. Space and time derivatives are coupled through Burgers' equation, so we can expect a method that is fourth order in time and space. However, solutions to Burgers' equation, which may initially be smooth, become discontinuous over time. Because of this, truncation error is half-order in space and time once a discontinuity develops. This is bad. But it gets worse. Gibbs oscillations develop around the discontinuity, and these oscillations will spread and grow because Burgers' equation is dispersive. Ultimately, the oscillations overwhelm the solution. We can implement the  $u_t = -\frac{1}{2} F^{-1} [i\xi F(u^2)]$  using the Dormand–Prince Runge–Kutta solver:

```
using FFTW, DifferentialEquations
m = 128; x = (1:m)/m*(2π) .- π
ξ = im*[0:(m÷2); (-m÷2+1):-1]
f = (u,p,t) -> -real(ifft(ξ.*fft(0.5u.^2)))
```

```
u = solve(ODEProblem(f,exp.(-x.^2),(0.0,2.0)),DP5());
```

which can be animated using the Plots.jl macro

```
@gif for t=0:.02:2
 plot(x,u(t),ylim=(-0.2,1.2))
end
```

The solution is shown on the facing page and in the QR link on the bottom of this page.

**16.3.** The following Julia code solves the KdV equation using integrating factors. We first set the initial conditions and parameters:

```
using FFTW, DifferentialEquations
ϕ = (x,x₀,c) -> 0.5c*sech(sqrt(c)/2*(x-x₀))2
L = 30; T = 2.0; m = 256
x = (1:m)/m*L .- L/2
u₀ = ϕ.(x,-4,4) + ϕ.(x,-9,9)
iξ = im*[0:(m÷2);(-m÷2+1):-1]*2π/L
F = plan_fft(u₀)
```

Next, we define integrating factor G and function f of the differential equation:

```
G = t -> exp.(-iξ.^3*t)
f = (w,_,t) -> -G(t) .\ (3iξ .* (F * (F \ (G(t).*w)).^2))
```

Then, we solve the problem

```
w = solve(ODEProblem(f,F*u₀,(0,T)),DP5())
u = t -> real(F\w(t).*G(t)))
```

Finally, we can plot an animation of the solution

```
@gif for t=0:.01:2
 plot(x,u(t),ylim=(0,5),legend=:none)
end
```

The soliton solution to the KdV equation is plotted in Figure A.21 and at the QR link on the next page. The top row shows the two independent solutions with  $u(x, 0) = \phi(x; -4, 4)$  and  $u(x, 0) = \phi(x; -9, 9)$  as separate initial conditions. The bottom row shows the two-soliton solution for  $u(x, 0) = \phi(x; -4, 4) + \phi(x; -9, 9)$ . After collision the taller, fast soliton in the bottom row is about 1 unit in front of the corresponding soliton in the top row. The smaller, slow soliton in the bottom row is about 1.5 units behind the corresponding soliton in the top row. Each soliton has the same energy after collision as it had before collision—their positions are merely shifted.



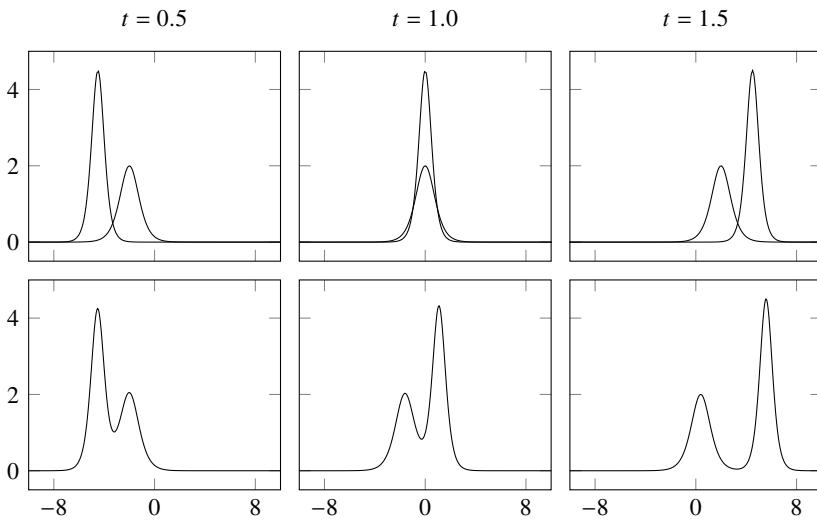


Figure A.21: Solutions of the KdV equation in exercise 16.3. The top row shows independent solitons. The bottom row shows interacting solitons.

**16.4.** Let's use Strang splitting on  $u' = Nu + Lu$ . There are two reasonable ways to define  $N$  and  $L$ . The first with  $Nu = \varepsilon u - u^3$  and  $Lu = -(\Delta + 1)^2 u$ , and the second with  $Nu = -u^3$  and  $Lu = \varepsilon u - (\Delta + 1)^2 u$ . In the first case, the analytic solution to  $u' = \varepsilon u - u^3$  is

$$u = \frac{u_0}{\sqrt{(1 - \varepsilon^{-1}u_0^2)e^{-2\varepsilon t} + \varepsilon^{-1}u_0^2}};$$

while in the second case, the solution to  $u' = -u^3$  is  $u = u_0(1 - 2tu_0^2)^{-1/2}$ . While the second case seems simpler, it does not accurately model the equilibrium dynamics  $u'(x, y, t) = 0$  as  $t \rightarrow \infty$ . The equilibrium solution  $\varepsilon u - u^3 - (\Delta + 1)^2 u = 0$  has a reaction component  $\varepsilon u - u^3 = 0$  and a diffusion component  $(\Delta + 1)^2 u = 0$ . The reaction component has two stable equilibrium at  $u = \pm\sqrt{\varepsilon}$  in addition to the unstable equilibrium at  $u = 0$ .

Strang splitting uses half-whole-half-step operator splitting at each iteration to get second-order in time error. The solution to  $u' = Nu$  over  $\Delta t/2$  is

$$f(u_0) = \frac{u_0}{\sqrt{(1 - \varepsilon^{-1}u_0^2)e^{-\varepsilon\Delta t} + \varepsilon^{-1}u_0^2}},$$

and the solution to  $\hat{u}' = \hat{L}\hat{u}$  over a time interval  $\Delta t$  is

$$E\hat{u}_0 = e^{-(\hat{\Delta}+1)^2\Delta t}\hat{u}_0.$$



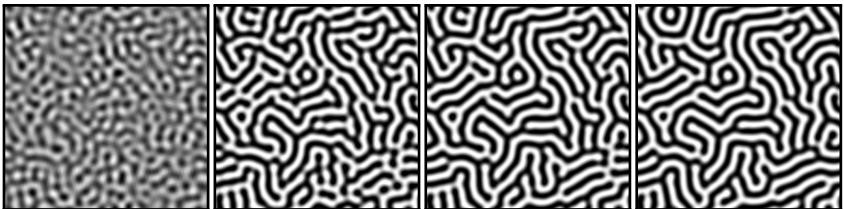


Figure A.22: Solution to the Swift–Hohenberg equation at  $t = 3, 10, 25$ , and  $100$ .

At each time step we have  $u = f(F^{-1}(E \circ F(f(u))))$ . The following Julia code produces the solution to the Swift–Hohenberg equation shown in Figure A.22 above and at the QR link at the bottom of this page.

```
using FFTW, Images
ε = 1; m = 256; ℓ = 100; n = 2000; Δt=100/n
U = (rand(m,m).>.5) .- 0.5
ξ = [0:(m÷2);(-m÷2+1):-1]*2π/ℓ
D² = -ξ.^2 .- (ξ.^2)'
E = exp.(-(D².+1).^2*Δt)
F = plan_fft(U)
f = U -> U./sqrt.(U.^2/ε + exp(-Δt*ε)*(1 .- U.^2/ε))
for i=1:600
 U = f(F\.(E.*(F*f(U))))
end
```

We can animate the solution by adding the following code inside the loop

```
save("temp"*lpad(i,3,"0")*.png",Gray.(clamp01.((real(U).+1)/2)))
```

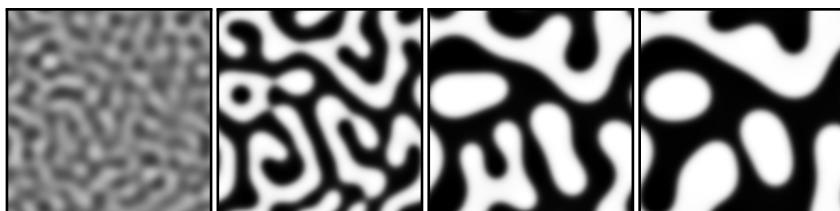
and then making a call to ffmpeg to produce a movie

```
run(`ffmpeg -i "temp%03d.png" -movflags faststart
-pix_fmt yuv420p swifthohenberg.mp4`)
```

## 16.5. We'll use a fourth-order ETDRK4 scheme

```
using FFTW, DifferentialEquations, DiffEqOperators
ℓ = 8; T = 8.0; m = 256; ε² = 0.01; α = 1
ξ = [0:(m÷2);(-m÷2+1):-1]*2π/ℓ
Δ = -(ξ.^2 .+ ξ'.^2); Δ² = Δ.^2
u₀ = randn(m,m)
F = plan_fft(u₀)
L = DiffEqArrayOperator(Diagonal((-ε²*Δ²+α*Δ)[:]))
N = (u,_,_) -> (v = F\reshape(u,m,m); Δ.*(F*(v.^3-(1+α)*v))[:])
```



Figure A.23: Solution to the Cahn–Hilliard equation at  $t = 0.1, 1, 4$ , and  $8$ .

```
problem = SplitODEProblem(L,N,(F*u₀)[:,(0,T))
solution = solve(problem,ETDRK4(),dt = 0.1)
u = t -> real(F\reshape(solution(t),m,m))
```

We can observe the solution over time using the `Interact.jl` library.

```
using Interact, ColorSchemes, Images
@manipulate for t in slider(0:0.1:8; value=0, label="time")
 get(colorschemes[:binary], u(t), :extrema)
end
```

The solution at time are shown in Figure A.23 and at the QR link at the bottom of this page. Notice how the dynamics slow down over time.

*16.6.* We'll use a fourth-order ETDRK method to solve the KSE.

```
using FFTW, DifferentialEquations, DiffEqOperators
ℓ = 50; T = 200.0; n = 128
x = LinRange(-ℓ/2, ℓ/2, n+1)[2:end]
u₀ = randn(n,n)
ξ = [0:(n÷2); (-n÷2+1):-1]*2π/ℓ
Δ = -(ξ.^2 .+ ξ'.^2); Δ² = Δ.^2
F = plan_fft(u₀)
L = DiffEqArrayOperator(Diagonal((-Δ-Δ²)[:, :]))
N = (u, _, _) ->
 (v = reshape(u, n, n);
 v = -0.5*abs((F\im*ξ.*v)).^2 + (F\im*ξ'.*v)).^2;
 w = (F*v)[:, :]; w[1] = 0.0; return w)
problem = SplitODEProblem(L,N,(F*u₀)[:,(0,T))
solution = solve(problem, ETDRK4(), dt=0.05, saveat=0.5);
u = t -> real(F\reshape(solution(t),n,n))
```

We can observe the time evolution of the solution using the `Interact.jl` library. See Figure A.24 on the next page.

```
using Interact, ColorSchemes, Images
```



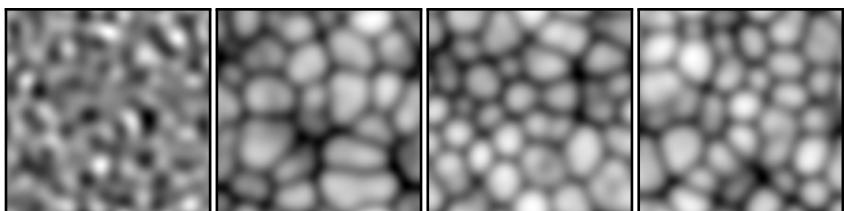


Figure A.24: Solution to the Kuramoto–Sivashinsky equation at  $t = 1, 50, 100$ , and  $150$ . Also, see the QR link at the bottom of this page.

```
@manipulate for t in slider(0:0.5:T; value=0, label="time")
 get(colorschemes[:magma], -u(t), :extrema)
end
```





## Appendix B

---

# Computing in Python and Matlab

## B.1 Scientific programming languages

### ► The trends

The evolution of modern scientific computing programming languages over the past seventy years is a product of several compounding and reinforcing factors: the exponential growth in computing speed and memory along with the considerable drop in computing cost; a paradigm shift towards open-source software and open access research; a widespread and fast internet, spurring greater information sharing and cloud computing; the shift from digital immigrants to digital natives and increase in diversity; the production and monetization of massive quantities of data; and the emergence of specialized fields of computational science like data science, bioinformatics, computational neuroscience, and computational sociology.

In 1965 Gordon Moore, the co-founder of Fairchild Semiconductor and later co-founder and CEO of Intel, observed that the number of transistors on an integrated circuit doubled every year. He revised his estimate ten years later, stating that the doubling occurred every two years. Moore's empirical law has more or less held since then. It is difficult to overstate the impact that technological change and economic growth have had on computing. ENIAC (Electronic Numerical Integrator and Computer), built in the mid-1940s, was the first programmable, digital, electronic computer. It cost \$6.6 million in today's dollars, weighed 27 tonnes, and would fill a small house. ENIAC had the equivalent of 40 bytes of internal memory and could execute roughly 500 floating-point operations per second. Cray-1, developed in the late 1970s, was one of the most commercially successful supercomputers with 80 units sold. It cost \$33 million in today's dollars, weighed 5 tonnes, and would fill a closet. Cray-1 had 8 million bytes of memory and could execute 160 million floating-point operations per second. Today, a typical smartphone costs \$500,

weighs less than 200 grams, and fits in a pocket. Smartphones often have 6 billion bytes of memory and can execute 10 billion floating-point operations per second. High-performance cloud computing, which can be accessed with those smartphones, has up to 30 trillion bytes of memory and can achieve a quintillion (billion billion) floating-point operations per second using specialized GPUs. And even now, scientists are developing algorithms that use nascent quantum computers to solve intractable high-dimensional problems in quantum simulation, cryptanalysis, and optimization.

The growth in open data, open standards, open access and reproducible research, and open-source software has further accelerated the evolution of scientific computing languages. Unlike proprietary programming languages and libraries, the open-source movement creates a virtuous innovation feedback loop powered through open collaboration, improved interoperability, and greater affordability. Even traditionally closed-source software companies such as IBM, Microsoft, and Google have embraced the open-source movement through Red Hat, GitHub, and Android. Former World Bank Chief Economist and Nobel laureate Paul Romer has noted, “The more I learn about the open source community, the more I trust its members. The more I learn about proprietary software, the more I worry that objective truth might perish from the earth.” Python, Julia, Matlab, and R all have a robust community of user-developers. The Comprehensive R Archive Network (CRAN) features 17 thousand contributed packages, the Python Python Package Index (PyPI) has over 200 thousand packages, and GitHub has over 100 million repositories. Nonprofit foundations like NumFOCUS (Numerical Foundation for Open Code and Useable Science) have further supported open-source scientific software projects.

In 1984 Donald Knuth, author of the *Art of Computer Programming* and the creator of *T<sub>E</sub>X*, introduced *literate programming* in which natural language exposition and source code are combined by considering programs to be “works of literature.” Knuth explained that “instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.” Mathematica, first released in the late 1980s, used notebooks that combined formatted text, typeset mathematics, and Wolfram Language code into a series of interactive cells. In 2001, Fernando Pérez developed a notebook for the Python programming language, called IPython, that allowed code, narrative text, mathematical expressions, and inline plots in a browser-based interface. In 2014, Pérez spun off the open-source Project Jupyter. Jupyter (a portmanteau of Julia, Python, and R and an homage to the notebooks that Galileo recorded his observations on moons of Jupiter) extends IPython to dozens of programming languages. Projects like Pluto for Julia and Observables for JavaScript have further developed notebooks to make them even more immersive as reactive notebooks. Code everywhere in a reactive notebook re-executes whenever a variable is changed anywhere in the notebook. The impact of notebooks in scientific programming is so significant that James

Somers, in a recent article in *The Atlantic* declared (too boldly) that the “scientific paper is obsolete.”

The widespread availability of high-speed internet has further improved collaboration. Crowdsourcing projects like Wikipedia and StackExchange have transformed how information gets disseminated. Massive open online courses (MOOCs) have made learning available anywhere at any time. Several Jupyter environments now support collaboration by synchronizing changes in real-time, like Google’s Colaboratory, Amazon’s SageMaker Notebooks, and William Stein’s CoCalc. Cloud computing, Software as a Service, and Infrastructure as a Service have all transformed scientific computing and enabled the democratization of data science. The next section provides an overview of the evolution of the major scientific programming languages from Fortran to Matlab to Python to Julia.

## ► The languages

Fortran (a portmanteau of Formula Translating System) was developed in the 1950s by a team at IBM led by then thirty-year-old John Backus. It came to dominate numerical computation for decades and is still frequently used, both directly in high-performance computing and as subroutines called by other scientific programming languages through wrapper functions. Fortran was the first widely-used, high-level “automatic programming” language, invented when computer code was almost entirely written in machine language or assembly language. While “automatic programming” was met with considerable skepticism at the time, the drop in cost of a computer relative to the cost in salaries for computer scientists, who might spend a significant time debugging code, was a significant driver behind the development of Fortran.

Realizing the need for portable, non-proprietary, mathematical software, researchers at Argonne National Laboratory, funded through the National Science Foundation, developed a set of Fortran libraries in the early 1970s. These libraries included EISPACK for computing eigenvalues and eigenvectors and LINPACK for performing linear algebra. The packages were subsequently expanded into a broader set of numerical software libraries called SLATEC (Sandia, Los Alamos, Air Force Weapons Laboratory Technical Exchange Committee). In the 1980s, MINPACK for solving systems of nonlinear equations, QUADPACK for numerical integration of one-dimensional functions, FFTPACK for the fast Fourier transform, and SLAP for sparse linear algebra, among others, were added to the SLATEC Common Mathematics Library. And in the early 1990s, EISPACK and LINPACK were combined into the general linear algebra package LAPACK. The GNU Scientific Library (GSL), was initiated in the mid-1990s, to provide a modern replacement to SLATEC.

Cleve Moler, one of the developers of LAPACK and EISPACK, created MATLAB (a portmanteau of Matrix Laboratory) in the 1970s to give his students

access to these libraries without needing to know Fortran.<sup>1</sup> You can appreciate how much easier it is to simply write

```
[1 2; 3 4]\[5;6]
```

and have the solution printed automatically, rather than to write in Fortran

```
program main
 implicit none
 external :: sgesv
 real :: A(2, 2)
 real :: b(2)
 real :: pivot(2)
 integer :: return_code

 A = reshape([1., 3., 2., 4.], [2, 2])
 b = [5., 6.]
 call sgesv(2, 1, A, 2, pivot, b, 2, return_code)
 if (return_code == 0) then
 print '(a, 2(f0.4, ", "))', 'solution: ', b
 else
 print '(a, i0)', 'error: ', return_code
 end if
end program main
```

which would then still need to be compiled, linked, and run. MATLAB also made data visualization and plotting easy. While at first MATLAB was little more than an interactive matrix calculator, Moler later developed MATLAB into a full computing environment in the early 1980s. MATLAB's commercial growth and evolution coincided with the introduction of affordable personal computers, which were initially barely powerful enough to run software like MATLAB. Over time, other features like sparse matrix operations and ODE solvers were added to make MATLAB a complete scientific programming language. Still, matrices are treated as the fundamental data type.

GNU Octave, developed by John Eaton and named after his former professor Octave Levenspiel, was initiated in the 1980s and first released in the 1990s as a free, open-source clone of MATLAB. While the syntax of Octave is nearly identical to MATLAB, there are a few differences. In general, Octave allows a little more freedom in syntax than MATLAB, while MATLAB tends to have more and newer features and functions. For example, Octave permits C-style

---

<sup>1</sup>In this book, “MATLAB” indicates the commercial numerical computing environment developed by MathWorks, and “Matlab” indicates the programming language inclusive of Octave. In modern orthography, uppercase often comes across as SHOUTY CAPS, but before the 1960s, limitations in computer memory and printing made uppercase a necessity. In the 1990s, ‘FORTRAN,’ on which MATLAB was based and styled, adopted the naming “Fortran,” and “MATLAB” stayed “MATLAB.”

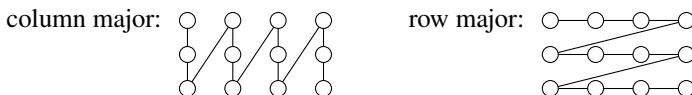
increment operators such as `x++` and addition assignment operators such as `+ =`, but MATLAB does not. MATLAB requires `...` for line continuation, while Octave also allows Python-style backslash for line continuation. Octave allows users to merge assignments such as `x=y=2`. MATLAB uses `~` for a logical negation, while Octave uses either `~` or `!`. And, MATLAB uses `%` to start a comment, while Octave can use either `%` or `#`. While such syntax provides greater programming flexibility, it does create greater compatibility issues between MATLAB and Octave.

Dennis Ritchie developed the C programming language at the Bell Telephone Laboratories in the early 1970s. C was named after B, which itself came from “Basic Combined Programming Language.” C was followed by D, C++, and C#. While not explicitly developed for scientific computing, C has been widely popular and influential in other derivatives like Python. The GNU Scientific Library (GSL) was written in C and has wrappers in many other languages, including Fortran, Python, R, Octave, and Julia.

The R programming language first appeared in the early 1990s as a GNU GPL modern implementation of the S programming language developed by Bell Laboratories in the mid-1970s. The name “S” is a reference to “statistical” and uses the programming language naming convention at Bell Laboratories at the time. The name “R” comes from both the first names of the language’s authors (Ross Ihaka and Robert Gentleman) and from a one-letter nod to S. In the mid-2000s, Hadley Wickham developed tidyverse that helped organize data into more intuitive and more readable syntax and `ggplot2` data visualization package that implemented Leland Wilkinson’s “grammar of graphics” to help users build visualizations. R has since become a favorite among statisticians and data scientists.

Python is a general-purpose programming, created in the early 1990s by Guido van Rossum with a design philosophy that focused on code readability. The name comes from the British comedy troupe Monty Python. While Python was not originally developed for scientific or numerical computing, the language attracted a user base who developed packages for technical computing. A matrix package called Numeric, developed in the mid-1990s, influenced Travis Oliphant to create NumPy the mid-2000s. SciPy was developed to provide tools for technical computing such as algorithms for signal processing, sparse matrices, special functions, optimization, and fast Fourier transforms. Matplotlib, first released in 2003, provides a plotting environment with a syntax familiar to Matlab, particularly through the `pyplot` module. Limited Unicode variable names were introduced in Python 3.0 in 2008, and the matrix multiplication `@` operator was introduced in Python 3.5 in 2015, further improving Python’s mathematical expressiveness. While Python’s improved functionality has made it ubiquitous for scientific computing, it still retains some of its vestigial mathematical clunkiness, such as using `**` instead of `^` for exponentiation and `@` instead of `*` for matrix

multiplication.<sup>2</sup> And while Julia has adopted Matlab’s syntax of the slash and backslash as convenient all-purpose left and right inverse operators, Python’s numpy.linalg module requires calls to specific functions such as `solve(A,b)` and `lstsq(A,b)`. Furthermore, while Fortran, Matlab, Julia, and R all use a column-major order convention for storing and accessing multidimensional arrays in computer memory, Python using NumPy (along with C) uses a row-major convention.



Finally, Python (like C) uses 0-based indexing, whereas many other languages including Fortran, Matlab, R, and Julia use 1-based indexing. Arguments can be made supporting either 0-indexing or 1-indexing. When asked why Python uses 0-based indexing, Guido van Rossum stated “I think I was swayed by the elegance of half-open intervals.”

Julia debuted in the early 2010s. When asked about the inspiration behind Julia’s name, one of the cocreators Stefan Karpinski replied, “There’s no good reason, really. It just seemed like a pretty name.” What Fortran is to the Silent Generation, what Matlab is to Baby Boomers, and what Python is to Generation X, one might say Julia is to Millennials. While Fortran and Matlab are both certainly showing their age, Julia is a true digital native. It’s designed in an era of cloud computing and GPUs. It uses Unicode and emojis that permit more expressive and more readable mathematical notation. The expression `2*pi` in Julia is simply  $2\pi$ , inputted using TeX conventions followed by a tab autocomplete. Useful binary operators, like `≈` for “approximately equal to,” are built-in, and you can define custom binary operators using Unicode symbols. Julia is still young, and its packages are still evolving.

While Fortran and C are compiled ahead of time, and Python and Matlab are interpreted scripts, Julia is compiled just in time (JIT) using the LLVM infrastructure. Because Julia code gets compiled the first time it is run, the first run can be slow.<sup>3</sup> But after that, Julia runs a much fast faster, cached compiled version. Julia has arbitrary precision arithmetic (arbitrary-precision integers and floating-point numbers using GNU Multiple Precision Arithmetic Library (GMP) and the GNU MPFR Library) with `BigInt` and `BigFloat`.

<sup>2</sup>The Python operator `^` is a bit-wise XOR. So  $7^3$  is 4 ( $111 \text{ XOR } 011 = 100$ ).

<sup>3</sup>Even excruciatingly slow when precompiling a libraries. This compile time latency is sometimes referred to as “time to first plot” (TTFP) given the amount of time import Plots.jl. But other packages are often slower. For example, DifferentialEquations.jl can take up to ten minutes on the first run and less than three seconds thereafter.

## B.2 Python

This section provides a Python supplement to the Julia commentary and code in the body of this book. Page references to the Julia commentary are listed in the left margins. For brevity, the following packages are assumed to have been imported as needed:

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.linalg as la
import scipy.sparse as sp
```

We'll also define the following variables:

```
bucket = "https://raw.githubusercontent.com/nmfsc/data/master/"
π = np.pi
```

The Python code in this section is available as a Jupyter notebook at

<https://nbviewer.jupyter.org/github/nmfsc/python/blob/main/python.ipynb>

The Jupyter viewer has a link in its menu to execute the node on Binder, which you can also access using the inner QR link at the bottom of this page. Note that it may take a few minutes for the Binder to launch the repository. Alternatively, you can download the python.ipynb file and run it on a local computer with Jupyter and Python. Or, you can upload the notebook to Google Colaboratory (Colab), Kaggle, or CoCalc, among others. Perhaps the easiest way to run the code is directly using Colab through the outer QR link at the bottom of this page.

- 5 Define the array `x = np.arange(n)`. There are several different ways to create a column vector using NumPy:

```
x[:,None]
x.reshape(n,1)
x.reshape(-1,1)
np.vstack(x)
```

- 5 Python uses `np.array` to create an array and `np.asarray` to cast a list as an array. The operators `+`, `-`, `*`, `/`, and `**` are all element-wise operators and broadcast by implicitly expanding arrays to be of compatible size. For example, by defining `A=np.array([[1,2],[3,4]])`, then `A*np.array([1,2])` is equivalent to `A*np.array([[1,2],[1,2]])`, and `A*np.array([[1],[2]])` is equivalent to `A*np.array([[1,1],[2,2]])`. NumPy uses the operator `@` for matrix multiplication and the operator `^` for matrix power.



- 6 The transpose of  $A$  is produced by  $A.T$ , and the conjugate transpose is produced by  $A.conj().T$ .
- 17 The numpy.linalg function `norm(A,p)` computes the matrix norm of  $A$ , where the default optional argument `p='fro'` returns the Frobenius norm. The function returns the induced  $p$ -norm when  $p$  is a number and the  $\infty$ -norm when `p=np.inf`.

```
23 M = [la.solve(la.hilbert(n),la.hilbert(n)) for n in [10,15,20,25,50]]
fig, ax = plt.subplots(1, 5)
for i in range(len(M)):
 ax[i].imshow(1-np.abs(M[i]),vmin=0,cmap="gray")
plt.tight_layout(); plt.show()
```

- 24 The numpy.linalg function `cond(A,p)` returns the  $p$ -condition number of  $A$ . The command `cond(A)` is equivalent to `cond(A, 2)`, and the  $\infty$ -condition number is given by `cond(A,np.inf)`.
- 29 The `getsource` function from the `inspect` module returns the text of the source code for an object.
- 30 The Jupyter magic command `%timeit` can be used to measure execution time.
- 30 The symbolic math library SymPy command `sympy.Matrix(A).rref()` returns the reduced row echelon form of  $A$ .
- 33 The following function overwrites the arguments  $A$  and  $b$ . Pass array copies of these objects `gaussian_elimination(A.copy(),b.copy())` if you wish to avoid overwriting them.

```
def gaussian_elimination(A,b):
 n = len(A)
 for j in range(n):
 A[j+1:,j] /= A[j,j]
 A[j+1:,j+1:] -= np.outer(A[j+1:,j],A[j,j+1:])
 for i in range(1,n):
 b[i] = b[i] - A[i,:i]@b[:i]
 for i in reversed(range(n)):
 b[i] = (b[i] - A[i,i+1:]@b[i+1:])/A[i,i]
 return b
```

- 42 The following Python code implements the simplex method. We start by defining functions used for pivot selection and row reduction in the simplex algorithm.

```
def get_pivot(tableau):
 j = np.argmax(tableau[-1,:-1]>0)
 a, b = tableau[:-1,j], tableau[:-1,-1]
```

```

k = np.argwhere(a > 0)
i = k[np.argmin(b[k]/a[k])]
return i,j

```

```

def row_reduce(tableau):
 i,j = get_pivot(tableau)
 G = tableau[i,:]/tableau[i,j]
 tableau -= tableau[:,j].reshape(-1,1)*G
 tableau[i,:] = G

```

```

from collections import namedtuple
def simplex(c,A,b):
 m,n = A.shape
 tableau = np.r_[np.c_[A,np.eye(m),b], \
 np.c_[c.T,np.zeros((1,m)),0]]
 while (any(tableau[-1,:n]>0)): row_reduce(tableau)
 p = np.argmax(tableau[-1,:n] != 0)
 x = np.zeros(n)
 for i in p.flatten():
 x[i] = np.dot(tableau[:,i],tableau[:, -1])
 z = -tableau[-1,-1]
 y = -tableau[-1,range(n,n+m)]
 solution = namedtuple("solution",["z","x","y"])
 return solution(z,x,y)

```

- 45 The `scipy.optimize` function `linprog` solves the linear programming problem using 'interior-point' (default), 'revised simplex', and 'simplex' (as a legacy method). Google has also developed a suite of OR-tools. The `ortools.linear_solver` function `pywraplp` is a C++ wrapper for solving linear programming problems.
- 45 The `scipy.sparse.linalg` function `spsolve` uses the scikit-umfpack wrapper of UMPFPACK to solve sparse linear systems.
- 45 The `matplotlib.pyplot` command `spy(A)` returns the sparsity plot of a matrix `A`.
- 48 The NetworkX package provides tools for constructing, analyzing, and visualizing graphs.
- 48 The following code draws the dolphin networks of the Doubtful Sound. We'll use dolphin gray (#828e84) to color the nodes.

```

import pandas as pd, networkx as nx
df = pd.read_csv(bucket+'dolphins.csv', header=None)
G = nx.from_pandas_edgelist(df,0,1)
nx.draw(G,nx.spectral_layout(G),alpha=0.5,node_color='#828e84')
plt.axis('equal'); plt.show()

```

We can change layouts using `nx.spring_layout(A)` and `nx.circular_layout(A)`.

- 52 The function `scipy.sparse.csgraph.reverse_cuthill_mckee` returns the permutation vector using the reverse Cuthill–McKee algorithm for sparse matrices.
- 54 The following code implements the revised simplex method.

```
import scipy.sparse as sp
from collections import namedtuple
def revised_simplex(c,A,b):
 c, A, b = c.astype(float), A.astype(float), b.astype(float)
 m, n = A.shape
 def xi(i): z=sp.lil_matrix((m,1)); z[i] = 1; return z.tocsr()
 N = np.arange(n); B = np.arange(n,n+m)
 A = sp.hstack([sp.csr_matrix(A),sp.identity(m)],format="csr")
 ABinv = sp.identity(m).tocsr()
 b = sp.csr_matrix(b)
 c = sp.vstack([sp.csr_matrix(c),sp.csr_matrix((m,1))])
 while True:
 J = np.argwhere((c[N].T-(c[B].T @ ABinv) @ A[:,N]) > 0)
 if len(J)==0: break
 j = J[0,1]
 q = ABinv @ A[:,N[j]]
 k = np.argwhere(q>0)[:,0]
 i = k[np.argmin(ABinv[k,:] @ b/q[k])]
 B[i], N[j] = N[j], B[i]
 ABinv -= ((q - xi(i))/q[i][0,0]) @ ABinv[i,:]
 i = np.argwhere(B<n)
 x = np.zeros(n)
 for k in i.flatten(): x[B[k]] = (ABinv[k,:] @ b)[0,0]
 y=(c[B].T@ABinv).toarray().flatten()
 solution = namedtuple("solution",["z","x","y"])
 return solution(z=x@c[:n],x=x,y=y)
```

- 57 For a  $2000 \times 1999$  matrix `la.lstsq(A1,b)` takes 3.4 seconds and `la.pinv(A1)@b` takes 6.3 seconds.
- 63 Python does not have a dedicated function for the Givens rotations matrix. Instead, use the `scipy.linalg QR` decomposition `Q,_ = qr([[x],[y]])`.
- 64 The `scipy.linalg` function `qr` implements LAPACK routines to compute the QR factorization of a matrix using Householder reflection.
- 65 The `numpy.linalg` and `scipy.linalg` function `lstsq` solves an overdetermined system using the LAPACK routines `gelsd` (using singular value decomposition) `gelsy` (using QR factorization), or `gelss` (singular value decomposition).

- 66 The Zipf's law coefficients  $c$  for the canon of Sherlock Holmes computed using ordinary least squares are

```
import pandas as pd
data = pd.read_csv(bucket+'sherlock.csv', sep='\t', header=None)
T = np.array(data[1])
n = len(T)
A = np.c_[np.ones((n,1)),np.log(np.arange(1,n+1)[:, np.newaxis])]
B = np.log(T)[:, np.newaxis]
c = la.lstsq(A,B)[0]
```

- 69 The constrained least squares problem is solved using

```
def constrained_lstsq(A,b,C,d):
 x = la.solve(np.r_[np.c_[A.T@A,C.T],
 np.c_[C,np.zeros((C.shape[0],C.shape[0]))]], np.r_[A.T@b,d])
 return x[:A.shape[1]]
```

- 71 The numpy.linalg or scipy.linalg command  $U,S,V=\text{svd}(A)$  returns the SVD decomposition of a matrix  $A$  and  $\text{la.svd}(A,0)$  returns the “economy” version of the SVD. The scipy.sparse.linalg command  $\text{svds}(A,k)$  returns the first  $k$  singular values and associated singular vectors.

- 73 The NumPy function  $\text{la.pinv}$  computes the Moore–Penrose pseudoinverse by computing the SVD using a default tolerance of  $1e-15*\text{norm}(A)$ .

- 77 The following code implements total least squares:

```
def tls(A,B):
 n = A.shape[1]
 ,,V = la.svd(np.c_[A,B])
 return -la.solve(V[n:,n:],V[n:,:n]).T
```

- 77  $A = \text{np.array}([[2,4],[1,-1],[3,1],[4,-8]])$ ;  
 $b = \text{np.array}([[1,1,4,1]]).T$   
 $x_{ols} = \text{la.lstsq}(A,b)[0]$   
 $x_{tls} = \text{tls}(A,b)$

- 78 Python has several libraries (matplotlib, cv2, and PIL) that will import images—we'll use matplotlib.

```
import urllib
f = urllib.request.urlopen(bucket+'red-fox.jpg')
def rgb2gray(rgb): return np.dot(rgb[...,:3], [0.2989,0.5870,0.1140])
A = rgb2gray(plt.imread(f,format='jpg'))
U,s,V = la.svd(A)
```

Take  $k$  to be a value such as 20. Let's evaluate  $\mathbf{A}_k$  and confirm that the error  $\|\mathbf{A} - \mathbf{A}_k\|_F^2$  matches  $\sum_{i=k+1}^n \sigma_i^2$ .

```
Ak = U[:, :k] @ np.diag(sigma[:k]) @ V[:, :]
la.norm(A-Ak, 'fro') - la.norm(sigma[k:])


```

Finally, we'll show the compressed image and plot the error curve.

```
plt.imshow(Ak, cmap=plt.get_cmap('gray'));
plt.show()
r = np.sum(A.shape)/np.prod(A.shape)*range(1,min(A.shape)+1)
error = 1 - np.sqrt(np.cumsum(sigma**2))/la.norm(sigma)
plt.semilogx(r,error,'.-');
plt.show()


```

- 84 The following function is a naïve implementation of NMF:

```
def nmf(X,p=6):
 W = np.random.rand(X.shape[0],p)
 H = np.random.rand(p,X.shape[1])
 for i in range(50):
 W = W*(X@H.T)/(W@(H@H.T) + (W==0))
 H = H*(W.T@X)/((W.T@W)@H + (H==0))
 return (W,H)


```

- 86 The NumPy function `vander` generates a Vandermonde matrix with rows given by  $[x_i^p \quad x_i^{p-1} \quad \dots \quad x_i \quad 1]$  for input  $(x_0, x_1, \dots, x_n)$ .
- 89 The NumPy function `roots` finds the roots of a polynomial  $p(x)$  by computing the eigenvalues of the companion matrix of  $p(x)$ .
- 90 Python does not have a function to compute the eigenvalue condition number. It can be computed using:

```
def condeig(A):
 w, vl, vr = la.eig(A, left=True, right=True)
 c = 1/np.sum(vl*vr, axis=0)
 return (c, vr, w)


```

- 97 The code to compute the PageRank of the graph in Figure 4.3 is

```
H = np.array([[0,0,0,0,1],[1,0,0,0,0], \
 [1,0,0,0,1],[1,0,1,0,0],[0,0,1,1,0]])
v = ~np.any(H,0)
H = H/(np.sum(H,0)+v)
n = len(H)
d = 0.85;
x = np.ones((n,1))/n
for i in range(9):
```

```
x = d*(H@x) + d/n*(v@x) + (1-d)/n
```

- 104 The `scipy.linalg` function `hessenberg` computes the unitarily similar upper Hessenberg form of a matrix.
- 109 Both `numpy.linalg` and `scipy.linalg` have the function `eig` that computes the eigenvalues and eigenvectors of a matrix.
- 114 The `scipy.sparse.linalg` command `eigs` computes several eigenvalues of a sparse matrix using the implicitly restarted Arnoldi process.
- 137 The `scipy.sparse.linalg` function `cg` implements the conjugate gradient method—preconditioned conjugate gradient method if a preconditioner is also provided.
- 140 The `scipy.sparse.linalg` function `gmres` implements the generalized minimum residual method and `minres` implements the minimum residual method.
- 148 The NumPy function `kron(A,B)` returns the Kronecker product  $\mathbf{A} \otimes \mathbf{B}$ .
- 148 The radix-2 FFT algorithm written as a recursive function in Python is

```
def fftx2(c):
 n = len(c)
 ω = np.exp(-2j*π/n);
 if np.mod(n,2) == 0:
 k = np.arange(n/2)
 u = fftx2(c[:-1:2])
 v = (ω**k)*fftx2(c[1::2])
 return np.concatenate((u+v, u-v))
 else:
 k = np.arange(n)[:,None]
 F = ω**(k*k.T);
 return F @ c
```

and the IFFT is

```
def ifftx2(y): return np.conj(fftx2(np.conj(y)))/len(y)
```

- 149 The `scipy.linalg` function `toeplitz` constructs a Toeplitz matrix.
- 150 The `scipy.linalg` function `circulant` constructs a circulant matrix.
- 152 The `scipy.signal` function `convolve` returns the convolution of two n-dimensional arrays using either FFTs or directly, depending on which is faster.
- 152 The following function returns the fast Toeplitz multiplication of the Toeplitz matrix with `c` as its first column and `r` as its first row and a vector `x`. We'll use `fftx2` and `ifftx2` that we developed earlier, although in practice it is much faster

to use the built-in NumPy or SciPy functions `fft` and `ifft` the `scipy.signals` function `convolve`.

```
def fasttoeplitz(c,r,x):
 n = len(x)
 m = (1<<(n-1).bit_length())-n
 x1 = np.concatenate((np.pad(c,(0,m)),r[:1:-1]))
 x2 = np.pad(x,(0,m+n-1))
 return ifftx2(fft2(x1)*fft2(x2))[:n]
```

```
155 def bluestein(x):
 n = len(x)
 ω = np.exp((1j*π/n)*(np.arange(n)**2))
 return ω*fasttoeplitz(conj(ω),conj(ω),ω)
```

- 158 The libraries `numpy.fft` and `scipy.fft` both have several functions for computing FFTs using Cooley–Tukey and Bluestein algorithms. These include `fft` and `ifft` for one-dimensional transforms, `fft2` and `ifft2` for two-dimensional transforms, and `fftn` and `ifftn` for multi-dimensional transforms. When `fft` and `ifft` are applied to multidimensional arrays, the transforms are along the last dimension by default. (In Matlab the transforms are along the first dimension by default.) Similar functions are available to compute FFTs for real inputs and as well as discrete sine and discrete cosine transforms. The PyFFTW package provides a Python wrapper to the FFTW routines, and according to the SciPy documentation: “users for whom the speed of FFT routines is critical should consider installing PyFFTW.”
- 158 The `scipy.fft` functions `fftshift` and `ifftshift` shift the zero frequency component to the center of the array.
- 159 The `scipy.fft` functions `dct` and `dst` return the type 1–4 DCT and DST, respectively.
- 178 The function `type` returns the data type of a variable.
- 178 The following function returns a double-precision floating-point representation as a string of bits:

```
def float_to_bin(x):
 if x == 0: return "0" * 64
 w, sign = (float.hex(x),0) if x > 0 else (float.hex(x)[1:],1)
 mantissa, exp = int(w[4:17], 16), int(w[18:])
 return "{}{:011b}{:052b}".format(sign, exp + 1023, mantissa)
```

- 179 The NumPy function `finfo` returns machine limits for floating-point types. For example, `np.finfo(float).eps` returns the double-precision machine epsilon

$2^{-52}$  and `np.finfo(float).precision` returns 15, the approximate precision in decimal digits.

- 179 The following function implements the Q\_rsqrt algorithm to approximate the reciprocal square root of a number. Because NumPy upcasts to double-precision integers, we'll explicitly recast intermediate results to single-precision.

```
def Q_rsqrt(x):
 i = np.float32(x).view(np.int32)
 i = (0x5f3759df - (i>>1)).astype(np.int32)
 y = i.view(np.float32)
 return y * (1.5 - (0.5 * x * y * y))
```

- 182 `a = 77617; b = 33096  
333.75*b**6+a**2*(11*a**2*b**2-b**6-121*b**4-2)+5.5*b**8+a/(2*b)`

- 182 The `sys` command `sys.float_info.max` returns the largest floating-point number. The command `sys.float_info.min` returns the smallest normalized floating-point number.

- 183 To check for overflows and `NaN`, use the NumPy commands `isinf` and `isnan`. You can use `NaN` to lift the “pen off the paper” in a `matplotlib` plot as in

```
plt.plot(np.array([1,2,2,2,3]),np.array([1,2,np.nan,1,2]));plt.show()
```

- 185 The NumPy functions `expm1` and `log1p` compute  $e^x - 1$  and  $\log(x + 1)$  more precisely than `exp(x)-1` and `log(x+1)` in a neighborhood of zero.

- 189 A Python implementation of the bisection method for a function `f` is

```
def bisection(f,a,b,tolerance):
 while abs(b-a)>tolerance:
 c = (a+b)/2
 if np.sign(f(c))==np.sign(f(a)): a = c
 else: b = c
 return (a+b)/2
```

- 194 The `scipy.optimize` function `fsolve(f,x0)` uses Powell's hybrid method to find the zero of the input function.

- 201 The following function takes the array `bb` for the lower-left and upper-right corners of the bounding box; `xpix` for the number of horizontal pixels; `n` for the maximum number of iterations; and `s` for the starting value  $z^{(0)}$ , which for the Mandelbrot set is 0. The function returns a two-dimensional array `M` that counts the number of iterations  $k$  to escape  $\{z \in \mathbb{C} \mid |z^{(k)}| > 2\}$ .

```

def mandelbrot(bb,xpix,n,s):
 ypix = int(np.round(xpix*(bb[3]-bb[1])/(bb[2]-bb[0])))
 M = np.zeros((ypix,xpix))
 z = s*np.ones((ypix,xpix),dtype=complex)
 c = np.linspace(bb[0],bb[2],xpix).reshape(1,-1) + \
 1j*np.linspace(bb[3],bb[1],ypix).reshape(-1,1)
 for k in range(n):
 mask = np.abs(z)<2
 M[mask] += 1
 z[mask] = z[mask]**2 + c[mask]
 return M

```

The following commands produces image (c) of Figure 8.4 on page 201:

```

import matplotlib.image as mpimg
M = mandelbrot([-0.1710,1.0228,-0.1494,1.0443],800,200,0)
mpimg.imsave('mandelbrot.png', -M, cmap='magma')

```

- 205 The NumPy function `polynomial.polynomial.polyval(x,p)` evaluates a polynomial with coefficients  $p = [a_0, a_1, \dots, a_n]$  at  $x$  using Horner's method.
- 208 The NumPy function `roots` returns the roots of a polynomial by finding the eigenvalues of the companion matrix.
- 214 The following code finds the roots using homotopy continuation:

```

from scipy.integrate import solve_ivp
def f(x): return (np.array([x[0]**3-3*x[0]*x[1]**2-1,
 x[1]**3-3*x[0]**2*x[1]]))
def df(t,x,p):
 A = np.array([[3*x[0]**2-3*x[1]**2,-6*x[0]*x[1]],
 [-6*x[0]*x[1],3*x[1]**2-3*x[0]**2]])
 return la.solve(-A,p)
x0 = np.array([1,1])
sol = solve_ivp(df,[0,1],x0,args=(f(x0),))
sol.y[:,1]

```

- 218 The following code uses the gradient descent method to find the minimum of the Rosenbrock function:

```

def df(x): return np.array([-2*(1-x[0])-400*x[0]*(x[1]-x[0]**2),
 200*(x[1]-x[0]**2)])
x = np.array([-1.8,3.0]); p = np.array([0.0,0.0])
alpha = 0.001; beta = 0.9
for i in range(500):
 p = -df(x) + beta*p

```

```
x += alpha*p
```

In practice, we can use the `scipy.optimize` function `minimize`, which provides several algorithms for finding the minimum of a multivariate function including the Nelder–Mead method and quasi-Newton methods. The `minimize_scalar` function includes the golden section method for finding the minimum of a univariate function.

```
from scipy.optimize import minimize
def f(x): return (1-x[0])**2 + 100*(x[1] - x[0]**2)**2
x0 = np.array([-1.8, 3.0])
x = minimize(f, x0)
```

- 223 The NumPy function `vander` constructs a Vandermonde matrix. Or you can build one yourself with `x**np.arange(n)`.
- 234 The following pair of functions determines the coefficients of a cubic spline with natural boundary conditions given arrays of nodes and then evaluates the spline using those coefficients. Because `C` is tridiagonal matrix, it is more efficient to use a banded Hermitian solver `solveh_banded` from `from scipy.linalg` than the general solver in `numpy.linalg`.

```
def spline_natural(x,y):
 h = np.diff(x)
 gamma = 6*np.diff(np.diff(y)/h)
 C = [h[:-1], 2*(h[:-1]+h[1:])]
 m = np.pad(la.solveh_banded(C,gamma),(1, 1))
 return m
```

```
def evaluate_spline(x,y,m,n):
 h = np.diff(x)
 B = y[:-1] - m[:-1]*h**2/6
 A = np.diff(y)/h-h/6*np.diff(m)
 X = np.linspace(np.min(x),np.max(x),n+1)
 i = np.array([np.argmax(i>=x)-1 for i in X])
 i[-1] = len(x)-2
 Y = (m[i]*(x[i+1]-X)**3 + m[i+1]*(X-x[i])**3)/(6*h[i]) \
 + A[i]*(X-x[i]) + B[i]
 return (X,Y)
```

- 235 The `scipy.interpolate` function `spline` returns a cubic (or any other order) spline. The two steps of first finding the coefficients for an interpolating spline and then evaluating the values of that spline can be broken up into `splprep` and `splev`. Perhaps the easiest approach is with the function `CubicSpline` that returns a cubic

spline interpolant class. The Spline classes in `scipy.interpolate` are wrappers for the Dierckx Fortran library.

- 237 The `scipy.signal` function `bspline(x,p)` evaluates a  $p$ th order B-spline. The `scipy.interpolate` function `Bspline(t,c,p)` constructs a spline using  $p$ th order B-splines with knots  $t$  and coefficients  $c$ .
- 238 The `scipy.interpolate` function `pchip` returns a cubic Hermite spline.
- 240 We can build a Bernstein matrix using the following function which takes a column vector such as  $t = \text{np.linspace}(0,1,50).[:,\text{None}]$ :

```
def bernstein(n,t):
 from scipy.special import comb
 k = np.arange(n+1)[None,:]
 return comb(n,k)*t**k*(1-t)**(n-k)
```

- 251 def legendre(x,n):
 if n==0:
 return np.ones\_like(x)
 elif n==1:
 return x
 else:
 return x\*legendre(x,n-1)-1/(4-1/(n-1)\*\*2)\*legendre(x,n-2)

- 257 We'll construct a Chebyshev differentiation matrix and use the matrix to solve a few simple problems.

```
def chebdiff(n):
 x = -np.cos(np.linspace(0,np.pi,n))[:,None]
 c = np.outer(np.r_[2,np.ones(n-2),2],(-1)**np.arange(n))
 D = c/c.T/(x - x.T + np.eye(n))
 return D - np.diag(np.sum(D,axis=1)), x
```

```
n = 15
D,x = chebdiff(n)
u = np.exp(-4*x**2);
plt.plot(x,D@u,'.-')
```

```
D[0,:] = np.zeros((1,n)); D[0,0] = 1; u[0] = 2
plt.plot(x,la.solve(D,u),'.-'); plt.show()
```

```
n = 15; k2 = 256
D,x = chebdiff(n)
L = D@D - k2*np.diag(x.flatten())
```

```
L[[0,-1],:] = 0; L[0,0] = 1; L[-1,-1] = 1
y = la.solve(L,np.r_[2,[0]*(n-2),1].T)
plt.plot(x,y, 'o');
```

- 259 The ChebPy package, a Python implementation of the Matlab Chebfun package, includes methods for manipulating functions in Chebyshev space.
- 265 The following function returns  $x$  and  $\phi(x)$  of a scaling function with coefficients given by  $c$  and values at integer values of  $x$  given  $z$ :

```
def scaling(c,z,n):
 m = len(c); L = 2**n
 phi = np.zeros(2*m*L)
 phi[0:m*L:L] = z
 for j in range(n):
 for i in range(m*2**j):
 x = (2*i+1)*2**(n-1-j)
 phi[x] = sum([c[k]*phi[(2*x-k*L)% (2*m*L)] for k in range(m)])
 return np.arange((m-1)*L)/L,phi[::(m-1)*L]
```

Let's use `scaling` to plot the Daubechies  $D_4$  scaling function.

```
sqrt3 = np.sqrt(3)
c = np.array([1+sqrt3,3+sqrt3,3-sqrt3,1-sqrt3])/4
z = np.array([0,1+sqrt3,1-sqrt3,0])/2
x,phi = scaling(c,z,8)
plt.plot(x,phi); plt.show()
```

- 267  $\psi = \text{np.zeros\_like}(\phi)$ ;  $n = \text{len}(c)-1$ ;  $L = \text{len}(\phi)/(2*n)$   
 $\text{for } k \text{ in range}(n):$   
 $\quad \psi[k*L:(k+n)*L] += (-1)**k*c[n-k]*\phi[::2]$

- 269 The `scipy.signals` library includes several utilities for wavelet transforms. The PyWavelets (`pywt`) package provides a comprehensive suite of utilities.
- 270 We'll use the PyWavelets package. The two-dimensional Haar DWT of a grayscale image is given by the following code:

```
import pywt, urllib, io
def rgb2gray(rgb): return np.dot(rgb[...,:3], [0.2989,0.5870,0.1140])
def adjustlevels(x): return 1-np.clip(np.sqrt(np.abs(x)),0,1)
f = urllib.request.urlopen(bucket+'laura_square.png')
img = rgb2gray(plt.imread(io.BytesIO(f.read())),format='png')
A = pywt.wavedec2(img,'haar')
c, slices = pywt.coeffs_to_array(A)
plt.imshow(adjustlevels(c), cmap='gray'); plt.axis('off'); plt.show()
```

Let's choose a threshold level of 0.5 and plot the resultant image after taking the inverse DWT.

```
level = 0.5
c = pywt.threshold(c,level)
c = pywt.array_to_coeffs(c,slices,output_format='wavedec2')
B = pywt.waverec2(c,'haar')
plt.imshow(B, cmap='gray'); plt.axis('off'); plt.show()
```

279 We first need a numerically approximation for the Jacobian.

```
def jacobian(f,x,c):
 J = np.empty([len(x), len(c)])
 n = np.arange(len(c))
 for i in n:
 J[:,i] = np.imag(f(x,c+1e-8j*(i==n)))/1e-8
 return J
```

Now we can implements the Levenberg–Marquardt method.

```
def gauss_newton(f,x,y,c):
 r = y - f(x,c)
 for j in range(100):
 G = jacobian(f,x,c)
 M = G.T @ G
 c += la.solve(M + np.diag(np.diag(M)),G.T@r)
 r, r0 = y - f(x,c), r
 if la.norm(r-r0) < 1e-10: return c
 print('Gauss–Newton did not converge.')
```

The problem can be solved using

```
def f(x,c): return (c[0]*np.exp(-c[1]*(x-c[2])**2) +
 c[3]*np.exp(-c[4]*(x-c[5])**2))
x = 8*np.random.rand(100)
y = f(x,np.array([1,3,3,2,3,6])) + np.random.normal(0,0.1,100)
c0 = np.array([2,0.3,2,1,0.3,7])
c = gauss_newton(f,x,y,c0)
```

Assuming that the method converges, we can plot the results using

```
X = np.linspace(0,8,100)
plt.plot(x,y,'.',X,f(X,c)); plt.show()
```

In practice, we can use the `scipy.optimize` function `curve_fit`, which uses the Levenberg–Marquardt method for unconstrained problems.

```
from scipy.optimize import curve_fit
def g(x,c0,c1,c2,c3,c4,c5): return f(x,[c0,c1,c2,c3,c4,c5])
c,_ = curve_fit(g, x, y, c0)
```

- 282 We'll first define the logistic function and generate synthetic data. Then, we apply Newton's method.

```
def σ(x): return 1/(1+np.exp(-x))
x = np.random.rand(30)[:,None]
y = (np.random.rand(30)[:,None] < σ(10*x-5))
```

```
X, w = np.c_[np.ones_like(x),x], np.zeros((2,1))
for i in range(10):
 S = σ(X@w)*(1-σ(X@w))
 w += la.solve(X.T@(S*X), X.T@(y-σ(X@w)))
```

In practice, we can use the statsmodels module to perform logistic regression:

```
import statsmodels.api as sm
results = sm.Logit(y, X).fit()
w = results.params
```

A summary of the results is provided with the command `results.summary()`.

- 284 Let's start by generating some training data and setting up the neural network architecture.

```
N = 100; θ = np.linspace(0,π,N)
x = np.cos(θ); x = np.c_[np.ones(N),x].T
y = np.sin(θ) + 0.05*np.random.randn(1,N)
```

```
n = 20; W1 = np.random.rand(n,2); W2 = np.random.randn(1,n)
def φ(x): return np.maximum(x,0)
def dφ(x): return (x>0)
α = 1e-3
```

Now, we train the model to determine the optimized parameters.

```
for epoch in range(10000):
 ŷ = W2 @ φ(W1@x)
 dLdy = 2*(ŷ-y)
 dLdW1 = dφ(W1@x)* (W2.T@ dLdy) @ x.T
 dLdW2 = dLdy @ φ(W1@x).T
 W1 -= 0.1 * α * dLdW1
 W2 -= α * dLdW2
```

Once we have determined the parameters  $w_1$  and  $w_2$ , we can plot the results.

```
plt.scatter(x[1,:],y,color='#ff000050')
x = np.linspace(-1.2,1.2,200); x = np.c_[np.ones_like(x),x].T
y = w2 @ phi(w1@x)
plt.plot(x[1,:],y[0],color='#000000')
```

Alternatively, we can swap out the activation and loss functions.

```
def phi(x): return 1/(1+np.exp(-x))
def dphi(x): return phi(x)*(1-phi(x))
```

```
L = la.norm(y-h)
dLdy = 2*(y-h)/L
```

Python has several open-source deep learning frameworks. The most popular include TensorFlow (developed by Google), PyTorch (developed by Facebook), and MXNet (developed by Apache). We'll use TensorFlow along with the Keras library, which provides a high-level interface to TensorFlow. First, let's load Keras and TensorFlow and generate some training data.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
n = 20; N = 100;
theta = tf.linspace(0.0,pi,N)
x = tf.math.cos(theta); y = tf.math.sin(theta) + 0.05*tf.random.normal([N])
```

Now, we can define the model, train it, and examine the outputs.

```
model = keras.Sequential(
 [
 layers.Dense(n, input_dim=1, activation='relu'),
 layers.Dense(1)
]
)
model.compile(loss='mean_squared_error', optimizer='SGD')
model.fit(x,y,epochs=2000,verbose=0)
y = model.predict(x)
plt.plot(x,y,color='#000000'); plt.scatter(x,y,color='#ff000050')
```

This example uses the default gradient descent model. We can also choose a specialized optimizer.

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
```

- 293 Coefficients to the third-order approximation to  $f'(x)$  using nodes at  $x - h$ ,  $x$ ,  $x + h$  and  $x + 2h$  are given by  $C[1, :]$  where

```
d = np.array([-1,0,1,2])[:,None]
n = len(d)
V = d**np.arange(n) / [np.math.factorial(i) for i in range(n)]
C = la.inv(V)
```

We can express floating-point numbers as fractions using the following function:

```
from fractions import Fraction
def rats(x): return str(Fraction(x).limit_denominator())
[rats(x) for x in C[1,:]]
```

The coefficients of the truncation are  $C @ d^{**n} / np.math.factorial(n)$ .

- 293 The `fractions` package facilitates arithmetic operations on rational numbers. The command `Fraction(3,4)` returns a representation for  $\frac{3}{4}$ . You can also convert a float to a rational using `x = Fraction(0.75)`, and express it as a string with `str(x)`.
- 294 The Python code for Richardson extrapolation taking  $\delta = \frac{1}{2}$  is

```
def richardson(f,x,m,n):
 if n==0: return phi(f,x,2**m)
 return (4**n*richardson(f,x,m,n-1)-richardson(f,x,m-1,n-1))/(4**n-1)
```

where we define

```
def phi(f,x,n): return (f(x+1/n) - f(x-1/n))/(2/n)
```

- 297 The Python package JAX and the older autograd implement forward and reverse automatic differentiation. We can build a minimal working example of forward accumulation automatic differentiation by defining a class and overloading the base operators:

```
class Dual:
 def __init__(self, value, deriv=1):
 self.value = value
 self.deriv = deriv
 def __add__(u, v):
 return Dual(value(u) + value(v), deriv(u) + deriv(v))
 __radd__ = __add__
 def __sub__(u, v):
 return Dual(value(u) - value(v), deriv(u) - deriv(v))
 __rsub__ = __sub__
 def __mul__(u, v):
 return Dual(value(u)*value(v),
 value(v)*deriv(u) + value(u)*deriv(v))
 __rmul__ = __mul__
```

```
def sin(u):
 return Dual(sin(value(u)),cos(value(u))*deriv(u))
```

And we'll need some helper functions:

```
def value(x):
 return (x.value if isinstance(x, Dual) else x)
def deriv(x):
 return (x.deriv if isinstance(x, Dual) else 0)
def sin(x): return np.sin(x)
def cos(x): return np.cos(x)
def auto_diff(f,x):
 return f(Dual(x)).deriv
```

We can compute the derivative of  $x + \sin x$  at  $x = 0$  using

```
auto_diff(lambda x: x + sin(x),0)
```

298 We can define the dual numbers as

```
x1 = Dual(2,np.array([1,0]))
x2 = Dual(np.pi,np.array([0,1]))
y1 = x1*x2 + sin(x2)
y2 = x1*x2 - sin(x2)
```

299 Python has several packages for automatic differentiation including dedicated libraries like JAX and machine learning frameworks like TensorFlow and PyTorch

302 We can use the following trapezoidal quadrature to make a Romberg method using Richardson extrapolation:

```
def trapezoidal(f,x,n):
 F = f(np.linspace(x[0],x[1],n+1))
 return (F[0]/2 + sum(F[1:-1]) + F[-1]/2)*(x[1]-x[0])/n
```

303 The following code examines the convergence rate of the composite trapezoidal rule using the function  $x + (x - x^2)^p$ :

```
n = np.logspace(1,2,num=10).astype(int)
error = np.zeros((10,7))
def f(x,p): return (x + x**p*(2-x)**p)
for p in range(1,8):
 S = trapezoidal(lambda x: f(x,p),(0,2),10**6)
 for i in range(len(n)):
 Sn = trapezoidal(lambda x: f(x,p),(0,2),n[i])
 error[i,p-1] = abs(Sn - S)/S
np.log(error)
```

```
A = np.c_[np.log(n),np.ones_like(n)]
x = np.log(error)
s = np.linalg.lstsq(A,x,rcond=None)[0][0]
info = ['{}: slope={1:0.1f}'.format(k+1,s[k]) for k in range(7)]
lines = plt.loglog(n,error)
plt.legend(lines, info); plt.show()
```

306 def clenshaw\_curtis(f,n):  
 x = np.cos(np.pi\*np.arange(n+1)/n)  
 w = np.zeros(n+1); w[0:n+1:2] = 2/(1-np.arange(0,n+1,2)\*\*2)  
 return ( 1/n \* np.dot(dctI(f(x)), w) )

```
from scipy.fft import dct
def dctI(f):
 g = dct(f,type=1)
 return (np.r_[g[0]/2, g[1:-1], g[-1]/2])
```

306 from scipy.fft import fft
def dctI(f):
 n = len(f)
 g = np.real(fft(np.r\_[f, f[n-2:0:-1]]))
 return ( np.r\_[g[0]/2, g[1:n-1], g[n-1]]/2 )

312 We can implement Gauss–Legendre quadrature by first defining the weights

```
def gauss_legendre(n):
 a = np.zeros(n)
 b = np.arange(1,n)**2 / (4*np.arange(1,n)**2 - 1)
 scaling = 2
 nodes, v = la.eigh_tridiagonal(a, np.sqrt(b))
 return (nodes, scaling*v[0,:]**2)
```

and then implementing the method as

```
def f(x): return np.cos(x)*np.exp(-x**2)
nodes, weights = gauss_legendre(n)
np.dot(f(nodes), weights)
```

Alternatively, the numpy.polynomial.legendre function leggauss computes nodes and weights for Gauss–Legendre quadrature. The function first computes the nodes  $x_k$  as eigenvalues of Jacobi matrix, followed by one Newton step to refine the calculation.

```
nodes, weights = np.polynomial.legendre.leggauss(n)
np.dot(f(nodes), weights)
```

- 321 The `numpy.random` library includes several distributions. Python’s TensorFlow package also has a large collection in the `distributions` module of the `tensorflow_probability` library.

```
341 r = np.exp(2j*np.linspace(0,1,100))
z = (3/2*r**2 - 2*r + 0.5)/r**2
plt.plot(z.real,z.imag); plt.axis('equal'); plt.show()
```

- 343 The following function determines the coefficients for stencil given by `m` and `n`:

```
def multistepcoeffs(m,n):
 s = len(m) + len(n)
 A = (np.r_[m]+1.0)**np.c_[range(s)]
 B = np.c_[range(s)]*(np.r_[n]+1.0)**(np.c_[range(s)]-1)
 c = la.solve(-np.c_[A,B],np.ones((s,1))).flatten()
 return (np.r_[1,c[:len(m)]], c[len(m):])
```

We’ll use the arrays returned by `multistepcoeffs` to plot the region of stability.

```
def plotstability(a,b):
 r = np.exp(1j*np.linspace(0,2*pi,200))
 z = [np.dot(a,r**np.arange(len(a))) / \
 np.dot(b, r**np.arange(len(b))) for r in r]
 plt.plot(np.real(z),np.imag(z));
 plt.axis('equal'); plt.show()
```

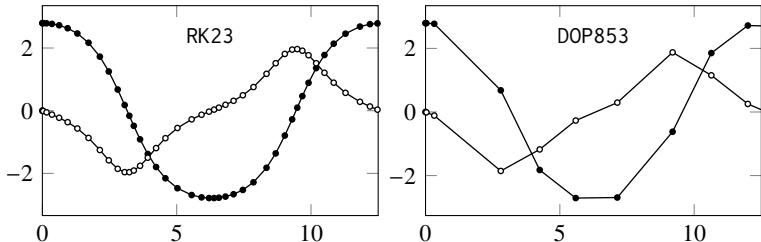
- 369 The Python recipe for solving a differential equation is quite similar to Julia’s recipe, with the exception that the “define the problem” step is merged into the “solve the problem” step. Python does not have the trapezoidal method, so the Bogacki–Shampine method is used instead.

1. Load the modules
2. Set up the parameters
3. Choose the method
4. Solve the problem
5. Present the solution

```
import numpy as np;
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
def pndlm(t,u): return u[1],-np.sin(u[0])
u0 = [8*pi/9,0]; tspan = [0,8*pi]
mthd = 'RK23'
sltn = solve_ivp(pndlm,tspan,u0,method=mthd)
plt.plot(sltn.t,sltn.y[0,:])
plt.plot(sltn.t,sltn.y[1,:])
plt.legend(labels=['θ','ω']); plt.show()
```

If a method is not explicitly stated, Python will use the default solver RK45. By default the values of `sltn.t` are those determined and used for adaptive time stepping. Because such steps can be quite large and far from uniform especially

for high-order methods, plots using `sltn.t` and `sltn.y` may not look smooth. Counterintuitively, higher-order methods such as DOP853 that use smoother interpolating polynomials produce rougher (though still accurate) plots than lower-order methods such as RK23:



We can request a continuous solution by setting the `dense_output` flag to `True`. In this case, `solve_ivp` returns an additional field `sol` that retains information from the method so that the solution later be interpolated between the discrete time steps. In practice, we can think of the field `sol` as a function of the independent variable:

```
sltn = solve_ivp(pndlm, tspan, u0, method=mthd, dense_output=True)
t = np.linspace(tspan[0], tspan[1], 200)
y = sltn.sol(t)
plt.plot(t,y[0],t,y[1])
plt.legend(labels=['θ', 'ω']); plt.show()
```

- 378 We can use `numpy.linalg.solve_banded` as a tridiagonal solver.

```
Δx = .01; Δt = .01; L = 2; c = Δt/Δx**2; uL = 0; uR = 0;
x = np.arange(-L,L,Δx); n = len(x)
u = (abs(x)<1)
u[0] += 2*c*uL; u[n-1] += 2*c*uR;
D = np.tile(np.array([[-c,1+2*c,-c]]).T,(1,n))
D[0,1] = 0; D[2,n-2] = 0
for i in range(20):
 u = la.solve_banded((1, 1), D, u)
```

We can compute the runtime by using the `timeit` package by adding

```
import timeit
start_time = timeit.default_timer()
```

before the block of code and

```
elapsed_time = timeit.default_timer() - start_time
```

after it. This code takes roughly 0.003 seconds on a typical laptop.

- 378 The `scipy.linalg` package has several routines for specialized matrices including `solve_banded` for banded matrices, `solveh_banded` for symmetric, banded matrices, and `solve_circulant` for circulant matrices (by doing division in Fourier space). The `scipy.sparse.linalg` function `spsolve` can be used to solve general sparse systems.
- 379 The following code solves the heat equation using the Crank–Nicolson method. While we can use `numpy.dot` and `numpy.linalg.solve_banded` for tridiagonal multiplication and inverse, let's instead use Python's sparse matrix routines.

```
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
Δx = .01; Δt = .01; L = 2; ν = Δt/Δx**2
x = np.arange(-L,L,Δx); n = len(x)
u = (abs(x)<1)
diagonals = [np.ones(n-1), -2*np.ones(n), np.ones(n-1)]
D = sp.diags(diagonals, [-1,0,1], format='csr')
D[0,1] *= 2; D[-1,-2] *= 2
A = 2*sp.identity(n) + ν*D
B = 2*sp.identity(n) - ν*D
for i in range(20):
 u = spsolve(B,A@u)
plt.plot(x,u); plt.show()
```

- 388 We'll use LSODA solver which automatically switches between Adams–Moulton and BDF routines for stiff nonlinear problems. We can plot the results in a Jupyter notebook as an interactive animation using the `ipywidgets` package.

```
from scipy.integrate import solve_ivp
n = 400; L = 2; x,Δx = np.linspace(-L,L,n,retstep=True)
def m(u): return u**2
def Du(t,u):
 return np.r_[0,np.diff(m((u[:-1]+u[1:])/2)*np.diff(u))/Δx**2,0]
u0 = (abs(x)<1)
sol = solve_ivp(Du, [0,2], u0, method='LSODA',\
 lband=1,uband=1,dense_output=True)
```

```
from ipywidgets import interactive
def anim(t=0):
 plt.fill_between(x,sol.sol(t),color='ff9999');
 plt.ylim(0,1);plt.show()
interactive(anim, t = (0,2,0.01))
```

- 423 PyClaw is a Python-based interface to Clawpack (Conservation Laws package), a collection of finite volume methods for hyperbolic systems of conservation laws. The collection was originally developed in Fortran by Randall LeVeque in the mid 1990s. FiPy is a finite volume PDE solver using Python developed at the National Institute of Standards and Technology.

- 437 The following code solves the heat equation using FEniCS:

```
from fenics import *
from mshr import *
square = Rectangle(Point(-0.5, 0), Point(0.5, 1))
circle = Circle(Point(0.0, 0.0), 1)
mesh = generate_mesh(circle - square, 16)
V = FunctionSpace(mesh, 'P', 1)
u, v, w = TrialFunction(V), TestFunction(V), Function(V)
def boundary(x, on_boundary): return on_boundary
bc = DirichletBC(V, Constant(0), boundary)
f = Expression("exp(-pow(2*x[0]+1,2)-pow(2*x[1]+1,2))", degree=2)
a = dot(grad(u),grad(v))*dx
L = f*v*dx
solve(a == L, w, bc)
File("poisson_solution.pvd") << w
```

- 444 The numpy.fft function fftfreq returns the DFT sample frequencies.

```
ik = 1j*np.fft.fftfreq(n,1/n)*(2*pi/L)
```

The Python implementation of the solution to the heat equation (16.4) is

```
from numpy.fft import fft,ifft, fftfreq
n = 256; l = 4
k2 = (fftfreq(n,1/n)*(2*pi/L))**2
def u(t,u0): return np.real(ifft(np.exp(-k2*t)*fft(u0)))
```

- 455 The Python code to solve the Navier–Stokes equation has three parts: define the functions, initialize the variables, and iterate over time. We start by defining functions for  $\hat{H}$  and for the flux used in the Lax–Wendroff scheme.

```
from numpy.fft import fft2,ifft2,fftfreq
def cdiff(Q,step=1): return Q-np.roll(Q,step,0)
def flux(Q,c): return c*cdiff(Q,1) - \
 0.5*c*(1-c)*(cdiff(Q,1)+cdiff(Q,-1))
def H(u,v,i\xi x,i\xi y): return -i\xi x*fft2(ifft2(u)**2) + \
 -i\xi y*fft2(ifft2(u)*ifft2(v))
```

The variable initialization is designed on equal  $x$  and  $y$  dimensions, but we can easily modify the code for arbitrary domain size.

```

 $\ell, n, \Delta t, \epsilon = 2, 128, 0.001, 0.001; \Delta x = \ell/n$
x = np.linspace($\Delta x, \ell, n$)[None,:]; y = x.T
q = 0.5*(1+np.tanh(10*(1-np.abs($\ell/2 - y$)/($\ell/4$))))
Q = np.tile(q, (1,n))
u = Q*(1+0.5*np.sin($\ell\pi x$))
v = np.zeros_like(u)
u,v = fft2(u),fft2(v)
us,vs = u,v
i ξ x = (1j*fftfreq(n)*n*(2 π/ℓ))[None,:]
i ξ y = i ξ x.T
 $\xi^2 = i\xi x^2 + i\xi y^2$
Hx, Hy = H(u,v,i ξ x,i ξ y), H(v,u,i ξ y,i ξ x)
M1 = 1/ Δt + ($\epsilon/2$)* ξ^2
M2 = 1/ Δt - ($\epsilon/2$)* ξ^2

```

Finally, we iterate on (16.14) and the Lax–Wendroff solver for (16.15).

```

for i in range(1200):
 Q -= flux(Q, ($\Delta t/\Delta x$)*np.real(ifft2(v))) + \
 flux(Q.T, ($\Delta t/\Delta x$)*np.real(ifft2(u)).T).T
 Hxo, Hyo = Hx, Hy
 Hx, Hy = H(u,v,i ξ x,i ξ y), H(v,u,i ξ y,i ξ x)
 us = u - us + (1.5*Hx - 0.5*Hxo + M1*u)/M2
 vs = v - vs + (1.5*Hy - 0.5*Hyo + M1*v)/M2
 $\phi = (i\xi x * us + i\xi y * vs) / (\xi^2 + (\xi^2 == 0))$
 u, v = us - i ξ x* ϕ , vs - i ξ y* ϕ
plt.imshow(Q,'seismic'); plt.show()

```

462 N = 10000; n = np.zeros(20)  
def mat\_01(d): return np.random.choice((0,1), size=(d,d))  
for d in range(20):  
 n[d] = sum([np.linalg.det(mat\_01(d))!=0 for i in range(N)])  
plt.plot(range(1,21),n/N,'.-'); plt.show()

463 An easy way to input  $\mathbf{D}_n$  in Python is with the diag function

```

D = np.diag(np.ones(n-1),1) \
 - 2*np.diag(np.ones(n),0) + np.diag(np.ones(n-1),-1)
la.eig(D)

```

Alternatively, to compute the eigenvalues of a real, symmetric tridiagonal matrix we can use the scipy.linalg function eigh\_tridiagonal.

```
464 def det(A):
 P,L,U = la.lu(A)
 s = 1
 for i in range(len(P)):
 try:
 m = np.argwhere(P[i+1:,i]).item(0)+1
 P[[i,i+m],:] = P[[i+m,i],:]
 s *= -1
 except:
 pass
 return s*np.prod(np.diagonal(U))
```

- 465 The following function implements a naïve reverse Cuthill–McKee algorithm for symmetric matrices:

```
import scipy.sparse as sp
def rcuthillmckee(A):
 r = np.argsort(np.bincount(A.nonzero()[0]))
 while r.size:
 q = np.atleast_1d(r[0])
 r = np.delete(r,0)
 while q.size:
 try: p = np.append(p,q[0])
 except: p = np.atleast_1d(q[0])
 k = sp.find(A[q[0],r])[1]
 q = np.append(q[1:],r[k])
 r = np.delete(r,k)
 return np.flip(p)
```

The Julia command  $A[p,p]$  permutes the rows and columns of  $A$  with the permutation array  $p$ . But, the similar looking Python command  $A[p,p]$  selects elements along the diagonal. Instead, we should take  $A[p[:,None],p]$ . We can test the solution using the following code:

```
A = sp.random(1000,1000,0.001); A += A.T
p = rcuthillmckee(A)
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.spy(A,ms=1); ax2.spy(A[p[:,None],p],ms=1)
plt.show()
```

- 465 We'll reuse the code on page 543.

```
A = nx.adjacency_matrix(G)
p = rcuthillmckee(A)
A = A[p[:,None],p]
G = nx.from_scipy_sparse_matrix(A)
```

```
nx.draw(G,nx.circular_layout(G),alpha=0.5,node_color='#828e84')
plt.axis('equal'); plt.show()
```

- 466 The following code solves the Stigler diet problem using the naïve simplex algorithm developed on page 42.

```
import pandas as pd
diet = pd.read_csv(bucket+'diet.csv')
A = diet.values[1:,3:].
T
b = diet.values[0,3:][:,None]
c = np.ones(((A.shape)[1],1))
food = diet.values[1:,0]
solution = simplex(b,A.T,c)
print("value: ", solution.z)
i = np.argwhere(solution.y!=0).flatten()
print("foods: ", food[i])
```

In practice, we can use the `scipy.optimize` function `linprog`.

```
from scipy import optimize
solution = optimize.linprog(c,-A,-b,method='revised simplex')
solution.fun, food[np.where(solution.x>1e-12)]
```

- 467 Let's first define a couple of helper functions to import the data.

```
import scipy.sparse as sp
def get_names(filename):
 return np.genfromtxt(bucket+filename+'.txt',delimiter='\n',\
 dtype="str",encoding="utf8").tolist()
def get_adjacency_matrix(filename):
 i = np.genfromtxt(bucket+filename+'.csv',delimiter=',',dtype=int)
 return sp.csr_matrix((np.ones_like(i[:,0]), (i[:,0]-1,i[:,1]-1)))
```

We use the `scipy.sparse` command `bmat` to form block matrix. Then we define functions to find the shortest path between nodes a and b.

```
actors = get_names("actors")
movies = get_names("movies")
B = get_adjacency_matrix("actor-movie")
A = sp.bmat([[None,B.T],[B,None]],format='csr')
actormovie = np.r_[actors,movies]

def findpath(A,a,b):
 p = -np.ones(A.shape[1],dtype=np.int64)
 q = [a]; p[a] = -9999; i = 0
 while i<len(q):
```

```

k = sp.find(A[q[i],:])[1]
k = k[p[k]==-1]
q.extend(k)
p[k] = q[i]; i += 1
if any(k==b): return backtrack(p,b)
display("No path.")

```

```

def backtrack(p,b):
 s = [b]; i = p[b]
 while i != -9999: s.append(i); i = p[i]
 return s[::-1]

```

```

a = actors.index("Bruce Lee"); b = actors.index("Tommy Wiseau")
actormovie[findpath(A,a,b)].tolist()

```

In practice, we can use the shortest\_path function from the scipy.sparse.csgraph library or networkx library.

468 The Python code for exercise 3.4 follows:

```

import pandas as pd
df = pd.read_csv(bucket+'filip.csv',header=None)
y,x = np.array(df[0]),np.array(df[1])
coef = pd.read_csv(bucket+'filip-coeffs.csv',header=None)
β = np.array(coef[0])[None,:]

```

We start by defining a function that determines the coefficients and residuals using three different methods and one that evaluate a polynomial given those coefficients.

```

def solve_filip(x,y):
 V = vandermonde(x,11)
 Q,R = la.qr(V,mode='economic')
 c = np.zeros((3,11),float)
 c[0,:] = la.solve(V.T@V,V.T@y)
 c[1,:] = la.solve(R,Q.T@y)
 c[2,:] = la.pinv(V,1e-14)@y
 r = [la.norm(V@c[i,:].T-y) for i in range(3)]
 return (c,r)
def build_poly(c,x):
 return vandermonde(x,len(c))@c
def vandermonde(x,n):
 return np.vander(x,n,increasing=True)

```

Now, we can solve the problem and plot the solutions.

```
b,r = solve_filip(x,y)
X = np.linspace(min(x),max(x),200)
b = np.r_[b,β]
plt.scatter(x,y,color='#0000ff40')
for i in range(4):
 plt.plot(X,build_poly(b[i],X))
plt.ylim(0.7,0.95);plt.show()
coef.assign(β1=b[0], β2=b[1], β3=b[2])
```

Let's also solve the problem and plot the solutions for the standardized data.

```
def zscore(X,x): return (X - x.mean())/x.std()
k1 = np.linalg.cond(vandermonde(x,11))
k2 = np.linalg.cond(vandermonde(zscore(x,x),11))
print("Condition numbers of the Vandermonde matrix:")
print("{:e}".format(k1))
print("{:e}".format(k2))
```

```
c,r = solve_filip(zscore(x,x),zscore(y,y))
plt.scatter(x,y,color='#0000ff40')
for i in range(3):
 Y = build_poly(c[i],zscore(X,x))*y.std() + y.mean()
 plt.plot(X, Y)
plt.show()
la.norm(c[0]-c[1]),la.norm(c[0]-c[2])
```

- 470 We'll use pandas to download CSV file. Pandas is built on top of NumPy, so converting from a pandas datafame to a NumPy array is straightforward.

```
import pandas as pd
df = pd.read_csv(bucket+'dailytemps.csv')
t = pd.to_datetime(df["date"]).values
day = (t - t[0])/np.timedelta64(365, 'D')
u = df["temperature"].values[:,None]
def tempsmodel(t): return np.c_[np.sin(2*π*t), \
 np.cos(2*π*t), np.ones_like(t)]
c = la.lstsq(tempsmodel(day),u)[0]
plt.plot(day,u,'o',color='#0000ff15');
plt.plot(day,tempsmodel(day)@c,'k'); plt.show()
```

- 470 We'll use the Keras library to load the MNIST data. We first compute the sparse SVD of the training set to get singular vectors.

```
from keras.datasets import mnist
import scipy.sparse as sp
```

```
from scipy.sparse.linalg import svds
(image_train,label_train),(image_test,label_test) = mnist.load_data()
image_train = np.reshape(image_train, (60000,-1));
V = np.zeros((12,784,10))
for i in range(10):
 D = sp.csr_matrix(image_train[label_train==i], dtype=float)
 U,S,V[:, :, i] = svds(D,12)
```

Let's plot examples of the principal components of "3."

```
import matplotlib.pyplot as plt
pix = [V[i,:,:3].reshape(28,28) for i in range(11,-1,-1)]
plt.imshow(np.hstack(pix), cmap="gray")
plt.axis('off'); plt.show()
```

We predict the best digit associated with each test image and build a confusion matrix to check the accuracy of the method.

```
image_test = np.reshape(image_test, (10000,-1));
r = np.zeros((10,10000))
for i in range(10):
 q = V[:, :, i].T @ (V[:, :, i] @ image_test.T) - image_test.T
 r[i, :] = np.sum(q**2, axis=0)
prediction = np.argmin(r, axis=0)
confusion = np.zeros((10,10)).astype(int)
for i in range(10):
 confusion[i, :] = np.bincount(prediction[label_test==i], minlength=10)
```

Finally, we can view the confusion matrix as a dataframe.

```
import pandas as pd
pd.DataFrame(confusion)
```

- 471 We use singular value decomposition to find a low-dimensional subspace relating actors and genres. Then we find the closest actors in that subspace using cosine similarity. We'll use the helper functions `get_names` and `get_adjacency_matrix` from page 566.

```
actors = get_names("actors")
genres = get_names("genres")
A = get_adjacency_matrix("movie-genre"); A /= A.sum(axis=0)
B = get_adjacency_matrix("actor-movie")
```

```
from scipy.sparse.linalg import svds
U,S,Vt = svds(A@B, 12)
Q = Vt/np.sqrt((Vt**2).sum(axis=0))
```

```

q = Q[:,actors.index("Steve Martin")]
z = Q.T @ q
r = np.argsort(-z)
[actors[i] for i in r[:10]]

```

```

p = (U*S) @ q
r = np.argsort(-p)
[(genres[i],p[i]/p.sum()) for i in r[:10]]

```

473    `xyt = np.array([[3,3,12],[1,15,14],[10,2,13],[12,15,13],[0,11,12]])
reference = xyt[0,:]; xyt = xyt - reference
A = np.array([2,2,-2])*xyt
b = (xyt**2)@np.array([[1],[1],[-1]])
x_ols = la.lstsq(A,b)[0] + reference[:,None]
x_tls = tls(A,b) + reference[:,None]`

473    `E = [la.eigvals(np.random.randn(n,n)) for i in range(2500)]
E = np.concatenate(E)
plt.plot(E.real, E.imag,'.',c='#0000ff10',mec='none')
plt.axis('equal'); plt.show()`

474    `def rayleigh(A,x=[]):
 n = len(A)
 if x==[]: x = np.random.randn(n,1)
 while True:
 x = x/la.norm(x)
 p = x.T @ A @ x
 M = A - p*np.eye(n)
 if abs(la.det(M))<1e-10:
 return (p,x)
 x = la.solve(M,x)`

474 We'll define a function for implicit QR method and verify it on a matrix with known eigenvalues.

```

def implicitqr(A):
 tolerance = 1e-12
 n = len(A)
 H = la.hessenberg(A)
 while True:
 if abs(H[n-1,n-2]) < tolerance:
 n -= 1

```

```

if n<2: return np.diag(H)
Q,_ = la.qr([[H[0,0]-H[n-1,n-1]], [H[1,0]]])
H[:2,:n] = Q @ H[:2,:n]
H[:n,:2] = H[:n,:2] @ Q.T
for i in range(1,n-1):
 Q,_ = la.qr([[H[i,i-1]], [H[i+1,i-1]]])
 H[i:i+2,:n] = Q @ H[i:i+2,:n]
 H[:n,i:i+2] = H[:n,i:i+2] @ Q.T

```

```

n = 20; S = np.random.randn(n,n);
D = np.diag(np.arange(1,n+1)); A = S @ D @ la.inv(S)
implicitqr(A)

```

- 475 We'll use the helper functions `get_names` and `get_adjacency_matrix` from page 566.

```

import scipy.sparse as sp
actors = get_names("actors")
B = get_adjacency_matrix("actor-movie")
r,c = (B.T@B).nonzero()
M = sp.csr_matrix((np.ones(len(r)),(r,c)))
v = np.ones(M.shape[0])
for k in range(10):
 v = M@v; v /= np.linalg.norm(v)
r = np.argsort(-v)
[actors[i] for i in r[:10]]

```

- 476 The following function approximates the SVD by starting with a set of  $k$  random vectors and performing a few steps of the naïve QR method to generate a  $k$ -dimensional subspace that is relatively close to the space of dominant singular values:

```

def randomizedsvd(A,k):
 Z = np.random.rand(A.shape[1],k)
 Q,R = la.qr(A@Z, mode='economic')
 for i in range(4):
 Q,R = la.qr(A.T @ Q, mode='economic')
 Q,R = la.qr(A @ Q, mode='economic')
 W,S,Vt = la.svd(Q.T @ A,full_matrices=False)
 U = Q @ W
 return (U,S,Vt)

```

Let's convert an image to an array, compute its randomized SVD, and then display the original image side-by-side with the rank-reduced version.

```

import urllib, PIL
f = urllib.request.urlopen(bucket+'red-fox.jpg')
A = rgb2gray(plt.imread(f,format='jpg'))
U, S, Vt = randomizedsvd(A,10);
img = np.c_[A, np.minimum(np.maximum((U*S)@Vt,0),255)]
PIL.Image.fromarray(np.int8(img), mode='L')

```

```

477 from scipy.sparse.linalg import svds
m = 500
A = np.array([[1/((i+j+1)*(i+j+2)/2 - j)
 for i in range(m)] for j in range(m)])
svds(A, 1)[1][0]

```

477 The following functions solve the Poisson equation:

```

import scipy.sparse as sp
n = 50; ξ = np.arange(1,n+1)/(n+1); Δx = 1/(n+1)
I = sp.identity(n)
D = sp.diags([1, -2, 1], [-1, 0, 1], shape=(n, n))
A = (sp.kron(sp.kron(D,I),I) + sp.kron(I,sp.kron(D,I)) +
 sp.kron(I,sp.kron(I,D)))/Δx**2
f = np.array([(x-x**2)*(y-y**2) + (x-x**2)*(z-z**2)+(y-y**2)*(z-z**2)
 for x in ξ for y in ξ for z in ξ])
ue = np.array([(x-x**2)*(y-y**2)*(z-z**2)/2
 for x in ξ for y in ξ for z in ξ])

```

```

from scipy.sparse.linalg import spsolve
def stationary(A,b,ω=0,n=400):
 ε = []; u = np.zeros_like(b)
 P = sp.diags(A.diagonal(),0) + ω*sp.tril(A,-1)
 for i in range(n):
 u += spsolve(P,b-A@u,'NATURAL')
 ε = np.append(ε,la.norm(u - ue,1))
 return ε

```

```

def conjugategradient(A,b,n=400):
 ε = []; u = np.zeros_like(b)
 r = b - A@u; p = np.copy(r)
 for i in range(n):
 Ap = A@p
 α = np.dot(r,p)/np.dot(Ap,p)
 u += α*p; r -= α*Ap
 β = np.dot(r,Ap)/np.dot(Ap,p)
 p = r - β*p

```

```

 ε = np.append(ε,la.norm(u - ue,1))
 return ε

ε = np.zeros((400,4))
ε[:,0] = stationary(A,-f,0)
ε[:,1] = stationary(A,-f,1)
ε[:,2] = stationary(A,-f,1.9)
ε[:,3] = conjugategradient(A,-f)
plt.semilogy(ε);
plt.legend(["Jacobi","Gauss-Seidel","SOR","Conj. Grad."]); plt.show()

```

The three stationary methods take about 13, 34, and 34 seconds, respectively, to complete 400 iterations. In contrast, the conjugate gradient method takes around 1.2 seconds to complete the same number of iterations—just 0.5 seconds to reach machine precision. The SciPy routine `spsolve` uses approximate minimum degree column ordering as the default, which would destroy the lower triangular structure. Instead, `stationary` sets the option `perm_c_spec = 'NATURAL'` to maintain the natural ordering. While SciPy has a dedicated sparse triangular matrix routine `spsolve_triangular`, it is excruciatingly slow, taking almost a second to complete just one iteration of the stationary method! And, directly solving the problem using `spsolve(A,-f)` takes over 12 minutes.

- 479 Python doesn't have a convenient function for generating primes. We could write a function that implements the Sieve of Eratosthenes, but it will be easier to import a list of the primes (pregenerated using Julia).

```

import scipy.sparse as sp
from scipy.sparse.linalg import cg
n = 20000
d = 2 ** np.arange(15); d = np.r_[-d,d]
P = sp.diags(np.loadtxt(bucket+"primes.csv"))
B = [np.ones(n - abs(d)) for d in d]
A = P + sp.diags(B, d)
b = np.zeros(n); b[0] = 1
cg(A, b, M=P)[0][0]

```

- 480 Python code for the radix-3 FFT in exercise 6.1 is

```

def fftx3(c):
 n = len(c)
 ω = np.exp(-2j*π/n);
 if np.mod(n,3) == 0:
 k = np.arange(n/3)
 u = np.stack((fftx3(c[:-2:3]),\
 ω**k * fftx3(c[1:-1:3]),\
 ω**k * fftx3(c[2::3])))

```

```

F = np.exp(-2j*pi/3)** \
 np.array([[0,0,0],[0,1,2],[0,2,4]])
return (F @ u).flatten()
else:
 k = np.arange(n)[:,None]
 F = omega**(k*k.T);
 return F @ c

```

481 Python code for exercise 6.2 is as follows

```

def multiply(p_,q_):
 from scipy.signal import fftconvolve
 p = np.flip(np.array([int(i) for i in list(p_)]))
 q = np.flip(np.array([int(i) for i in list(q_)]))
 pq = np.rint(fftconvolve(p,q)).astype(int)
 pq = np.r_[pq,0]
 carry = pq//10
 while (np.any(carry)):
 pq -= carry*10
 pq[1:] += carry[:-1]
 carry = pq//10
 return ''.join([str(i) for i in np.flip(pq)]).lstrip('0')

```

We can alternatively use the numpy.fft commands `rfft` and `irfft`. We just need ensure that the zero-padded array has even length  $n$ . In this case, we use

```

from numpy.fft import rfft,irfft
m = (len(p)+len(q)) + (len(p)+len(q))%2
pq = np.round(irfft(rfft(p,m)*rfft(q,m))).astype(int)

```

in the place of `pq = convolve(p,q)`. Note that Python uses arbitrary-precision integers, so we can simply multiply the numbers directly. To multiply integers, Python uses the recursive Karatsuba algorithm, which is significantly faster than grade school multiplication but still slower than the Schönhage–Strassen algorithm for larger numbers.

482 Unlike in Julia and Matlab, the SciPy `fft` and `ifft` functions operate on the last dimension by default. So, we'll need to tell these functions to use the first dimension.

```

from scipy.fft import fft,ifft
def dct(f):
 n = f.shape[0]
 omega = np.exp(-0.5j*pi*np.arange(n)/n).reshape(-1,1)
 i = [*range(0,n,2),*range(n-1-n%2,0,-2)]
 return np.real(omega*fft(f[i,:],axis=0))

```

```
def idct(f):
 n = f.shape[0]
 ω = np.exp(-0.5j*π*np.arange(n)/n).reshape(-1,1)
 i = [n-(i+1)//2 if i%2 else i//2 for i in range(n)]
 f[0,:] = f[0,:]/2
 return np.real(ifft(f/ω, axis=0))[i,:]*2
```

The two-dimensional DCT and inverse DCT are

```
def dct2(f): return dct(dct(f.T).T)
def idct2(f): return idct(idct(f.T).T)
```

```
import urllib
f = urllib.request.urlopen(bucket+'red-fox.jpg')
image = plt.imread(f, format='jpg')
A = np.dot(image[...,:3], [0.2989, 0.5870, 0.1140])
B = dct2(A)[:50,:50]
C = idct2(np.pad(B, ((0,A.shape[0]-50), (0,A.shape[1]-50))))
plt.figure(figsize=(8, 6), dpi=160)
plt.imshow(np.c_[A,C], cmap=plt.get_cmap('gray'))
plt.axis('off'); plt.show()
```

486 We'll find the solution to the system of equations in exercise 8.11. Let's first define the system and the gradient.

```
def f(x,y): return (np.array([(x**2+y**2)**2-2*(x**2-y**2),
 (x**2+y**2-1)**3-x**2*y**3]))
def df(x,y): return(np.array([
 [4*x*(x**2+y**2-1), 4*y*(x**2+y**2+1)],
 [6*x*(x**2+y**2-1)**2-2*x*y**3, \
 6*y*(x**2+y**2-1)**2-3*x**2*y**2]]))
```

The homotopy continuation method is

```
def homotopy(f,df,x):
 from scipy.integrate import solve_ivp
 def dxdt(t,x,p): return(la.solve(-df(x[0],x[1]),p))
 sol = solve_ivp(dxdt,[0,1],x,args=(f(x[0],x[1]),))
 return sol.y[:, -1]
```

and Newton's method is

```
def newton(f,df,x):
 for i in range(100):
 Δx = -la.solve(df(x[0],x[1]),f(x[0],x[1]))
 x += Δx
 if (la.norm(Δx) < 1e-8): return x
```

487 We'll write a function that computes point addition and point doubling.

```
def addpoint(P,Q):
 a = 0
 r = (1<<256) - (1<<32) - 977
 if P[0] == Q[0]:
 d = pow(2*P[1], -1, r)
 λ = ((3*pow(P[0],2,r)+a)*d) % r
 else:
 d = pow(Q[0] - P[0], -1, r)
 λ = ((Q[1] - P[1])*d) % r
 x = (pow(λ,2,r) - P[0] - Q[0]) % r
 y = (-λ*(x - P[0]) - P[1]) % r
 return [x,y]
```

Now, we can implement the double-and-add algorithm.

```
def isodd(m): return ((m&1)==1)
def dbl_add(m,P):
 if m>1:
 Q = dbl_add(m>>1,P)
 return addpoint(addpoint(Q,Q),P) if isodd(m) else addpoint(Q,Q)
 else:
 return P
```

Finally, we test the algorithm using Alice and Bob.

```
Px = int("79BE667EF9DCBBAC55A06295CE87"\n + "0B07029BFCDB2DCE28D959F2815B16F81798",16)
Py = int("483ADA7726A3C4655DA4FBFC0E11"\n + "08A8FD17B448A68554199C47D08FFB10D4B8",16)
P = [Px,Py]
m, n = 1829442, 3727472
mP = dbl_add(m,P)
nP = dbl_add(n,mP)
```

491 The following function modifies `spline_natural` on page 551 to compute the coefficients  $\{m_0, m_1, \dots, m_{n-1}\}$  for a spline with periodic boundary conditions:

```
def spline_periodic(x,y):
 h = np.diff(x)
 d = 6*np.diff(np.diff(np.r_[y[-2],y])/np.r_[h[-1],h])
 α = h[:-1]
 β = h + np.r_[h[-1],h[:-1]]
 C = np.diag(β)+np.diag(α,1)
 C[0,-1]=h[-1]; C += C.T
 m = la.solve(C,d)
 return np.r_[m,m[0]]
```

Now we can compute a parametric spline interpolant with  $nx \times n$  points through a set of  $n$  random points using the function `evaluate_spline` defined on page 551:

```
n, nx = 10, 20
x, y = np.random.rand(n), np.random.rand(n)
x, y = np.r_[x,x[0]], np.r_[y,y[0]]
t = np.cumsum(np.sqrt(np.diff(x)**2+np.diff(y)**2))
t = np.r_[0,t]
T,X = evaluate_spline(t,x,spline_periodic(t,x),nx*n)
T,Y = evaluate_spline(t,y,spline_periodic(t,y),nx*n)
plt.plot(X,Y,x,y,'o'); plt.show()
```

492 The following code computes and plots the interpolating curves:

```
n = 20; N = 200
x = np.linspace(-1,1,n)[:,None]
X = np.linspace(-1,1,N)[:,None]
y = (x>0)

def phi1(x,a): return abs(x-a)**3
def phi2(x,a): return np.exp(-20*(x-a)**2)
def phi3(x,a): return x**a
def interp(phi,a):
 return phi(X,a.T)@la.solve(phi(x,a.T),y)
```

```
Y1 = interp(phi1,x)
Y2 = interp(phi2,x)
Y3 = interp(phi3,np.arange(n))
plt.plot(x,y,X,Y1,X,Y2,X,Y3)
plt.ylim((-0.5,1.5)); plt.show()
```

493 We define a function to solve linear boundary value problems.

```
def solve(L,f,bc,x):
 h = x[1]-x[0]
 S = np.array([[1,-1/2,1/6],[-2,0,2/3],[1,1/2,1/6]]) \
 /np.array([h**2,h,1])
 S = np.r_[np.zeros((1,3)),L(x)@S.T,np.zeros((1,3))]
 d = np.r_[bc[0], f(x), bc[1]]
 A = np.diag(S[1:,0],-1) + np.diag(S[:,1]) + np.diag(S[:-1,2],1)
 A[0,:3] , A[-1,-3:] = np.array([1,4,1])/6 , np.array([1,4,1])/6
 return la.solve(A,d)
```

Next, we define a function that will interpolate between collocation points.

```

def build(c,x,N):
 X = np.linspace(x[0],x[-1],N)
 h = x[1] - x[0]
 i = (X // h).astype(int)
 i[-1] = i[-2]
 C = np.c_[c[i],c[i+1],c[i+2],c[i+3]]
 B = lambda x: np.c_[(1-x)**3, 4-3*(2-x)*x**2, \
 4-3*(1+x)*(1-x)**2, x**3]/6
 Y = np.sum(C*B((X-x[i])/h),axis=1)
 return (X,Y)

```

Now, we can solve the Bessel equation.

```

from scipy.special import jn_zeros, j0
n = 15; N = 141
L = lambda x: np.c_[x,np.ones_like(x),x]
f = lambda x: np.zeros_like(x)
b = jn_zeros(0,4)[-1]
x = np.linspace(0,b,n)
c = solve(L,f,[1,0],x)
X,Y = build(c,x,N)
plt.plot(X,Y,X,j0(X)); plt.show()

```

Finally, we examine the error and convergence rate.

```

N = 10*2**np.arange(6)
epsilon = []
for n in N:
 x = np.linspace(0,b,n)
 c = solve(L,f,[1,0],x)
 [X,Y] = build(c,x,n)
 epsilon.append(np.linalg.norm(Y-j0(X))/n)
plt.loglog(N,epsilon,'.-'); plt.show()

```

```

from numpy.polynomial.polynomial import polyfit
s = polyfit(np.log(N),np.log(epsilon),1)[1]
print("slope = " + "{:4.4f}".format(s))

```

496 Let's compute the fractional derivatives of  $e^{-16x^2}$  and  $x(1 - |x|)$ .

```

from numpy.fft import fft,ifft,fftshift
def f(x): return np.exp(-16*x**2)
def f(x): return x*(1-np.abs(x))
n = 2000; l = 2
x = np.arange(n)/n*l-l/2
xi = fftshift(np.arange(n)-n/2)*2*pi/l

```

We can use Jupyter widgets to create an interactive plot of the derivatives.

```
from ipywidgets import interactive
def anim(derivative=0):
 u = np.real(ifft((1j*x)**derivative*fft(f(x))))
 plt.plot(x,u,color='k'); plt.show()
interactive(anim, derivative = (0,2,0.01))
```

497 We can define the model architecture in Keras as

```
import tensorflow as tf
from tensorflow import keras
from keras.layers import Conv2D, AvgPool2D, Dense, Flatten
model = keras.models.Sequential([
 Conv2D(6,5,activation='tanh',padding='same',input_shape=(28,28,1)),
 AvgPool2D(),
 Conv2D(16, 5, activation='tanh'),
 AvgPool2D(),
 Conv2D(120, 5, activation='tanh'),
 Flatten(),
 Dense(84, activation='tanh'),
 Dense(10, activation='sigmoid')
])
```

Keras provides a summary of the model.

```
model.build(); model.summary()
```

Let's load the MNIST training and test data. We'll convert each  $28 \times 28$ -pixel image into a  $28 \times 28 \times 1$ -element floating-point array with values between zero and one.

```
from keras.datasets import mnist
(image_train,label_train),(image_test,label_test) = mnist.load_data()
image_train = tf.expand_dims(image_train/255.0, 3)
image_test = tf.expand_dims(image_test/255.0, 3)
```

We compile the model, defining the loss function, the optimizer, and the metrics.

```
loss = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer="adam", loss=loss, metrics=["accuracy"])
```

Now, we can train the model.

```
model.fit(image_train, label_train, epochs=5)
```

Finally, let's see how well the model works on an independent set of similar data.

```
model.evaluate(image_test,label_test)
```

The LeNet-5 model has about 2 percent error rate. With more epochs, it has about 1 percent error rate.

```
499 d = np.array([0,1,2,3])[:,None]; n = len(d)
factorial = [np.math.factorial(i) for i in range(n+1)]
V = d**np.arange(n) / factorial[:-1]
C = la.inv(V)
C = np.c_[C,C@d**n/factorial[-1]]
```

The coefficients of the finite difference approximation of the derivative and coefficient of the truncation error are given by `rats(C[1, :])` where the function `rats` is defined on page 557.

```
500 def richardson(f,x,m):
 D = np.zeros(m)
 for i in range(m):
 D[i] = phi(f,x,2***(i+1))
 for j in range(i-1,-1,-1):
 D[j] = (4***(i-j)*D[j+1] - D[j])/(4***(i-j) - 1)
 return D[1]
```

500 We'll extend the Dual class on page 557 by adding methods for division, cosine, and square root to the class definition.

```
def __truediv__(u, v):
 return Dual(value(u) / value(v),
 (value(v)*deriv(u)-value(u)*deriv(v))/(value(v)*value(v)))
__rtruediv__ = __truediv__
def cos(u):
 return Dual(cos(value(u)), -1*sin(value(u))*deriv(u))
def sqrt(u):
 return Dual(sqrt(value(u)), deriv(u)/(2*sqrt(value(u))))
```

We also define the helper functions

```
def cos(u): return np.cos(u)
def sqrt(u): return np.sqrt(u)
```

Note that in the definition of `cos(u)` we multiplied by the `-1` instead of using a unitary negative operator, because we haven't defined such an operator for the Dual class. Now, we can implement Newton's method.

```
def get_zero(f,x):
 epsilon = 1e-14; delta = 1
```

```

while abs(δ) > ε:
 fx = f(Dual(x))
 δ = value(fx)/deriv(fx)
 x -= δ
return x

```

```
get_zero(lambda x: 4*sin(x) + sqrt(x), 4)
```

To find a minimum or maximum of  $f(x)$ , we substitute the following two lines into the Newton solver:

```

fx = f(Dual(Dual(x)))
δ = deriv(value(fx))/deriv(deriv(fx))

```

501 def cauchyderivative(f, a, p, n = 20, ε = 0.1):
 ω = np.exp(2\*π\*1j\*np.arange(n)/n)
 return np.math.factorial(p)/(n\*ε\*\*p)\*sum(f(a+ε\*ω)/ω\*\*p)

```
f = lambda x: np.exp(x)/(np.cos(x)**3 + np.sin(x)**3)
cauchyderivative(f, 0, 6)
```

502 The following function computes the nodes and weights for Gauss–Legendre quadrature by using Newton’s method to find the roots of  $P_n(x)$ :

```

def gauss_legendre(n):
 x = -np.cos((4*np.arange(n)+3)*π/(4*n+2))
 Δ = np.ones_like(x)
 dP = 0
 while(max(abs(Δ))>1e-16):
 P0, P1 = x, np.ones_like(x)
 for k in range(2,n+1):
 P0, P1 = ((2*k - 1)*x*P0-(k-1)*P1)/k, P0
 dP = n*(x*P0 - P1)/(x**2-1)
 Δ = P0 / dP
 x -= Δ
 return (x, 2/((1-x**2)*dP**2))

```

502  $\xi, w = \text{np.polynomial.hermite.hermgauss}(40)$   
 $\text{def } u0(x): \text{return np.sin}(x)$   
 $\text{def } u(t,x):$   
 $\text{ return [np.dot}(w,u0(x-2*np.sqrt(t)*ξ)/np.sqrt(π)) \text{ for } x \text{ in } x]$   
 $x = \text{np.linspace}(-12,12,100)$   
 $\text{plt.plot}(x,u(1,x)); \text{plt.show}()$

502 Let's define a general function

```
def mc_pi(n,d,m):
 return sum(sum(np.random.rand(d,n,m)**2)<1)/n**2*d
```

that computes the volume of an  $d$ -sphere using  $n$  samples repeated  $m$  times. We can verify the convergence rate of by looping over several values of  $n$ .

```
m = 20; error = []
N = 2**np.arange(20)
error = [sum(abs(pi - mc_pi(n,2,m)))/m for n in N]
plt.loglog(N,error,marker=".",linestyle="None")
s = np.polyfit(np.log(N),np.log(error),1)
plt.loglog(N,np.exp(s[1])*N**s[0])
plt.show()
```

504 The following code plots the region of absolute stability for a Runge–Kutta method with tableau A and b:

```
N = 100; n = b.shape[1]
r = np.zeros((N,N))
E = np.ones((n,1))
x,y = np.linspace(-4,4,N),np.linspace(-4,4,N)
for i in range(N):
 for j in range(N):
 z = x[i] + 1j*y[j]
 r[j,i] = abs(1 + z*b@(la.solve(np.eye(n) - z*A,E)))
plt.contour(x,y,r,[1]); plt.show()
```

```
i = np.arange(4)[:,None]
def factorial(k): return np.cumprod(np.r_[1,range(1,k)])
c1 = la.solve(((i)**i.T/factorial(4)).T,np.array([0,1,0,0]))
c2 = la.solve(((i+1)**i.T/factorial(4)).T,np.array([1,0,0,0]))
```

507 The following function returns the orbit of points in the complex plane for an  $n$ th order Adams–Bashforth–Moulton PE(CE) $^m$ . It calls the function `multistepcoeffs` defined on page 560.

```
def PECE(n,m):
 _,a = multistepcoeffs([1],range(1,n+1))
 _,b = multistepcoeffs([1],range(0,n+1))
 def c(r): return np.r_[r-1,\n np.full(m, r + (b[1:] @ r**np.arange(1,n+1))/b[0]),\n (a @ r**np.arange(1,n+1))/b[0]]
 z = []
 for r in np.exp(1j*np.linspace(0,2*pi,200)):
 z = np.append(z,np.roots(np.flip(c(r)))/b[0])
```

```

 return z

for i in range(5):
 z = PECE(4,i)
 plt.scatter(np.real(z),np.imag(z),s=0.5)
 plt.axis('equal'); plt.show()

```

- 508 The `scipy.interpolate` library has a function `pade(a, m, n)` to compute a Padé approximant from Taylor polynomial coefficients. For the purpose of instruction, let's write our own:

```

def pade(a,m,n):
 A = np.eye(m+n+1);
 for i in range(n): A[i+1:,m+i+1] = -a[:m+n-i]
 pq = la.solve(A,a[:m+n+1])
 return pq[:m+1], np.r_[1,pq[m+1:]]

```

Now, compute the coefficients of  $P_m(x)$  and  $Q_n(x)$  for Taylor polynomial approximation of  $\log(x + 1)$ .

```

m = 3; n = 2
a = np.r_[0, (-1)**np.arange(m+n)/(1+np.arange(m+n))]
p,q = pade(a,m,n)

```

Finally, shift and combine the coefficients using upper inverse Pascal matrices.

```

def S(n): return la.invpascal(n+1, kind='upper')
S(m)@p, S(n)@q

```

- 508 The solution to the SIR problem is

```

from scipy.integrate import solve_ivp
def SIR(t,u,beta,gamma): return (-beta*u[0]*u[1],beta*u[0]*u[1]-gamma*u[1],gamma*u[1])
sol = solve_ivp(SIR, [0, 15], [0.99, 0.01, 0],\
 args=(2,0.4), dense_output=True)
t = np.linspace(0,15,200); u = sol.sol(t)
plt.plot(t,u[0,:],t,u[1,:],t,u[2,:]); plt.show()

```

We can add the optional argument `t_eval=np.linspace(0,15,100)` to `solve_ivp` to evaluate the solution at additional points for a smoother plot.

- 509 The solution to the Duffing equation is

```

from scipy.integrate import solve_ivp
def duff(t,x,g): return(x[1],-g*x[1]+x[0]-x[0]**3+0.3*np.cos(t))
sol = solve_ivp(duff,[0,200], [1, 0], args=(0.37,),\
 method='DOP853',dense_output=True)

```

```
t = np.linspace(0,200,2000); y = sol.sol(t)
plt.plot(y[0,:],y[1,:]); plt.show()
```

- 510 The following code solves the Airy equation over the domain  $x = (-12, 0)$  using the shooting method. The function `shoot_airy` solves the initial value problem using two initial conditions—the given boundary condition  $y(-12)$  and our guess for  $y'(-12)$ . The function returns the difference in the value  $y(0)$  computed by the `solve_ivp` and our given boundary condition  $y(0)$ . We then use `fsolve` to find the zero-error initial value.

```
from scipy.integrate import solve_ivp
from scipy.optimize import fsolve
def airy(x,y): return (y[1],x*y[0])
domain = [-12,0]; bc = [1,1]; guess = 5
def shoot_airy(guess):
 sol = solve_ivp(airy,domain,[bc[0],guess[0]])
 return sol.y[0,-1] - bc[1]
v = fsolve(shoot_airy,guess)[0]
```

The `scipy.integrate` function `solve_bvp` solves the boundary problem using a different approach, solving a collocation system with Newton's method.

- 512 The following code implements the Dufort–Frankel method to solve the heat equation. (As mentioned elsewhere, this approach is not recommended.)

```
import scipy.sparse as sp
dx = 0.01; dt = 0.01; n = 400
L = 1; x = np.arange(-L,L,dx); m = len(x)
U = np.empty((n,m))
U[0,:] = np.exp(-8*x**2); U[1,:] = U[0,:]
c = dt/dx**2; a = 0.5 + c; b = 0.5 - c
start_time = timeit.default_timer()
B = c*sp.diags([1, 1], [-1, 1], shape=(m, m)).tocsr()
B[0,1] *=2; B[-1,-2] *=2
for i in range(1,n-1):
 U[i+1,:] = (B@U[i,:]+b*U[i-1,:])/a
```

We can use the `ipywidgets` library to build an interactive plot of the solution.

```
from ipywidgets import interactive
def anim(i=0):
 plt.fill_between(x,U[i,:],color='ff9999');
 plt.ylim(0,1);plt.show()
interactive(anim, i=(0,n-1))
```

- 513 The following code solves the Schrödinger equation:

```

import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
def ψ₀(x, ε): return np.exp(-(x-1)**2/(2*ε))/(π*ε)**(1/4)
def schroedinger(n, m, ε):
 x, dx = np.linspace(-4, 4, n, retstep=True); Δt = 2*π/m; V = x**2/2
 ψ = ψ₀(x, ε)
 D = 0.5j*ε*sp.diags([1, -2, 1], [-1, 0, 1], shape=(n, n))/dx**2 \
 - 1j/ε*sp.diags(V, 0)
 D[0, 1] *= 2; D[-1, -2] *= 2
 A = sp.eye(n) + (Δt/2)*D
 B = sp.eye(n) - (Δt/2)*D
 for i in range(m):
 ψ = spsolve(B, A*ψ)
 return ψ

```

We'll loop over several values for time steps and mesh sizes and plot the error.

```

ε = 0.3; m = 20000; N = np.logspace(2, 3.7, 6).astype(int)
x = np.linspace(-4, 4, m)
ψ_m = -ψ₀(x, ε)
error_t = []; error_x = []
for n in N:
 x = np.linspace(-4, 4, n)
 ψ_n = -ψ₀(x, ε)
 error_t.append(la.norm(ψ_m - schroedinger(m, n, ε))/m)
 error_x.append(la.norm(ψ_n - schroedinger(n, m, ε))/n)
plt.loglog(2*π/N, error_t, '-r', 8/N, error_x, '-k');
plt.show()

```

- 515 We'll solve a radially symmetric heat equation. Although we divide by zero at  $r = 0$  when constructing the Laplacian operator, the resulting `inf` is overwritten when we apply the boundary condition.

```

import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
T = 0.5; m = 100; n = 100
r = np.linspace(0, 2, m); Δr = r[1]-r[0]; Δt = T/n
u = np.tanh(32*(1-r))[:, None]
D = sp.diags([1, -2, 1], [-1, 0, 1], shape=(m, m))/Δr**2 \
 + sp.diags([-1/r[1:], 1/r[:-1], [-1, 1]])/(2*Δr)
D = D.tocsr()
D[0, 0:2] = np.array([-4, 4])/Δr**2;
D[-1, -2:] = np.array([2, -2])/Δr**2
A = sp.eye(m) - 0.5*Δt*D
B = sp.eye(m) + 0.5*Δt*D
for i in range(n):
 u = spsolve(A, B@u)

```

```
plt.fill_between(r,u,-1,color='#ff9999'); plt.show()
```

517 We'll first define a function as a logit analogue to np.linspace.

```
def logitspace(x,n,p):
 return x*np.arctanh(np.linspace(-p,p,n))/np.arctanh(p)
```

Next, we define a general discrete Laplacian operator. The following function returns a sparse matrix in diagonal format (DIA). Two inconsequential elements of array d are replaced with nonzero numbers to avoid divide by zero warnings.

```
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
def laplacian(x):
 h = np.diff(x); h1 = h[:-1]; h2 = h[1:]; n = len(x)
 d = np.c_[\
 np.r_[h1[0]**2, h2*(h1+h2), 0], \
 np.r_[-h1[0]**2, -h1*h2,-h2[-1]**2], \
 np.r_[h1*(h1+h2), h2[-1]**2,0]].T
 d[0,-1], d[2,-1] = 999, 999
 return sp.diags(2/d,[-1,0,1],shape=(n, n))
```

Then, we write a function to solve the heat equation.

```
def heat_equation(x,t,u):
 n = 40; Δt = t/n
 u = φ(x,0,10)
 D = laplacian(x)
 A = sp.eye(len(x)) - 0.5*Δt*D
 B = sp.eye(len(x)) + 0.5*Δt*D
 for i in range(n):
 u = spsolve(A,B@u)
 return u
```

Finally, we compare the solutions using uniformly-spaced grid points and logit-spaced grid points.

```
def φ(x,t,s):
 return np.exp(-s*x**2/(1+4*s*t))/np.sqrt(1+4*s*t)
t = 15; m = 40
x = logitspace(20,m,.999)
laplacian(x).toarray()
u = heat_equation(x,t,φ(x,0,10))
plt.plot(x,u,'.-',x,φ(x,t,10),'k'); plt.show()
x = np.linspace(-20,20,m)
u = heat_equation(x,t,φ(x,0,10))
plt.plot(x,u,'.-',x,φ(x,t,10),'k'); plt.show()
```

518 Here's a solution to the Allen–Cahn equation using Strang splitting:

```
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
L = 16; m = 200; Δx = L/m
T = 8; n = 1600; Δt = T/n
x = np.linspace(-L/2,L/2,m)[None,:]
u = np.tanh(x**4 - 16*(2*x**2-x.T**2))
#u = np.random.standard_normal((m,m))
D = sp.diags([1, -2, 1], [-1, 0, 1], shape=(m,m)).tocsr() / Δx**2
D[0,1] *= 2; D[-1,-2] *= 2;
A = sp.eye(m) + 0.5*Δt*D
B = sp.eye(m) - 0.5*Δt*D
def f(u,Δt):
 return u/np.sqrt(u**2 - (u**2-1)*np.exp(-50*Δt))
u = f(u,Δt/2)
for i in range(n):
 if (i%8==0): U[:, :, i//10] = u
 u = spsolve(B,(A@spesolve(B,A@u).T)).T
 if (i<n): u = f(u,Δt)
u = f(u,Δt/2)
```

We can plot the solution using the code

```
plt.imshow(u, cmap="gray"); plt.axis('off'); plt.show()
```

524 The following code uses the local Lax–Friedrichs method to solve Burgers' equation.

```
m = 100; x,Δx = np.linspace(-1,3,m,retstep=True)
n = 100; Lt = 4; Δt = Lt/n; c = Δt/Δx
def f(u): return u**2/2
def fp(u): return u
u = ((x>=0)&(x<=1)).astype(float)
for i in range(n):
 fu = f(np.r_[u[0],u]); fpu = fp(np.r_[u[0],u])
 α = np.maximum(abs(fu[1:-1]),abs(fu[:-2]))
 F = (fu[1:-1]+fu[:-2])/2 - α*(u[1:]-u[:-1])/2
 u -= c*(np.diff(np.r_[0,F,0]))
```

524 Let's first define a few functions.

```
def limiter(t): return (abs(t)+t)/(1+abs(t))
def fixzero(u): return u + (u==0).astype(float)
def diff(u): return np.diff(u, axis=0)
def slope(u):
 du = diff(u)
```

```

 return np.r_[np.c_[0,0], \
 np.c_[du[1:,:]*limiter(du[:-1,:]/fixzero(du[1:,:])),\
 np.c_[0,0]]
def F(u):
 return np.c_[u[:,0]*u[:,1], u[:,0]*u[:,1]**2+0.5*u[:,0]**2]

```

Now, we can solve the dam-break problem.

```

m = 1000; x,Δx = np.linspace(-.5,.5,m,retstep=True)
T = 0.25; n = (T/(Δx/2)).astype(int); Δt = (T/n)/2; c = Δt/Δx
u = np.c_[0.8*(x<0)+0.2,0*x]
for i in range(n):
 v = u-0.5*c*slope(F(u))
 u[1:,:]=(u[:-1,:]+u[1:,:])/2 - diff(slope(u))/8 - c*diff(F(v))
 v = u-0.5*c*slope(F(u))
 u[:-1,:]=(u[:-1,:]+u[1:,:])/2 - diff(slope(u))/8 - c*diff(F(v))
plt.plot(x,u); plt.show()

```

```

526 m = 10; x,h = np.linspace(0,1,m,retstep=True)
A = np.tile(np.r_[-1/h-h/6,2/h-2/3*h,-1/h-h/6],(m,1)).T
A[1,0]/=2; A[1,-1] /= 2
b=np.r_-[-2/3*h**3,-4/3*h**3-8*h*x[1:-1]**2,-4*h+8/3*h**2-2/3*h**3+1]
u=la.solve_banded((1,1),A,b)
s=(-16)+8*x**2+15*np.cos(x)/np.sin(1)
plt.plot(x,s,'o-',x,u,'.-'); plt.show()

```

```

527 m = 8; x,h = np.linspace(0,1,m+2,retstep=True)
def tridiag(a,b,c): return np.diag(a,-1)+np.diag(b,0)+np.diag(c,1)
def D(a,b,c):
 return tridiag(a*np.ones(m-1), b*np.ones(m), c*np.ones(m-1))/h**3
M = np.r_[np.c_[D(-12,24,-12),D(-6,0,6)],
 np.c_[D(6,0,-6),D(2,8,2)]]
b = np.r_[np.ones(m)*h*384,np.zeros(m)]
u = la.solve(M,b)
plt.plot(x,16*(x**4 - 2*x**3 + x**2),'o-',x,np.r_[0,u[:m],0],'.-')
plt.show()

```

```

528 from scipy.integrate import solve_ivp
from scipy.fft import fftshift, fft, ifft
m = 128; x = np.linspace(-π,π,m,endpoint=False)
ξ = 1j*fftshift(np.arange(-m/2,m/2))
def f(t,u): return -np.real(ifft(ξ*fft(0.5*u**2)))
sol = solve_ivp(f, [0,1.5], np.exp(-x**2), method = 'RK45')
plt.plot(x,sol.y[:,-1]); plt.show()

```

- 529 The following code solves the KdV equation using integrating factors. We first set the parameters.

```
from scipy.fft import fftshift, fft, ifft
def φ(x,x₀,c): return 0.5*c/np.cosh(np.sqrt(c)/2*(x-x₀))**2
L = 30; T = 2.0; m = 256
x = np.linspace(-L/2,L/2,m,endpoint=False)
iξ = 1j*fftshift(np.arange(-m/2,m/2))*(2*π/L)
```

Next, we define the integrating factor  $G$  and the right-hand side function  $f$ .

```
def G(t): return np.exp(-iξ**3*t)
def f(t,w): return -(3*iξ*fft(ifft(G(t)*w)**2))/G(t)
```

Then we solve the problem using RK45.

```
from scipy.integrate import solve_ivp
u = φ(x,-4,4) + φ(x,-9,9)
w = fft(u)
sol = solve_ivp(f,[0,T],w,t_eval=np.linspace(0,T,200))
u = [np.real(ifft(G(sol.t[i])*sol.y[:,i])) for i in range(200)]
```

We can animate the solution using matplotlib.animation.

```
plt.rcParams["animation.html"] = "jshtml"
from matplotlib.animation import FuncAnimation
fig, ax = plt.subplots()
l, = ax.plot([-15,15],[0,5])
def animate(i): l.set_data(x, u[i])
FuncAnimation(fig, animate, frames=len(u), interval=50)
```

The first line of this code block display the animation as HTML using JavaScript. Alternatively, we can replace this line with

```
plt.rcParams["animation.html"] = "html5"
```

to convert the animation to HTML5, which will require to have ffmpeg video codecs available.

```

531 from scipy.fft import fftshift, fft2, ifft2
 $\epsilon = 1; m = 256; \ell = 100; n = 2000; \Delta t = 100/n$
U = (np.random.rand(m,m)>0.5)-0.5
 $\xi = fftshift(np.arange(-m/2,m/2))*(2\pi/\ell)$
D2 = - $\xi[:,None]**2-\xi[None,:]**2$
E = np.exp(-(D2+1)**2* Δt)
def f(U):
 return U/np.sqrt(U**2/ $\epsilon + np.exp(-\Delta t*\epsilon)*(1-U**2/\epsilon)$)
for i in range(n):
 U = f(ifft2(E*fft2(f(U))))
plt.imshow(np.real(U), cmap="gray"); plt.axis('off'); plt.show()

```

### B.3 Matlab

This section provides a MATLAB/Octave supplement to the Julia commentary and code elsewhere in the book. Page references to the Julia commentary are listed in the left margins. For brevity, the following variable is assumed to be defined:

```
bucket = "https://raw.githubusercontent.com/nmfsc/data/master/";
```

The Matlab code in this section is available as a Jupyter notebook at

<https://nbviewer.jupyter.org/github/nmfsc/octave/blob/main/octave.ipynb>

You can download the ipynb file and run it on a local computer with Octave and Jupyter. Alternatively, you can upload the file to CoCalc. You can also run the code directly on Binder using the QR link at the bottom of this page. Note that it may take a few minutes for the Binder to launch the repository.

- 5 While Julia's syntax tends to prefer column vectors, Matlab's syntax tends to prefer row vectors. The syntax for a row vector includes [1 2 3 4], [1,2,3,4], and 1:4. Syntax for a column vector is [1;2;3:4].
- 5 Matlab broadcasts arithmetic operators and some functions by implicitly expanding arrays to compatible sizes. For example, if we define A = rand(3,4), then A - mean(A) will subtract the row vector mean(A) from each row of A. Similarly, A - mean(A,2) subtracts a column vector from each column of A. Unlike Julia which explicitly requires a dot (.-) to denote broadcasting, Matlab does not. The operator \* is used for matrix multiplication and .\* is used for the element-wise Hadamard product. Similarly, ^ is used for matrix power and .^ is used for element-wise power. Matlab implicitly broadcasts functions such as sin(A) across each element of A.



23 `n = [10,15,20,25,50];  
set(gcf,'position',[0,0,1000,200])  
for i = 1:5,  
 subplot(1,5,i)  
 imshow(1-abs(hilb(n(i))\hilb(n(i))),[0 1])  
end`

- 29 The type and edit commands display the contents of an m-file, if available. Many basic functions are built-in and do not have an m-file.
- 30 The Matlab command rref returns the reduced row echelon form.
- 33 The corresponding Matlab code is

```
function b = gaussian_elimination(A,b)
n = length(A);
for j=1:n
 A(j+1:n,j) = A(j+1:n,j)/A(j,j);
 A(j+1:n,j+1:n) = A(j+1:n,j+1:n) - A(j+1:n,j).*A(j,j+1:n);
end
for i=2:n
 b(i) = b(i) - A(i,1:i-1)*b(1:i-1);
end
for i=n:-1:1
 b(i) = (b(i) - A(i,i+1:n)*b(i+1:n))/A(i,i);
end
```

- 42 The following Matlab code implements the simplex method. We start by defining functions used for pivot selection and row reduction in the simplex algorithm.

```
function [tableau] = row_reduce(tableau)
[i,j] = get_pivot(tableau);
G = tableau(i,:)/tableau(i,j);
tableau = tableau - tableau(:,j)*G;
tableau(i,:) = G;
end

function [i,j] = get_pivot(tableau)
[_,j] = max(tableau(end,1:end-1));
a = tableau(1:end-1,j); b = tableau(1:end-1,end);
k = find(a > 0);
[_,i] = min(b(k)./a(k));
i = k(i);
```

```

end

function [z,x,y] = simplex(c,A,b)
[m,n] = size(A);
tableau = [A eye(m) b; c' zeros(1,m) 0];
while (any(tableau(end,1:n)>0)),
 tableau = row_reduce(tableau);
end
p = find(tableau(end,1:n)==0);
x = zeros(n,1);
x(p) = tableau(:,p)'*tableau(:,end);
z = -tableau(end,end);
y = -tableau(end,n+(1:m));
end

```

- 45 The MATLAB function `linprog` solves the LP problem. The built-in Octave function `glpk` (GNU Linear Programming Kit) can solve LP problems using either the revised simplex method or the interior point method.
- 45 Matlab's backslash and forwardslash operators use UMPFPACK to solve sparse linear systems.
- 45 The command `spy(A)` returns the sparsity plot of a matrix `A`.
- 48 The function `gplot` plots the vertices and edges of an adjacency matrix. MATLAB, but not Octave, has additional functions for constructing and analyzing graphs.
- 48 Matlab lacks many of the tools for drawing graphs but we can construct them. First, let's write naïve functions that will draw force-directed, spectral, and circular graphs layouts. Although the force-directed layout can be implemented using a forward Euler scheme, it may be easier to use an adaptive timestep ODE solver to manage stability.

```

function f = spring(A,z)
n = length(z); k = 2*sqrt(1/n);
d = z - z.'; D = abs(d)/k;
F = -(A.*D - 1./(D+eye(n)).^2);
f = sum(F.*d,2);
end
function xy = spring_layout(A,z)
n = size(A,1);
z = rand(n,1) + 1i*rand(n,1);
[t,z] = ode45(@(t,z) spring(A,z),[0,10],z);
xy = [real(z(end,:));imag(z(end,:))]';

```

```

end

function xy = spectral_layout(A)
D = diag(sum(A,2));
[v,d] = eig(D - A);
[_,i] = sort(diag(d));
xy = v(:,i(2:3));
end

function xy = circular_layout(A)
n = size(A,1); t = (2*pi*(1:n)/n)';
xy = [cos(t) sin(t)];
end

```

Let's also define a function to download and construct an adjacency matrix.

```

function M = get_adjacency_matrix(bucket,filename)
data = urlread([bucket filename '.csv']);
ij = cell2mat(textscan(data,'%d,%d'));
M = sparse(ij(:,1),ij(:,2),1);
end

```

Now, we can draw the dolphin networks of the Doubtful Sound.

```

A = get_adjacency_matrix(bucket,"dolphins");
gplot(A,spring_layout(A),'.-'); axis equal; axis off;

```

- 52 The Matlab function `symrcm` returns the permutation vector using the reverse Cuthill–McKee algorithm for symmetric matrices, and `symamd` returns the permutation vector using the symmetric approximate minimum degree permutation algorithm.
- 54 The following code implements the revised simplex method.

```

function [z,x,y] = revised_simplex(c,A,b)
[m,n] = size(A);
N = 1:n; B = n + (1:m);
A = [A speye(m)];
ABinv = speye(m);
c = [c;zeros(m,1)];
while true
y = c(B)'*ABinv;
if isempty(j=find(c(N)'-y*A(:,N)>0,1)), break; end
k = find((q = ABinv*A(:,N(j))) > 0);
[_,i] = min(ABinv(k,:)*b./q(k));

```

```

i = k(i);
p = B(i); B(i) = N(j); N(j) = p;
ABinv = ABinv - ((q - sparse(i,1,1,m,1))/q(i))*ABinv(i,:);
end
i = find(B<=n);
x = zeros(n,1);
x(B(i)) = ABinv(i,:)*b;
z = c(1:n)'*x;
end

```

- 57 For a  $2000 \times 1999$  matrix  $A \setminus b$  takes 3.8 seconds and  $\text{pinv}(A) * b$  takes almost 56 seconds.
- 63 The function `givens(x,y)` returns the Givens rotation matrix for  $(x,y)$ .
- 64 The function `qr` implements LAPACK routines to compute the QR factorization of a matrix using Householder reflection.
- 65 Matlab's `mldivide (\)` solves an overdetermined system using a QR solver.
- 66 The Zipf's law coefficients  $c$  for the canon of Sherlock Holmes computed using ordinary least squares are

```

data = urlread([bucket 'sherlock.csv']);
T = cell2mat(textscan(data, '%s\t%d')(2));
n = length(T);
A = [ones(n,1) log(1:n)];
B = log(T);
c1 = A\B

```

- 69 The constrained least squares problem is solved using

```

function x = constrained_lstsq(A,b,C,d)
 x = [A'*A C'; C zeros(size(C,1))] \ ([A'*b;d])
 x = x(1:size(A,2))
end

```

- 63 The command `[U,S,V]=svd(A)` returns the SVD of a matrix  $A$  and `svd(A,0)` returns the “economy” version of the SVD. The command `svds(A,k)` returns the first  $k$  singular values and associated singular vectors.

73 The function `pinv` computes the Moore–Penrose pseudoinverse by computing the SVD using a default tolerance of `max(size(A))*norm(A)*eps`, i.e., machine epsilon scaled by the largest singular value times the largest dimension of  $A$ .

77 The following code implements total least squares:

```
function X = tls(A,B)
n = size(A,2);
[~,~,V] = svd([A B],0);
X = -V(1:n,n+1:end)/V(n+1:end,n+1:end);
end
```

77  $A = [2 \ 4; \ 1 \ -1; \ 3 \ 1; \ 4 \ -8]; \ b = [1; \ 1; \ 4; \ 1];$   
 $x_{ols} = A \backslash b$   
 $x_{tls} = \text{tls}(A,b)$

78 Matlab reads a color image as a 3-dimensional RGB array of unsigned integers between 0 and 255. It will be easiest to work with floating-point values between 0 and 1, so we'll first convert to grayscale and normalize. We can use the function `rgb2gray` to make the conversion. The function is a standard function in MATLAB it is available in Octave in the image toolbox, or we can write it ourselves. Take  $k$  to be some nominal value like 20.

```
rgb2gray = @(x) 0.2989*x(:,:,1) + 0.5870*x(:,:,2) + 0.1140*x(:,:,3);
A = rgb2gray(double(imread(['bucket "laura.png'])))/255;
[U, S, V] = svd(A,0);
sigma = diag(S);
```

We can confirm that the error  $\|A - A_k\|_F^2$  matches  $\sum_{i=k+1}^n \sigma^2$  by computing

```
Ak = U(:,1:k) * S(1:k,1:k) * V(:,1:k)';
norm(double(A)-Ak,'fro') - norm(sigma(k+1:end))
imshow(Ak)
```

Although values of the compressed image no longer lie between 0 and 1—instead ranging between -0.11 and 1.18—the command `imshow(Ak)` clamps the values between 0 and 1 for a floating-point array. Finally, let's plot the error:

```
r = sum(size(A))/prod(size(A))*(1:min(size(A)));
error = 1 - sqrt(cumsum(sigma.^2))/norm(sigma);
semilogx(r,error,'.-');
```

84 The following function is a naïve implementation of NMF:

```

function [W,H] = nmf(X,p)
 W = rand(size(X,1),p);
 H = rand(p,size(X,2));
 for i=1:50,
 W = W.* (X*H')./(W*(H*H') + (W==0));
 H = H.* (W'*X)./((W'*W)*H + (H==0));
 end
end

```

- 86 The function `vander` generates a Vandermonde matrix for input  $(x_0, x_1, \dots, x_n)$  with rows given by  $\begin{bmatrix} x_i^p & x_i^{p-1} & \cdots & x_i & 1 \end{bmatrix}$ .
- 89 The function `roots` finds the roots of a polynomial  $p(x)$  by computing `eig` for the companion matrix of  $p(x)$ .
- 90 The Matlab function `condeig` returns the eigenvalue condition number.
- 97 The following code computes the PageRank of the graph in Figure 4.3.

```

H = [0 0 0 0 1; 1 0 0 0 0; 1 0 0 0 1; 1 0 1 0 0; 0 0 1 1 0];
v = ~any(H);
H = H./(sum(H)+v);
n = length(H);
d = 0.85;
x = ones([n 1])/n;
for i = 1:n
 x = d*(H*x) + d/n*(v*x) + (1-d)/n;
end

```

- 104 The function `hess` returns the unitarily similar upper Hessenberg form of a matrix using LAPACK.
- 109 The function `eig` computes the eigenvalues and eigenvectors of a matrix with a LAPACK library that implements the shifted QR method.
- 114 The function `eigs` computes several eigenvalues of a sparse matrix using the implicitly restarted Arnoldi process.
- 137 The function `pcg` implements the conjugate gradient method—preconditioned conjugate gradient method if a preconditioner is also provided.

140 The function `gmres` implements the generalized minimum residual method and `minres` implements the minimum residual method.

148 The function `kron(A,B)` returns the Kronecker product  $\mathbf{A} \otimes \mathbf{B}$ .

148 The radix-2 FFT algorithm written as a recursive function in Matlab is

```
function y = fftx2(c)
 n = length(c);
 omega = exp(-2i*pi/n);
 if mod(n,2) == 0
 k = (0:n/2-1)';
 u = fftx2(c(1:2:n-1));
 v = (omega.^k).*fftx2(c(2:2:n));
 y = [u+v; u-v];
 else
 k = (0:n-1)';
 F = omega.^(k*k');
 y = F*c;
 end
end
```

and the IFFT is

```
function c = ifftx2(y)
 c = conj(fftx2(conj(y)))/length(y);
end
```

149 The Matlab function `toeplitz` constructs a Toeplitz matrix.

150 A circulant matrix can be built in Matlab from a vector  $v$  using

```
toeplitz(v,circshift(flipud(v),1))
```

152 The convolution of  $u$  and  $v$  is `ifft(fft(u).*fft(v))`.

```
152 function y = fasttoeplitz(c,r,x)
 n = length(x);
 m = 2^(ceil(log2(n)))-n;
 x1 = [c; zeros([m 1]); r(end:-1:2)];
 x2 = [x; zeros([m+n-1 1])];
 y = ifftx2(fftx2(x1).*fftx2(x2));
 y = y(1:n);
end
```

```
155 function y = bluestein(x)
 n = length(x);
 w = exp((1i*pi/n)*((0:n-1).^2))';
 y = w.*fasttoeplitz(conj(w),conj(w),w.*x);
end
```

- 158 The Matlab function `fft` implements the Fastest Fourier Transform in the West (FFTW) library. The inverse FFT can be computed using `ifft` and multidimensional FFTs can be computed using `fftn` and `ifftn`.
- 158 The functions `fftshift` and `ifftshift` shift the zero frequency component to the center of the array.
- 160 Matlab does not have a DST function, so we'll need to build one ourselves. The function `dst3` computes the three-dimensional DST (6.4) by computing the DST over each dimension. The function `dstx3` computes a DST in one-dimension for a three-dimensional array. The real command in the function `dstx3` is only needed to remove the round-off error from the FFT. By distributing the scaling constant  $\sqrt{2/(n+1)}$  across both the DST and the inverse DST, we only need to implement one function for both.

```
function a = dst3(a)
 a = dstx3(shiftdim(a,1));
 a = dstx3(shiftdim(a,1));
 a = dstx3(shiftdim(a,1));
end
```

```
function a = dstx3(a)
 n = size(a); o = zeros(1,n(2),n(3));
 y = [o;a;o;-a(end:-1:1,:,:)];
 y = fft(y);
 a = imag(y(2:n(1)+1,:,:)*(sqrt(2*(n(1)+1))));
```

```
n = 50; x = (1:n)'/(n+1); dx = 1/(n+1);
[x,y,z] = meshgrid(x);
v = @(k) 2 - 2*cos(k*pi/(n+1));
[ix,iy,iz] = meshgrid(1:n);
lambda = (v(ix)+v(iy)+v(iz))./dx.^2;
f = 2*((x-x.^2).*(y-y.^2) + (x-x.^2).*(z-z.^2) + (y-y.^2).*(z-z.^2));
u = dst3(dst3(f)./lambda);
```

The program takes less than a third of a second to run for  $n = 50$ .

- 159 The MATLAB function `dct` returns the type 1–4 DCT. The equivalent function in Octave’s `signal` package returns a DCT-2.
- 178 The function `class` returns the data type of a variable.
- 178 The Matlab function `num2hex` converts a floating-point number to a hexadecimal string. We can further convert the string to a string of bits using

```
function b = float_to_bin(x)
 b = sprintf("%s",dec2bin(hex2dec(num2cell(num2hex(x))),4)');
 if x<0, b(1) = '1'; end
end
```

- 179 The constant `eps` returns the double-precision machine epsilon of  $2^{-52} \approx 10^{-16}$ . The function `eps(x)` returns the double-precision round-off error at  $x$ . For example, `eps(0)` returns  $2^{-1074} \approx 10^{-323}$ . Of course, `eps(1)` is the same as `eps`.

- 179 

```
function y = Q_rsqrt(x)
 i = typecast(single(x),'int32');
 i = 0x5f3759df - bitshift(i,-1);
 y = typecast(i,'single');
 y = y * (1.5 - (0.5 * x * y * y));
end
```

- 182 

```
a = 77617; b = 33096;
333.75*b^6+a^2*(11*a^2*b^2-b^6-121*b^4-2)+5.5*b^8+a/(2*b)
```

- 182 The command `realmax` returns the largest floating-point number. The command `realmin` returns the smallest normalized floating-point number.

- 183 To check for overflows and `NaN`, use the Matlab logical commands `isinf` and `isnan`. You can use `NaN` to lift the “pen off the paper” in a Matlab plot as in

```
plot([1 2 2 2 3],[1 2 NaN 1 2])
```

- 185 The functions `expm1` and `log1p` compute  $e^x - 1$  and  $\log(x + 1)$  more precisely than  $\exp(x)-1$  and  $\log(x+1)$  in a neighborhood of zero.

```
189 function x = bisection(f,a,b,tolerance)
 while abs(b-a) > tolerance
 c = (a+b)/2;
 if sign(f(c)) == sign(f(a)), a = c; else b = c; end
 end
 x = (a+b)/2;
end
```

- 194 The function `fzero(f,x0)` uses the Dekker–Brent method to find a zero of the input function. The function `f` can either be a string (which is a function of `x`), an anonymous function, or the name of an m-file.
- 201 The following Matlab code takes the complex-valued array `bb` for the lower-left and upper-right corners of the bounding box; `xpix` for the number of horizontal pixels; `n` for the maximum number of iterations; and `s` for the starting value  $z^{(0)}$ , which for the Mandelbrot set is 0. The function returns a two-dimensional array `M` that counts the number of iterations  $k$  to escape:  $|z^{(k)}| > 2$ .

```
function M = mandelbrot(bb,xpix,n,s)
 ypix = round(xpix*(bb(4)-bb(2))/(bb(3)-(bb(1))));
 M = zeros(ypix,xpix);
 z = s*ones(ypix,xpix);
 c = linspace(bb(1),bb(3),xpix) + 1i*linspace(bb(4),bb(2),ypix)';
 for k=1:n
 mask = abs(z)<2;
 M(mask) = M(mask) + 1;
 z(mask) = z(mask).^2 + c(mask);
 end
end
```

The following produces image (c) of Figure 8.4 on page 201:

```
M = mandelbrot([-0.1710,1.0228,-0.1494,1.0443],800,200,0);
imwrite(1-M/max(M(:)),'mandelbrot.png');
```

- 205 The function `polyval(x,p)` uses Horner's method to evaluate a polynomial with coefficients  $p = [a_0, a_1, \dots, a_n]$  at  $x$ .
- 208 The function `roots` returns the roots of a polynomial by finding the eigenvalues of the companion matrix.

```
214 f = @(x) [x(1)^3-3*x(1)*x(2)^2-1; x(2)^3-3*x(1)^2*x(2)];
df = @(t,x,p) [-3*x(1)^2-3*x(2)^2, -6*x(1)*x(2);...
-6*x(1)*x(2), 3*x(2)^2-3*x(1)^2]\p;
x0 = [1;1];
[t,y] = ode45(@(t,x) df(t,x,f(x0)),[0;1],x0);
y(end,:)
```

To reduce the error, we can add the option `opt=odeset("RelTol",1e-10)` to the solver.

- 218 The following code uses the gradient descent method to find the minimum of the Rosenbrock function:

```
df = @(x) [-2*(1-x(1))-400*x(1)*(x(2)-x(1)^2), 200*(x(2)-x(1)^2)];
x = [-1.8,3.0]; p = [0,0]; a = 0.001; b = 0.9;
for i = 1:500
 p = -df(x) + b*p;
 x = x + a*p;
end
```

In practice, we can use the optimization toolbox. The `fminbnd` function uses the golden section search method to find the minimum of a univariate function. The `fminsearch` function uses the Nelder–Mead method to find the minimum of a multivariate function. The `fminunc` function uses the BFGS method to find the minimum of a multivariate function.

```
pkg load optim
f = @(x) (1-x(1)).^2 + 100*(x(2) - x(1)).^2;
x0 = [-1.8,3.0];
fminunc(f, x0)
```

- 223 The function `vander` generates a Vandermonde matrix with rows given by  $[x_i^n \quad x_i^{n-1} \quad \cdots \quad 1]$ . Add `fliplr` to reverse them  $[1 \quad \cdots \quad x_{i-1}^{n-1} \quad x_i^n]$ .
- 224 The following function computes the coefficients  $m$  of a cubic spline with natural boundary conditions through the nodes given by the arrays  $x$  and  $y$ .

```
function m = spline_natural(x,y)
h = diff(x(:));
gamma = 6*diff(diff(y(:)) ./ h);
alpha = diag(h(2:end-1),-1);
beta = 2*diag(h(1:end-1)+h(2:end),0);
m = [0;(alpha+beta+alpha')\gammama;0];
end
```

We can then use (9.3) and (9.4) to evaluate that spline using  $n$  points.

```

function [X,Y] = evaluate_spline(x,y,m,n)
x = x(:); y = y(:); h = diff(x);
B = y(1:end-1) - m(1:end-1).*h.^2/6;
A = diff(y)./h-h./6.*diff(m);
X = linspace(min(x),max(x),n+1)';
i = sum(x<=X');
i(end) = length(x)-1;
Y = (m(i).*(x(i+1)-X).^3 + m(i+1).*(X-x(i)).^3)./(6*h(i)) ...
+ A(i).* (X-x(i)) + B(i);
end

```

The function takes column-vector inputs for  $x$  and  $y$  and outputs column vectors  $X$  and  $Y$  with length  $N$ . Note that the line  $i = \text{sum}(x \leq X')$ ; broadcasts the element-wise operator  $\leq$  across a row and a column array.

- 235 The function `spline` returns a cubic spline with the not-a-knot condition.
  - 237 The function `pp = mkpp(b,C)` returns a structured array `pp` that can be used to build piecewise polynomials of order  $n$  using the function `y = ppval(pp,x)`. The input `b` is a vector of breaks (including endpoints) with length  $p + 1$  and `C` is an  $p \times n$  array of the polynomial coefficients in each segment.
  - 238 The Matlab function `pchip` returns a cubic Hermite spline.
  - 240 We can build a Bernstein matrix in Matlab from a column vector `t` with the following function:
- $$B = @(n,t)[1 \text{ cumprod}((n:-1:1)/(1:n))].*t.^{(0:n)}.*{(1-t).^{(n:-1:0)}};$$
- 251 

```
function P = legendre(x,n)
if n==0, P = ones(size(x));
elseif n==1, P = x;
else P = x.*legendre(x,n-1)-1/(4-1/(n-1)^2).*legendre(x,n-2);
end
end
```
  - 257 We'll construct a Chebyshev differentiation matrix and use the matrix to solve a few simple problems.

```

function [D,x] = chebdiff(n)
x = -cos(linspace(0,pi,n))';
c = [2;ones(n-2,1);2].*(-1).^(0:n-1);
D = c./c'./(x - x' + eye(n));

```

```
D = D - diag(sum(D,2));
end
```

```
n = 15;
[D,x] = chebdiff(n);
u = exp(-4*x.^2);
plot(x,D*u,'.-')
```

```
B = zeros(1,n); B(1) = 1;
plot(x,[B;D]\[2;u],'.-')
```

```
n = 15; k2 = 256;
[D,x] = chebdiff(n);
L = (D^2 - k2*diag(x));
L([1,n],:) = 0; L(1,1) = 1; L(n,n) = 1;
y = L\[2;zeros(n-2,1);1];
plot(x,y,'.-')
```

- 259 The Chebfun package (<https://www.chebfun.org>) includes methods for manipulating functions in Chebyshev space.

```
265 function [x,phi] = scaling(c,z,n)
m = length(c); L = 2^n;
phi = zeros(2*m*L,1);
k = (0:m-1)*L;
phi(k+1) = z;
for j = 1:n
 for i = 1:m*2^(j-1)
 x = (2*i-1)*2^(n-j);
 phi(x+1) = c * phi(mod(2*x-k,2*m*L)+1);
 end
end
x = (1:(m-1)*L)/L;
phi = phi(1:(m-1)*L);
end
```

Let's use scaling to plot the Daubechies  $D_4$  scaling function.

```
c = [1+sqrt(3),3+sqrt(3),3-sqrt(3),1-sqrt(3)]/4;
z = [0,1+sqrt(3),1-sqrt(3),0]/2;
[x,phi] = scaling(c,z,8);
plot(x,phi)
```

```

267 psi = zeros(size(phi)); n = length(c)-1; L = length(phi)/(2*n)
for k=0:n
 psi((k*L+1):(k+n)*L) += (-1)^k*c(n-k+1)*phi(1:2:end);
end
plot(x,psi)

```

269 The Large Time-Frequency Analysis Toolbox (LTFAT) includes several utilities for wavelet transforms and is available from <http://ltfat.github.io> under the GNU General Public License.

270 We'll use the LTFAT package.

```

pkg load ltfat
adjustlevels = @(x) 1 - min(max((sqrt(abs(x))),0),1);
A = double(rgb2gray(imread(['bucket "laura_square.png'])))/255;
c = fwt2(A,"db2",9);
image(adjustlevels(c)); colormap(gray)

```

Let's choose a threshold level of 0.5 and plot the resultant image after taking the inverse DWT.

```

c = thresh(c,0.5);
B = ifwt2(c,"db2",9);
image(max(min(B,1),0)); colormap(gray)

```

279 The Jacobian can be computed using complex-step approximation.

```

function J = jacobian(f,x,c)
 for k = (n = 1:length(c))
 J(:,k) = imag(f(x,c+1e-8i*(k==n)'))/1e-8;
 end
end
\end{lstlisting}
We can then implement the Levenberg--Marquardt method.
\begin{lstlisting}
function c = gauss_newton(f,x,y,c)
 r = y - f(x,c);
 for j = 1:100
 G = jacobian(f,x,c);
 M = G'*G;
 c = c + (M+diag(diag(M)))\ (G'*r);
 if norm(r-(r=y-f(x,c))) < 1e-10, return; end
 end
 display('Gauss-Newton did not converge.')
end

```

Now, the problem can be solved with the following code:

```
f = @(x,c) c(1)*exp(-c(2).*(x-c(3)).^2) + ...
 c(4)*exp(-c(5).*(x-c(6)).^2);
x = 8*rand([100 1]);
y = f(x,[1 3 3 2 3 6]) + 0.1*randn([100 1]);
c0 = [2 0.3 2 1 0.3 7]';
c = gauss_newton(f,x,y,c0);
```

Assuming that the Gauss–Newton method converged, we can plot the results.

```
X = linspace(0,8,100);
plot(x,y,'.',X,f(X,c));
```

In practice, we can use the `lsqcurvefit` function in the `optim` toolkit:

```
pkg load optim
c = lsqcurvefit(@(c,x) f(x,c), c0, x, y)
```

Alternatively, we can use the `leasqr` function in Octave's `optim` toolkit:

```
pkg load optim
[_,c] = leasqr (x, y, c0, f)
```

- 282 We'll first define the logistic function and generate synthetic data. Then, we apply Newton's method.

```
sigma = @(x) 1./(1+exp(-x));
x = rand(30,1); y = (rand(30,1) < sigma(10*x-5));

X = [ones(size(x)) x]; w = zeros(2,1);
for i=1:10
 S = sigma(X*w).* (1 - sigma(X*w));
 w = w + (X'*(S.*X))\ (X'*(y - sigma(X*w)));
end
```

- 284 Let's start by generating some training data and setting up the neural network architecture.

```
N = 100; t = linspace(0,pi,N);
x = cos(t); x = [ones(1,N);x];
y = sin(t) + 0.05*randn(1,N);
n = 20; W1 = rand(n,2); W2 = randn(1,n);
phi = @(x) max(x,0);
dphi = @(x) (x>0);
```

Now, we train the model to determine the optimized parameters.

```
alpha = 1e-3;
for epoch = 1:10000
 s = W2 * phi(W1*x);
 dLdy = 2*(s-y);
 dLdW1 = dphi(W1*x) .* (W2' * dLdy) * x';
 dLdW2 = dLdy * phi(W1*x)';
 W1 -= 0.1 * alpha * dLdW1;
 W2 -= alpha * dLdW2;
end
```

Once we have determined the parameters  $W_1$  and  $W_2$ , we can plot the results.

```
s = W2 * phi(W1*x);
scatter(x(2,:),y,'r','filled'); hold on
plot(x(2,:),s)
```

Alternatively, we can swap out the activation and loss functions.

```
phi = @(x) 1./(1+exp(-x));
dphi = @(x) phi(x).*(1-phi(x));
```

```
L = norm(s- y);
dLdy = 2*(s-y)/L;
```

- 293 Coefficients to the third-order approximation to  $f'(x)$  using nodes at  $x - h$ ,  $x$ ,  $x + h$  and  $x + 2h$  are given by  $C[2, :]$  where

```
d = [-1;0;1;2];
n = length(d);
V = fliplr(vander(d)) ./ factorial([0:n-1]);
C = inv(V);
```

Coefficients are given by `rats(C)` and the coefficients of the truncation error is given by `rats(C*d.^n/factorial(n))`:

- 293 The command `rats` returns the rational approximation of number as a string by using continued fraction expansion. For example `rats(0.75)` returns  $3/4$ .

- 294 Matlab code for Richardson extrapolation taking  $\delta = \frac{1}{2}$  is

```
function D = richardson(f,x,m,n)
if n > 0
 D = (4^n*richardson(f,x,m,n-1) - richardson(f,x,m-1,n-1))/(4^n-1);
else
```

```
D = phi(f,x,2^m);
end
end
```

where the central difference scheme is

```
function p = phi(f,x,n)
 p = (f(x+1/n) - f(x-1/n))/(2/n);
end
```

- 297 We can build a minimal working example of forward accumulation automatic differentiation in Matlab following an example in Neidinger [2010]. We first define a class, which we'll call dual, with the properties value and deriv for the value and derivative of a variable. We can overload the built-in functions + (plus), \* (mtimes), and sin to operate on the dual class. We save the following code as the m-file dual.m. When working in Jupyter, we can add %%file dual.m to top of a cell to write the cell to a file.

```
%%file dual.m
classdef dual
properties
 value
 deriv
end
methods
 function obj = dual(a,b)
 obj.value = a;
 obj.deriv = b;
 end
 function h = plus(u,v)
 if ~isa(u,'dual'), u = dual(u,0); end
 if ~isa(v,'dual'), v = dual(v,0); end
 h = dual(u.value + v.value, u.deriv + v.deriv);
 end
 function h = mtimes(u,v)
 if ~isa(u,'dual'), u = dual(u,0); end
 if ~isa(v,'dual'), v = dual(v,0); end
 h = dual(u.value*v.value, u.deriv*v.value + u.value*v.deriv);
 end
 function h = sin(u)
 h = dual(sin(u.value), cos(u.value)*u.deriv);
 end
 function h = minus(u,v)
 h = u + (-1)*v;
 end
end
```

```
end
```

Now, let's define a function for the autodiff.

```
x = dual(0,1);
y = x + sin(x);
```

Then  $y.value$  returns 0 and  $y.deriv$  returns 2, which agrees with  $y(0) = 0$  and  $y'(0) = 2$ .

298 We can define the dual numbers as

```
x1 = dual(2,[1 0]);
x2 = dual(pi,[0 1]);
y1 = x1*x2 + sin(x2);
y2 = x1*x2 - sin(x2);
```

302 

```
function p = trapezoidal(f,x,n)
F = f(linspace(x(1),x(2),n+1));
p = (F(1)/2 + sum(F(2:end-1)) + F(end)/2) * (x(2)-x(1))/n;
end
```

303 

```
n = floor(logspace(1,2,10));
for p = 1:7,
 f = @(x) x + x.^p.*(2-x).^p;
 S = trapezoidal(f,[0,2],1e6);
 for i = 1:length(n)
 Sn = trapezoidal(f,[0,2],n(i));
 error(i) = abs(Sn - S)/S;
 end
 slope(p) = [log(error)/[log(n);ones(size(n))]](1);
 loglog(n,error,'.-k');
 hold on;
end
```

306 

```
function S = clenshaw_curtis(f,n)
x = cos(pi*(0:n)'/n);
w = zeros(1,n+1); w(1:2:n+1) = 2 ./ (1 - (0:2:n).^2);
S = 2/n * w * dctI(f(x));
end
```

```
function a = dctI(f)
n = length(f);
```

```

g = real(fft([f;f(n-1:-1:2)])) / 2;
a = [g(1)/2; g(2:n-1); g(n)/2];
end

```

- 312 We can implement Gauss–Legendre quadrature by first defining the weights

```

function [nodes,weights] = gauss_legendre(n)
b = (1:n-1).^2./(4*(1:n-1).^2-1);
a = zeros(n,1);
scaling = 2;
[v,s] = eig(diag(sqrt(b),1) + diag(a) + diag(sqrt(b),-1));
weights = scaling*v(1,:).^2;
nodes = diag(s);
end

```

and then implementing the method as

```

f = @(x) cos(x).*exp(-x.^2);
[nodes,weights] = gauss_legendre(n);
weights * f(nodes)

```

- 341  $r = \exp(2i\pi(0:0.01:1));$   
 $\text{plot}((1.5\cdot r.^2 - 2\cdot r + 0.5)./r.^2); \text{axis equal};$

- 343 The following function returns the coefficients for a stencil given by  $m$  and  $n$ :

```

function [a,b] = multistepcoeffs(m,n)
s = length(m)+ length(n);
A = (m+1).^(0:s-1)';
B = (0:s-1)'.*(n+1).^(0:s-2)';
c = -[A B]\ones(s,1);
a = [1;c(1:length(m))];
b = [c(length(m)+1:end)];
end

```

We can express floating-point numbers  $c$  as fractions using the `rats(c)`.

```

function plotstability(a,b)
r = exp(1i*pi*(.01,2*pi-0.01,400));
z = (r'.^(1:length(a))*a) ./ (r'.^(1:length(b))*b);
plot(z); axis equal
end

```

- 369 The Matlab recipe for solving a differential equation is similar to the Julia recipe, but rather than calling a general solver and specifying the method, each method in Matlab is its own solver. Fortunately, the syntax for each solver including the options syntax is almost identical.

1. Define the problem pendulum = @(t,u)[u(2);-sin(u(1))];
2. Set up the parameters u0 = [8\*pi/9,0]; tspan = [0,8\*pi];
3. Solve the problem [t,u] = ode23(pendulum,tspan,u0);
4. Present the solution plot(t,u(:,1),'.-',t,u(:,2),'.-')  
legend('\theta','\omega')

The parameter `tspan` may be either a two-element vector specifying initial and final times, or a longer, strictly increasing or decreasing vector. In the two-vector case, the solver returns the intermediate solutions at each adaptive time step. In the longer-vector case, the solver interpolates the values between the adaptive time steps to provide solutions at the points given in `tspan`. We can alternatively request a structure `sol` as output of the ODE solver, which we can subsequently evaluate at an arbitrary array of values `t` using the function `u = deval(sol,t)`. This approach is similar to the modern approaches in Julia and Python. Unfortunately, the function `deval` is not implemented in Octave.

- 378 The following Matlab code implements the backward Euler method:

```
dx = .01; dt = .01; L = 2; lambda = dt/dx^2; uL = 0; uR = 0;
x = (-L:dx:L)'; n = length(x);
u = (abs(x)<1);
u(1) += 2*lambda*uL; u(n) += 2*lambda*uR;
D = spdiags(repmat([1 -2 1],[n 1]),-1:1,n,n);
D(1,2) = 0; D(n,n-1)= 0;
A = speye(n) - lambda*D;
for i = 1:20
 u = A\u;
end
area(x,u,"edgecolor",[1 .5 .5],"facecolor",[1 .8 .8])
```

The runtime using Octave on a typical laptop is about 0.35 seconds. The runtime using a non-sparse matrix instead is about 0.45 seconds, so there doesn't appear to be a great advantage in using a sparse solver. But, for a larger system with `dx = .001`, the runtime using sparse matrices is still about 0.35 seconds, whereas for a nonsparse matrix it is almost 30 seconds.

- 378 The command `mldivide (\)` will automatically implement a tridiagonal solver on a tridiagonal matrix if the matrix is formatted as a sparse matrix. The command `help sparfun` returns a list of matrix functions for working with sparse matrices.

379 The following code solves the heat equation using the Crank–Nicolson method:

```
dx = .01; dt = .01; L = 2; lambda = dt/dx^2;
x = (-L:dx:L)'; n = length(x);
u = (abs(x)<1);
D = spdiags(repmat([1 -2 1],[n 1]),-1:1,n,n);
D(1,2) = 2; D(n,n-1) = 2;
A = 2*speye(n) + lambda*D;
B = 2*speye(n) - lambda*D;
for i = 1:20
 u = B\ (A*u);
end
plot(x,u);
```

388 The following code implements (13.17) with  $m(u) = u^2$  using `ode15s`, which uses a variation of the adaptive-step BDF methods:

```
n = 400; L = 2; x = linspace(-L,L,n)'; dx = x(2)-x(1);
m = @(u) u.^2;
Du = @(t,u) [0;diff(m((u(1:n-1)+u(2:n))/2).*diff(u))/dx.^2;0];
u0 = double(abs(x)<1);
options = odeset('JPattern',spdiags(ones([n 3]),-1:1,n,n));
[t,U] = ode15s(Du,linspace(0,2,10),u0,options);
for i=1:size(U,1), plot(x,U(i,:),'r'); hold on; ylim([0 1]); end
```

We could also define the right-hand side of the porous medium equation as

```
m = @(u) u.^3/3;
Du = @(t,u) [0;diff(m(u),2)/dx.^2;0];
```

444 We can express the discrete indices of  $\xi$  as

```
ik = 1i*[0:floor(n/2) floor(-n/2+1):-1]*(2*pi/L);
```

or

```
ik = 1i*ifftshift([floor(-n/2+1):floor(n/2)])*(2*pi/L);
```

The Matlab implementation of the solution to the heat equation (16.4) is

```
n = 256; L = 4;
k2 = ([0:floor(n/2) floor(-n/2+1):-1]*(2*pi/L)).^2;
u = @(t,u0) real(ifft(exp(-k2*t).*fft(u0)))
```

455 The Matlab code that solves the Navier–Stokes equation has three parts: define the functions, initialize the variables, and iterate over time. We start by defining functions for  $\hat{\mathbf{H}}$  and for the flux used in the Lax–Wendroff scheme.

```
flux = @(Q,c) c.*diff([Q(end,:);Q]) ...
+ 0.5*c.*((1-c).*diff([Q(end,:);Q;Q(1,:)],2));
H = @(u,v,ikx,iky) -ikx.*fft2(ifft2(u).^2) ...
-iky.*fft2(ifft2(u).*ifft2(v));
```

Now define initial conditions. The variable  $e$  is used for the scaling parameter  $\varepsilon$ , and  $k = \xi$  is a temporary variable used to construct  $ikx = i\xi_x$ ,  $iky = i\xi_y$ , and  $k^2 = -|\xi|^2 = -\xi_x^2 - \xi_y^2$ . We'll take the  $x$ - and  $y$ -dimensions to be the same, but they can easily be modified.

```
L = 2; n = 128; e = 0.001; dt = .001; dx = L/n;
x = linspace(dx,L,n); y = x';
Q = 0.5*(1+tanh(10*(1-abs(L/2 -y)/(L/4)))).*ones(size(x));
u = Q.*((1+0.5*sin(L*pi*x));
v = zeros(size(u));
u = fft2(u); v = fft2(v);
us = u; vs = v;
ikx = 1i*[0:n/2 (-n/2+1):-1]*(2*pi/L);
iky = ikx.';
k2 = ikx.^2+iky.^2;
Hx = H(u,v,ikx,iky); Hy = H(v,u,iky,ikx);
M1 = 1/dt + (e/2)*k2;
M2 = 1/dt - (e/2)*k2;
```

Now, we iterate. At each time step, we evaluate (16.14) and use the Lax–Wendroff method to evaluate the advection equation (16.15). The variable  $Q$  gives the density  $Q$  of the tracer in the advection equation. The variable  $Hx$ , computed using the function  $H$ , is the  $x$ -component of  $\mathbf{H}^n$  and  $Hxo$  is  $x$ -component of  $\mathbf{H}^{n-1}$ . Similarly,  $Hy$  and  $Hy0$  are the  $y$ -components of  $\mathbf{H}^n$  and  $\mathbf{H}^{n-1}$ . The variables  $u$  and  $v$  are the  $x$ - and  $y$ -components of the velocity  $\mathbf{u}^n = (u^n, v^n)$ . Similarly,  $us$  and  $vs$  are  $u^*$  and  $v^*$ , respectively. The variable  $\phi$  is an intermediate variable for  $\Delta^{-1}\nabla \cdot \mathbf{u}^*$ . The tracer  $Q$  is plotted in Figure 16.2 using a contour plot.

```
for i = 1:1200
Q = Q - flux(Q,(dt/dx)*real(ifft2(v))) ...
- flux(Q',(dt/dx)*real(ifft2(u))')';
Hxo = Hx; Hyo = Hy;
Hx = H(u,v,ikx,iky); Hy = H(v,u,iky,ikx);
us = u - us + (1.5*Hx - 0.5*Hxo + M1.*u)./M2;
vs = v - vs + (1.5*Hy - 0.5*Hyo + M1.*v)./M2;
phi = (ikx.*us + iky.*vs)./(k2+(k2==0));
u = us - ikx.*phi;
```

```
v = vs - iky.*phi;
end
contourf(x,y,Q,[.5 .5]); axis equal;
```

462 N = 10000; n = zeros([1 20]);  
for d = 1:20  
for j = 1:N  
n(d) = n(d) + (det(rand(d)>0.5)~-0);  
end  
end  
plot(1:20,n/N,'.-')

463 An easy way to input  $\mathbf{D}_n$  in Matlab is with the diag function:

```
D = diag(ones([n-1 1]),1) ...
- 2*diag(ones([n 1]),0) + diag(ones([n-1 1]),-1);
eig(D)
```

464 The following function implements a naïve determinant:

```
function D = detx(A)
[L,U,P] = lu(A);
s = 1;
for i = 1:length(P)
m = find(P(i+1:end,i));
if m, P([i i+m],:) = P([i+m i],:); s = -1*s; end
end
D = s * prod(diag(U));
end
```

465 The following function implements a naïve reverse Cuthill–McKee algorithm for symmetric matrices:

```
function p = rcuthillmckee(A)
A = spones(A);
[~,r] = sort(sum(A));
p = [];
while ~isempty(r)
q = r(1); r(1) = [];
while ~isempty(q)
p = [p q(1)];
[~,k] = find(A(q(1),r));
q = [q(2:end) r(k)]; r(k) = [];
end
end
```

```

end
end
p = fliplr(p);
end

```

```

A = sprand(1000,1000,0.001); A = A + A';
p = rcuthillmckee(A);
subplot(1,2,1); spy(A,'.',2); axis equal
subplot(1,2,2); spy(A(p,p),'.',2); axis equal

```

- 465 We'll reuse the code on page 593.

```

p = rcuthillmckee(A);
gplot(A(p,p),circular_layout(A),'.-')
axis equal; axis off;

```

- 466 Reading mixed data into MATLAB or Octave can be frustrating even for relatively simple structures. In Octave, we can use the `csvcell` function (from the `io` package) to convert a CSV file into a cell. Then, we can use a couple of loops to divide the cell up into meaningful pieces. The file `diet.mat` contains these variables. The solution to the diet problem is then

```

urlwrite([bucket "diet.mat"],"diet.mat");
load diet.mat
A = values'; b = minimums; c = ones(size(A,2),1);
[z,x,y] = simplex(b,A',c);
cost = z, food{find(y!=0)}

```

or we can use the built-in Octave `glpk` function

```

[x,z] = glpk(c,A,b,[],[],repmat("L",size(b')));
cost = z, food{find(x!=0)}

```

- 467 We'll reuse the function `get_adjacency_matrix` from page 593. Let's also define a function to get the names.

```

function r = get_names(bucket,filename)
 data = urlread([bucket filename '.txt']);
 r = cell2mat(textscan(data,'%s','delimiter', '\n')(1));
end

```

We'll create a sparse, block matrix and define a function to find the shortest path between nodes `a` and `b`.

```

actors = get_names(bucket,"actors");
movies = get_names(bucket,"movies");
B = get_adjacency_matrix(bucket,"actor-movie");
[m,n] = size(B);
A = [sparse(n,n) B';B sparse(m,m)];
actormovie = [actors;movies];

```

```

function path = findpath(A,a,b)
p = -ones(size(A,2),1);
q = a; p(a) = -9999; i = 1;
while i<=length(q),
 k = find(A(q(i),:));
 k = k(p(k)==-1);
 q = [q k]; p(k) = q(i); i = i + 1;
 if any(k==b),
 path = backtrack(p,b); return;
 end
end
display("No path.");
end

```

```

function path = backtrack(p,b)
s = b; i = p(b);
while i != -9999, s = [s i]; i = p(i); end
path = s(end:-1:1);
end

```

```

a = find(ismember(actors,"Bruce Lee"));
b = find(ismember(actors,"Tommy Wiseau"));
actormovie(findpath(A,a,b)){:}

```

468 The Matlab code for exercise 3.4 follows. We'll first download the CSV files.

```

data = urlread([bucket 'filip.csv']);
T = cell2mat(textscan(data,'%f,%f'));
y = T(:,1); x = T(:,2);
data = urlread([bucket 'filip-coeffs.csv']);
beta = cell2mat(textscan(data,'%f'));

```

Now, let's define a function that determines the coefficients and residuals using three different methods and one that evaluate a polynomial given those coefficients. The command  $qr(V, \theta)$  returns an economy-size QR decomposition.

```

function [c,r] = solve_filip(x,y,n)
V = vander(x, n);
c(:,1) = (V'*V)\(V'*y);
[Q,R] = qr(V,0);
c(:,2) = R\Q'*y;
c(:,3) = pinv(V,1e-10)*y;
for i=1:3, r(i) = norm(V*c(:,i)-y); end
end

```

```
build_poly = @(c,X) vander(X,length(c))*c;
```

Now, we can solve the problem and plot the solutions.

```

n = 11;
[c,r] = solve_filip(x,y,n);
X = linspace(min(x),max(x),200);
Y = build_poly(c,X);
plot(X,Y,x,y,'.'); ylim([0.7,0.95])

```

Let's also solve the problem and plot the solutions for the standardized data.

```

zscore = @(X,x) (X .- mean(x))/std(x);
[cond(vander(x,11)) cond(vander(zscore(x,x),11))]
[c,r] = solve_filip(zscore(x,x), zscore(y,y), n);
for i=1:3,
 Y(:,i) = build_poly(c(:,i),zscore(X,x))*std(y).+mean(y);
end
plot(X,Y,x,y,'.'); ylim([0.7,0.95])

```

```

470 data = urlread([bucket 'dailytemps.csv']);
T = textscan(data, '%s%f', 'HeaderLines', 1, 'Delimiter', ',');
day = datenum(T{1}, 'yyyy-mm-dd'); temp = T{2};
day = (day-day(1))/365;
tempmodel = @(t) [sin(2*pi*t) cos(2*pi*t) ones(size(t))];
c = tempmodel(day)\temp;
plot(day,temp,'.', day,tempmodel(day)*c, 'k');

```

- 470 We'll first load the variables into the workspace and build a matrix  $\mathbf{D}_i$  for each  $i \in \{0, 1, \dots, 9\}$ . Then we'll use svds to compute the first  $k = 12$  singular matrices. We only need to keep  $\mathbf{V}_i$ , which contain the a low-dimensional column spaces of the space of digits.

```
k = 12; V = [];
```

```
urlwrite([bucket "emnist-mnist.mat"], "emnist-mnist.mat")
load emnist-mnist.mat
for i = 1:10
 D = dataset.train.images(dataset.train.labels==i-1,:);
 [_,_,V(:, :, i)] = svds(double(D), k);
end
```

Let's display the principal components for the “3” image subspace.

```
pix = reshape(V(:, :, 3+1), [28, 28*k]);
imshow(pix, []); axis off
```

Now, let's find the closest column spaces to our test images  $\mathbf{d}$  by finding the  $i$  with the smallest the residual  $\|\mathbf{V}_i \mathbf{V}_i^T \mathbf{d} - \mathbf{d}\|_2$ .

```
r = [];
d = double(dataset.test.images)';
for i = 1:10
 r(i, :) = sum((V(:, :, i) * (V(:, :, i)' * d) - d).^2);
end
[c, predicted] = min(r);
```

We can examine the accuracy of the solution by building a confusion matrix:

```
for i = 1:10
 x = predicted(find(dataset.test.labels == i-1));
 confusion(i, :) = histc(x, 1:10);
end
```

Each row of the matrix represents the predicted class and each column represents the actual class.

- 471 We use singular value decomposition to find a low-dimensional subspace relating actors and genres. Then we find the closest actors in that subspace using cosine similarity. We'll use the helper functions `get_names` and `get_adjacency_matrix` from page 614.

```
actors = get_names(bucket, "actors");
genres = get_names(bucket, "genres");
A = get_adjacency_matrix(bucket, "movie-genre"); A = A/diag(sum(A));
B = get_adjacency_matrix(bucket, "actor-movie");

[U, S, V] = svds(A*B, 12);
Q = V' ./ sqrt(sum(V' .^ 2));
q = Q(:, find(ismember(actors, "Steve Martin")));
[_, r] = sort(Q'*q, 'descend');
actors(r(1:10)){}:
```

```
[p,r] = sort(U*S*q,'descend');
for i=1:10, printf("%s: %4.3f\n",genres(r(i)){:},p(i)/sum(p)), end
```

```
471 xyt = [3 3 12; 1 15 14; 10 2 13; 12 15 14; 0 11 12];
reference = xyt(1,:); xyt = xyt - reference;
A = [2 2 -2].*xyt; b = (xyt.^2)*[1; 1; -1];
x_ols = A\b + reference';
x_tls = tls(A,b) + reference'
```

```
473 n = 8; N = 2500; E = zeros(n*N,1);
for i = 0:N-1, E(n*i+(1:n)) = eig(randn(n,n)); end
plot(E,'.'); axis equal
```

```
474 function [rho,x] = rayleigh(A)
n = length(A); x = randn([n 1]);
while true
 x = x/norm(x);
 rho = x'*A*x;
 M = A-rho*eye(n);
 if rcond(M)<eps, break; end
 x = M\x;
end
end
```

474 The Matlab function `hess(A)` returns an upper Hessenberg matrix that is unitarily similar to `A`. The function `givens(a,b)` returns the Givens rotation matrix for the vector  $(a, b)$ . The following Matlab code implements the implicit QR method:

```
function eigenvalues = implicitqr(A)
n = length(A);
tolerance = 1e-12;
H = hess(A);
while true
 if abs(H(n,n-1))<tolerance,
 n = n-1; if n<2, break; end;
 end
 Q = givens(H(1,1)-H(n,n),H(2,1));
 H(1:2,1:n) = Q*H(1:2,1:n);
 H(1:n,1:2) = H(1:n,1:2)*Q';
 for i = 2:n-1
 Q = givens(H(i,i-1),H(i+1,i-1));
```

```

H(i:i+1,1:n) = Q*H(i:i+1,1:n);
H(1:n,i:i+1) = H(1:n,i:i+1)*Q';
end
end
eigenvalues = diag(H);
end

n = 20; S = randn(n);
D = diag(1:n); A = S*D*inv(S);
implicitqr(A)

```

- 475 We'll use the helper functions `get_names` and `get_adjacency_matrix` from page 614.

```

actors = get_names("actors");
B = get_adjacency_matrix("actor-movie");
M = sparse(B'*B);
v = ones(size(M,1),1);
for k = 1:10,
 v = M*v; v /= norm(v);
end
[_,r] = sort(v,'descend');
actors(r(1:10)){:}

```

- 476 We approximate the SVD by starting with a set of  $k$  random vectors and performing a few steps of the naïve QR method to generate a  $k$ -dimensional subspace that is relatively close to the space of dominant singular values:

```

function [U,S,V] = randomizedsvd(A,k)
Z = rand([size(A,2) k]);
[Q,R] = qr(A*Z,0);
for i = 1:3
 [Q,R] = qr(A'*Q,0);
 [Q,R] = qr(A*Q,0);
end
[W,S,V] = svd(Q'*A,0);
U = Q*W;
end

```

Let's convert an image to an array, compute its randomized SVD, and then display the original image side-by-side with the rank-reduced version.

```

A = double(rgb2gray(imread(["bucket "red-fox.jpg"]))/255;
[U,S,V] = randomizedsvd(A,10);
imshow([A U*S*V'],[])

```

```
477 m = 5000; j = 1:m;
A = zeros(m,m);
for i = 1:m
 A(i,j) = 1./(1 + (i+j-1).*(i+j)/2 - j);
end
c = svds(A,1)
```

477 The following functions solve the Poisson equation:

```
n = 50; x = (1:n)'/(n+1); dx = 1/(n+1);
[x,y,z] = meshgrid(x);
I = speye(n);
D = spdiags(repmat([1 -2 1],[n 1]),-1:1,n,n);
A = (kron(kron(D,I),I) + kron(I,kron(D,I)) + kron(I,kron(I,D)))/dx^2;
f = ((x-x.^2).* (y-y.^2)+(x-x.^2).* (z-z.^2)+(y-y.^2).* (z-z.^2))(:);
ue = ((x-x.^2).* (y-y.^2).* (z-z.^2)/2)(:);
```

```
function e = stationary(A,b,w,n,ue)
e = zeros(n,1); u = zeros(size(b));
P = tril(A,0) - (1-w)*tril(A,-1);
for i=1:n
 u = u + P\ (b-A*u);
 e(i) = norm(u - ue, 1);
end
end
```

```
function e = conjugategradient(A,b,n,ue)
e = zeros(n,1); u = zeros(size(b));
r = b-A*u; p = r;
for i=1:n
 Ap = A*p;
 a = (r'*p)/(Ap'*p);
 u = u + a.*p; r = r - a.*Ap;
 b = (r'*Ap)/(Ap'*p);
 p = r - b.*p;
 e(i) = norm(u - ue, 1);
end
end
```

```
tic; err(:,1) = stationary(A,-f,0,400,ue); toc
tic; err(:,2) = stationary(A,-f,1,400,ue); toc
tic; err(:,3) = stationary(A,-f,1.9,400,ue); toc
tic; err(:,4) = conjugategradient(A,-f,400,ue); toc
semilogy(err); legend("Jacobi","Gauss-Seidel","SOR","Conj. Grad.")
```

Each function takes around two seconds to complete 400 iterations. Direct computation `tic; A\(-f); toc` takes around 20 seconds, considerably shorter than either Julia or Python.

```
479 n = 20000
d = 2 .^ (0:14); d = [-d;d];
P = spdiags(primes(224737)',0,n,n);
B = spdiags(ones(n,length(d)),d,n,n);
A = P + B;
b = sparse(1,1,1,n,1);
x = pcg(A, b, 1e-15, 100, P); x(1)
```

480 The Matlab code for the radix-3 FFT in exercise 6.1 is

```
function y = fftx3(c)
n = length(c);
omega = exp(-2i*pi/n);
if mod(n,3) == 0
 k = 0:n/3-1;
 u = [fftx3(c(1:3:n-2)).';
 (omega.^k).*fftx3(c(2:3:n-1)).';
 (omega.^(2*k)).*fftx3(c(3:3:n)).'];
 F = exp(-2i*pi/3).^(0:2)'*(0:2));
 y = (F*u).';
else
 F = omega.^[0:n-1]'*[0:n-1];
 y = F*c;
end
end
```

481 Matlab typically works with floating-point numbers, so we can't easily input a large integer directly. Instead, we can input it as a string and then convert the string to an array by subtracting '`'0'`'. For example, '`5472`' - '`'0'`' becomes `[5 4 7 2]`. Because `fft` operates on floating-point numbers we use `round` to convert the `ifft` to nearest integer. Putting everything together we get the following:

```
function [pq] = multiply(p,q)
np = [fliplr(p-'0') zeros([1 length(q)])];
nq = [fliplr(q-'0') zeros([1 length(p)])];
pq = round(ifft(fft(np).*fft(nq)));
carry = fix(pq/10);
while sum(carry)>0,
 pq = (pq - carry*10) + [0 carry(1:end-1)];
```

```

 carry = fix(pq/10);
end
n = max(find(pq));
pq = char(fliplr(pq(1:n))+'0');
end

```

482    **function** f = dct(f)  
       n = size(f,1);  
       w = exp(-0.5i\*pi\*(0:n-1)'/n);  
       f = real(w.\*fft(f([1:2:n n-mod(n,2):-2:2],:)));  
**end**

```

function f = idct(f)
n = size(f,1);
f(1,:) = f(1,:)/2;
w = exp(-0.5i*pi*(0:n-1)'/n);
f([1:2:n n-mod(n,2):-2:2],:) = 2*real(ifft(f./w));
end

```

For images, we make two-dimensional DCT and IDCT operators by applying the DCT or IDCT once in each dimension

```

dct2 = @(f) dct(dct(f')');
idct2 = @(f) idct(idct(f')');

```

Eliminating rows and columns of  $\hat{f}_\xi$  gives a lossy compression. Adding zero rows and columns to  $\hat{f}_\xi$  will rescale the image.

```

A = double(rgb2gray(imread(['bucket "red-fox.jpg'])))/255;
B = dct2(A); B = B(1:50,1:50);
C = zeros(size(A));
C(1:50,1:50) = B;
C = idct2(C);
imshow([A C],[])

```

486 We'll find the solution to the system of equations in exercise 8.11. Let's first define the system and the gradient.

```

f = @(x,y) [(x^2+y^2)^2-2*(x^2-y^2); (x^2+y^2-1)^3-x^2*y^3];
df = @(x,y) [4*x*(x^2+y^2-1), 4*y*(x^2+y^2+1);
 6*x*(x^2+y^2-1)^2-2*x*y^3, 6*y*(x^2+y^2-1)^2-3*x^2*y^2];

```

The homotopy continuation method is

```
function x = homotopy(f,dx,x)
dxdt = @(t,x) -df(x(1),x(2))\f(x(1),x(2));
[t,y] = ode45(dxdt,[0;1],x);
x = y(end,:);
end
```

and Newton's method is

```
function x = newton(f,dx,x)
for i = 1:100
 dx = -df(x(1),x(2))\f(x(1),x(2));
 x = x + dx;
 if norm(dx)<1e-8, return; end
end
```

- 491 The following function computes the coefficients  $\{m_0, m_1, \dots, m_{n-1}\}$  for a spline with periodic boundary conditions. It is assumed that  $y_0 = y_n$ .

```
function m = spline_periodic(x,y)
h = diff(x);
C = circshift(diag(h),[1 0]) + 2*diag(h+circshift(h,[1 0])) ...
 + circshift(diag(h),[0 1]);
d = 6.*diff(diff([y(end-1);y])./[h(end);h]);
m = C\d; m = [m;m(1)];
end
```

To following script picks  $n$  random points and interpolates a parametric spline with periodic boundary conditions using  $nx \times n$  points. To evaluate the spline we use `evaluate_spline` discussed on page 601.

```
n = 10; nx = 20;
x = rand(n,1); y = rand(n,1);
x = [x;x(1)]; y = [y;y(1)];
t = [0;cumsum(sqrt(diff(x).^2+diff(y).^2))];
[_,X] = evaluate_spline(t,x,spline_periodic(t,x),nx*n);
[_,Y] = evaluate_spline(t,y,spline_periodic(t,y),nx*n);
plot(X,Y,x,y,'.','markersize',6);
```

- 492 The following code computes and plots the interpolating curves:

```
n = 20; N = 200;
x = linspace(-1,1,n)'; X = linspace(-1,1,N)';
y = (x>0);
```

```

phi1 = @(x,a) abs(x-a).^3;
phi2 = @(x,a) exp(-20*(x-a).^2);
phi3 = @(x,a) x.^a;
interp = @(phi,a) phi(X,a')*(phi(x,a')\y);

```

```

Y1 = interp(phi1,x);
Y2 = interp(phi2,x);
Y3 = interp(phi3,(0:n-1)');
plot(x,y,X,Y1,X,Y2,X,Y3); ylim([- .5 1.5]);

```

493 We define a function to solve linear boundary value problems.

```

function c = solve(L,f,bc,x)
h = x(2)-x(1); n = length(x);
S = ([1 -1/2 1/6; -2 0 2/3; 1 1/2 1/6]./[h^2 h 1])*L(x);
A(2:n+1,1:n+2) = spdiags(S',[0 1 2],n,n+2);
A(1,1:3) = [1/6 2/3 1/6];
A(n+2,n:n+2) = [1/6 2/3 1/6];
d = [bc(1) f(x) bc(2)];
c = A\d';
end

```

Let's also define a function that will interpolate between collocation points.

```

function [X,Y] = build(c,x,N)
X = linspace(x(1),x(end),N);
h = x(2) - x(1);
i = floor(X/h)+1; i(N) = i(N-1);
C = [c(i) c(i+1) c(i+2) c(i+3)]';
B = @(x) [(1-x).^3; 4-3*(2-x).*x.^2; 4-3*(1+x).*x.^2; x.^3]/6;
Y = sum(C.*B((X-x(i))/h));
end

```

Now, we can solve the Bessel equation.

```

n = 15; N = 141
L = @(x) [x;ones(size(x));x];
f = @(x) zeros(size(x));
b = fzero(@(z) besselj(0, z), 11);
x = linspace(0,b,n);
c = solve(L,f,[1,0],x);
[X,Y] = build(c,x,N);
plot(X,Y,X,besselj(0,X))

```

Finally, we examine the error and convergence rate.

```

n = 10*2.^ (1:6);
for i = 1:length(n)
 x = linspace(0,b,n(i));
 c = solve(L,f,[1,0],x);
 [X,Y] = build(c,x,n(i));
 e(i) = sqrt(sum((Y-besselj(0,X)).^2)/n(i));
end
loglog(n,e,'.-');
s = polyfit(log(n),log(e),1);
printf("slope: %f",s(1))

```

496 n = 128; L = 2;  
 x = (0:n-1)/n\*L-L/2;  
 k = [0:(n/2-1) (-n/2):-1]\*(2\*pi/L);  
 f = exp(-6\*x.^2);  
 for p = 0:0.1:1  
 d = real(ifft((1i\*k).^p.\*fft(f)));  
 plot(x,d,'m'); hold on;  
 end

499 d = [0,1,2,3]; n = length(d);  
 V = fliplr(vander(d)) ./ factorial([0:n-1]);  
 coeffs = inv(V);  
 trunc = coeffs\*d'.^n/factorial(n);

Coefficients are given by `rats(coeffs)` and the coefficients of the truncation error is given by `rats(trunc)`.

500 function a = richardson(f,x,m)  
 for i=1:m  
 D(i) = phi(f,x,2^i);  
 for j = i-1:-1:1  
 D(j) = (4^(i-j)\*D(j+1) - D(j))/(4^(i-j) - 1);  
 end  
 end  
 a = D(1);  
end

500 We'll extend the dual class on page 607 by adding methods for division, square root, and cosine.

```

function h = mrdivide(u,v)
 if ~isa(u,'dual'), u = dual(u,0); end
 if ~isa(v,'dual'), v = dual(v,0); end
 h = dual(u.value/v.value, ...
 (v.value*u.deriv-1*u.value*v.deriv)/(v.value*v.value)));
end
function h = cos(u)
 h = dual(cos(u.value), -1*sin(u.value)*u.deriv);
end
function h = sqrt(u)
 h = dual(sqrt(u.value), u.deriv/(2*sqrt(u.value)));
end

```

If an earlier definition of `dual` is still in our workspace, we may need to refresh the workspace with the command `clear functions`. Now, we can implement Newton's method .

```

function x = get_zero(f,x)
 tolerance = 1e-12; delta = 1;
 while abs(delta)>tolerance,
 fx = f(dual(x,1));
 delta = fx.value/fx.deriv;
 x -= delta;
 end
end

```

```
get_zero(@(x) 4*sin(x) + sqrt(x), 4)
```

To find a minimum or maximum of  $f(x)$ , we substitute the following two lines into the Newton solver:

```

fx = f(dual(dual(x,1),1));
delta = fx.deriv.value/fx.deriv.deriv;

```

```

501 function d = cauchyderivative(f, a, p, n = 20, r = 0.1)
 w = exp(2*pi*1i*(0:(n-1))/n);
 d = factorial(p)/(n*r^p)*sum(f(a+r*w)./w.^p)
end

```

```
f = @(x) exp(x)/(cos(x).^3 + sin(x).^3)
cauchyderivative(f, 0, 6)
```

- 502 The following function computes the nodes and weights for Gaussian–Legendre quadrature by using Newton’s method to find the roots of  $P_n(x)$ :

```
function [x,w] = gauss_legendre(n)
 x = -cos((4*(1:n)-1)*pi/(4*n+2))';
 dx = ones(n,1);
 dP = 0;
 while(max(abs(dx))>1e-16),
 P = [x ones(n,1)];
 for k=2:n
 P = [((2*k - 1)*x.*P(:,1)-(k-1)*P(:,2))/k, P(:,1)];
 end
 dP = n*(x.*P(:,1) - P(:,2))./(x.^2-1);
 dx = P(:,1) ./ dP(:,1);
 x = x - dx;
 end
 w = 2./((1-x.^2).*dP(:,1).^2);
end
```

- 502 We’ll modify the implementation of the Golub–Welsch algorithm from page 609 to Gauss–Hermite polynomials.

```
function [nodes,weights] = gauss_hermite(n)
 b = (1:n-1)/2;
 a = zeros(n,1);
 scaling = sqrt(pi);
 [v,s] = eig(diag(sqrt(b),1) + diag(a) + diag(sqrt(b),-1));
 weights = scaling*v(1,:).^2;
 nodes = diag(s);
end
```

Now, we can solve the problem.

```
[s,w] = gauss_hermite(20);
u0 = @(x) sin(x);
u = @(t,x) w * u0(x-2*sqrt(t)*s)/sqrt(pi);
x = linspace(-12,12,100);
plot(x,u(1,x))
```

- 502 Let’s define a general function

```
mc_pi = @(n,d,m) sum(sum(rand(d,n,m).^2,1)<1)./n*2^d;
```

that computes the volume of an  $d$ -sphere using  $n$  samples repeated  $m$  times. We can verify the convergence rate of by looping over several values of  $n$ .

```
m = 20; error = []; N = 2 .^ (1:20);
for n = N
 error = [error sum(abs(pi - mc_pi(n,2,m)))/m];
end
s = polyfit(log(N),log(error),1);
printf("slope: %f",s(1))
loglog(N,exp(s(2)).*N.^s(1),N,error, '.');
```

- 504 The following code plots the region of absolute stability for a Runge–Kutta method with tableau A and b:

```
n = length(b);
N = 100;
[x,y] = meshgrid(linspace(-4,4,N));
lk = x + 1i*y;
E = ones(n,1);
for i = 1:N, for j=1:N
 r(i,j) = 1+ lk(i,j) * b*((eye(n) - lk(i,j)*A)\E);
end, end
contour(x,y,abs(r),[1 1], 'k');
axis([-4 4 -4 4]);
```

- 505  $i = (0:3)';$   
 $a = ((-(i+1)').^i ./ factorial(i))\ [1;0;0;0];$   
 $b = ((-i').^i ./ factorial(i))\ [0;1;0;0];$

- 507 The following function returns the orbit of points in the complex plane for an  $n$ th order Adams–Bashforth–Moulton PE(CE) $^m$ . It calls the function multistepcoeffs defined on page 609.

```
function z = PECE(n,m)
[_,a] = multistepcoeffs(1,1:n);
[_,b] = multistepcoeffs(1,0:n);
for i = 1:200
 r = exp(2i*pi*(i/200));
 c(1) = r - 1;
 c(2:m+1) = r + r.^(1:n)*b(2:end)/b(1);
 c(m+2) = r.^(1:n)*a/b(1);
 z(i,:) = roots(fliplr(c))'/b(1);
end
end
```

```
for i= 1:4
 plot(PECE(1,i),'.k'); hold on; axis equal
end
```

- 508 Let's write a function that computes the Padé coefficients.

```
function [p,q] = pade(a,m,n)
A = eye(m+n+1);
for i=1:n, A(i+1:end,m+1+i) = -a(1:m+n+1-i); end
pq = A\a(1:m+n+1);
p = pq(1:m+1); q = [1; pq(m+2:end)];
end
```

Now, compute the coefficients of  $P_m(x)$  and  $Q_n(x)$  for Taylor polynomial approximation of  $\log(x + 1)$ .

```
m = 3; n = 2
a = [0; ((-1).^(0:m+n)./(1:m+n+1))]
(p,q) = pade_approx(a,m,n)
```

Finally, shift and combine the coefficients using upper inverse Pascal matrices.

```
S = @(n) inv(pascal(n,-1)');
S(m+1)*p, S(n+1)*q
```

- 508 The solution to the SIR problem is

```
SIR = @(t,y,b,g) [-b*y(1)*y(2);b*y(1)*y(2)-g*y(2);g*y(2)];
tspan = linspace(0,15,100); y0 = [0.99, 0.01, 0];
[t,y] = ode45(@(t,y) SIR(t,y,2,0.4),tspan,y0);
plot(t,y(:,1),t,y(:,2),t,y(:,3));
```

We can replace the interval  $[0 \ 15]$  in  $\text{ode45}$  with  $\text{linspace}(0,15,100)$  to evaluate the solution at additional points for a smoother plot.

- 509 The solution to the Duffing equation is

```
duffing = @(t,x,g) [x(2); -g*x(2)+x(1)-x(1).^3+0.3*cos(t)];
tspan = linspace(0,200,2000);
[t,x] = ode45(@(t,x) duffing(t,x,0.37), tspan, [1,0]);
plot(x(:,1),x(:,2));
```

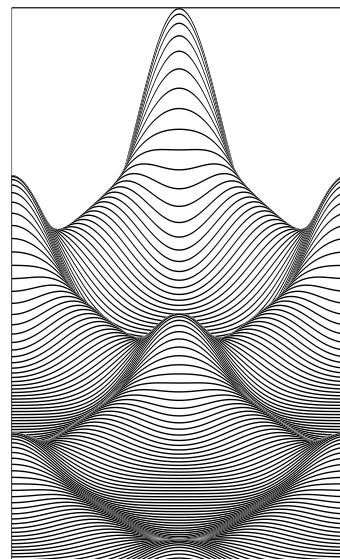
- 510 The following code solves the initial value problem and computes the error at the second boundary point:

```
function error = shooting(x,f,xspan,bc)
 [t,y] = ode45(f,xspan,[bc(1),x]);
 error = y(end,1)- bc(2);
end
```

The function `fsolve` calls the function `shooting` that solves an initial value problem and computes the error in the solutions at the second boundary point.

```
xspan = [-12, 0]; bc = [1, 1]; guess = 5;
airy = @(x,y) [y(2);x*y(1)];
v = fsolve(@(x) shooting(x,airy,xspan,bc),guess)
```

- 512 Displaying time-varying dynamics can be challenging. A simple approach in Matlab is adding a plot command along with a `drawnow` statement inside a loop. This approach can be slow and doesn't always work in different Octave environments. Another approach is saving the plots as PDF or PNG files, and then using an external program like `ffmpeg` to convert the stack of images into a gif or mp4. A different approach to capturing time-varying dynamics is by layering snapshots as a still image. Because such a visualization gets messy when there are many layers, we can offset each layer horizontally or vertically. The figure on the right layers the solutions one on top of the other like stacked paper cut-outs, shifting each new one down slightly. From it we see the undulating behavior of the Dufort–Frankel solution.



```
dx = 0.01; dt = 0.01;
L = 1; x = (-L:dx:L)'; m = length(x);
V = exp(-8*x.^2); U = V;
c = dt/dx^2; a = 0.5 + c; b = 0.5 - c;
B = c*spdiags([ones(m,1),zeros(m,1),ones(m,1)],-1:1,m,m);
B(1,2) = B(1,2)*2; B(end,end-1) = B(end,end-1)*2;
for i = 1:420,
 if mod(i,3)==1, area(x, (V-i/300),-1,'facecolor','w'); hold on; end
 Vo = V; V = (B*V+b*U)/a; U = Vo;
end
ylim([-1,1]); set(gca,'xtick',[],'ytick',[])
```

- 513 The following Matlab code solves the Schrödinger equation:

```
function psi = schroedinger(n,m,eps)
x = linspace(-4,4,n)'; dx = x(2)-x(1); dt = 2*pi/m; V = x.^2/2;
psi = exp(-(x-1).^2/(2*eps))/(pi*eps)^(1/4);
diags = repmat([1 -2 1],[n 1])/dx^2;
D = 0.5i*eps*spdiags(diags,-1:1,n,n) - 1i/eps*spdiags(V,0,n,n);
D(1,2) = 2*D(1,2); D(end,end-1) = 2*D(end,end-1);
A = speye(n) + (dt/2)*D;
B = speye(n) - (dt/2)*D;
for i = 1:m
 psi = B\A*psi;
end
```

We'll loop over several values for time steps and mesh sizes and plot the error.

```
eps = 0.3; m = 20000; n = floor(logspace(2,3.7,6));
x = linspace(-4,4,m)';
psi_m = -exp(-(x-1).^2/(2*eps))/(pi*eps)^(1/4);
for i = 1:length(n),
 x = linspace(-4,4,n(i))';
 psi_n = -exp(-(x-1).^2/(2*eps))/(pi*eps)^(1/4);
 error_t(i) = norm(psi_m - schroedinger(m,n(i),eps))/m;
 error_x(i) = norm(psi_n - schroedinger(n(i),m,eps))/n(i);
end
loglog(2*pi./n,error_t,'.-r',8./n,error_x,'.-k');
```

- 515 Let's solve a radially symmetric heat equation. Although we divide by zero at  $r = 0$  when constructing the Laplacian operator, the resulting `inf` is overwritten when we apply the boundary condition.

```
T = 0.5; m = 100; n = 100;
r = linspace(0,2,m)'; dr = r(2)-r(1); dt = T/n;
u = tanh(32*(1-r));
tridiag = [1 -2 1]/dr^2 + (1./r).*[-1 0 1]/(2*dr);
D = spdiags(tridiag,-1:1,m,m)';
D(1,1:2) = [-4 4]/dr^2; D(m,m-1:m) = [2 -2]/dr^2;
A = speye(m) - 0.5*dt*D;
B = speye(m) + 0.5*dt*D;
for i = 1:n
 u = A\B*u;
end
area(r,u,-1,"edgecolor",[1 .5 .5],"facecolor",[1 .8 .8]);
```

Alternatively, a much slower, but high-order BDF routine can be used in place of the Crank–Nicolson routine:

```
options = odeset('JPattern', spdiags(ones([n 3]), -1:1,m,m));
[t,u] = ode15s(@(t,u) D*u,[0 T],u,options);
```

- 517 We'll first define a function as a logit analogue to `linspace`.

```
logitspace = @(x,n,p) x*atanh(linspace(-p,p,n)')/atanh(p);
```

Next, we'll define a general discrete Laplacian operator. The following function returns a sparse matrix in diagonal format (DIA). Two inconsequential elements of array `d` are replaced with nonzero numbers to avoid divide by zero warnings.

```
function D = laplacian(x)
 h = diff(x); h1 = h(1:end-1); h2 = h(2:end); n = length(x);
 diags = 2./[h1(1).^2, -h1(1).^2, 0;
 h2.*(h1+h2), -h1.*h2, h1.*(h1+h2);
 0, -h2(end).^2, h2(end).^2];
 D = spdiags(diags,-1:1,n,n)';
end
```

Now, we write a function to solve the heat equation.

```
function u = heat_equation(x,t,u):
 n = 40; dt = t/n;
 D = laplacian(x);
 A = speye(n) - 0.5*dt*D;
 B = speye(n) + 0.5*dt*D;
 for i = 1:m
 u = A\B*u;
 end
end
```

Finally, we generate a solution using logit spacing.

```
phi = @(x,t,s) exp(-s*x.^2/(1+4*s*t))/sqrt(1+4*s*t);
t = 15; m = 40;
x = logitspace(20,m,.999);
u = heat_equation(x,t,phi(x,0,10));
plot(x,u,'.-',x,phi(x,t,10),'k')
```

- 518 Here's a solution to the Allen–Cahn equation using Strang splitting:

```
L = 16; n = 400; dx = L/m;
T = 4; m = 1600; dt = T/n;
x = linspace(-L/2,L/2,m);
u = tanh(x.^4 - 16*(2*x.^2-x'.^2));
```

```

D = spdiags(repmat([1 -2 1],[m 1]),-1:1,m,m)/dx^2;
D(1,2) = 2*D(1,2); D(end,end-1) = 2*D(end,end-1);
A = speye(m) + 0.5*dt*D;
B = speye(m) - 0.5*dt*D;
f = @(u,dt) u./sqrt(u.^2 - (u.^2-1).*exp(-50*dt));
u = f(u,dt/2);
for i = 1:n
 u = (B\A*(B\A*u))';
 if i<n, u = f(u,dt); end
end
u = f(u,dt/2);

```

We can plot the solution using the code

```
image((u+1)/2*100); colormap(gray(100));
```

We can animate the time evolution of the solution by adding an `imwrite` command inside the loop to save each iteration to a stack of sequential pngs and then using a program such as ffmpeg to compile the pngs into an mp4.

- 524 The following code uses the local Lax–Friedrichs method to solve Burgers' equation.

```

m = 100; x = linspace(-1,3,m); dx = x(2)-x(1);
n = 100; Lt = 4; dt = Lt/n;
lambda = dt/dx;
f = @(u) u.^2/2; fp = @(u) u;
u = (x>=0)&(x<=1);
for i = 1:n
 fu = f([u(1) u]); fpu = fp([u(1) u]);
 a = max(abs(fu(1:n-1)),abs(fu(2:n)));
 F = (fu(1:n-1)+fu(2:n))/2 - a.*diff(u)/2;
 u = u - lambda*(diff([0 F 0]));
end
area(x,u,"edgecolor", [.3 .5 1], "facecolor", [.6 .8 1]);

```

- 524 Let's first define a few functions.

```

function s = slope(u)
 limiter = @(t) (abs(t)+t)./(1+abs(t));
 du = diff(u);
 s = [[0 0];du(2:end,:).*limiter(du(1:end-1,:))./(du(2:end,:)) ...
 + (du(2:end,:)==0));[0 0]];
end

```

Now, we can solve the dam-break problem.

```

F = @(u) [u(:,1).*u(:,2), u(:,1).*u(:,2).^2+0.5*u(:,1).^2];
m = 1000; x = linspace(-.5,.5,m)'; dx = x(2)-x(1);
T = 0.25; n = ceil(T/(dx/2)); dt = (T/n)/2; c = dt/dx;
u = [0.8*(x<0)+0.2,0*x];
j = 1:m-1;
for i = 1:n
 v = u-0.5*c*slope(F(u));
 u(j+1,:)=(u(j,:)+u(j+1,:))/2 - diff(slope(u))/8-c*diff(F(v));
 v = u-0.5*c*slope(F(u));
 u(j,:)=(u(j,:)+u(j+1,:))/2 - diff(slope(u))/8-c*diff(F(v));
end
plot(x,u(:,1));

```

Other slope limiters we might try include the superbee and the minmod:

```

limiter = @(t) max(0,max(min(2*t,1),min(t,2)));
limiter = @(t) max(0,min(1,t));

```

526 m=10;  
 $x = \text{linspace}(0,1,m)'$ ;  $h = x(2)-x(1)$ ;  
 $A = \text{diag}(\text{repmat}(-1/h-h/6,[m-1 1]),-1) + \text{diag}(\text{repmat}(1/h-h/3,[m 1]))$ ;  
 $A = A + A'$ ;  $A(1,1)=A(1,1)/2$ ;  $A(m,m)=A(m,m)/2$ ;  
 $b=[-2/3*h^3;-4/3*h^3-8*h*x(2:m-1).^2;-4*h+8*h^2/3-2*h^3/3+1]$ ;  
 $u=A\b$ ;  
 $s=(-16)+8.*x.^2+15.*\cos(x).*\csc(1)$ ;  
 $\text{plot}(x,s,'o-',x,u,'.-')$ ;

527 m = 8;  
 $x = \text{linspace}(0,1,m+2)$ ;  $h = x(2)-x(1)$ ;  
 $D = @(a,b,c) (\text{diag}(a*\text{ones}(m-1,1),-1) + \dots$   
 $\text{diag}(b*\text{ones}(m,1),0) + \text{diag}(c*\text{ones}(m-1,1),1))/h^3$ ;  
 $M = [D(-12,24,-12) D(-6,0,6);D(6,0,-6) D(2,8,2)]$ ;  
 $b = [\text{ones}([m 1])*h*384;\text{zeros}([m 1])]$ ;  
 $u = M\b$ ;  
 $\text{plot}(x,16*(x.^4 - 2*x.^3 + x.^2), 'o-', x, [0;u(1:m);0], '-.')$ ;

528 m = 128;  $x = (1:m)'/m*(2*pi)-pi$ ;  
 $k = 1i*[0:m/2 -m/2+1:-1]'$ ;  
 $f = @(t,u) -(i\text{fft}(k.*\text{fft}(0.5*u.^2)))$ ;  
 $[t,u] = \text{ode45}(f,[0 1.5],\exp(-x.^2))$ ;  
 $\text{plot}(x,u(\text{end},:))$

- 529 The following Matlab code solves the KdV equation using integrating factors.  
We first set the parameters:

```
phi = @(x,x0,c) 0.5*c*sech(sqrt(c)/2*(x-x0)).^2;
L = 30; T = 1.0; m = 256;
x = (1:m)'./m*L-L/2;
k = 1i*[0:(m/2) (-m/2+1):-1]'.*(2*pi/L);
```

We define the integrating factor and right-hand side of the differential equation.

```
G = @(t) exp(-k.^3*t);
f = @(t,w) -G(t).'\(3*k.*fft(ifft(G(t).*w).^2));
```

Then, we solve the problem using ode45.

```
u = phi(x,-4,4) + phi(x,-9,9);
w = fft(u);
[t,w] = ode45(f,linspace(0,T,40),w);
u = real(ifft(G(t').*w.').');
```

Finally, we present the solution as a waterfall plot.

```
for i=40:-1:1,
 area(x, T*(u(i,:)+i)/40,'facecolor','w'); hold on;
end
```

Be careful about the dimensions of the output arrays t and w. This and using conjugate transpose ('.') instead of a regular transpose (.') provided me with what seemed like hours of debugging amusement.

- 531 We'll use Strang splitting to solve the Swift–Hohenberg equation.

```
eps = 1; m = 256; L = 100; n = 2000; dt=100/n;
U = (rand(m)>.5)-0.5;
colormap(gray(256))
k = [0:(m/2) (-m/2+1):-1]'.*(2*pi/L);
D2= (1i*k).^2+(1i*k').^2;
E = exp(-(D2+1).^2*dt);
f = @(U) U./sqrt(U.^2/eps + exp(-dt*eps)*(1-U.^2/eps));
for i=1:n
 U = f(ifft2(E.*fft2(f(U))));
end
imshow(real(U))
```



## References

- Forman S. Acton. *Numerical Methods that Work*. Mathematical Association of America, 1990.
- Uri M. Ascher, Steven J. Ruuth, and Brian T. R. Wetton. Implicit-explicit methods for time-dependent partial differential equations. *SIAM Journal on Numerical Analysis*, 32(3):797–823, 1995.
- Uri M. Ascher, Steven J. Ruuth, and Raymond J. Spiteri. Implicit-explicit Runge–Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2):151–167, 1997.
- Kendall Atkinson and Weimin Han. *Theoretical Numerical Analysis*, volume 39 of *Texts in Applied Mathematics*. Springer, New York, second edition, 2005.
- Jared L. Aurentz, Thomas Mach, Raf Vandebril, and David S. Watkins. Fast and backward stable computation of roots of polynomials. *SIAM Journal on Matrix Analysis and Applications*, 36(3):942–973, 2015.
- Santiago Badia and Francesc Verdugo. Gridap: An extensible finite element toolbox in Julia. *Journal of Open Source Software*, 5(52):2520, 2020.
- Serge Bernstein. Démonstration du théorème de Weierstrass fondée sur le calcul des probabilités. *Communications of the Kharkov Mathematical Society*, 13(1):1–2, 1912.
- Michael Blair, Sally Obenski, and Paula Bridickas. Patriot missile defense: Software problem led to system failure at Dhahran. Technical report, GAO/IMTEC-92-26, United States General Accounting Office, 1992.
- Ignace Bogaert. Iteration-free computation of Gauss–Legendre quadrature nodes and weights. *SIAM Journal on Scientific Computing*, 36(3):A1008–A1026, 2014.

- Max Born. *The Restless Universe*. Courier Corporation, 1951.
- George E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29:610–611, 1958.
- Daniel R. L. Brown. SEC 2: Recommended elliptic curve domain parameters. *Standards for Efficient Cryptography*, 2010. <https://www.secg.org/sec2-v2.pdf>.
- Adhemar Bultheel. Learning to swim in a sea of wavelets. *Bulletin of the Belgian Mathematical Society*, 2(1):1–46, 1995.
- John C. Butcher. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, 2008.
- Emmanuel J. Candès and Michael B. Wakin. An introduction to compressive sampling. *IEEE Signal Processing Magazine*, 25(2):21–30, 2008.
- Yoonsuck Choe, B. H. McCormick, and W. Koh. Network connectivity analysis on the temporally augmented *C. elegans* web: A pilot study. In *Society for Neuroscience Abstracts*, volume 30. 2004. Program No. 921.9.
- Fan R. K. Chung and Fan Chung Graham. *Spectral Graph Theory*. American Mathematical Society, 1997.
- Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: an extension of MNIST to handwritten letters. *CoRR*, abs/1702.05373, 2017. <http://arxiv.org/abs/1702.05373>.
- James W. Cooley. The re-discovery of the fast Fourier transform algorithm. *Microchimica Acta*, 93(1):33–45, 1987.
- James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.
- Robert M. Corless, Gaston H. Gonnet, David E. G. Hare, David J. Jeffrey, and Donald E. Knuth. On the lambertw function. *Advances in Computational Mathematics*, 5(1):329–359, 1996.
- Steven M. Cox and Paul C. Matthews. Exponential time differencing for stiff systems. *Journal of Computational Physics*, 176(2):430–455, 2002.
- Charles Francis Curtiss and Joseph O. Hirschfelder. Integration of stiff equations. *Proceedings of the National Academy of Sciences of the United States of America*, 38(3):235, 1952.

- Germund Dahlquist. Convergence and stability in the numerical integration of ordinary differential equations. *Mathematica Scandinavica*, pages 33–53, 1956.
- Germund Dahlquist. A special stability problem for linear multistep methods. *BIT Numerical Mathematics*, 3(1):27–43, 1963.
- George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
- George B. Dantzig. Reminiscences about the origins of linear programming. Technical report, Stanford University Systems Optimization Laboratory, April 1981. Technical report SOL 81-5.
- George B. Dantzig. The diet problem. *Interfaces*, 20(4):43–47, 1990.
- George Bernard Dantzig. *Linear Programming and Extensions*, volume 48. Princeton University Press, 1998.
- Ingrid Daubechies. *Ten Lectures on Wavelets*. SIAM, 1992.
- James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.
- Peter Deuflhard and Andreas Hohmann. *Numerical Analysis in Modern Scientific Computing*, volume 43 of *Texts in Applied Mathematics*. Springer–Verlag, New York, second edition, 2003.
- Persi Diaconis, R. L. Graham, and William M. Kantor. The mathematics of perfect shuffles. *Advances in Applied Mathematics*, 4(2):175–196, 1983.
- Paul Adrien Maurice Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35(3):416–418, 1939.
- Arthur Conan Doyle. *The Complete Sherlock Holmes*. Doubleday Books, 1930.
- Laurent Duval. WITS = where is the starlet?, 2019. <http://www.laurent-duval.eu/siva-wits-where-is-the-starlet.html>.
- Roger Eckhardt. Stan Ulam, John von Neumann, and the Monte Carlo method. *Los Alamos Science*, 15:131, 1987.
- Björn Engquist and Andrew Majda. Absorbing boundary conditions for the numerical simulation of waves. *Mathematics of Computation*, 31(139):629–651, July 1977.
- Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Imperialis Petropolitanae*, pages 128–140, 1741.

- Lawrence C. Evans. *Partial differential equations*. American Mathematical Society, 2010.
- Rida T. Farouki. The Bernstein polynomial basis: A centennial retrospective. *Computer Aided Geometric Design*, 29(6):379–419, 2012.
- Etienne Forest and Ronald D. Ruth. Fourth-order symplectic integration. *Physica D: Nonlinear Phenomena*, 43(1):105–117, 1990.
- John G. F. Francis. The QR transformation a unitary analogue to the I<sub>r</sub> transformation—part 1. *The Computer Journal*, 4(3):265–271, 1961.
- John G. F. Francis. The QR transformation—part 2. *The Computer Journal*, 4(4):332–345, 1962.
- Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics, New York, second edition, 2016.
- Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.
- Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.
- Hannah Fry. *Hello World: Being Human in the Age of Algorithms*. WW Norton & Company, 2018.
- Walter Gautschi. *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press on Demand, 2004.
- Walter Gautschi. Orthogonal polynomials, quadrature, and approximation: Computational methods and software (in MATLAB). In *Orthogonal polynomials and special functions*, pages 1–77. Springer, 2006.
- Walter Gautschi. *A Software Repository for Orthogonal Polynomials*. SIAM, 2018.
- Walter Gautschi. *A Software Repository for Gaussian Quadratures and Christoffel Functions*. SIAM, 2020.
- Amanda Gefter. The man who tried to redeem the world with logic. *Nutilus Quarterly*, 21, 2015.
- Bernard R. Gelbaum and John M. H. Olmsted. *Counterexamples in Analysis*. Courier Corporation, 2003.

- John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.
- Nicolas Gillis. The why and how of nonnegative matrix factorization. *Regularization, Optimization, Kernels, and Support Vector Machines*, 12(257):257–291, 2014.
- Gabriel Goh. Why momentum really works. *Distill*, 2017. doi: 10.23915/distill.00006. <http://distill.pub/2017/momentum>.
- Gene Golub and Frank Uhlig. The QR algorithm: 50 years later its genesis by John Francis and Vera Kublanovskaya and subsequent developments. *IMA Journal of Numerical Analysis*, 29(3):467–485, 2009.
- Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.
- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.
- Joseph F. Grcar. John von Neumann’s analysis of Gaussian elimination and the origins of modern numerical analysis. *SIAM Review*, 53(4):607–682, 2011.
- John A. Gubner. Gaussian quadrature and the eigenvalue problem. Technical report, University of Wisconsin, 2009.
- Alfred Haar. *Zur Theorie der Orthogonalen Funktionensysteme*. Georg-August-Universitat, Gottingen., 1909.
- Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, volume 14. Springer–Verlag, Berlin, 2010.
- Ernst Hairer, Syvert Paul Norsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer–Verlag, Berlin, 1993.
- Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration: Structure-preserving Algorithms for Ordinary Differential Equations*, volume 31. Springer, 2006.
- Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.
- Paul R. Halmos. How to write mathematics. *L’Enseignement Mathématique*, 16 (2):123–152, 1970.

- Godfrey Harold Hardy. Weierstrass's non-differentiable function. *Transactions of the American Mathematical Society*, 17(3):301–325, 1916.
- Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media, 2009.
- Deanna Haunsperger and Robert Thompson. *101 Careers in Mathematics*, volume 64. American Mathematical Society, 2019.
- Oliver Heaviside. XI. On the forces, stresses, and fluxes of energy in the electromagnetic field. *Philosophical Transactions of the Royal Society of London.(A.)*, 183:423–480, 1892.
- Nicholas J. Higham and Fran oise Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1185–1201, 2000.
- Marlis Hochbruck and Alexander Ostermann. Exponential integrators. *Acta Numerica*, 19(May):209–286, 2010.
- Yifan Hu. Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71, 2005.
- T. Huckle and T. Neckel. *Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science*. Software, Environments, and Tools. Society for Industrial and Applied Mathematics, 2019.
- William Kahan and K. C. Ng. Freely distributable LIBM e\_sqrt.c, 1995. Available at the Netlib repository [http://www.netlib.org/fdlibm/e\\_sqrt.c](http://www.netlib.org/fdlibm/e_sqrt.c).
- Anna Kalogirou. *Nonlinear dynamics of surfactant-laden multilayer shear flows and related systems*. PhD thesis, Queensland University of Technology, 2014.
- Aly-Khan Kassam and Lloyd N. Trefethen. Fourth-order time-stepping for stiff PDEs. *SIAM Journal on Scientific Computing*, 26(4):1214–1233, 2005.
- Christopher A. Kennedy and Mark H. Carpenter. Additive Runge–Kutta schemes for convection–diffusion–reaction equations. *Applied Numerical Mathematics*, 44(1-2):139–181, 2003.
- Christopher A. Kennedy and Mark H. Carpenter. Diagonally implicit runge–kutta methods for ordinary differential equations, a review. Technical report, National Aeronautics and Space Administration, Langley Research Center, 2016.
- A. Kiely and M. Klimesh. The ICER progressive wavelet image compressor. *The Interplanetary Network Progress Report*, 42(155), 2003.

- A. Kiely, M. Klimesh, H. Xie, and N. Aranki. Icer-3d: A progressive wavelet-based compressor for hyperspectral images. *The Interplanetary Network Progress Report*, 42(164), 2006.
- David Kincaid and Ward Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Brooks/Cole Publishing Co., Pacific Grove, CA, third edition, 2001.
- Diederik P. Kingma and Jimmy Ba. Adam: a method for stochastic optimization, 2017.
- Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
- John Denholm Lambert. *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem*. John Wiley & Sons, 1991.
- Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Springer, 2017a. ISBN 978-3-319-52461-0. doi: 10.1007/978-3-319-52462-7.
- Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python: The FEniCS Tutorial I*. Springer Nature, 2017b.
- Amy N. Langville and Carl D. Meyer. Deeper inside PageRank. *Internet Mathematics*, 1(3):335–380, 2004.
- Dirk Laurie. Calculation of Gauss–Kronrod quadrature rules. *Mathematics of Computation*, 66(219):1133–1145, 1997.
- Peter Lax. *Linear Algebra and Its Applications*. Pure and Applied Mathematics. Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, second edition, 2007.
- Peter Lax and Burton Wendroff. Systems of conservation laws. Technical report, Los Alamos National Laboratory, 1959.
- Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998.
- Dongsun Lee, Joo-Youl Huh, Darae Jeong, Jaemin Shin, Ana Yun, and Junseok Kim. Physical, mathematical, and numerical derivations of the Cahn–Hilliard equation. *Computational Materials Science*, 81:216–225, 2014.
- Randall J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002.
- Mario Livio. *The Equation that Couldn't Be Solved: How Mathematical Genius Discovered the Language of Symmetry*. Simon and Schuster, 2005.

- Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN 978-3-642-23098-1. doi: 10.1007/978-3-642-23099-8.
- Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of Atmospheric Sciences*, 20(2):130–141, 1963.
- David Lusseau, Karsten Schneider, Oliver J. Boisseau, Patti Haase, Elisabeth Slooten, and Steve M. Dawson. The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405, 2003.
- James N. Lyness and Cleve B. Moler. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210, 1967.
- Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, New York, 1982.
- Kyle T. Mandli, Aron J. Ahmadia, Marsha Berger, Donna Calhoun, David L. George, Yiannis Hadjimichael, David I. Ketcheson, Grady I. Lemoine, and Randall J. LeVeque. Clawpack: Building an open source ecosystem for solving hyperbolic PDEs. *PeerJ Computer Science*, 2:e68, 2016.
- Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.
- Robert M. May. Simple mathematical models with very complicated dynamics. *Nature*, 261:459, 1976.
- Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.
- Cleve B. Moler. *Numerical Computing with MATLAB*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2004. Available online at <https://www.mathworks.com/moler/chapters.html>.
- Cleve B. Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix. *SIAM Review*, 20(4):801–836, 1978.
- Cleve B. Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003.
- K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations*. Cambridge University Press, Cambridge, second edition, 2005.

- Richard D. Neidinger. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Review*, 52(3):545–563, 2010.
- John A. Nelder and Roger Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.
- Haim Nessyahu and Eitan Tadmor. Non-oscillatory central differencing for hyperbolic conservation laws. *Journal of Computational Physics*, 87(2):408–463, 1990.
- F. T. Nieuwstadt and J. A. Steketee. *Selected Papers of J. M. Burgers*. Springer Science & Business Media, 2012.
- Kyle A. Novak and Laura J. Fox. *Special Functions of Mathematical Physics: A Tourist’s Guidebook*. Equal Share Press, 2019.
- Karlheinz Ochs. A comprehensive analytical solution of the nonlinear pendulum. *European Journal of Physics*, 32(2):479, 2011.
- Robert E. O’Malley Jr. *Singular Perturbation Methods for Ordinary Differential Equations*. Springer-Verlag, 1991.
- Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
- Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.
- Thomas N. L. Patterson. The optimum addition of points to quadrature formulae. *Mathematics of Computation*, 22(104):847–856, 1968.
- Donald W. Peaceman and Henry H. Rachford, Jr. The numerical solution of parabolic and elliptic differential equations. *Journal of the Society for Industrial and Applied Mathematics*, 3(1):28–41, 1955.
- Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.
- Alfio Quarteroni and Fausto Saleri. *Scientific Computing with MATLAB and Octave*, volume 2 of *Texts in Computational Science and Engineering*. Springer-Verlag, Berlin, second edition, 2006.
- Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical Mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer-Verlag, Berlin, second edition, 2007.

- Christopher Rackauckas. A comparison between differential equation solver suites in MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran, 2018. <https://thewinnow.com>.
- Christopher Rackauckas and Qing Nie. Differentialequations.jl—a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software*, 5(1):15, 2017.
- John K. Reid and Jennifer A. Scott. Reducing the total bandwidth of a sparse unsymmetric matrix. *SIAM Journal on Matrix Analysis and Applications*, 28(3): 805–821, 2006. <https://www.numerical.rl.ac.uk/reports/rsRAL2005001.pdf>.
- Edward L Reiss. A new asymptotic method for jump phenomena. *SIAM Journal on Applied Mathematics*, 39(3):440–455, 1980.
- Philip L. Roe. Modelling of discontinuous flows. *Lectures in Applied Mathematics*, 22, 1985.
- Paul Romer. Jupyter, Mathematica, and the future of the research paper, April 2018. <https://paulromer.net/jupyter-mathematica-and-the-future-of-the-research-paper>.
- Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386, 1958.
- H. H. Rosenbrock. Some general implicit processes for the numerical solution of differential equations. *The Computer Journal*, 5(4):329–330, 1963.
- David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.
- Siegfried M. Rump. Algorithms for verified inclusions: Theory and practice. In *Reliability in Computing*, pages 109–126. Elsevier, 1988.
- Youcef Saad. *Numerical Methods for Large Eigenvalue Problems*. Algorithms and Architectures for Advanced Scientific Computing. Manchester University Press, Manchester, 1992.
- Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.
- Lawrence F. Shampine. Design of software for ODEs. *Journal of Computational and Applied Mathematics*, 205(2):901–911, 2007.
- Lawrence F. Shampine and Robert M. Corless. Initial value problems for ODEs in problem solving environments. *Journal of Computational and Applied Mathematics*, 125(1-2):31–40, 2000.

- Lawrence F. Shampine and Mark W. Reichelt. The MATLAB ODE suite. *SIAM Journal on Scientific Computing*, 18(1):1–22, 1997.
- Chi-Wang Shu and Stanley Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of Computational Physics*, 77(2):439–471, 1988.
- Bonnie Shulman. Math-alive! Using original sources to teach mathematics in social context. *Problems, Resources, and Issues in Mathematics Undergraduate Studies*, 8(1):1–14, 1998.
- Lawrence Sirovich and Michael Kirby. Low-dimensional procedure for the characterization of human faces. *Josa A*, 4(3):519–524, 1987.
- Gustaf Söderlind, Laurent Jay, and Manuel Calvo. Stiffness 1952–2012: Sixty years in search of a definition. *BIT Numerical Mathematics*, 55(2):531–558, 2015.
- Pavel Šolín. *Partial Differential Equations and the Finite Element Method*. Pure and Applied Mathematics. Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, 2006.
- James Somers. The scientific paper is obsolete. *The Atlantic*, 4, 2018.
- Peter L. Søndergaard, Bruno Torrésani, and Peter Balazs. The linear time frequency analysis toolbox. *International Journal of Wavelets, Multiresolution Analysis and Information Processing*, 10(4), 2012.
- William Squire and George Trapp. Using complex variables to estimate derivatives of real functions. *SIAM review*, 40(1):110–112, 1998.
- George J. Stigler. The cost of subsistence. *Journal of Farm Economics*, 27(2):303–314, 1945.
- Gilbert Strang. Wavelets and dilation equations: A brief introduction. *SIAM Review*, 31(4):614–627, 1989.
- Gilbert Strang. *Linear Algebra and Its Applications*. Wellesley–Cambridge Press, fifth edition, 2016.
- Fabio Toscano. *The Secret Formula: How a Mathematical Duel Inflamed Renaissance Italy and Uncovered the Cubic Equation*. Princeton University Press, 2020.
- Alex Townsend. The race for high order Gauss–Legendre quadrature. *SIAM News*, 48:1–3, 2015.

- Lloyd N. Trefethen. The definition of numerical analysis. Technical report, Cornell University, 1992.
- Lloyd N. Trefethen. Finite difference and spectral methods for ordinary and partial differential equations. Unpublished text, 1996.
- Lloyd N. Trefethen. *Spectral Methods in MATLAB*, volume 10 of *Software, Environments, and Tools*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.
- Lloyd N. Trefethen. A hundred-dollar, hundred-digit challenge. *SIAM News*, 35(1):1, 2002.
- Lloyd N. Trefethen. Ten digit algorithms. In *21st Biennial Conference on Numerical Analysis*. University of Dundee, June 2005. A. R. Mitchell Lecture.
- Lloyd N. Trefethen. Computing numerically with functions instead of numbers. *Mathematics in Computer Science*, 1(1):9–19, 2007.
- Lloyd N. Trefethen. Is Gauss quadrature better than Clenshaw–Curtis? *SIAM Review*, 50(1):67–87, 2008.
- Lloyd N. Trefethen. Ten digit problems. In *An Invitation to Mathematics*, pages 119–136. Springer, 2011. Available online at <https://www.mathworks.com/moler/chapters.html>.
- Lloyd N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition*. SIAM, 2019.
- Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.
- Lloyd N. Trefethen and Kristine Embree. The (unfinished) PDE coffee table book. <https://people.maths.ox.ac.uk/trefethen/pdectb.html>, 2001.
- Lloyd N. Trefethen and J. A. C. Weideman. The exponentially convergent trapezoidal rule. *SIAM Review*, 56(3):385–458, 2014.
- Adam Usadi and Clint Dawson. Years of ADI methods: Celebrating the contributions of Jim Douglas, Don Peaceman and Henry Rachford. *SIAM News*, 39(2):2006, 2006.
- Charles Van Loan. *Computational Frameworks for the fast Fourier Transform*, volume 10. SIAM, 1992.
- Loup Verlet. Computer “experiments” on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules. *Physical Review*, 159(1):98, 1967.

- Sebastiano Vigna. Spectral ranking. *Network Science*, 4(4):433–445, 2016.
- Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, 2020.
- John Von Neumann and Herman H. Goldstine. Numerical inverting of matrices of high order. *Bulletin of the American Mathematical Society*, 53(11):1021–1099, 1947.
- David S. Watkins. Francis’s algorithm. *The American Mathematical Monthly*, 118(5):387–403, 2011.
- David S. Watkins. *Fundamentals of Matrix Computations*. John Wiley & Sons, third edition, 2014.
- X. Wu, G. J. Zwieteren, and K. G. Zee. Stabilized second-order convex splitting schemes for Cahn–Hilliard models with application to diffuse-interface tumor-growth models. *International Journal for Numerical Methods in Biomedical Engineering*, 30(2):180–203, 2014.
- Haruo Yoshida. Construction of higher order symplectic integrators. *Physics Letters A*, 150(5):262–268, 1990.
- Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning, 2021. <https://d2l.ai>.



# Index

## A

A( $\alpha$ )-stable, 341  
A-conjugate, 135  
A-energy inner product, 13  
A-stable, 335  
Abel, Niels, 204  
Abel–Ruffini theorem, 89, 103, 189, 204  
absolutely stable, 333  
acceptance-rejection sampling, 319  
activation function, 272  
Adams–Bashforth method, 345, 507  
Adams–Moulton method, 345, 507  
adaptive methods, 303, 314, 315, 358, 359, 561, 592, 611  
adjacency matrix, 46  
adjoint, 6, 299  
affine transformation, 24, 272, 312  
Agnesi, Maria, 231  
Airy equation, 258, 372, 584  
Aitken extrapolation, 197  
algorithms  
    Arnoldi method, 112  
    Babylonian method, 191  
    bisection method, 189  
    Bluestein factorization, 153–155  
    Broyden’s method, 211

Cholesky decomposition, 37  
conjugate gradient method, 136  
Cooley–Tukey radix-2 FFT, 148  
Cuthill–McKee algorithm, 51–52, 464, 544, 565, 593, 613  
de Boor’s algorithm, 236  
de Casteljau’s algorithm, 240  
Dekker–Brent method, 193  
fast inverse square root, 179  
fast Toeplitz multiplication, 152  
full orthogonalization method, 139  
Gauss–Seidel method, 121  
Gaussian elimination, 33  
general minimized residual, 140  
Golub–Kahan–Reinisch algorithm, 115  
Golub–Welsch algorithm, 311, 627  
gradient descent, 132, 216  
Gram–Schmidt method, 62, 250  
implicit QR method, 108  
multigrid method, 128  
Newton’s method, 191, 209, 485–487, 575, 580, 581, 584, 623, 626, 627  
Newton–Horner method, 206

- non-negative matrix factorization, 84
  - Rader factorization, 155–157
  - randomized SVD algorithm, 116
  - Rayleigh quotient iteration, 100
  - revised simplex method, 54
  - Richardson extrapolation, 294
  - simplex method, 42
  - successive over-relaxation, 126
  - Allen–Cahn equation, 394, 448, 518
  - almost L-stable, 360
  - ansatz, 403
  - Arnoldi method, 110–114
    - implicitly restarted, 113
  - Arnoldi process, 138
  - artificial intelligence, 272
  - asymptotic convergence, 196
    - rate, 124
  - autodiff, *see* automatic differentiation
  - automatic differentiation, 287, 295–299
  - automatic programming, 537
- B**
- B-splines, 236–237, 262, 274, 447, 493, 552
  - Bézier curves, 238–242
  - Bézier, Pierre, 241
  - Babbage, Charles, xx, 225, 305
  - Babylonian method, 191
  - Backus, John, 537
  - backward differentiation formula, 340
  - backward Euler method, 329–337, 347, 360, 365, 377, 382, 399, 435, 503, 513
  - Bacon number, 56, 566, 614
  - Bacon, Kevin, 56
  - Banach fixed-point theorem, 93, 195, 356
  - Banach space, 249
  - banded matrix, 8, 50
  - basic feasible solution, 41
  - basic variables, 41
  - basin of attraction, 6, 210
  - BDF, *see* backward differentiation formula
  - Bernoulli number, xx, 305
  - Bernoulli trial, 241, 281
  - Bernoulli’s theorem, 241
  - Bernstein matrix, 240, 552, 602
  - Bernstein polynomial, 240, 241, 495
  - Bernstein, Serge, 241
  - Bessel’s equation, 243, 493
  - best approximation, 245
  - bifurcation diagram, 199
  - big O notation, 169
  - bilinear form, 429
  - biomimicry, 272
  - bisection method, 189
  - Bitcoin, 220
  - Bluestein factorization, 153–155
  - Bogacki–Shampine method, 352, 353, 367
  - Bonnet’s formula, 322
  - Boole’s rule, 301
  - Born, Max, xxiv
  - boundary conditions, 377–380
    - open, 393, 416
  - boundary value problem
    - Chebyshev polynomials, 258
    - collocation, 243
    - finite element method, 425
    - shooting method, 372, 510
  - Box, George, 320, 472
  - Box–Muller transform, 320
  - breadth-first search, 51
  - Broyden’s method, 211
  - Broyden–Fletcher–Goldfarb–Shanno algorithm, 216
  - Buffon’s needle, 315
  - Burgers’ equation, 407–408
  - Burgers, Jan, 407

- Butcher tableau, 349  
 butterfly matrix, 147
- C**  
 C programming language, 539  
 Céa's lemma, 434  
*Caenorhabditis elegans*, 55  
 Cahn–Hilliard equation, 458, 531  
 Cantor set, 201  
 Cardano's formula, 204  
 Cardano, Girolamo, 204  
 cardioid, 199, 511  
 catastrophic cancellation, 181  
 Cauchy differentiation formula, 322  
 Cauchy problem, 408  
 Cauchy sequence, 249  
 Cauchy–Schwarz inequality, 14, 433, 436  
 centrality, 95  
 CFL condition, 382, 397  
 CFL number, 397  
 chaotic map, 199  
 characteristic curves, 396  
 characteristic decomposition, 406  
 characteristic functions, 236  
 characteristic polynomial, 89, 207, 370, 462  
 characteristic speed, 396, 405  
 Chebyshev approximation, 245  
 Chebyshev differentiation matrix, 256  
 Chebyshev equioscillation theorem, 230  
 Chebyshev node, 230, 462  
 Chebyshev polynomial, 229, 255–259, 307, 462  
 Chebyshev spectral method, 255  
 Cholesky decomposition, 35–37  
 circulant matrix, 11, 149–153  
 ciardauq formula, 172  
 Clawpack, 563  
 Clenshaw–Curtis quadrature, 305–307, 315, 559, 608  
 coercive, 433  
 colleague matrix, 259  
 collocation, 242, 352, 492  
 column space, 6, 24, 41, 58, 59, 71, 75, 80, 110, 116  
 companion matrix, 207, 259  
 compatible norm, 16  
 complete metric space, 249  
 complete pivoting, 35  
 complex-step derivative, 187, 279  
 composite Simpson's rule, 302  
 composite trapezoidal rule, 302  
 compressed sparse column, 50  
 comrade matrix, 259  
 condition number, 22, 171–173, 175  
 eigenvalue, 90  
 conditioning, *see* condition number,  
 ill-conditioned,  
 well-conditioned  
 confusion matrix, 471, 569, 617  
 congruent, 19, 36  
 conjugate gradient method, 134–137  
 conjugate transpose, 6  
 connectivity, 46  
 conservation law, 406  
 conservative, 418  
 consistent, 175, 332, 338  
 contact discontinuity, 418  
 continuous, 433  
 contraction mapping, 93, 123, 195, 356  
 convergence, 175, 338  
 global, 190  
 local, 190  
 order of, 190  
 spectral, *see* spectral accuracy  
 convergence factor, 124  
 convolution, 12, 151, 152, 287, 455, 547, 597  
 convolutional neural network, 289, 496, 579

- Cooley–Tukey radix-2 FFT, 148  
 coset, 145  
 cosine similarity, 80  
 Courant–Fischer theorem, 99  
 Courant–Friedrichs–Lowy, 382  
 covector, 20  
 Crank–Nicolson method, 435, 515, 517, 518  
 cubature, 300  
 cubic splines, 231–235  
 cumulative length spline, 235  
 curse of dimensionality, 284, 316  
 Cuthill–McKee algorithm, 51–52, 464, 544, 565, 593, 613  
 Cybenko, George, 274
- D**
- da Vinci, Leonardo, 272  
 Dahlquist barrier, 344  
 dam break problem, 424  
 Dantzig, George, 37  
 de Boor’s algorithm, 236  
 de Casteljau’s algorithm, 240  
 decision variables, 38  
 decomposition  
     incomplete LU, 122  
 deep learning, 179, 286, 556  
 defective, 13  
 defective matrix, 7  
 deflation  
     of a matrix, 106  
     of a polynomial, 205  
 degree  
     of a polynomial, 204, 221, 228, 307  
     of a vertex, 46  
     of freedom, 108, 231, 276  
     of separation, 55  
 degree matrix, 47  
 Dekker–Brent method, 193  
 Deming regression, 76  
 determinant, 24, 55, 464  
 de Broglie, Louis, 161
- de Casteljau, Paul, 241  
 DFT, *see* discrete Fourier transform  
 diagonally dominant, 123  
 diet problem, *see* Stigler diet problem  
 Diffie–Hellman key exchange, 202  
 Dijkstra’s algorithm, 51  
 Dirac notation, 20  
 direct problem, 171  
 Dirichlet boundary condition, 377  
 discrete Fourier transform, 144  
 discrete wavelet transform, 268–272  
 dispersion relation, 403  
 divided differences, 225  
 dolphins of Doubtful Sound, 48  
 domain of dependence/influence, 398  
 dominant eigenvalue, 93  
 Dormand–Prince method, 352, 358, 367, 389  
 double-and-add, 204, 488  
 dual number, 295  
 dual problem, 44  
 dual space, 20  
 Duffing equation, 371, 509, 583, 629  
 Dufort–Frankel method, 383–384, 512  
 DWT, *see* discrete wavelet transform
- E**
- Eaton, John, 538  
 Eckart–Young–Mirsky theorem, 74  
 eigenface, 79  
 eigenspace, 7  
 eigenvalue, 6, 25  
     condition number, 90  
     estimation, 91  
     methods for computing, 93–114  
 eigenvector, 6, 25  
     left eigenvector, 90

- eigenvector centrality, 96  
 elementary row operations, 30  
 elliptic curve, 203  
 energy method, 390  
 energy norm, 14  
 ENIAC, xiv, 316, 535  
 entropy, 412  
 epoch, 285  
 equivalent matrices, 8, 12  
 equivalent norms, 15, 391  
 Erdős, Paul, 56  
 Euclidean inner product, 13  
 Euclidean norm, 14, 25  
 Euler equations, 424  
 Euler method, *see* backward Euler method, forward Euler method  
 Euler, Leonhard, 45  
 Euler–Maclaurin formula, 305  
 explicit Euler method, *see* forward Euler method
- F**  
 fast Fourier transform, 143–162, 255, 444  
 fast inverse square root, 179, 180, 187  
 fast Toeplitz multiplication, 152  
 father wavelet, *see* scaling function  
 Favard’s theorem, 250, 309  
 Feigenbaum constant, 199  
 FEniCS, 436–439, 563  
 FFT, *see* fast Fourier transform  
 FFTW, 158, 160, 445, 548, 598  
 Fickian diffusion, 374  
 Filippelli problem, 85  
 finite volume method, 420  
 fixed point, 93, 198, 486  
 fixed-point method, 194–197, 356  
 floating-point number, 177–179  
 flux, 406  
 Fontana, Niccolò, 204  
 force-directed graph drawing, 48  
 Fortran, 182, 208, 235, 315, 368, 537–538, 552, 563  
 forward Euler method, 174, 329–337, 347, 365, 382, 397, 398, 401, 503, 513  
 Fourier polynomial, 252–254, 442  
 Fourier series, 252  
 Fourier transform, 252  
 Fourier, Joseph, xix, 161  
 Fox, Laura, 68, 79, 162, 270, 271  
 foxed-point number, 182  
 Francis, John, 101  
*Frejér quadrature, see*  
 Clenshaw–Curtis quadrature  
 Frobenius norm, 17, 27, 74, 83, 161, 211, 463, 542  
 Fry, Hannah, 286  
 full orthogonalization method, 139  
 fundamental subspaces, 6, 71  
 fundamental theorem of  
     Gaussian quadrature, 308  
     linear algebra, 6  
     linear programming, 40  
     numerical analysis, 176, 337
- G**  
 Galerkin formulation, 427  
 Galois field, 3  
 Galois, Évariste, 204  
 Gauss, Carl Friedrich, xix, 31, 146, 308  
 Gauss–Chebyshev, 308  
*Gauss–Christoffel quadrature, see*  
     Gaussian quadrature  
 Gauss–Hermite, 308, 311, 323, 627  
 Gauss–Jacobi, 308  
 Gauss–Kronrod, 313–315  
 Gauss–Laguerre, 308, 312  
 Gauss–Legendre, 308, 311, 312, 314, 315, 322, 502, 581  
 Gauss–Lobatto, 314, 315  
 Gauss–Newton method, 278  
 Gauss–Radau, 314

Gauss–Seidel method, 121  
 Gaussian distribution, 58, 161, 273, 278, 445  
 Gaussian elimination, 30–35, 119  
 Gaussian quadrature, *see also*  
     Gauss–Chebyshev,  
     Gauss–Hermite, Gauss–Jacobi,  
     Gauss–Kronrod,  
     Gauss–Laguerre, Gauss–Legendre, Gauss–Lobatto,  
     Gauss–Radau, 307–315  
 Gaussian–Legendre, 627  
 Gear’s method, *see* backward differentiation formula  
 general minimized residual, 140  
 generalized linear model, 281  
 generator  
     pseudo-random number, 321  
     subgroup, 144  
     subspace, 4  
 Gershgorin circle theorem, 91  
 ghost point, 376  
 Gibbs phenomenon, 161, 448  
 Givens rotation, 116  
 global truncation error, 342  
 golden ratio, 215  
 golden section search method, 215  
 Goldstine, Herman, 31  
 Golub–Kahan–Reinisch algorithm, 115  
 Golub–Welsch algorithm, 311, 627  
 Google matrix, 96  
 gradient descent, 130–134, 137, 216–218, 284, 285, 498, 556  
 Gram matrix, 70, 247  
 Gram–Schmidt method, 61–62, 116, 250  
 graph Laplacian, 47  
 graphs, 45–49  
 Green’s first identity, 429  
 Green’s theorem, 410  
 group velocity, 404

**H**

Haar, Alfréd, 260  
 Hadamard product, 5, 70, 151  
 Hadamard, Jacques, 21, 171  
 Halley’s method, 219  
 Halmos, Paul, xiv, xxiii  
 Hamiltonian, 20, 365  
 heat equation, 374–375  
     backward Euler solution, 377  
     Crank–Nicolson solution, 379  
     Dufort–Frankel solution, 512  
     Fourier solution, 444  
     fundamental solution, 391  
     polar coordinates, 393  
 heavy ball method, 217  
 Hein, Piet, 6  
 Hermite interpolation, 226  
 Hermitian matrix, 8  
 Hessian matrix, 216  
 Heun’s method, 348  
 Hilbert approximation, 245  
 Hilbert basis, 249  
 Hilbert matrix, 23, 72, 248  
 Hilbert space, 249  
 homotopy, 212  
 Hopf–Cole transformation, 408  
 Horner form, 205, 488  
 Horner’s method, 205  
 Householder matrix, 64, 104  
 Householder method, 219  
 Householder reflection, 64, 116  
 hundred-dollar problem, 118, 142, 170  
 hyperbolic equation, 395–402  
 hyperbolic system, 404–407, 414–419  
 hyperoperation, 168  
 hyperspectral images, 83

**I**

identification problem, 171  
 IDFT, *see* discrete Fourier transform

- IEEE 754 standard, 178–179
- ill-conditioned, 22, 172
- ill-posed, 21, 171
- image compression
  - discrete cosine transform, 161
  - discrete wavelet transform, 269
  - singular value decomposition, 78
- implicit Euler method, *see*
  - backward Euler method
- implicit Q theorem, 108
- implicit QR method, 108
- incomplete LU factorization, 122
- induced norm, 16, 22, 25, 123, 250
- infinitesimal number, 296
- infinity norm, 14, 17, 58, 92, 187, 229, 245, 288, 391
- inner product, 13, 246
- inner product space, 246
- integral curve, 330
- integration, *see* Clenshaw–Curtis
  - quadrature, Gaussian
  - quadrature, Monte Carlo
  - integration, Newton–Cotes
  - quadrature
- intrinsic dimension, 276
- invariant, 7
- inverse problem, 171
- iteratively reweighted least squares, 282
- J**
  - Jacobi method, 121
    - weighted, 125
  - Jacobi operator, 310
  - Jenkins–Traub method, 208
  - JPEG 2000, 78, 269
  - Julia set, 201, 210
  - Jupyter, 536, 541, 590
- K**
  - Königsberg bridge puzzle, 46
  - Kahan, William, 179
  - Kantorovich, Leonid, 37
  - Karinty, Frigyes, 55
  - Karpinski, Stefan, 540
  - Karush–Kuhn–Tucker conditions, 70, 83
  - Katz centrality, 96
  - Kelvin–Helmholtz instability, 455
  - kernel, 6
  - Kirchhoff matrix, 47
  - Kirchhoff, Gustav, 46
  - KKT conditions, *see*
    - Karush–Kuhn–Tucker conditions
  - knots, 231
  - Knuth, Donald, xiv, 168, 536
  - Kortweg–deVries equation, 457, 529
  - Kronecker product, 148
  - Kronrod, Aleksandr, 313
  - Krylov matrix, 111, 150
  - Krylov methods, 137–140
  - Krylov subspace, 25, 111, 138
  - Kublanovskaya, Vera, 101
  - Kuramoto–Sivashinsky equation, 451, 458, 532
  - L**
    - L-stable, 360
    - Lagrange multiplier, 70, 211
    - Lambert W function, 357
    - Lanczos method, 114, 138
    - Landau notation, 169
    - Landau, Edmund, 95
    - LAPACK, 33, 537, 544, 594, 596
    - Laplace equation, 425
    - Laplace expansion, 207
    - Laplacian matrix, 47
    - Laplacian operator, 51, 120, 124, 159, 394, 585
    - Larry, Shampine, 357
    - latent semantic analysis, 80
    - law of large numbers, 241, 317
    - law of the unconscious statistician, 317

- Lax entropy condition, 412  
 Lax equivalence theorem, 176, 338  
 Lax, Peter, 3  
 Lax–Friedrichs method, 399–405, 419, 519  
 Lax–Milgram lemma, 433  
 Lax–Wendroff method, 400–405, 419, 455  
 Lax–Wendroff theorem, 419  
 leapfrog method, 329–337, 383, 513  
 learning rate, 216  
 least squares  
     constrained, 69  
     multiobjective, 69  
     regularized, 67  
     sparse, 70  
 LeCun, Yann, 289  
 left eigenvector, 90, 93, 415  
 left null space, 6, 71, 72  
 Legendre polynomial, 4, 248, 251, 314, 354  
 Leland, Wilkinson, 539  
 LeNet, 289, 496  
 Levenberg–Marquardt method, 216, 279  
 LeVeque, Randall, 563  
 likelihood function, 281  
 Lindemann–Weierstrass theorem, 299  
 linear functional, 428  
 linear independence, 4  
 linear map, 5  
 linear programming, 37–45  
     equational form, 40  
     standard form, 39  
 linear regression, 58, 66, 277, 469, 568, 616  
 link function, 281  
 Lipschitz continuous, 171, 328  
 literate programming, 536  
 little O notation, 169  
 load vector, 428  
 local Lax–Friedrichs method, 419, 524  
 local truncation error, 342  
 logistic equation, 173  
 logistic map, 174, 198  
 logit function, 517  
 Lorenz equation, 371  
 Lovelace, Ada, xx, 225, 305  
 LU decomposition, 31–35
- M**
- Müller’s method, 193  
 machine epsilon, 179, 183, 210, 548, 595, 599  
 magic number, 181, 187  
 Mandelbrot set, 201, 549, 600  
 Mandelbrot, Benoit, xix  
 manifold hypothesis, 276  
 Markov chain, 93  
 Markov matrix, 93  
 Mathematica, *see* Wolfram Language  
 Matplotlib, 539  
 matrices  
     adjacency, 46  
     banded, 8, 50  
     Bernstein, 240, 552, 602  
     Chebyshev differentiation, 256  
     circulant, 11, 149–153  
     colleague, 259  
     companion, 207, 259  
     comrade, 259  
     defective, 7  
     degree, 47  
     Gram, 70, 247  
     Hermitian, 8  
     Hilbert, 23, 72, 248  
     Householder, 64, 104  
     Krylov, 111, 150  
     Laplacian, 47  
     normal, 8  
     orthogonal, 8, 24  
     permutation, 8, 24

- positive definite, 8, 19, 35
  - primitive, 93
  - projection, 8, 24, 454
  - semisimple, 7
  - sparse, 45–52
  - Toeplitz, 149–153
  - tridiagonal, 8, 29, 121, 159, 237, 310, 377, 379, 388, 399
  - unitary, 8
  - upper Hessenberg, 8, 104, 108, 113, 139, 547, 596
  - upper triangular, 8, 24
  - Vandermonde, 85, 86, 222, 224, 242, 292
  - zero-one, 26, 46
  - matrix decomposition, *see*
    - Cholesky, LU, non-negative matrix, QR, Schur, singular value, spectral decompositions
  - matrix norm, 15
  - maximum likelihood, 58
  - maximum principle, 375
  - May, Robert, 198
  - McColluch, Warren, 272
  - mean, 287
  - mean value theorem, 227
  - median, 287
  - Mersenne twister, 321
  - Merson method, 370
  - method of characteristics, 396
  - method of cofactors, 26
  - method of lines, 375
  - Metropolis algorithm, 319
  - Metropolis, Nicholas, 316
  - midrange, 287
  - Millennium Prize, 375
  - million-dollar problem, 375
  - min-max theorem, 99
  - mindblowingly awesome, *see* Fox, Laura
  - minimization problem, 425
  - mixed boundary condition, 380
  - MNIST, 86, 290, 470, 497, 568
  - mode, 287
  - Moler, Cleve, 27, 179, 357, 537
  - Monte Carlo integration, 315–321, 323
  - Monty Python, 539
  - Moore–Penrose pseudoinverse, *see* pseudoinverse
  - mother wavelet, *see* wavelet function
  - motion by mean curvature, 519
  - multigrid method, 128
  - multilateration, 87
  - multiresolution, 261
- N**
- Nakamoto, Satoshi, 220
  - NaN, *see* not a number
  - Navier–Stokes equation, 375, 407, 452–456
  - Nelder–Mead method, 215
  - neural network, 272–277, 283–287, 496–498
  - Newton’s method, 191, 209, 216, 485–487, 575, 580, 581, 584, 623, 626, 627
  - Newton, Isaac, 219
  - Newton–Cotes formula, 300
  - Newton–Cotes quadrature, 299–307
  - Newton–Horner method, 206
  - Newton–Raphson method, *see* Newton’s method
  - node, 46
  - non-negative matrix factorization, 83–84
  - nonbasic variables, 41
  - nonstandard analysis, 295
  - norm
    - seealso* energy norm, Euclidean norm, Frobenius norm, induced norm, infinity norm, matrix norm, p-norm, spectral norm, vector norm, 13

normal equation, 58–60, 247, 277  
 normal matrix, 8  
 not a number, 182, 549, 599  
 null space, 6, 24, 67, 70–72, 92  
 nullity, 6  
 numerical diffusion/dispersion,  
     402–404  
 numerical method of lines, 327  
 numerical viscosity, 402  
 numerically stable, 175

**O**

objective function, 38  
 Octave, 538  
 Octave, Levenspiel, 538  
 Oliphant, Travis, 539  
 operator splitting, *see* splitting  
     methods  
 optimal vector, 131  
 orbit, 196  
 orthogonal, 13, 246  
 orthogonal complement, 246  
 orthogonal matrix, 8, 24  
 orthogonal polynomial, 249–251,  
     307–309, 313  
 orthogonal regression, 76  
 orthonormal, 249  
 overflow, 182  
     Arianne 5 rocket, 183

**P**

p-norm, 14, 22, 542  
 Pólya, George, 199  
 Padé approximation, 346  
 PageRank algorithm, 96  
 Parseval’s theorem, 249, 447  
 partial pivoting, 35  
 partition of unity, 236, 265, 427  
 path, 46  
 pathological function, 167  
 PCA, *see* principal component  
     analysis  
 permutation matrix, 8, 24

Perron–Frobenius theorem, 93, 97  
 phase velocity, 404  
 Pitts, Walter, 272  
 pivoting, 34–35  
 Poincaré inequality, 433  
 Poisson equation, 120, 126, 141,  
     159, 428, 430, 434, 437  
 Poisson summation formula, 443  
 polynomial, *see* Bernstein,  
     characteristic, Chebyshev,  
     Fourier, Lagrange, Legendre,  
     orthogonal, Stieltjes, Taylor  
     polynomials  
 polynomial interpolation error  
     theorem, 228  
 positive definite, 130, 458  
 positive definite matrix, 8, 19, 35  
 positive matrix, 93  
 power method, 93–101  
 preconditioner, 122  
 predictor-corrector method, 345  
 prestige, 95  
 primal problem, 44  
 primitive matrix, 93  
 principal component analysis,  
     73–75, 86, 471  
 probability, 20, 93, 97, 241, 315,  
     317  
 probability vector, 97  
 projection matrix, 8, 24, 454  
 pseudo-inner product, 246  
 pseudoinverse, 57, 68, 72–73, 85,  
     278, 279, 545, 595  
 pseudospectral, 444  
 Python, 539

**Q**

Q\_rsqrt algorithm, *see* fast inverse  
     square root  
 QR code, xxiv, 78, 81, 82, 100, 105,  
     113, 130, 162, 168, 170, 199,  
     240, 271, 285, 365, 374, 379,  
     390, 404, 405, 407, 452, 456,

- 496, 512, 516, 519, 524, 525,  
529–533, 541, 590
- QR decomposition, 60–64
- QR method, 101–109  
  implicit QR, 107–109  
  shifted QR, 105
- quadratic approximation, 193
- quadratic form, 18
- quadratic map, 198
- quadrature, 299, *see also*  
  Clenshaw–Curtis quadrature,  
  Gaussian quadrature, Monte  
  Carlo integration,  
  Newton–Cotes quadrature
- R**
- R programming language, 539
- Rader factorization, 155–157
- radial basis function, 242, 273
- randomized SVD, 476
- randomized SVD algorithm, 116
- rank of a matrix, 6
- Rankine–Hugoniot condition, 410
- Rayleigh quotient, 6, 99
- Rayleigh quotient iteration, 98–101
- Rayleigh quotient shifting, 105
- Rayleigh–Bénard convection, 6,  
  457
- Rayleigh–Ritz formulation, 427
- Rayleigh–Ritz method, 110
- RBF, *see* radial basis function
- recursion, 229, 265, 276, 294, 322
- reduced row echelon form, 30
- region of absolute stability, 334,  
  335
- regression  
  Deming, 76  
  linear, 58, 277, 469, 568, 616  
  orthogonal, 76  
  total least squares, 76
- regula falsi, 218
- Reiss combustion model, 357
- relaxation, 126
- ReLU, 273
- residual, 58
- reverse Cuthill–McKee algorithm,  
  *see* Cuthill–McKee algorithm
- revised simplex method, 54
- Richardson extrapolation, 294, 303,  
  322
- ridge regression, 67
- Riemann invariant, 396, 415
- Riemann problem, 410
- Riemann, Bernhard, 167
- Ritchie, Dennis, 539
- Ritz estimate, 113
- Ritz pair, 110
- Rolle’s theorem, 228
- Romberg’s method, 303, 315
- root of unity, 144, 253
- root-finding methods  
  bisection method, 189  
  Broyden’s method, 211  
  Companion matrix method, 207  
  fixed-point method, 194  
  Halley’s method, 219  
  Homotopy continuation, 212  
  Horner’s method, 205  
  Householder method, 219  
  Newton’s method, 190  
  secant method, 192  
  Steffensen’s method, 219
- Rosenbrock function, 217
- Rosenbrock method, 355
- rough number, 153
- round-off error, 183–185  
  Patriot missile, 182  
  Vancouver Stock Exchange, 184
- row space, 6, 71
- Ruffini’s rule, 206
- Rump, Siegfried, 181
- Runge’s phenomenon, 229, 307
- Runge, Carl, 230
- Runge–Kutta method, 347–356
- Runge–Kutta methods

- Bogacki–Shampine, 352, 353, 367
- Dormand–Prince, 367, 528
- ETDRK, 450, 531, 532
- Heun, 348
- Radau, 355
- RK4, 350
- Rosenbrock, 355
- trapezoidal, 349
- S**
- S programming language, 539
- scaling function, 261
- Schönhage–Strassen algorithm, 164
- Schrödinger equation, 20, 392, 449, 456, 513
- Schur decomposition, 9–10, 102, 109
- Schur decomposition theorem, 9
- secant method, 193, 211
- self-adjoint, 8
- self-similar solution, 411
- self-similarity, 168, 201, 261, 268, 276
- semi-norm, 246
- semisimple matrix, 7
- Sherlock Holmes, 66
- Sherman–Morrison formula, 54, 212
- shift-and-invert, 98
- shock, 409
- speed, 410
- shooting method, 372, 510
- sigmoid function, 273, 274, 380, 518
- similar matrices, 7
- similarity transform, 7
- simplex, 215
- simplex method, 40–45
- revised, 53–54
- Simpson’s rule, 301, 302, 315, 349
- sinc, 446
- singular value, 12, 17, 25, 71, 72
- singular value decomposition, 12–13, 23, 70–82, 115–116
- singular vectors, 7
- smooth number, 153
- Sobolev space, 254, 429
- soliton, 457
- Sonneborn–Berger system, 95
- sparse matrix, 45–52, 119, 431
- sparse SVD, 116, 472, 476
- spectral accuracy, 305, 441, 528
- spectral decomposition, 10–12, 90, 99
- spectral layout drawing, 48
- spectral norm, 16, 74
- spectral radius, 7
- spectral theorem, 10
- spectrum, 7
- splines, 231–238
- cardinal B-spline, 237
- computation, 233
- cubic Hermite spline, 237
- cumulative length, 235
- parametric, 235
- splitting methods, 360–364
- ADI, 386
- fractional step, 385
- IMEX, 363, 450
- integrating factors, 364, 449
- Strang splitting, 362
- stable, 21, 328, 333, 338
- Stan programming language, 317
- Steffensen’s method, 219
- stencil, 330
- Stieltjes polynomial, 313
- stiff equation, 357–360, 388, 448–452
- stiffness matrix, 428
- stiffness ratio, 359
- Stigler diet problem, 38, 55, 466, 566, 614
- Stigler, George, 38
- stochastic matrix, 93
- Strang splitting, 362, 518

- Strang, Gilbert, 3, 6, 143  
 strictly hyperbolic, 404  
 strong duality theorem, 45  
 strong solution, 409  
 submultiplicative, 15  
 subspace, 4  
 successive over-relaxation, 126  
 SVD, *see* singular value decomposition  
 Swift–Hohenberg equation, 6, 457  
 swish function, 273  
 symmetric, 433  
 symplectic Euler method, 366  
 symplectic methods, 364–366
- T**  
 Taylor series expansion, 516  
 Taylor’s theorem, 435  
 ten-digit algorithm, xv  
 ten-digit problem, 323  
 test function, 409, 427  
 tetration, 168  
 theorems  
   Abel–Ruffini theorem, 89, 103,  
 189, 204  
   Banach fixed-point theorem, 93,  
 195, 356  
   Bernoulli’s theorem, 241  
   Céa’s lemma, 434  
   Chebyshev equioscillation theorem, 230  
   Courant–Fischer theorem, 99  
   Eckart–Young–Mirsky theorem, 74  
   Favard’s theorem, 250, 309  
   fundamental theorem of Gaussian quadrature, 308  
   linear programming, 40  
   numerical analysis, 176  
   Gershgorin circle theorem, 91  
   Golub–Welsch algorithm, 311  
   Green’s first identity, 429  
   Green’s theorem, 410  
   implicit Q theorem, 108  
   Lax equivalence theorem, 176,  
 338  
   Lax–Wendroff theorem, 419  
   Lindemann–Weierstrass theorem, 299  
   mean value theorem, 227  
   min–max theorem, 99  
   Parseval’s theorem, 249, 447  
   Perron–Frobenius theorem, 93,  
 97  
   Poincaré inequality, 433  
   polynomial interpolation error theorem, 228  
   Rolle’s theorem, 228  
   Schur decomposition theorem, 9  
   spectral theorem, 10  
   strong duality theorem, 45  
   Taylor’s theorem, 435  
   universal approximation theorem, 273  
   Weierstrass approximation theorem, 241  
   theta method, 369, 503  
   Thomae’s function, 168  
   three-term recurrence relation, 250,  
 289, 309, 462  
   Tikhonov regularization, 67, 279  
   time to first plot, 540  
   time-marching, 376  
   Toeplitz matrix, 149–153  
   total least squares regression, 76  
   total potential energy, 426  
   traffic equation, 412  
   transpose, 5  
   trapezoidal method, 329–337, 359,  
 503, 513  
   trapezoidal rule, 300  
   Trefethen, Nick, xv, 118, 142, 170,  
 256, 259, 323  
   trial function, 427  
   tridiagonal matrix, 8, 29, 121, 159,  
 237, 310, 377, 379, 388, 399

truncation error, 185, 338, 402  
Turing machine, 272

**U**  
Ulam, Stanislaw, 316  
ultimate shift, 109  
UMFPACK, 45, 543, 592  
unconditionally stable, 382  
underflow, 182  
unit roundoff, *see* machine epsilon  
unitarily similar, 7  
unitary matrix, 8  
universal approximation theorem, 273  
upper Hessenberg matrix, 8, 104, 108, 113, 139, 547, 596  
upper triangular matrix, 8, 24  
upwind method, 396–398, 402, 405

**V**  
V-elliptic, 433  
Vandermonde matrix, 85, 86, 143, 222, 224, 242, 292, 468, 490, 499, 546, 596  
van Gogh, Vincent, 455  
van Loan, Charles, 27  
van Rossum, Guido, 539  
variational problem, 425  
vector norm, 14, 25  
vector space, 3  
Verlet method, 366  
versine function, 231  
vertex, 46  
von Neumann analysis, 380–385, 397, 400  
von Neumann, John, 31, 37, 319, 321

**W**  
w-orthogonal, 246  
wavelet function, 267  
wavelets  
B-spline, 262  
Cohen–Daubechies–Feauveau, 269  
Coiflet, 272  
Daubechies, 265  
Haar, 260, 270  
LeGall–Tabatabai, 269  
weak solution, 409  
Weierstrass approximation theorem, 241  
Weierstrass function, 167  
Weierstrass, Karl, 230  
well-conditioned, 34, 122, 138, 172  
well-posed, 21–24, 173–177, 327  
Wickham, Hadley, 539  
Wiener weight, 73  
Wikipedia, xiv, 56, 472, 537  
Wilkinson shift, 106  
Wilkinson, James, 45  
wisdom, 158  
witch of Agnesi, 231  
Wolfram Language, 208, 346, 495, 536

**Z**  
zero-based indexing, 540  
zero-one matrix, 26, 46  
zero-stability, 337  
zeros of polynomials  
companion matrix method, 207  
Newton–Horner method, 206  
Zipf’s law, 66

# Julia Index

This index is an incomplete listing of Julia functions, macros, and packages referenced in this book.

## Functions

- append, 465
- argmax, 42
- argmin, 42
- bitstring, 179
- cg, 137
- Circulant, 150
- cond, 24
- conv, 152
- curve\_fit, 280
- dct, 159
- det, 462
- Diagonal, 476, 478
- diff, 491
- eigen, 109
- eigs, 114
- eigvals, 109
- eigvecs, 109
- eps, 179
- evalpoly, 205
- expm1, 185
- fft, 158, 496
- fftshift, 158
- findall, 465
- findfirst, 54
- floatmax, 183
- floatmin, 183
- fzero, 194
- gausslegendre, 312
- givens, 63, 474
- gmres, 140
- gplot, 49
- Gray, 476
- hcat, 482
- hessenberg, 105, 474
- hilbert, 24
- idct, 159
- ifft, 158, 496
- ifftshift, 158
- interp, 492
- irfft, 158
- isinf, 183
- isnan, 183
- kron, 148, 477
- log1p, 185
- LowerTriangular, 478
- Matrix, 378, 476, 491
- minres, 140

nlsolve, 194  
 norm, 17  
 ones, 477  
 opnorm, 17  
 optimize, 218  
 parse, 481  
 permutedims, 470  
 pinv, 73  
 popfirst, 465  
 qr, 64, 476  
 rand, 476  
 randn, 491  
 randstring, 481  
 Rational, 293  
 readdlm, 469  
 reshape, 470  
 rfft, 158  
 Roots, 494  
 roots, 89, 208  
 rref, 30  
 scaleminmax, 470  
 solve, 486  
 sortperm, 465  
 sparse, 54, 477, 478  
 spdiags, 477  
 SpecialFunctions, 494  
 SplitODEProblem, 451  
 spy, 45  
 spzeros, 54  
 svd, 72, 476  
 svds, 72, 470, 477  
 SymTridiagonal, 491  
 Toeplitz, 149  
 transpose, 6  
 Tridiagonal, 378  
 typeof, 178  
 Vandermonde, 86, 224  
 Vector, 54

## Macros

@animate, 389, 496  
 @., 5

@btime, 378  
 @constraint, 466  
 @elapsed, 30  
 @gif, 512  
 @less, 29  
 @manipulate, 389  
 @objective, 466  
 @time, 30, 378, 476  
 @variable, 466

## Packages

AbstractFFTs, 445  
 ApproxFun, 259  
 Arpack, 114, 470, 472, 476, 477  
 BenchmarkTools, 181, 378  
 COSMO, 45  
 CSV, 466, 470  
 DataFrames, 466, 470  
 Dates, 470  
 DelimitedFiles, 49, 465, 469  
 Dierckx, 235  
 DiffEqOperators, 369, 451  
 DifferentialEquations, 366, 451,  
 486  
 Distributions, 321  
 DSP, 152  
 FastGaussQuadrature, 312, 502  
 FastPolynomialRoots, 208  
 FEniCS, 437  
 Ferrite, 437  
 FFTW, 158, 445, 481, 482, 496  
 FileIO, 476, 482  
 FinEtools, 52  
 Flux, 286, 497  
 ForwardDiff, 299  
 GLPK, 45, 466  
 GraphPlot, 48, 49, 465  
 Gridap, 437  
 HomotopyContinuation, 215  
 Images, 470, 476, 482  
 Interact, 389, 496, 532  
 Interpolations, 237

- IterativeSolvers, 137, 140  
JuliaFEM, 52  
JuMP, 45, 466  
Kinetic, 423  
LightGraphs, 48, 49, 465  
LinearAlgebra, xxiv, 17, 474  
LsqFit, 280  
MLDatasets, 86, 470, 497  
NLsolve, 194  
NMF, 84  
Oceananigans, 423  
OffsetArrays, 265  
Optim, 218  
Plots, xxiv, 540  
Polynomials, 89, 208, 508  
PolynomialsRoots, 208  
ProgressMeter, 498  
Random, 481  
ReverseDiff, 299  
Roots, 194, 372  
RowEchelon, 30  
SciPyDiffEq, 368  
SparseArrays, 49, 54, 465, 477  
SpecialMatrices, 24, 86, 224  
Sundials, 368, 388  
ToeplitzMatrices, 149, 150  
Trixi, 423  
ViscousFlow, 423  
Wavelets, 269  
Zygote, 299



# Python Index

This index is an incomplete listing of Python functions and packages referenced in this book.

## Functions

|                    |                    |
|--------------------|--------------------|
| bspline, 552       | hilbert, 542       |
| Bspline, 552       | ifft, 548          |
| cg, 547            | ifftshift, 548     |
| circulant, 547     | isinf, 549         |
| comb, 552          | isnan, 549         |
| cond, 542          | kron, 547          |
| conj, 542          | leggauss, 559      |
| convolve, 547, 548 | linear_solver, 543 |
| CubicSpline, 552   | linprog, 543       |
| curve_fit, 554     | log1p, 549         |
| dct, 548           | lstsq, 544         |
| dot, 543, 562      | max, 549           |
| dst, 548           | min, 549           |
| eig, 547           | minimize, 551      |
| eigs, 547          | minres, 547        |
| eps, 549           | norm, 542          |
| expm1, 549         | outer, 542         |
| fft, 548           | pchip, 552         |
| fftfreq, 563       | pinv, 544, 545     |
| fftshift, 548      | polyval, 550       |
| finfo, 549         | pywraplp, 543      |
| fsolve, 549        | qr, 544            |
| hessenberg, 547    | quad_vec, 315      |
|                    | reshape, 541, 590  |

reverse\_cuthill\_mckee, 544  
romb, 315  
romberg, 315  
roots, 546, 550  
rref, 542  
simpson, 315  
solve, 542  
solve\_banded, 562  
solve\_circulant, 562  
solve\_ivp, 368, 550, 560, 561  
solveh\_banded, 551, 562  
splev, 552  
spline, 552  
splprep, 552  
spsolve, 543, 562  
spy, 543  
svd, 545  
timeit, 542  
toeplitz, 547  
type, 548  
vander, 546, 551  
vstack, 541, 590

**Packages**  
Autograd, 557  
ChebPy, 553  
FEniCS, 563  
FiPy, 563  
fractions, 557  
ipywidgets, 562  
JAX, 557  
Matplotlib, 541  
Mshr, 563  
NetworkX, 543  
networkx, 544  
NumPy, 541  
OR-Tools, 543  
Pandas, 544  
PyWavelets (pywt), 553  
scikits, 368  
SciPy, 541  
sys, 549  
TensorFlow, 556  
timeit, 561  
urllib, 545

# Matlab Index

This index is an incomplete listing of MATLAB/Octave functions and packages referenced in this book.

## Functions

|             |          |                                |          |
|-------------|----------|--------------------------------|----------|
| class,      | 599      | kron,                          | 597      |
| condeig,    | 596      | leasqr,                        | 605      |
| dct,        | 599      | linprog,                       | 592      |
| eig,        | 596      | log1p,                         | 599      |
| eigs,       | 596      | lsode,                         | 368      |
| eps,        | 599      | lsqcurvefit,                   | 605      |
| expm1,      | 599      | minres,                        | 597      |
| fft,        | 598      | mkpp,                          | 602      |
| fftshift,   | 598      | mldivide () ,                  | 592, 594 |
| fminbnd,    | 601      | mlrdivide () ,                 | 592      |
| fminsearch, | 601      | num2hex,                       | 599      |
| fminunc,    | 601      | ode113, ode15s, ode23, ode23i, |          |
| fzero,      | 600      | ode23s, ode23t, ode23tb,       |          |
| givens,     | 594      | ode45,                         | 367      |
| glpk,       | 592      | pcg,                           | 596      |
| gplot,      | 592      | pchip,                         | 602      |
| hess,       | 596      | pinv,                          | 594, 595 |
| hilb,       | 542, 591 | polyval,                       | 600      |
| ifft,       | 598      | ppval,                         | 602      |
| ifftshift,  | 598      | qr,                            | 594      |
| integral,   | 315      | quad, quadcc, quadl, quadgk,   |          |
| isinf,      | 599      | quadv,                         | 315      |
| isnan,      | 599      | rats,                          | 606      |
|             |          | realmax, realmin,              | 599      |

rgb2gray, 595  
roots, 596, 600  
rref, 591  
spline, 602  
spy, 592  
svd, 594  
svds, 594  
symrcm, 593  
toeplitz, 597

trapz, 315  
vander, 596, 601

### Packages

Chebfun, 603  
image, 595  
LTFAT, 604  
optim, 601, 605  
signal, 599



