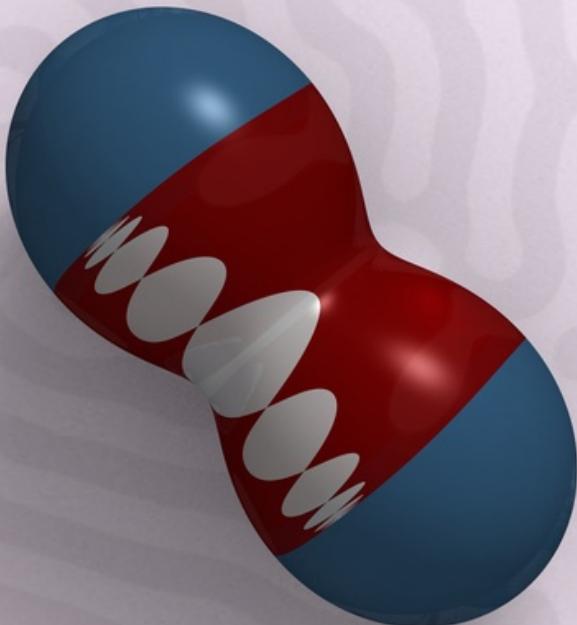


# NUMERICAL METHODS FOR SCIENTIFIC COMPUTING

*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 2-220623

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 99. The solution to the Swift–Hohenberg equation with random initial conditions, which models Rayleigh–Bénard convection, is discussed on page 463.

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](mailto: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](http://www.equalsharepress.com)



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

[Buy this book on Amazon](#)

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	Overspecified systems	57
3.2	Underspecified systems	67
3.3	Singular value decomposition	71
3.4	Nonnegative matrix factorization	83
3.5	Exercises	86

## 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	146
6.3	Toeplitz and circulant matrices	150
6.4	Bluestein and Rader factorization	153
6.5	Applications	158
6.6	Exercises	163

# Numerical Analysis

## 7 Preliminaries 169

7.1	Well-behaved functions	169
7.2	Well-posed problems	172
7.3	Well-posed methods	175
7.4	Floating-point arithmetic	180

7.5	Computational error	186
7.6	Exercises	189
8	Solutions to Nonlinear Equations	191
8.1	Bisection method	191
8.2	Newton's method	193
8.3	Fixed-point iterations	197
8.4	Dynamical systems	201
8.5	Roots of polynomials	207
8.6	Systems of nonlinear equations	211
8.7	Homotopy continuation	215
8.8	Finding a minimum (or maximum)	218
8.9	Exercises	221
9	Interpolation	225
9.1	Polynomial interpolation	226
9.2	How good is polynomial interpolation?	232
9.3	Splines	235
9.4	Bézier curves	243
9.5	Exercises	246
10	Approximating Functions	249
10.1	Least squares approximation	250
10.2	Orthonormal systems	253
10.3	Fourier polynomials	256
10.4	Chebyshev polynomials	259
10.5	Wavelets	263
10.6	Neural networks	276
10.7	Data fitting	281
10.8	Exercises	292
11	Differentiation and Integration	295
11.1	Numerical differentiation	295
11.2	Automatic differentiation	299
11.3	Newton–Cotes quadrature	303
11.4	Gaussian quadrature	311
11.5	Monte Carlo integration	320
11.6	Exercises	326

## Numerical Differential Equations

### 12 Ordinary Differential Equations 331

- 12.1 Well-posed problems 331
- 12.2 Single-step methods 333
- 12.3 Lax equivalence theorem 341
- 12.4 Multistep methods 343
- 12.5 Runge–Kutta methods 351
- 12.6 Nonlinear equations 360
- 12.7 Stiff equations 361
- 12.8 Splitting methods 364
- 12.9 Symplectic integrators 368
- 12.10 Practical implementation 370
- 12.11 Exercises 376

### 13 Parabolic Equations 379

- 13.1 Method of lines 381
- 13.2 Von Neumann analysis 386
- 13.3 Higher-dimensional methods 391
- 13.4 Nonlinear diffusion equations 394
- 13.5 Exercises 397

### 14 Hyperbolic Equations 401

- 14.1 Linear hyperbolic equations 401
- 14.2 Methods for linear hyperbolic equations 402
- 14.3 Numerical diffusion and dispersion 408
- 14.4 Linear hyperbolic systems 410
- 14.5 Hyperbolic systems of conservation laws 420
- 14.6 Methods for nonlinear hyperbolic systems 425
- 14.7 Exercises 429

### 15 Elliptic Equations 431

- 15.1 A one-dimensional example 431
- 15.2 A two-dimensional example 434
- 15.3 Stability and convergence 439
- 15.4 Time-dependent problems 441
- 15.5 Practical implementation 442
- 15.6 Exercises 445

**16 Fourier Spectral Methods 447**

- 16.1 Discrete Fourier transform 447
- 16.2 Nonlinear stiff equations 454
- 16.3 Incompressible Navier–Stokes equation 458
- 16.4 Exercises 463

**Back Matter****A Solutions 467**

- A.1 Linear algebra 467
- A.2 Analysis 491
- A.3 Differential equations 513

**B Computing in Python and Matlab 543**

- B.1 Scientific programming languages 543
- B.2 Python 549
- B.3 Matlab 599

**References 647****Index 661****Julia Index 675****Python Index 679****Matlab Index 681**



# 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, 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 errors 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<sup>A</sup>T<sub>E</sub>X 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 included 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 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 to the support and patience of my mind-blowingly awesome wife. Thank you, Laura. I am grateful to the open-source community for developing tools such as tools Octave, R, Python, Julia, L<sup>A</sup>T<sub>E</sub>X, and Inkscape. I am grateful to 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  $\pi$  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 primary direct solver is Gaussian elimination. We can get more efficient methods such as banded solvers, block solvers, and Cholesky decomposition by taking advantage of a matrix's properties, 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  $\lambda$  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. It is impossible to solve the eigenvalue problem directly, so we must instead use iterative methods. In Chapter 4, we look at how to do this.

Finally, Chapter 6 looks at the fast Fourier transform, which has revolutionized computational mathematics, signal processing, imaging, and a host of other fields.

## ► Numerical methods for analysis

Numerical analysis develops tools and methods for problem-solving methods 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, such as handling complicated boundaries. Hence, finite element methods provide attractive solutions to engineering problems. Finite element methods often require 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 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 that 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. 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 Matlbesque, 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>

The code in this book was written and tested in Julia version 1.7.2. To cut down on redundancy, the `LinearAlgebra.jl` and `Plots.jl` packages are implicitly assumed always to be imported and 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 in 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.



gas molecules film  
from *The Restless Universe*

# Numerical Methods for Linear Algebra



## Chapter 1

---

# A Review of Linear Algebra



We'll start by reviewing some essential 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 Watkins' 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  $\mathbb{F}$ . We often say that  $V$  is a vector space over the field  $\mathbb{F}$  to remove the ambiguity. 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  $4 \times 4$  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  $4 \times 4$  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  $\mathbf{A}$  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 have<sup>1</sup>

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

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  $\text{span}\{(0, 0, 1)\}$ . ◀

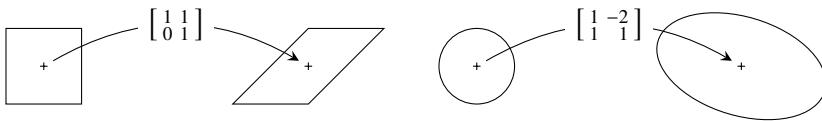
Several important types of matrices are listed on page 8. Remember, all vector spaces over  $\mathbb{R}^n$  are isomorphic to  $\mathbb{R}^n$ . So by considering maps in  $\mathbb{R}^n$ , we have a nice geometric interpretation of linear maps. Take  $\mathbb{R}^2$ . A circle (centered at the origin) 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 parallelepiped.

## 1.2 Eigenspaces

A number  $\lambda$  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

---

<sup>1</sup>These 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.

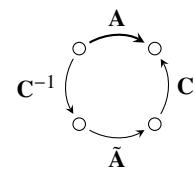


associated with an eigenvalue  $\lambda$  is called the *eigenspace*. A subspace  $S$  is said to be *invariant* with respect to a square matrix  $\mathbf{A}$  if any vector in  $S$  stays in  $S$  under mapping by  $\mathbf{A}$ . In other words  $\mathbf{AS} \subset S$ . An eigenspace is invariant, 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  $\mathbf{A} = \tilde{\mathbf{A}}\mathbf{C}^{-1}$ . 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,  $\tilde{\mathbf{A}}\mathbf{C}^{-1}$  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 *nondefective*. For nondefective matrices, we can rewrite the eigenvalue problem  $\mathbf{Ax} = \lambda\mathbf{x}$  as  $\mathbf{AS} = \mathbf{S}\Lambda$ , where  $\mathbf{S}$  is a matrix of the eigenvectors and  $\Lambda$  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  $\Lambda$  in its eigenvector basis. The system is uncoupled in this eigenvector basis and much easier to manipulate.

### 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 nonzero 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  $\text{diag}(a_{11}, a_{22}, \dots, a_{nn})$ .

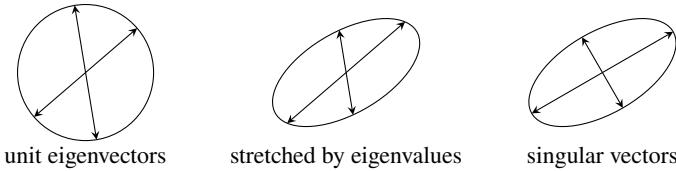
*Upper triangular:*  $a_{ij} = 0$  for  $i > j$ .

*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
$\begin{bmatrix} \bullet & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \bullet \end{bmatrix}$	$\begin{bmatrix} \bullet & & & \\ & \bullet & & \\ & & \ddots & \\ & & & \bullet \end{bmatrix}$	$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ & \ddots & & \\ & & \ddots & \\ & & & \bullet \end{bmatrix}$	$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ & \ddots & & \\ & & \ddots & \\ & & & \bullet \end{bmatrix}$	$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$



**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 Shur 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  $1 \times 1$  matrix with  $\mathbf{U} = 1$ . Now, assume that the hypothesis holds for  $n - 1$  and show that it also holds for  $n$ . Let  $\lambda$  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  $\hat{\mathbf{A}} = \mathbf{W}^H \mathbf{A} \mathbf{W} \in \mathbf{C}^{(n-1) \times (n-1)}$ . 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

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

Then  $\mathbf{U}_2$  is unitary and

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

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

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

□

**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 eigenvalues of  $\mathbf{A}$  and  $\Lambda$  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  $\Lambda$  is an upper triangular matrix whose diagonal elements are eigenvalues of  $\mathbf{A}$  and  $\mathbf{U}$  is a unitary matrix. It follows that

$$\begin{aligned} \mathbf{A}^H \mathbf{A} &= \mathbf{U} \Lambda^H \mathbf{U}^H \mathbf{U} \Lambda \mathbf{U}^H = \mathbf{U} \Lambda^H \mathbf{U} \Lambda^H \quad \text{and} \\ \mathbf{A} \mathbf{A}^H &= \mathbf{U} \Lambda \mathbf{U}^H \mathbf{U} \Lambda^H \mathbf{U}^H = \mathbf{U} \Lambda \Lambda^H \mathbf{U}^H. \end{aligned}$$

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

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

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

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

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

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

Continuing like this, it follows that  $\Lambda$  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  $\Lambda = \mathbf{F}\mathbf{C}\mathbf{F}^H$  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  $\Lambda$ . 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

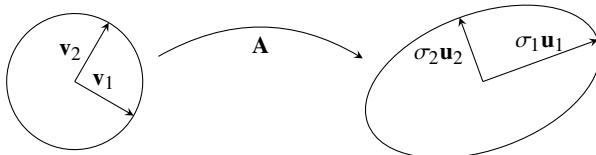
$$\mathbf{U}^H \mathbf{A} \mathbf{V} = \Sigma = \text{diag}(\sigma_1, \dots, \sigma_n)$$

with  $\sigma_1 \geq \dots \geq \sigma_n \geq 0$ . The numbers  $\sigma_i$  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  $2 \times 2$  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 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

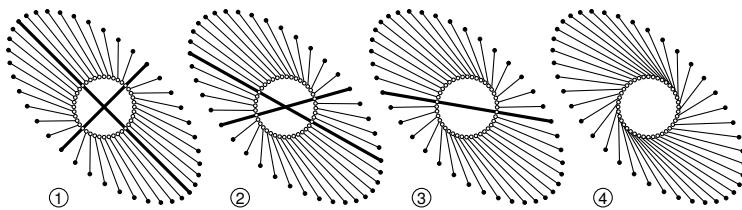


Figure 1.1: A matrix maps points on the unit circle to points on an ellipse. Such a mapping has two, one, or no real eigenvectors (thick line segments) running along the radial directions.

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.  $\blacktriangleleft$

### 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*  $(\mathbf{u}, \mathbf{v})_2 = \mathbf{u}^T \mathbf{v}$  and the *A-energy inner product*  $(\mathbf{u}, \mathbf{v})_A = \mathbf{u}^T \mathbf{A} \mathbf{v}$  where  $\mathbf{A}$  is symmetric, positive definite. Vectors that are orthogonal under the  $\mathbf{A}$ -energy inner product are said to be  $\mathbf{A}$ -orthogonal or  $\mathbf{A}$ -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  $\mathbf{A}$ -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  $(\Lambda^{1/2}\mathbf{Q}^{-1}\mathbf{u})^T(\Lambda^{1/2}\mathbf{Q}^{-1}\mathbf{v})$ . 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:<sup>2</sup>

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

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  $\infty$ -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  $\alpha$  we have

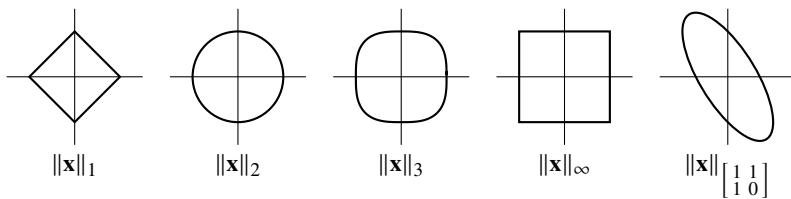
$$0 \leq \|\mathbf{x} - \alpha\mathbf{y}\|^2 = \|\mathbf{x}\|^2 - 2\alpha(\mathbf{x}, \mathbf{y}) + |\alpha|^2\|\mathbf{y}\|^2.$$

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  $\|\mathbf{Qx}\|_2 = \|\mathbf{x}\|_2$ . This result should be clear from the geometric interpretation of the Euclidean norm and orthogonal transformation, but you can crosscheck it with simple algebra:  $\|\mathbf{Qx}\|_2^2 = \|\mathbf{x}^\top \mathbf{Q}^\top \mathbf{Qx}\|_2 = \|\mathbf{x}^\top \mathbf{x}\|_2 = \|\mathbf{x}\|_2^2$ .

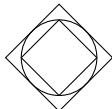
Two norms  $\|\cdot\|_\alpha$  and  $\|\cdot\|_\beta$  on a vector space  $V$  are *equivalent* if there are positive constants  $c$  and  $C$  such that  $c\|\mathbf{u}\|_\alpha \leq \|\mathbf{u}\|_\beta \leq C\|\mathbf{u}\|_\alpha$  for all vectors  $\mathbf{u} \in V$ . All norms on finite-dimensional 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.

---

<sup>2</sup>The  $p$ -norm is the discrete, finite-dimensional analog to an  $\ell^p$ -norm for an infinite-dimensional vector space and an  $L^p$ -norm for a continuous vector space.

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

**Example.** Let's first show that the 1- and 2-norms are equivalent and then show that the 2- and  $\infty$ -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

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.*

$$1. \quad \|\mathbf{Ax}\| = \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} \|\mathbf{x}\| \leq \left( \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} \right) \|\mathbf{x}\| = \|\mathbf{A}\| \|\mathbf{x}\|$$

$$2. \quad \|\mathbf{I}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{x}\| = 1$$

$$3. \quad \|\mathbf{AB}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{ABx}\| \leq \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}\| \|\mathbf{Bx}\| = \|\mathbf{A}\| \|\mathbf{B}\| \quad \square$$

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  $\infty$ -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,

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

Finally, consider the unit circle for the  $\infty$ -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

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

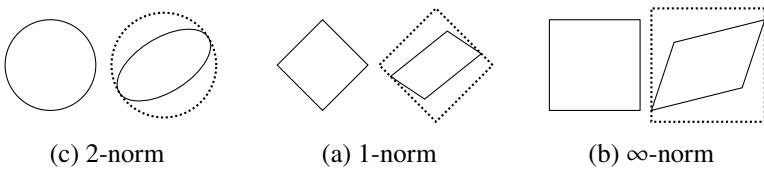


Figure 1.3: Mapping of unit circles by  $\frac{1}{4} \begin{bmatrix} 1 & 1 \\ 1 & 3 \end{bmatrix}$ .

Note that  $\|A\|_\infty = \|A^T\|_1$ . You can use the superscripts of the 1- and  $\infty$ -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 “ $\infty$ ” 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  $A$ , 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  $\lambda_i$  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  $\lambda$  is an eigenvalue of  $\mathbf{A}$ , then  $\mathbf{Ax} = \lambda\mathbf{x}$  for some vector  $\mathbf{x}$ . So  $\|\mathbf{Ax}\| = |\lambda|\|\mathbf{x}\|$ . From this it follows that  $|\lambda| = \|\mathbf{Ax}\|/\|\mathbf{x}\|$ , 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  $\xi_j$  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  $\xi_j$ . 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  $\rho(\mathbf{A}) \approx 10.03$ .  $\blacktriangleleft$

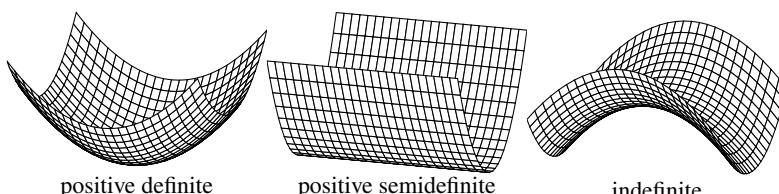
## ► Quadratic forms

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  $2 \times 2$  matrix,

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



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  $\lambda_i$ . 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  $\psi$  is denoted by a special symbol  $| \rangle$ . We can then express the vector  $a\psi$  as  $a| \rangle$ . Because this notation is ambiguous when working with more than one vector, say  $\psi_1$  and  $\psi_2$ , 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  $a\psi_1$  as  $a|1\rangle$ . Bear in mind that anything between  $|$  and  $\rangle$  is simply a label, so  $|a\rangle \neq a|1\rangle$  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  $\langle \cdot \rangle$ . Define a linear functional  $\langle \cdot \rangle = (\phi, \cdot) = \int_{-\infty}^{\infty} \phi^* \cdot dx$ , 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  $\langle \cdot \rangle$ . Just as we used labels to make  $\rangle$  unambiguous, we can do the same for  $\langle \cdot \rangle$ . To denote a specific operator  $\langle \cdot \rangle$ , we can similarly enclose a label  $a$  between  $\langle$  and  $|$  to get  $\langle a | \cdot \rangle$ . 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  $\psi$  are called wave functions, and by convention are normalized over space  $\int_{-\infty}^{\infty} \psi^* \psi dx = 1$ . This gives a wavefunction  $\psi$  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  $\psi$  a superposition of states, they are saying that the vector  $\psi$  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  $x$ :  $\langle \mathbf{x} \rangle = \langle x \rangle$ . The composition  $\langle \mathbf{x} \rangle$  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.  $\blacktriangleleft$

## 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 fail 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, a unique solution exists. 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  $\kappa_p(\mathbf{A})$ .

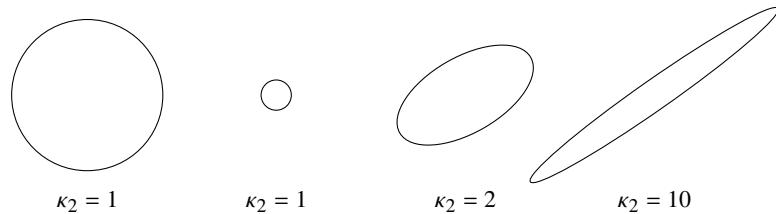


Figure 1.4: Images of unit circles under different linear transformations and their associated condition numbers.

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  $\kappa_2(\mathbf{A}) = \sigma_{\max}/\sigma_{\min}$ , the ratio of radii of the semi-major to semi-minor axes of an ellipsoid image of a matrix. See the figure above.

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  $\mathbf{A}$  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 quite 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' book *Fundamentals of Matrix Computations* for an 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  $4 \times 4$  Hilbert matrix is

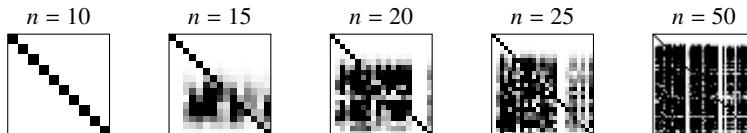
$$\mathbf{H} = \begin{bmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{bmatrix}$$

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 with magnitude 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  $n = 15$ . ▶

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

• 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 the 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  $\infty$ -norm. The 1-norm is the sum of the absolute values of the coordinates of a vector in the standard basis, and the  $\infty$ -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  $\infty$ -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  $\lambda$ , 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  $m$ ? 

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  $1 \times 1$  (0,1)-matrices, namely,  $[0]$  and  $[1]$ . So half are invertible. There are sixteen  $2 \times 2$  matrices whose entries are ones and zeros:

$$\text{Singular: } \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\text{Invertible: } \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

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  $2^{n^2}$   $n \times n$  (0,1)-matrices—for instance, roughly  $10^{19}$   $8 \times 8$  matrices. To check each  $8 \times 8$  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  $n = 1, 2, \dots, 20$ .



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  $\mathbf{D}$ .
- (b) Compute the spectral radius  $\rho(\mathbf{D} + 2\mathbf{I})$ .
- (c) Determine the condition number  $\kappa_2(\mathbf{D})$  and discuss its 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  $\|\mathbf{A}\|_{\text{F}} = \sqrt{\sum_{i,j} a_{ij}^2}$ . 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  $\|\mathbf{Q}\|_{\text{F}}$  where  $\mathbf{Q}$  is an  $n \times n$  orthogonal matrix.
- (c) Prove that the Frobenius norm is invariant under orthogonal transformations  $\|\mathbf{Q}\mathbf{A}\|_{\text{F}} = \|\mathbf{A}\|_{\text{F}}$ .

- (d) Prove that  $\|\mathbf{A}\|_F = \sqrt{\sum_i \sigma_i^2}$ , where  $\sigma_i$  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). 

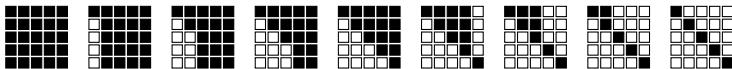
1.10. Cleve Moler and Charles van Loan's article "Nineteen dubious ways to compute the exponential of a matrix" presents several methods of computing  $e^{\mathbf{A}}$  for a matrix  $\mathbf{A}$ . Discuss or implement one of the methods. How is a matrix exponent computed in practice?



## 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 program any of these algorithms explicitly. However, it is worthwhile to understand the mathematics underpinning these packages to be aware of potential pitfalls. This chapter discusses direct methods for solving the problem  $\mathbf{Ax} = \mathbf{b}$  when  $\mathbf{A}$  is invertible. We will discuss methods for solving  $\mathbf{Ax} = \mathbf{b}$  when  $\mathbf{A}$  is not invertible in Chapter 3. And we will 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  $2000 \times 2000$  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 the square matrix  $\mathbf{A}$ . It then applies 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  $2000 \times 2000$  tridiagonal matrix 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 happening 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

Elementary row operations leave a matrix in row equivalent form, i.e., they 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.

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

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

$$\left[ \begin{array}{cccc|c} x & x & x & x & | & x \\ x & x & x & x & | & x \\ x & x & x & x & | & x \\ x & x & x & x & | & x \end{array} \right] \rightarrow \left[ \begin{array}{cccc|c} x & x & x & x & | & x \\ 0 & x & x & x & | & x \\ 0 & 0 & x & x & | & x \\ 0 & 0 & 0 & x & | & x \end{array} \right] \rightarrow \left[ \begin{array}{cccc|c} x & x & x & x & | & x \\ 0 & x & x & x & | & x \\ 0 & 0 & x & x & | & x \\ 0 & 0 & 0 & x & | & x \end{array} \right] \rightarrow \left[ \begin{array}{cccc|c} x & x & x & x & | & x \\ 0 & x & x & x & | & x \\ 0 & 0 & x & x & | & x \\ 0 & 0 & 0 & x & | & x \end{array} \right]$$

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

$$\left[ \begin{array}{cccc|c} x & x & x & x & | & x \\ x & x & x & x & | & x \\ x & x & x & x & | & x \\ x & x & x & x & | & x \end{array} \right] \rightarrow \left[ \begin{array}{ccc|cc} x & x & x & | & x \\ 0 & x & x & | & x \\ 0 & 0 & x & | & x \\ 0 & 0 & 0 & | & x \end{array} \right] \rightarrow \left[ \begin{array}{cc|cc} x & x & | & x \\ 0 & x & | & x \\ 0 & 0 & | & x \\ 0 & 0 & | & x \end{array} \right] \rightarrow \left[ \begin{array}{c|c} x & x \\ 0 & x \\ 0 & 0 \\ 0 & 0 \end{array} \right]$$

• 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. This 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. John von Neumann and Herman Goldstine developed Gaussian elimination in its modern matrix form 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. The lower triangular system

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

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

$$l_{i1}y_1 + \dots + l_{i,i-1}y_{i-1} + y_i = b_i,$$

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. It is often convenient to overwrite the array  $\mathbf{b}$  with the values  $\mathbf{y}$  to save computer memory.

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 when  $n$  is large. So it's important to make efficient use of it. Storing the full matrices **A**, **L**, and **U** in computer memory takes  $3n^2$  floating-point numbers (at eight bytes each). The matrices **L** and **U** together have the same effective information **A**, so keeping **A** is unnecessary. Also, **L** and **U** are almost half-filled with zeros—wasted memory. Now, the smart idea! Note that except along the diagonal, the nonzero 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

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

Now, the really smart idea! Overwrite **A** with the elements of **L** and **U**. When performing 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
         $a_{ij} \leftarrow a_{ij}/a_{jj}$ 
        for k = j + 1, ..., n
             $a_{ik} \leftarrow a_{ik} - a_{ij}a_{jk}$ 
    } LU decomposition
for i = 2, ..., n
     $b_i \leftarrow b_i - \sum_{j=1}^{i-1} a_{ij}b_j$ 
} forward elimination
for i = n, n - 1, ..., 1
     $b_i \leftarrow (b_i - \sum_{j=i+1}^n a_{ij}b_j) / a_{ii}$ 
} 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  $1000 \times 1000$  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 requires

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

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 saves time when solving 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

Gaussian elimination fails if the pivot  $a_{ii}$  is zero at any step because we are dividing by zero. In practice, even if a pivot is close to zero, Gaussian elimination may be unstable because round-off errors get 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 2-norm condition number of this matrix is about 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 reconstructed matrix  $\tilde{\mathbf{A}}$  differ in the bottom-right elements. Let's see how well our naïve function `gaussian_elimination` does in solving  $\mathbf{Ax} = \mathbf{b}$  when  $\mathbf{b} = (-5, 10, 0)$ .

```

 $\epsilon = 1e-15; A = [1 1 2; 2 2+\epsilon 0; 4 14 4]; b = [-5; 10; 0.0]$ 
A\b, gaussian_elimination(A,b)

```

The function fails horribly, returning a solution of about  $(1, 4, -5)$  instead of the correct answer  $(5, 0, -5)$ . The numerical instability results from dividing by a small pivot. We can avoid it by permuting the rows or columns at each iteration so that a large number is in the pivot position.

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 magnitudes 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  $\mathbf{R}^T \mathbf{R}$ , where  $\mathbf{R}$  is an upper triangular matrix. This factorization is called the Cholesky decomposition. Let's prove this property and then examine how to implement it numerically.

**Theorem 7.** *A symmetric, positive-definite matrix  $\mathbf{A}$  has the decomposition  $\mathbf{R}^T \mathbf{R}$  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  $1 \times 1$  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  $\mathbf{A} = \mathbf{LDL}^T$ , where  $\mathbf{L}$  is a unit lower triangular matrix and  $\mathbf{D}$  is a diagonal matrix.  $\mathbf{A}$  has the decomposition  $\mathbf{LDM}^T$ , 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  $a_{11}$ :

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

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

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:

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

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

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

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

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

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

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

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

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
         $r_{ij} \leftarrow \frac{1}{r_{ii}} \left( a_{ji} - \sum_{k=1}^{i-1} r_{kj}r_{ki} \right)$ 

```

We can successively overwrite the matrix  $\mathbf{A}$ , just as in LU decomposition. Cholesky decomposition requires  $\frac{1}{3}n^3$  operations instead of  $\frac{2}{3}n^3$  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 large-scale industrial and military production, management, and logistics problems. 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.<sup>1</sup> By analyzing a system, determining the objective to be fulfilled, and

---

<sup>1</sup>The 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).<sup>2</sup>

As a typical example of an LP problem, imagine that you are tasked with shipping canned tomatoes from several factories to different warehouses 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). The decision variables of the cannery problem 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 the least amount of money a typical person would need to spend on food to maintain good health was. 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, working 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

---

<sup>2</sup>The term *linear programming* says nothing about the problems' primary challenge, that of solving a system of inequalities. Just a few years after its introduction, physicist Phillip M. Morse lamented, “it seems to me that the term ‘linear programming’ is a most unfortunate phrase for this promising technique, particularly since many possible extensions appear to be in nonlinear directions. A more general yet more descriptive term, such as ‘bounded optimization,’ might have been a happier choice.”

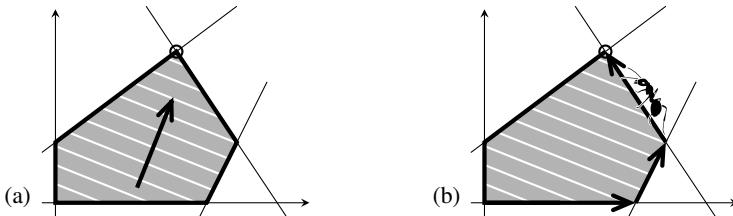


Figure 2.1: A two-dimensional simplex. (a) The objective function reaches its maximum and minimum at vertices of the simplex. (b) The simplex method steps along edges from vertex to vertex in a direction that increases the objective function, just as an ant might crawl, until it finally reaches a maximum.

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. 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  $\mathbf{A}\mathbf{x} \leq \mathbf{b}$  and nonnegativity 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 better understand 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  $n$ -polytype, also known as a simplex, in the positive orthant (a convex polygon in the upper right quadrant in two dimensions). The objective function  $\mathbf{c}^\top \mathbf{x}$  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 water's surface 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 to a face or edge of the polyhedron (infinitely many solutions). This characterization 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  $\varepsilon$  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. 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 nonnegativity restriction  $\mathbf{x}, \mathbf{s} \geq 0$ .

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

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

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 simplify the discussion. 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 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[:,:] = 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∈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, we can add the command

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

to the while loop. The initial tableau consists of six blocks **A**, **I**, **b**,  $\mathbf{c}^T$ , **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 one. 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 nonnegativity 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 nonnegativity 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 several 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 implementing 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.

Suppose that we have the  $4 \times 6$  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	2
value	11	41	22	33	43	14	34	44

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

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

• Julia's backslash and forward slash operators use UMFPACK (pronounced "Umph Pack") to solve nonsymmetric, sparse systems directly.

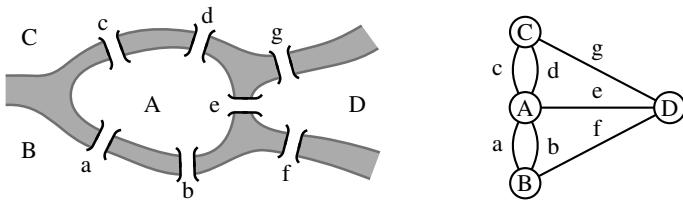
The *density* of a sparse matrix is the number of nonzero elements divided by the total number of elements. Conversely, the *sparsity* is the number of zero elements divided by the total number of elements. Julia's SparseArrays.jl function `nnz` returns the number of nonzero elements of sparse matrix. For example, `A = sprand(60, 80, 0.2)` will construct a 60-by-80 sparse, random matrix with a density of 0.2 and nonzero elements uniformly distributed over the unit interval. The command `nnz(A)` returns an integer in the neighborhood of 960. The Plots.jl function `spy(A)` draws the sparsity plot of matrix `A`, a row-column scatter plot of its nonzero values. The `spy` command is not always needed as simply outputting `A` to the console in Julia will provide a sparsity pattern printed using the Unicode braille pattern.

## ► 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.<sup>3</sup>

---

<sup>3</sup>Euler published his paper 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 two others were later demolished to make way for a modern highway.



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, we use graph theory 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 a path exists 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.

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. It follows that the vector of ones is in the nullspace, i.e., the vector of ones is an eigenvector. Finally, the number of connected components equals the dimension of the nullspace, i.e., the multiplicity of the zero 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  $\Delta$ . We can write the solution  $\mathbf{u}(t)$  using the eigenvector basis  $\mathbf{L}$ . Namely,

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

for eigenvalues  $\mathbf{v}$ , 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.<sup>4</sup>

In moving from 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 the next smoothest, and so on. The solution can be approximated using the smallest eigenvalues  $\lambda_2$  and  $\lambda_3$  near equilibrium. This approximation 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 with 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 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

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

---

<sup>4</sup>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.

and the parameters  $k$  and  $\alpha$  adjust the optimal distance between vertices. 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 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  $\Delta t$  is 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 potentially take hundreds of iterations to reach the equilibrium solution, force-directed graph drawing may require significant computation time.

- The `Graphs.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 southwest 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.<sup>5</sup> Let's draw the graph of the frequent associations of the dolphins. First, we'll define a helper 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], ones.(ij[:,1]))
end
```

Then, we'll plot the graph.

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

---

<sup>5</sup>The collection is a curated set of around 3000 sparse matrices that occur in applications such as computational 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.

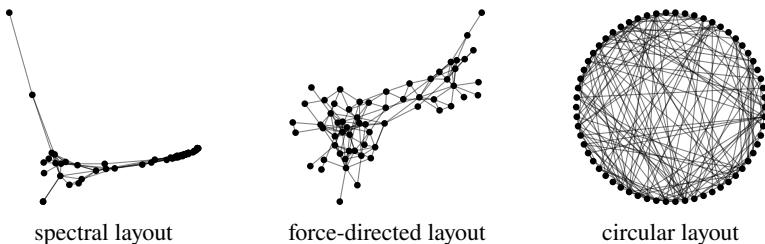


Figure 2.2: Graph drawing layouts for the bottlenose dolphin network.

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 appears to consist of two smaller two pods. The circular layout draws the vertices at equally spaced points along a circle's circumference. 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 a method of sorting the vertices to minimize the bandwidth. ◀

### ► Preserving sparsity

A matrix is a *banded matrix* with lower bandwidth  $p$  and upper bandwidth  $q$  if the  $ij$ -elements are zero except when  $j - q \leq i \leq j + p$ . 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 approximations to one-dimensional partial differential equations.

A banded matrix with lower bandwidth  $p$  and an upper bandwidth  $q$  has an LU decomposition with a lower bandwidth of  $p$  and an upper bandwidth of  $q$ . The number of multiplications and additions is approximately  $2npq$ . Forward and backward substitution requires an additional  $2np + 2nq$  operations.

Consider the simple example of the following sparse matrix:

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \quad \text{has the LU decomposition} \quad \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$$

with a tremendous fill-in. But, if we reorder the rows and columns first by moving

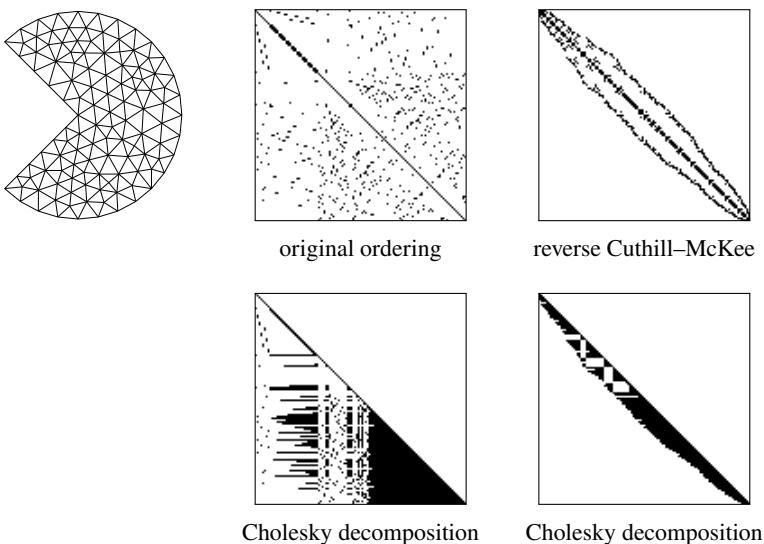


Figure 2.3: The original ordering and the reverse Cuthill–McKee ordering of the sparsity plot for the Laplacian operator in a finite element solution for a pacman domain. The Cholesky decompositions for each ordering are directly below.

the first row to the bottom and the first column to the end, then

$$\begin{bmatrix} \bullet & & & \\ \cdot & \ddots & & \\ & \cdot & \ddots & \\ & & \ddots & \ddots \end{bmatrix} \quad \text{has the LU decomposition} \quad \begin{bmatrix} \bullet & & & \\ \cdot & \ddots & & \\ & \cdot & \ddots & \\ & & \ddots & \ddots \end{bmatrix}.$$

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 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.<sup>6</sup> The algorithm builds a permutation sequence starting with an empty set:

---

<sup>6</sup>Dijkstra'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

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 from lowest degree to highest degree, skipping any already vertex.
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  $12 \times 12$  matrix in the figure on the facing page. We start the Cuthill–McKee algorithm by selecting a column of the matrix with the fewest off-diagonal nonzero elements. Column 5 has one nonzero element in row 10, so we start with that one. 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 we’ve already counted. We continue in this fashion until we’ve run through all 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 sorting 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 so 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 nonnegativity restriction  $\bar{\mathbf{x}}, \mathbf{b} \geq 0$ ,

which has the tableau

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

---

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 a different path—a strategy one might use to solve mazes.

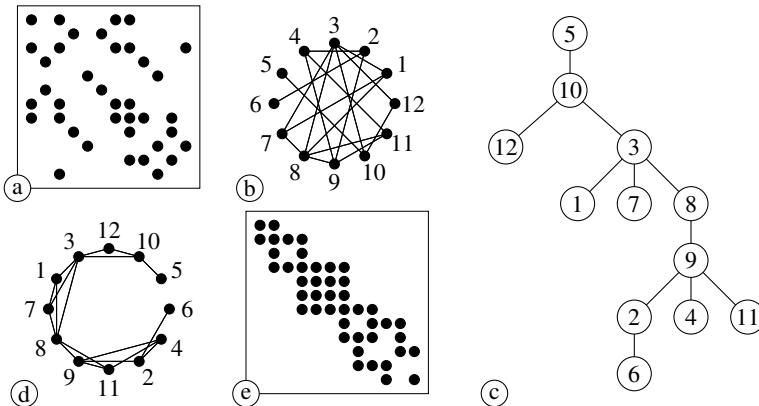


Figure 2.4: Cuthill–McKee sorting: (a) original sparsity pattern; (b) associated graph; (c) completed breadth-first search tree; (d) reordered graph; (e) sparsity pattern of column-and-row-permuted matrix.

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 nonbasis 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 nonbasic 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):

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

At each iteration, we need to keep track of which variables are basic and which ones are nonbasic, 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  $\xi_j$  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  $\mathbf{x}_B = \mathbf{A}_B^{-1} \mathbf{b}$  remains nonnegative. Let  $\mathbf{p}$  be the  $i$ th column of  $\mathbf{A}_B$ , the column associated with the leaving variable.

After swapping columns  $\mathbf{p}$  and  $\mathbf{q}$  between  $\mathbf{A}_B$  and  $\mathbf{A}_N$ , we'll need to update  $\mathbf{A}_B^{-1}$ . 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:

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

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

$$(\mathbf{A}_B + (\mathbf{q} - \mathbf{p})\xi_i^T)^{-1} = \mathbf{A}_B^{-1} - \hat{q}_i^{-1}(\hat{\mathbf{q}} - \xi_i)\xi_i^T\mathbf{A}_B^{-1}$$

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

Let's implement the revised simplex method in Julia. We'll take  $\text{ABinv}$  to be  $\mathbf{A}_B^{-1}$  and  $B$  and  $N$  to be the list of columns of  $\bar{\mathbf{A}}$  in  $\mathbf{A}_B$  and  $\mathbf{A}_N$ , respectively. We define a method  $\xi(i)$  to return a unit vector. In practice, we can save and update  $[\mathbf{A}_B^{-1} \quad \mathbf{A}_B^{-1} \mathbf{b}]$  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 isnothing(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 function `gaussian_elimination` program on page 33 to include partial pivoting. Use both functions to solve the system  $\mathbf{Ax} = \mathbf{b}$ , where

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

with  $\varepsilon = 10^{-15}$  (or about five times machine epsilon).

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 provides an efficient way to compute the determinant of a matrix. (Julia, Python, and Matlab all use this approach.) 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 49. 

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 as edges of a bipartite graph *celegans.csv* from <https://github.com/nmfsc/data>. 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* from <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.<sup>7</sup> In a contemporary version of Karinthy’s game called “six degrees of Kevin

---

<sup>7</sup>In 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.

Bacon,” players attempt to trace an arbitrary actor to Kevin Bacon by identifying a chain of movies coappearances.<sup>8</sup> 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 a Bacon number of one. Anyone who has acted with someone who has acted with Kevin Bacon has a 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.

2.9. The CSC and CSR formats require some overhead to store a sparse matrix. What is the break-even point in terms of the density?

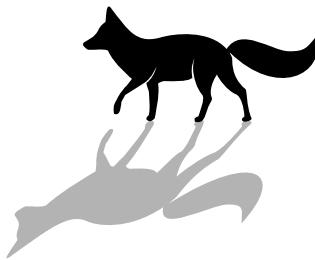
---

<sup>8</sup>Mathematicians sometimes play a nerdier game by tracking the chain of co-authors on research papers to the prolific Hungarian mathematician Paul Erdős.

## Chapter 3

---

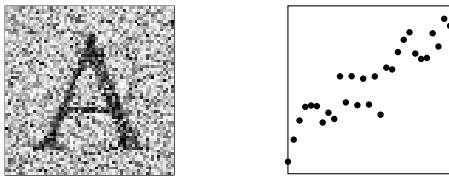
# Inconsistent Systems



In the last chapter, we examined solutions to square, consistent systems. Now, we turn our attention to nonsquare matrices. We can solve a nonsquare system  $\mathbf{Ax} = \mathbf{b}$  like the square system in Julia using the `\` function. Computing  $\mathbf{A}\backslash\mathbf{b}$  for a  $2000 \times 1999$  matrix takes 1.54 seconds, considerably longer than the 0.15 seconds to solve a  $2000 \times 2000$  matrix. Alternatively, solving the same nonsquare 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 has a unique solution. This chapter examines 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

Perhaps we want to solve a problem with noisy input data. The problem might be curve fitting, image recognition, or statistical modeling. We may have sufficient 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 next 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  $\mathbf{b}$ . In this case, 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  $\infty$ -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  $\infty$ -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  $\infty$ -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.<sup>1</sup>

## ► 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 expanding,

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

which says

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

---

<sup>1</sup>In 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. Taking the gradient of

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

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  $\mathbf{r} = \mathbf{A}\mathbf{e} = \mathbf{b} - \delta\mathbf{b}$  where  $\delta\mathbf{b} = \mathbf{A}\mathbf{e}$ . 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  $\mathbf{A}$ . Whatever of the origins of the term *normal equation*,<sup>2</sup> it serves as a useful mnemonic that 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

---

<sup>2</sup>According to William Kruskal and Stephen Stigler, Gauss introduced term (*Normalgleichungen*) in an 1822 paper without motive. They offer possibilities about its origin but nothing dominates.

$\mathbf{Ax} = \mathbf{b} - \delta\mathbf{b}$  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! Furthermore, zeroing out  $\mathbf{x}_{\text{null}}$  will 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  $\kappa(\mathbf{A}^T\mathbf{A})$  equals  $\kappa(\mathbf{A})^2$ , and the condition number  $\kappa((\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T)$  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  $\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$ . Taking  $\mathbf{A} = \mathbf{QR}$ , then

$$(\mathbf{QR})^T \mathbf{QRx} = (\mathbf{QR})^T \mathbf{b},$$

which reduces to

$$\mathbf{R}^T \mathbf{Rx} = \mathbf{R}^T \mathbf{Q}^T \mathbf{b}.$$

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

$$\mathbf{R}^T (\mathbf{Rx} - \mathbf{Q}^T \mathbf{b}) = \mathbf{0},$$

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

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

So  $\mathbf{Rx}$  must equal  $\mathbf{Q}^T \mathbf{b}$  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  $\mathbf{Rx} = \mathbf{Q}^\top \mathbf{b}$ :

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

We discard the inconsistent bottom equations and solve the system

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

In the next chapter, we'll use orthogonal matrices to help us find eigenvalues. One reason orthogonal matrices are particularly lovely 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  $\mathbf{Q}$ ? We typically use 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 generates 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 have 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, ..., n
  q_k ← v_k
  for j = 1, ..., k - 1
    r_jk = (q_j, v_k)
    q_k ← q_k - r_jk q_j
  r_kk ← ||q_k||_2
  q_k ← 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  $\theta$  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  $2 \times 2$  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  $5 \times 5$  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. The product  $\mathbf{Q}_{52}\mathbf{A}$  equals

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

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

For example, consider a  $4 \times 4$  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:

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

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

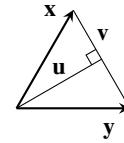
While Givens rotations zero out one element at a time, *Householder reflections* speed things up by changing whole columns all at once

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

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  $\mathbf{y} = \pm \|\mathbf{x}\|_2 \boldsymbol{\xi}$  with  $\boldsymbol{\xi} = (1, 0, \dots, 0)$  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:  $\mathbf{y} = (\mathbf{I} - 2\mathbf{P}_\mathbf{v})\mathbf{x}$ . In general, for any vector in the direction of  $\mathbf{v}$ , we define the *Householder reflection matrix*:

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



Because we want  $\mathbf{y}$  to be  $\pm \|\mathbf{x}\|_2 \boldsymbol{\xi}$ , we'll take  $\mathbf{v} = \mathbf{x} \pm \|\mathbf{x}\|_2 \boldsymbol{\xi}$ . Technically, we ought to take  $\mathbf{v} = (\mathbf{x} \pm \|\mathbf{x}\|_2 \boldsymbol{\xi})/2$ , but the one-half scaling factor drops out when we compute  $\mathbf{P}_\mathbf{v}$ . 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 second column's subdiagonal elements, 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  $\pm \|\mathbf{x}\| \boldsymbol{\xi}$ .

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

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

From this expression, we see that the problem of solving  $\mathbf{Ax} = \mathbf{b}$  can be replaced with the equivalent problem of solving the upper triangular system  $\mathbf{Rx} = \mathbf{c}$  where  $\mathbf{c} = \mathbf{Q}^\top \mathbf{b}$ . The residual  $\mathbf{r}$  of this system

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

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

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

The term  $\|\mathbf{c}_2\|_2^2$  is independent of  $\mathbf{x}$ —it doesn't change  $\|\mathbf{r}\|_2^2$  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  $\|\mathbf{r}\|_2$ . 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

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

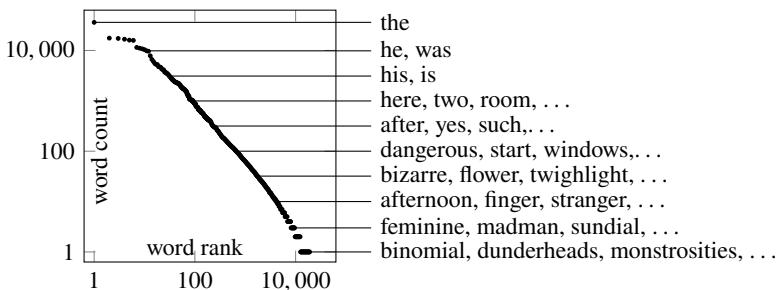


Figure 3.1: Frequency versus rank of words in Sherlock Holmes.

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

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

and hence  $\|\mathbf{r}\|_2^2$  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 solution will be the "best" solution. We develop a selection 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.

**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. 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*<sup>3</sup> appears 54 times. And

<sup>3</sup>Professor James Moriarty—criminal mastermind and archenemy of Sherlock Holmes—was a brilliant mathematician who at the 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.”

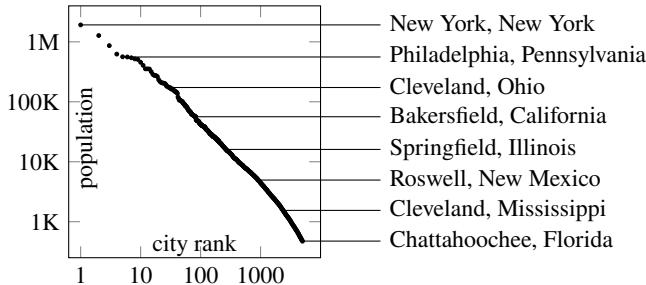


Figure 3.2: Population versus rank of cities in the United States.

there are 6610 words like *binomial*, *dunderheads*, *monstrosities*, *sunburnt*, and *vendetta* that appear only once. See Figure 3.1 on the facing page.

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 log-linear 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
```

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.<sup>4</sup> We find that  $s$  is  $-0.770$ . ◀

## 3.2 Underdetermined systems

So far, 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 valid. If the null space of  $\mathbf{A}$  has more than the zero vector, then  $\mathbf{A}^T \mathbf{A}$  is not invertible. And the upper triangular submatrix  $\mathbf{R}_1$  discussed in the previous section does not have full rank. In this case, the problem has infinitely many solutions. So then,

<sup>4</sup>See *UScities.csv* at <https://github.com/nmfsc/data>

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 works exceptionally well in problems where  $\mathbf{x}$  has a zero mean, which frequently appear 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  $\alpha$  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 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 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  $2\mathbf{A}^\top \mathbf{A} \mathbf{x} - 2\mathbf{A}^\top \mathbf{b} + 2\alpha^2 \mathbf{x} = \mathbf{0}$ , which we can solve to get

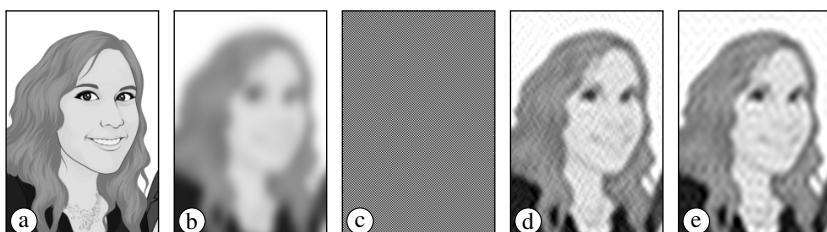
$$\mathbf{x} = (\mathbf{A}^\top \mathbf{A} + \alpha^2 \mathbf{I})^{-1} \mathbf{A}^\top \mathbf{b} = \mathbf{R}_\alpha \mathbf{b}. \quad (3.2)$$

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  $\mathbf{A}^\top (\mathbf{A} \mathbf{A}^\top + \alpha^2 \mathbf{I}) = (\mathbf{A}^\top \mathbf{A} + \alpha^2 \mathbf{I}) \mathbf{A}^\top$ , we have

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

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

**Example.** Photographs sometimes have motion or focal blur and noise, something we might want to remove or minimize. Suppose that  $\mathbf{x}$  is some original image and  $\mathbf{y} = \mathbf{Ax} + \mathbf{n}$  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, so the problem is ill-posed. Compare the following images:



The first image (a) shows the original image  $\mathbf{x}$ . Blur and noise have been added to this image to get the second image  $\mathbf{y} = \mathbf{Ax} + \mathbf{n}$ . Image (c) shows the solution using the least squares method  $(\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{y}$ . 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 fails. Image (d) shows the regularized least squares solution  $(\mathbf{A}^T \mathbf{A} + \alpha^2 \mathbf{I})^{-1} \mathbf{A}^T \mathbf{y}$  for a well-chosen  $\alpha$ . I think that this one is the winner. Image (e) shows the pseudoinverse  $\mathbf{A}^+ \mathbf{y}$ . This one is also good, although perhaps not as good as the regularized least squares solution.  $\blacktriangleleft$

### ► 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  $\alpha$  to 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  $\alpha^2$  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

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

where the  $\lambda$  is a vector of Lagrange multipliers. The pair  $(\tilde{\mathbf{x}}, \lambda)$  is a solution to the constrained minimization problem when  $\nabla \mathcal{L}(\mathbf{x}, \lambda) = \mathbf{0}$ . The zero gradient

gives us the systems of equation  $\mathbf{A}^T \mathbf{A} \tilde{\mathbf{x}} - \mathbf{A}^T \mathbf{b} + \mathbf{C}^T \lambda = 0$ , which we can rewrite along with the system of constraints  $\mathbf{C} \tilde{\mathbf{x}} = \mathbf{d}$  in block matrix form

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

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  $\mathcal{L}(\mathbf{x}, \mu, \lambda) = f(\mathbf{x}) + \mu^T \mathbf{g}(\mathbf{x}) + \lambda^T \mathbf{h}(\mathbf{x})$ , the value  $\mathbf{x}^*$  is a local minimum if the following KKT conditions hold:

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

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

## ► Sparse least squares

QR decomposition and Gram matrices both fill in sparse matrices. We can avoid fill-in by rewriting the normal equation  $\mathbf{A}^T \mathbf{A} \mathbf{x} = \mathbf{A}^T \mathbf{b}$  as a expanded block system using the residual. Because the residual  $\mathbf{r} = \mathbf{b} - \mathbf{A} \mathbf{x}$  is in the left null space of  $\mathbf{A}$ , it follows that  $\mathbf{A}^T \mathbf{r} = \mathbf{0}$ . From these two equations, we have

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

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

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

### 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  $\|\mathbf{b} - \mathbf{Ax}\|_2$  by forcing  $\mathbf{b}_{\text{left null}} = \mathbf{0}$  and ensure a unique solution  $\|\mathbf{x}\|_2$  by forcing  $\mathbf{x}_{\text{null}} = \mathbf{0}$ . We can use singular value decomposition to both zero out the left null space and zero out the null space. A matrix's singular value decomposition is  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ , where  $\mathbf{U}$  and  $\mathbf{V}$  are orthogonal matrices and  $\Sigma$  is a diagonal matrix of singular values. By convention, the singular values  $\sigma_i$  are 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 developing tools for finding eigenvalues. In this section, we look at applications of singular value decomposition.

The SVD of an  $m \times n$  matrix  $\mathbf{A}$  is an  $m \times m$  matrix  $\mathbf{U}$ , an  $m \times n$  diagonal matrix  $\Sigma$ , 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  $m > n$ , only the first  $n$  left singular vectors are needed to reconstruct  $\mathbf{A}$ . The other columns are superfluous. 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  $\Sigma$ , and an  $n \times n$  matrix  $\mathbf{V}^T$ . See Figure 3.3 on the next page. The same holds for low-rank matrices. For a matrix  $\mathbf{A}$  with rank  $r$ ,

- 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  $\mathbf{A}$ .

We can reconstruct  $\mathbf{A}$  using an  $m \times r$  matrix  $\mathbf{U}$ , an  $r \times r$  diagonal matrix  $\Sigma$ , and an  $r \times n$  matrix  $\mathbf{V}^T$ . This type of a reduced SVD is called a *compact* SVD.

Small singular values do not contribute much information 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  $\Sigma_k$ , and an  $k \times n$  matrix  $\mathbf{V}_k^T$ . Truncated SVDs have several different uses—reducing the memory needed to store a matrix, making computations faster, simplifying problems by projecting the data to lower-dimensional subspaces, and regularizing problems by reducing a matrix's condition number.

• 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 a economy SVD. The function `svds(A,k)` returns an object containing the first  $k$  (default is 6) singular values and associated singular vectors.

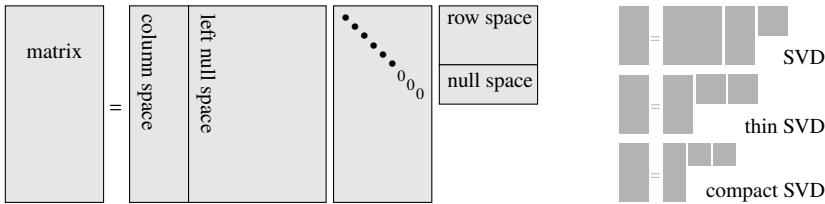


Figure 3.3: The SVD breaks a matrix into the orthonormal bases of its four fundamental subspaces—its column space and left null space along with its row space and null space—coupled through a diagonal matrix of singular values.

### ► Pseudoinverse

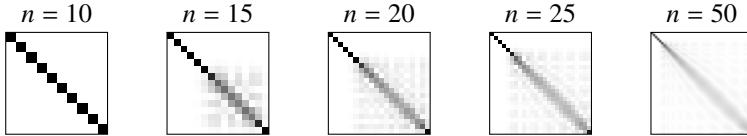
Let  $\mathbf{A}$  be an  $m \times n$  matrix with the singular value decomposition  $\mathbf{U}\Sigma\mathbf{V}^T$ . When  $\mathbf{A}$  is nonsingular, its inverse  $\mathbf{A}^{-1} = \mathbf{V}\Sigma^{-1}\mathbf{U}^T$ . The matrix  $\Sigma^{-1}$  is a diagonal matrix of the reciprocals of the singular values. The *Moore–Penrose pseudoinverse* of  $\mathbf{A}$  is an  $n \times m$  matrix defined as  $\mathbf{A}^+ = \mathbf{V}\Sigma^+\mathbf{U}^T$ . The matrix  $\Sigma^+$  is a diagonal matrix of the reciprocals of the nonzero elements of  $\Sigma$ , leaving the zero elements unchanged. The singular vectors corresponding to the zero singular values are in the null space and left null space of  $\mathbf{A}$ . By fixing the zero elements of  $\Sigma$ , we are choosing the zero vectors from the null space and the left null space of  $\mathbf{A}$ , which ensures that a pseudoinverse exists and that it is unique. 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 $\mathbf{A}^+ = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$
$n > m = r$	rowspace projection of $\mathbf{b}$ and $\mathbf{A}^+ = \mathbf{A}(\mathbf{AA}^T)^{-1}$
$n \geq m > r$	$\arg \min_{\mathbf{x}} \ \mathbf{b} - \mathbf{Ax}\ _2$ using the rowspace projection of $\mathbf{b}$

We also can use the pseudoinverse to regularize an ill-conditioned problem. The 2-condition number of a matrix is given by the ratio of the largest to smallest singular values to  $\sigma_1/\sigma_r$ . By zeroing out the singular values below a threshold  $\alpha$ , we lower the condition number to under  $\sigma_1/\alpha$  and solve the problem in a lower-dimensional subspace.

**Example.** Consider the  $n \times n$  Hilbert matrix, whose  $ij$ -element equals  $(i+j-1)^{-1}$ . We examined this matrix and its inverse using Gaussian elimination 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  $n = 0, 15, 20, 25$ , and 50

are below. Zero values are white, values with magnitude one or greater are black, and intermediate values are shades of gray.



The pseudoinverse has noticeable error even for small matrices. But unlike Gaussian elimination, which falls flat on its face, the pseudoinverse solution fails with grace. The pseudoinverse truncates all singular values less than  $10^{-15}$  which happens near  $\sigma_8$ , 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 with a default tolerance of `minimum(size(A))*eps()`. A tolerance of `sqrt(eps())` is recommended for least squares problems.

## ► Tikhonov regularization

Tikhonov regularization introduced the previous section and the pseudoinverse discussed above each provide ways to solve underdetermined problems. The two procedures are 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  $\Sigma_\alpha^{-1}$  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. The truncated SVD filter

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

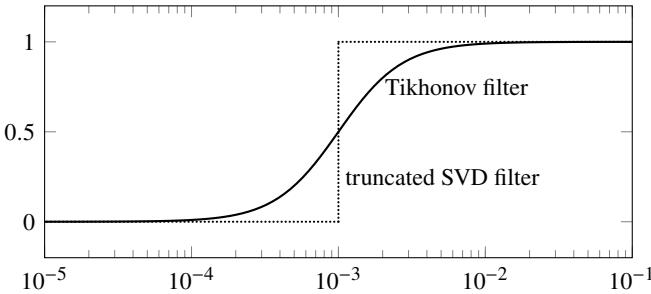


Figure 3.4: Comparison of the Tikhonov filter for  $\alpha = 10^{-3}$  and the truncated SVD filter with singular values zeroed out below  $10^{-3}$ .

exhibits a sharp cut-off at  $\alpha$ . See Figure 3.4. 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. We can write an  $m \times n$  matrix  $\mathbf{A}$  as the sum of rank-one, mutually orthogonal *principal components*  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top = \mathbf{E}_1 + \mathbf{E}_2 + \dots + \mathbf{E}_r$ , where  $\mathbf{E}_i = \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$  and  $r$  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  $\mathbf{A} = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$ . 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

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

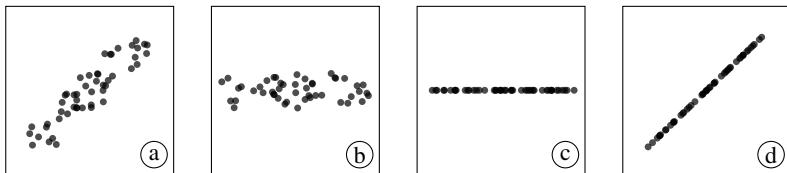
Now, consider the Frobenius norm. Let a subscript  $i$  denote an approximation using the first  $i$  principal components.

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

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,

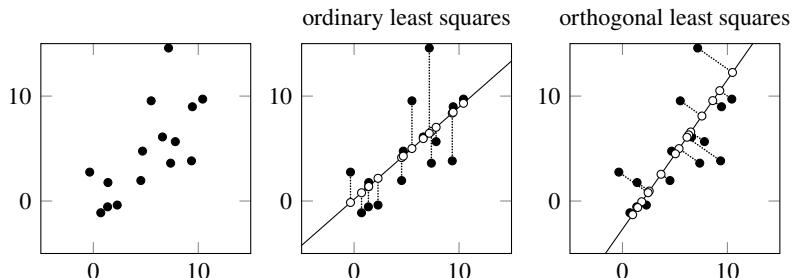
$$\|\mathbf{A} - \mathbf{B}\|_F^2 = \sum_{i=1}^r \sigma_i(\mathbf{A} - \mathbf{B})^2 \geq \sum_{i=k+1}^r \sigma_i(\mathbf{A})^2 = \|\mathbf{A} - \mathbf{A}_k\|_F^2. \quad \square$$

Suppose that we measure the heights  $\mathbf{h}$  and weights  $\mathbf{w}$  of 50 people. Let  $\mathbf{A}$  be the  $50 \times 2$  standardized data matrix, whose first column is the mean-centered heights  $\mathbf{h} - \bar{\mathbf{h}}$  and whose second column is the mean-centered weights  $\mathbf{w} - \bar{\mathbf{w}}$ . The matrix  $\mathbf{A}^\top \mathbf{A}$  is the covariance matrix. If height is linearly correlated to weight, a variable  $\sigma_1 \mathbf{u}_1$  in the direction of the first left singular vector incorporates everyone's heights and weights. The vector  $\mathbf{u}_1$  is a point on a 50-dimensional unit sphere, and every person gets their own axis. A second variable  $\sigma_2 \mathbf{u}_2$  in the direction of the second left singular vector accounts for the variation in the heights and weights. The vector  $\mathbf{u}_2$  is also on the unit sphere, orthogonal to  $\mathbf{u}_1$ . A rank-one approximation  $\mathbf{A}_1 = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^\top$  eliminates the contribution from that second variable. Consider the following figures:



(a) The data in  $\mathbf{A}$  lie for the most part along a direction  $\mathbf{v}_1$  with a small orthogonal component in direction  $\mathbf{v}_2$ . (b) The matrix  $\mathbf{AV}$  is 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 second principal component—the offsets in the y-direction. (d) After rotating back to our original basis, all the data lie along the  $\mathbf{v}_1$ -direction.

Unlike least squares fit, which would project the data “vertically” onto the height-weight line, principal component analysis projects the data orthogonally onto the height-weight line. Consider the scatter plots below.



The original data  $\bullet$  are 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  $15 \times 2$  Vandermonde matrix. Principal component analysis is an orthogonal projection into the first right 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 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 the norm of the residual the 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. In two dimensions, total least squares regression is called Deming regression or orthogonal regression. Let's examine the general case and 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.

We can write the total least squares problem  $(\mathbf{A} + \delta\mathbf{A})\mathbf{X} = \mathbf{B} + \delta\mathbf{B}$  using block matrices

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

We want to find

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

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

$$[\mathbf{A} \quad \mathbf{B}] = [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \boldsymbol{\Sigma}_1 & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma}_2 \end{bmatrix} \begin{bmatrix} \mathbf{V}_{11} & \mathbf{V}_{12} \\ \mathbf{V}_{21} & \mathbf{V}_{22} \end{bmatrix}^\top,$$

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

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

By linearity,

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

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

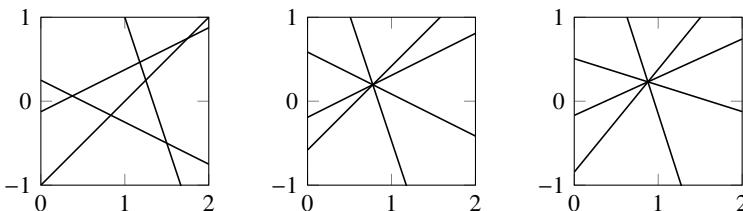
$$\mathbf{X} = -\mathbf{V}_{12} \mathbf{V}_{22}^{-1}.$$

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; xtls = 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.  $\blacktriangleleft$

### ▷ 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 significantly reduce 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.<sup>5</sup> On the other hand, singular value decomposition is slow, and even moderate compression can result in noticeable artifacts. Furthermore, singular value decomposition produces matrices containing positive and negative floating-point values that must be quantized. Storing these matrices may 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 format 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 produce  $2^{24}$  (or roughly 16 million) color variations.<sup>6</sup> 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

---

<sup>5</sup>Deflate 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.

<sup>6</sup>A 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 16 bits per channel.

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  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ :

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

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

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

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

```
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 often be much smaller than an SVD-compressed image, without any loss in quality.

## ► Image recognition

Image recognition algorithms have been around since the late 1950s, when psychologist Frank Rosenblatt built the Perceptron, a device with 400 photocells and a rudimentary mathematical model capable of classifying basic shapes. In 1987, mathematicians Lawrence Sirovich and Michael Kirby developed a methodology to characterize human faces. The methodology, initially called *eigenpictures*, was subsequently popularized as *eigenfaces* or *eigenimages*. Two years later, computer scientist Yann LeCun and fellow researchers developed a neural network image classifier called LeNet. We'll return to LeNet in Chapter 10. Here, we examine Sirovich and Kirby's method.

The eigenimage method applies singular value decomposition to training images to produce a smaller set of orthogonal images. 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



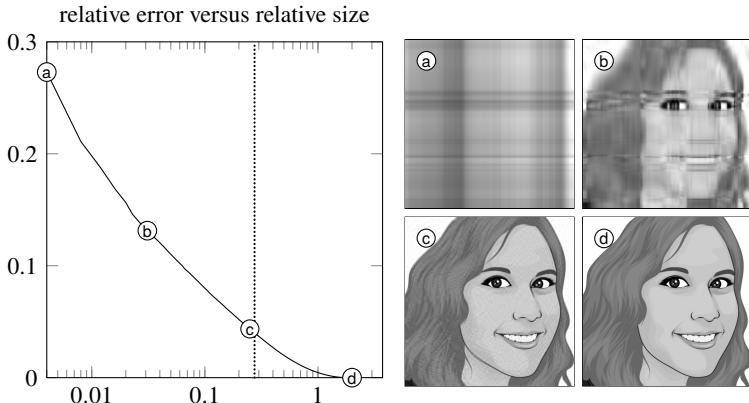


Figure 3.5: The relative error of an SVD-compressed image as a function of relative storage size. The dotted line shows equivalent lossless PNG compression.

standardize these vectors by subtracting the element-wise means of  $\{\mathbf{d}_j\}$  and dividing by the element-wise nonzero standard deviations of  $\{\mathbf{d}_j\}$  to reduce the condition number. Construct the matrix  $\mathbf{D} = [\mathbf{d}_1 \ \mathbf{d}_2 \ \dots \ \mathbf{d}_n]$ , which has the singular value decomposition  $\mathbf{U}\Sigma\mathbf{V}^\top$ .

Although we may include many training images, we can use a substantially lower-dimensional subspace by discarding the singular vectors corresponding to the smallest singular values. By keeping the largest  $k$  singular values and discarding the rest, we create a low-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 save the  $k \times n$  signature matrix  $\mathbf{W}_k$  and the  $pq \times k$  eigenimage matrix  $\mathbf{U}_k$ .

Figure 3.6 on page 82 shows a set of 26 training images—each image  $\mathbf{d}_i$  is a 120-by-120 pixel rasterized Times Roman capital letter. These images span a 26-dimensional subspace of a 14400-dimensional space. The matrix  $\mathbf{D}$  has the singular value decomposition into a sum of its principal components

$$\mathbf{D} = \mathbf{E}_1 + \mathbf{E}_2 + \dots + \mathbf{E}_{26} = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^\top + \sigma_2 \mathbf{u}_2 \mathbf{v}_2^\top + \dots + \sigma_{26} \mathbf{u}_{26} \mathbf{v}_{26}^\top.$$

The eigenimages  $\mathbf{u}_i$  and the signatures  $\sigma_i \mathbf{v}_i^\top$  contain the information needed to reconstruct the training images.

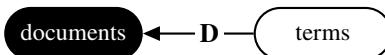
Take the 8-dimensional projection of  $\mathbf{D}$  using the first eight eigenimages and their respective signatures:  $\mathbf{D}_8 = \mathbf{E}_1 + \mathbf{E}_2 + \dots + \mathbf{E}_8$ . The projected images are

decent representations of the training images, although several—the C and G, the E and F, and the O and Q—are not well differentiated.

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 tie together underlying concepts. The lower-dimensional space of these variables is called the latent space. Figure 3.7 on page 83 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 contributions 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 close together in the latent space. The O is 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 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 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  $\mathbf{D} = [\mathbf{d}_1 \ \mathbf{d}_2 \ \dots \ \mathbf{d}_n]$  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  $\mathbf{q}$ . Using *cosine similarity*, we identify the document  $i$  that maximizes  $\cos \theta_i = \mathbf{q}^T \mathbf{d}_i / \|\mathbf{q}\| \|\mathbf{d}_i\|$ . Values close to one represent a good 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*, introduced by computer scientist Susan Dumais in the late-1980s. We can think of concepts as intermediaries between text and documents. Take the singular

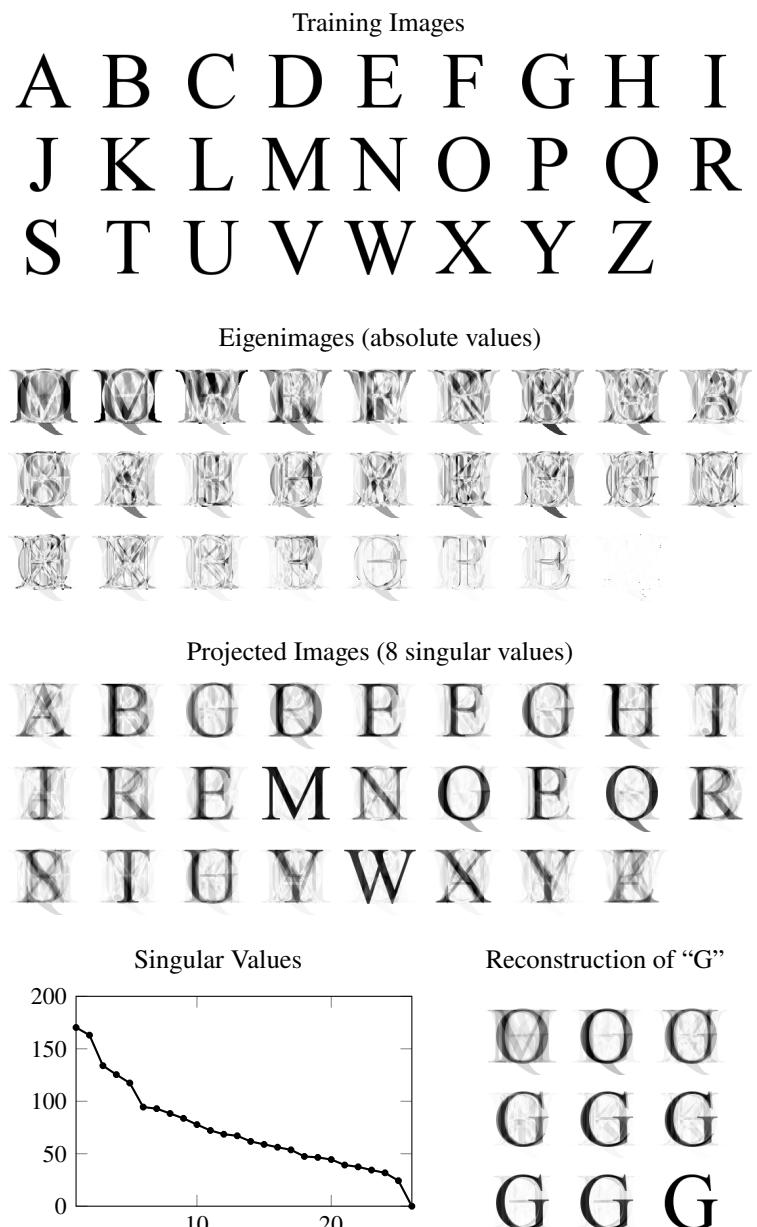


Figure 3.6: These  $120 \times 120$  pixel images span a 26-dimensional subspace of  $\mathbb{R}^{14400}$ . Eigenimages (ordered by their associated singular values) form an orthogonal basis for the subspace spanned by the training images. Also, see the QR code below.



alphabet eigenimages  
and low rank  
projections

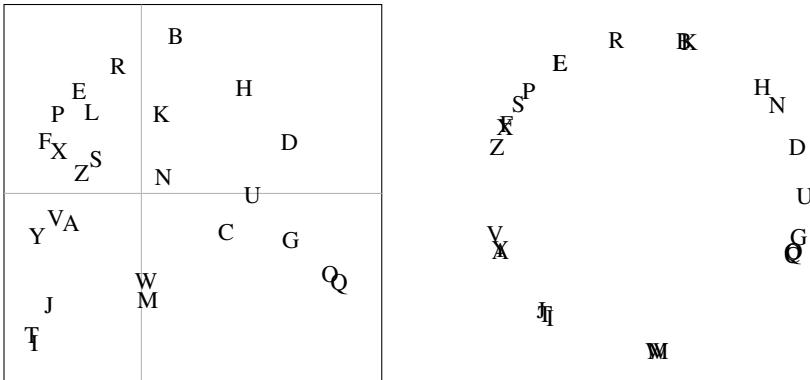
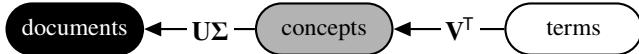


Figure 3.7: Left: The two-dimensional latent space of the letters with the  $v_1$  and  $v_2$ -axes of the right-singular vectors shown. Notice how letters with similar appearances are quite close together in the latent space such as the O and Q and the T, I, and J. Right: The directions of the same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

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 Nonnegative 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 is that the factors typically do not preserve sparsity, a possibly undesirable trait. It will be helpful to have a method of factorization that maintains nonnegativity and sparsity. Take an  $m \times n$



matrix  $\mathbf{X}$  whose elements are all nonnegative. Nonnegative matrix factorization (NMF) finds a rank  $p$  approximate matrix factorization  $\mathbf{WH}$  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 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 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 uses 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  $\textcircled{5}$  into  $\textcircled{6}$ , we have  $\mathbf{H} = \mathbf{H} \circ \mathbf{W}^T \mathbf{X} \oslash \mathbf{W}^T \mathbf{WH}$ . These two equations can be implemented as iterative multiplicative updates:

$$\begin{aligned} \mathbf{W} &\leftarrow \mathbf{W} \circ \mathbf{XH}^T \oslash \mathbf{WHH}^T \\ \mathbf{H} &\leftarrow \mathbf{H} \circ \mathbf{W}^T \mathbf{X} \oslash \mathbf{W}^T \mathbf{WH}. \end{aligned}$$

The method is relatively slow to converge and not guaranteed to converge, but its 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.

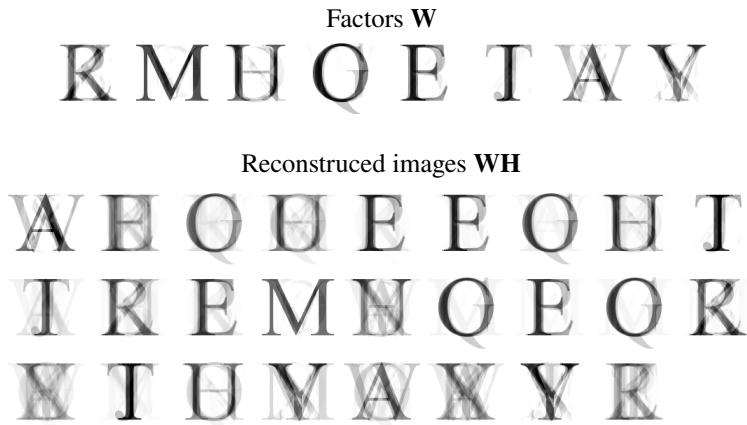


Figure 3.8: Nonnegative matrix factorization with rank  $p = 8$  was applied to the 26 training images in Figure 3.6 on page 82. The reconstructed images are combinations of the eight factors  $\mathbf{W}$  using positive weights  $\mathbf{H}$ .

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 nonnegative 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  $\infty$ -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

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

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:

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

3.4. Fill in the steps missing from the example on page 68. Hint: let  $\mathbf{Y} = \mathbf{A}\mathbf{X}\mathbf{B} + \mathbf{N}$ , where  $\mathbf{X}$  and  $\mathbf{Y}$  are image arrays,  $\mathbf{A}$  and  $\mathbf{B}$  are appropriately sized Gaussian blur (Toeplitz) matrices, and  $\mathbf{N}$  is a random matrix. The matrix  $\mathbf{A}$  acts on the columns of  $\mathbf{X}$ , and the matrix  $\mathbf{B}$  acts on the rows of  $\mathbf{X}$ . 

3.5. The Filippelli problem was contrived to benchmark statistical software packages. The National Institute of Standards and Technology (NIST) “Filip” Statistical Reference Dataset<sup>7</sup> 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  $\mathbf{V}_{ij} = x_i^{p-j}$  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 cut-off values for the pseudoinverse? What happens if you standardize the data first by subtracting the respective means and dividing

---

<sup>7</sup><http://www.itl.nist.gov/div898/strd/l1s/data/Filip.shtml>

them by their standard deviations? The Filip dataset and certified parameters are available as *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  $[1 \ x_i \ \cdots \ x_i^{P-1} \ x_i^P]$ .

3.6. The daily temperature of a city over several years can be modeled as a sinusoidal function. Use linear least squares to develop such a model using historical data recorded in Washington, DC, between 1967 to 1971, available as *dailytemps.csv* from <https://github.com/nmfsc/data>. Or choose your own data from <https://www.ncdc.noaa.gov>. 

3.7. 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  $28 \times 28$ -pixel grayscale image. For reference, the following 9s are sampled from the dataset:



Use the `MLDatasets.jl` package to load the MNIST dataset.<sup>8</sup> 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. 

3.8. Apply Zipf's law to the frequency of surnames in the United States. You can find a 2010 census data set as *surname.csv* from <https://github.com/nmfsc/data>.

3.9. A model is a simplification of the real world. Take Hollywood actors, for example. We can characterize actors by the roles they often play. Russell Crowe

---

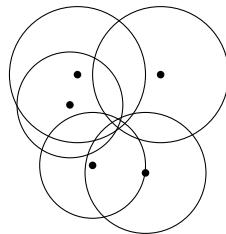
<sup>8</sup>Alternatively, you can download the training and test data as the MAT-file *emnist-mnist.mat* from <https://www.nist.gov/itl/products-and-services/emnist-dataset> or <https://github.com/nmfsc/data>.

is primarily thought of as an action film actor, Ben Stiller as a comedic actor, and Kate Winslet as a 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?



3.10. Multilateration determines an object's position by comparing the arrival times 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. The system will be inconsistent, because of variations in signal speed, the diffraction of waves around objects, measurement errors, etc. 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  $4 \times 4$ . The Abel–Ruffini theorem states that no general solutions exist to degree-five or higher polynomial equations using a finite number of algebraic operations (adding, subtracting, multiplying, dividing, and taking an integer or fractional power). Consequently, 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, this 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 discussing methods for determining eigenvalues, let's look at the estimation and stability of eigenvalues. Suppose that  $\mathbf{A}$  is nondefective, with the diagonalization  $\mathbf{A} = \mathbf{S}^{-1}\mathbf{AS}$ . Let  $\delta\mathbf{A}$  be 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, we have

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

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  $\lambda$  vary with some perturbation parameter and let  $\delta A$ ,  $\delta x$  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* as

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

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  $A = SAS^{-1}$ :

```
function condeig(A)
    Sr = eigvecs(A)
    S1 = inv(Sr')
    S1 ./= sqrt.(sum(abs.(S1.^2), dims=1))
    1 ./ abs.(sum(Sr.*S1, dims=1))
end
```

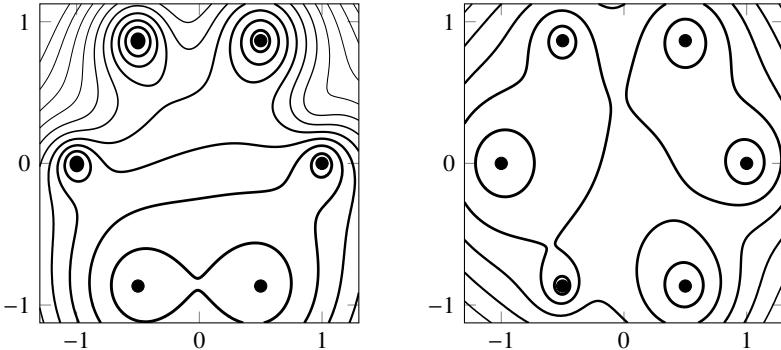


Figure 4.1: Contours of  $\varepsilon$ -pseudospectra of two similar matrices. Each contour represents a change of  $\varepsilon = 0.0005$ . For each matrix the 2-condition number is on the order of  $10^6$  and the condition number of the matrix of eigenvectors is on the order of  $10^3$ .

Finding the spectrum of a matrix  $\mathbf{A}$  is the same as asking which values  $\lambda$  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, i.e., when  $\|(\mathbf{A} - \lambda\mathbf{I})^{-1}\|$  is large. In this case, it is helpful to talk about the pseudospectrum of the matrix  $\mathbf{A}$ .

We define the  $\varepsilon$ -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*

$$\lambda(\mathbf{A}) \subseteq \mathcal{S}_R = \bigcup_{i=1}^n \mathcal{R}_i \quad \text{where} \quad \mathcal{R}_i = \{z \in \mathbb{C} : |z - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ij}|\}.$$

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  $\mathbf{B}_\lambda$ , 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  $\infty$ -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,  $\lambda \in \mathcal{R}_i$ . □

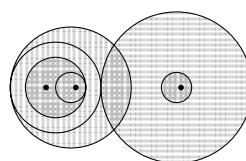
**Corollary 11.** Let  $\mathbf{A} \in \mathbb{C}^{n \times n}$ . Then  $\lambda(\mathbf{A}) \subseteq \mathcal{S}_{\mathcal{R}} \cap \mathcal{S}_{\mathcal{C}}$  where

$$\mathcal{S}_{\mathcal{C}} = \bigcup_{i=1}^n C_i \quad \text{where} \quad C_i = \{z \in \mathbb{C} : |z - a_{ii}| \leq \sum_{\substack{j=1 \\ j \neq i}}^n |a_{ji}|\}.$$

*Proof.* Substitute the 1-norm for  $\infty$ -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 1 is the largest eigenvalue of a stochastic matrix.  $\blacktriangleleft$

**Theorem 12** (Perron–Frobenius theorem). *A positive square matrix (a matrix with all positive elements) 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 198) 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  $\lambda$ . We can also consider a similar contraction mapping that generates a unique left eigenvector  $\mathbf{u}$  with strictly positive components and eigenvalue  $\lambda'$ . 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  $\mu$  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 nonnegative 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  $\lambda_1$  is the *dominant eigenvalue* of  $\mathbf{A}$ . An eigenvector associated with  $\lambda_1$  is a dominant eigenvector. 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

$$\begin{aligned}\mathbf{x}^{(1)} &= \mathbf{A}\mathbf{x}^{(0)} = \alpha_1 \mathbf{A}\mathbf{x}_1 + \alpha_2 \mathbf{A}\mathbf{x}_2 + \cdots + \alpha_n \mathbf{A}\mathbf{x}_n \\ &= \alpha_1 \lambda_1 \mathbf{x}_1 + \alpha_2 \lambda_2 \mathbf{x}_2 + \cdots + \alpha_n \lambda_n \mathbf{x}_n.\end{aligned}$$

After  $k$  iterations,

$$\begin{aligned}\mathbf{x}^{(k)} &= \mathbf{A}^k \mathbf{x}^{(0)} = \alpha_1 \lambda_1^k \mathbf{x}_1 + \alpha_2 \lambda_2^k \mathbf{x}_2 + \cdots + \alpha_n \lambda_n^k \mathbf{x}_n \\ &= \alpha_1 \lambda_1^k \left( \mathbf{x}_1 + \frac{\alpha_2}{\alpha_1} \left( \frac{\lambda_2}{\lambda_1} \right)^k \mathbf{x}_2 + \cdots + \frac{\alpha_n}{\alpha_1} \left( \frac{\lambda_n}{\lambda_1} \right)^k \mathbf{x}_n \right) \\ &= \lambda_1^k \alpha_1 \mathbf{x}_1 + O\left(|\lambda_2/\lambda_1|^k\right) \text{ terms.}\end{aligned}$$

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

$$\left\| \mathbf{x}^{(k)} - \mathbf{x}_1 \right\| = \left\| \sum_{i=2}^n \frac{\alpha_i}{\alpha_1} \left( \frac{\lambda_i}{\lambda_1} \right)^k \mathbf{x}_i \right\| \leq \left| \frac{\lambda_2}{\lambda_1} \right|^k \left\| \sum_{i=2}^n \frac{\alpha_i}{\alpha_1} \mathbf{x}_i \right\| \leq C \left| \frac{\lambda_2}{\lambda_1} \right|^k.$$

For the Euclidean norm,  $C = \alpha_1^{-1} \sqrt{\alpha_2^2 + \alpha_3^2 + \cdots + \alpha_n^2}$ .

In practice, the power method is straightforward. We multiply the current eigenvector approximation by our matrix and then normalize to prevent over or underflow at each iteration. Starting with an initial guess  $\mathbf{x}^{(0)}$ ,

$$\begin{cases} \text{multiply} & \mathbf{x}^{(k+1)} = \mathbf{A}\mathbf{x}^{(k)} \\ \text{normalize} & \mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k+1)} / \|\mathbf{x}^{(k+1)}\|_2 \end{cases}.$$

At each step, the approximate dominant eigenvalue is  $\mathbf{x}^{(k)\top} \mathbf{A} \mathbf{x}^{(k)}$ . We stop when the difference in iterates  $\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|$  or the angle between iterates  $1 - \mathbf{x}^{(k+1)\top} \mathbf{x}^{(k)}$  is within a prescribed tolerance.

**Example.** The results of round-robin chess tournaments are often recorded on square matrices with 1 for a victory,  $\frac{1}{2}$  for a draw, and 0 for a defeat. A

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.*)

player’s tournament score is given by the sum of the scores along the rows. 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 a weak one. 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

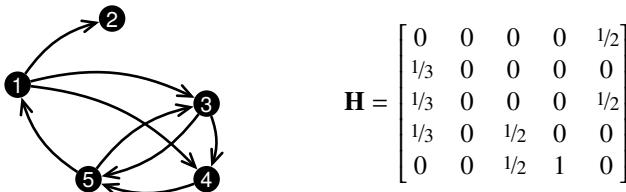
$$1 \cdot 5\frac{1}{2} + \frac{1}{2} \cdot 5 + \frac{1}{2} \cdot 4\frac{1}{2} + \frac{1}{2} \cdot 2\frac{1}{2} + 1 \cdot 2 + 1 \cdot 1\frac{1}{2} + 1 \cdot 1\frac{1}{2} = 16\frac{1}{2},$$

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 an iterative refinement of 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 a proportional sum of the rankings of those players defeated by that player. Of course, this is simply an eigenvector problem.<sup>1</sup> 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 that guaranteed the unique existence of a solution

---

<sup>1</sup>The 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.”

Figure 4.3: A graph and its normalized hyperlink matrix  $\mathbf{H}$ .

*Centrality* is a measure of influence in a network.<sup>2</sup> It could measure the strength of chess masters or the ranking of football teams, it could measure peer status among kindergarteners or trust among colleagues, and it could measure the 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.<sup>3</sup>

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 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  $\mathbf{S} = \mathbf{H} + \mathbf{e}\mathbf{v}^T$  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 they are 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  $\alpha$  is the likelihood of following any link on the

<sup>2</sup>The term *prestige* is sometimes used for directed networks, with *measure of influence* for outgoing edges and *measure of support* for incoming edges.

<sup>3</sup>Katz 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.

page and  $\mathbf{E}$  is an  $n \times n$  matrix whose elements all equal  $1/n$ . A typical value for the damping factor  $\alpha$  is 0.85. The matrix  $\mathbf{G}$  is often called the Google matrix.

A nonnegative 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 the  $i$ 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 the 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

$$\mathbf{x}^{(k+1)} \leftarrow \alpha \mathbf{Hx}^{(k)} + \alpha \mathbf{v}^\top \mathbf{x}^{(k)} \mathbf{e} + (1 - \alpha) \mathbf{e}. \quad (4.1)$$

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
```

The full matrix form is fine for smallish matrices like this one, but for larger matrices we should use the sparse form. After a few iterations, the solution  $x$  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

$$\lambda(\mathbf{A}) = \{\lambda_1, \lambda_2, \dots, \lambda_{n-1}, \lambda_n\}$$

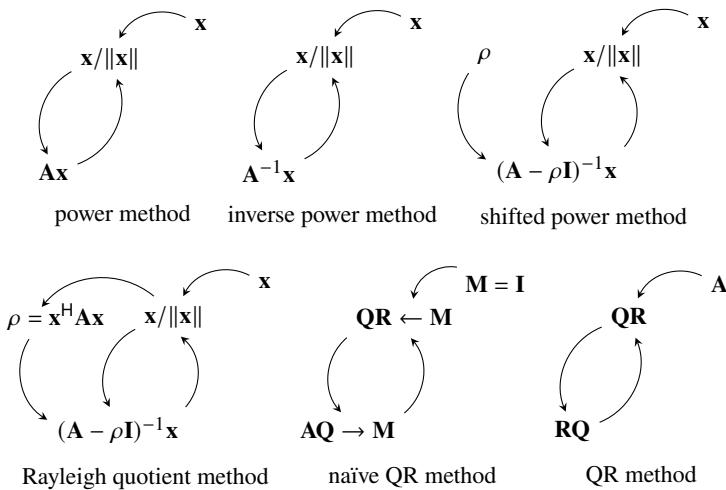


Figure 4.4: Power method and some of its variations.

where  $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_{n-1}| > |\lambda_n|$ . Then, the spectrum of  $\mathbf{A}^{-1}$  is

$$\lambda(\mathbf{A}^{-1}) = \left\{ \frac{1}{\lambda_1}, \frac{1}{\lambda_2}, \dots, \frac{1}{\lambda_{n-1}}, \frac{1}{\lambda_n} \right\}$$

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 for any value  $\rho$ , 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 as  $\mathbf{A}$ . We can use this property in a clever approach called *shift-and-invert*. We make a good guess of one of the eigenvalues, say  $\rho \approx \lambda_i$  for some  $i$ . Then, we apply power iteration to  $(\mathbf{A} - \rho \mathbf{I})^{-1}$ . We will recover the eigenvector associated with the eigenvalue  $\lambda_i$ . The ratio of convergence is  $|\lambda_i - \rho|/|\lambda_k - \rho|$  where  $\lambda_k$  is the next closest eigenvalue to  $\rho$ . In practice, for each iteration, we solve

$$(\mathbf{A} - \rho \mathbf{I}) \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}, \quad \text{where} \quad \mathbf{x}^{(k)} \leftarrow \frac{\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|}.$$

Computation requires  $\frac{2}{3}n^3$  flops for LU-decomposition and then only  $2n^2$  flops for each iteration

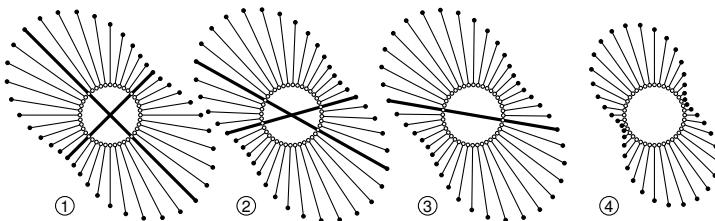


Figure 4.5: Rayleigh quotient  $\rho_A(\mathbf{x})\mathbf{x}$  for the matrices in Figure 1.1. Eigenvectors are depicted with thick line segments.

### ► Rayleigh quotient iteration

The convergence rate for the shift-and-invert method depends on having a good initial guess for the shift  $\rho$ . It seems reasonable that we could improve the method by updating the shift  $\rho$  at each iteration. One way we can do this is by using the eigenvalue equation  $\mathbf{Ax} = \rho\mathbf{x}$ . Unless  $(\rho, \mathbf{x})$  is already an eigenvalue-eigenvector pair of  $\mathbf{A}$ , this equation is inconsistent. Instead, we will find the “best”  $\rho$ . Let’s determine which  $\rho$  minimizes the 2-norm of the residual

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

At the minimum,

$$0 = \frac{d}{d\rho} \|\mathbf{r}\|_2^2 = \frac{d}{d\rho} \|\mathbf{Ax} - \rho\mathbf{x}\|_2^2 = -2\mathbf{x}^H \mathbf{Ax} + 2\rho\mathbf{x}^H \mathbf{x}.$$

Therefore, the “best” choice for  $\rho$  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, establishes bounds on the Rayleigh quotient.

**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  $\mathbf{A} = \mathbf{Q}^H \boldsymbol{\Lambda} \mathbf{Q}$  where  $\mathbf{Q}$  is an orthogonal matrix. Because  $\mathbf{x}^H \mathbf{Ax} = (\mathbf{Q}\mathbf{x})^H \boldsymbol{\Lambda} (\mathbf{Q}\mathbf{x})$  and  $\|\mathbf{Q}\mathbf{x}\|_2 = \|\mathbf{x}\|_2$ , it is sufficient to consider the case of  $\mathbf{A} = \boldsymbol{\Lambda}$ , i.e.,  $\mathbf{A}$  as a diagonal matrix. In this case,

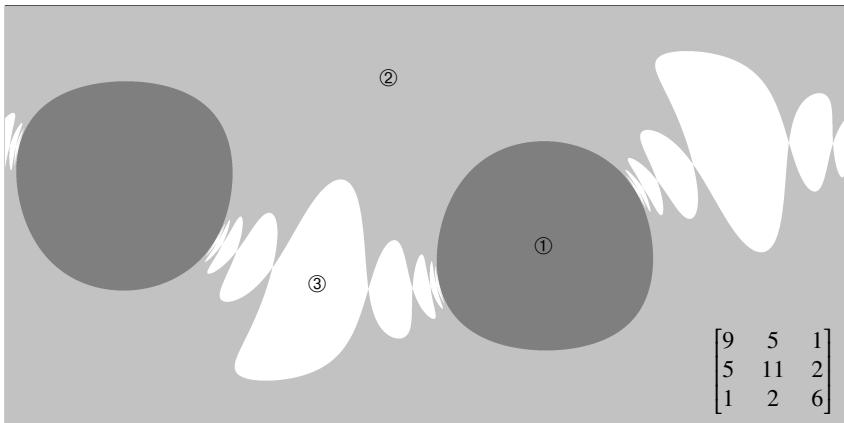


Figure 4.6: Basins of attraction for Rayleigh iteration in the  $(\varphi, \theta)$ -plane. Initial guesses in ■ regions converge to the eigenvector at ①, guesses in ▨ regions converge to the eigenvector at ②, and guesses in □ regions converge to the eigenvector at ③. See the front cover and the QR code at the bottom of this page.

$\mathbf{x}^H \mathbf{A} \mathbf{x} = \sum_{i=1}^n \lambda_i x_i^2$  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

$$\mathbf{x}^H \mathbf{A} \mathbf{x} = \sum_{i=1}^n \lambda_i x_i^2 = \sum_{i=1}^k \lambda_i x_i^2 \geq \lambda_k \sum_{i=1}^k x_i^2 = \lambda_k \|\mathbf{x}\|_2^2.$$

And, if  $\mathbf{x}$  happens to be the eigenvector  $\xi_k$ , 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  $\lambda_1$ .

$$\mathbf{x}^H \mathbf{A} \mathbf{x} = \sum_{i=1}^n \lambda_i x_i^2 \leq \lambda_1 \sum_{i=1}^n x_i^2 = \lambda_1 \|\mathbf{x}\|_2^2$$

and  $\rho_A(\mathbf{x}) = \lambda_1$  for  $\mathbf{x} = \xi_1$ . So,  $\lambda_1 = \max_{\mathbf{x}} \rho_A(\mathbf{x})$ . □

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 the best approximation to the eigenvalue, and it 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.



basins of attraction  
painted onto the  
Rayleigh quotient

The following method combines Rayleigh quotient approximation with the shift-and-invert method, starting with a unit vector  $\mathbf{x}^{(0)}$ :

$$\begin{cases} \text{calculate} & \rho^{(k)} = \mathbf{x}^{(k)\top} \mathbf{A} \mathbf{x}^{(k)} \\ \text{solve} & (\mathbf{A} - \rho^{(k)} \mathbf{I}) \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} \\ \text{normalize} & \mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k+1)} / \|\mathbf{x}^{(k+1)}\|_2 \end{cases}.$$

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.<sup>4</sup> (Watkins [2011]) The QR method extends 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.

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

---

<sup>4</sup>Remarkably, 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 southeast 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.

$\mathbf{x}_1^{(k+1)}$  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 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  $\mathbf{A}$ :  $\mathbf{M}_{k+1} = \mathbf{A}\mathbf{Q}_k$ .
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

$$\mathbf{Q}_{k+1}\mathbf{R}_{k+1} = \mathbf{M}_{k+1} = \mathbf{A}\mathbf{Q}_k. \quad (4.2)$$

It's not yet clear what this equation means. Let's change the basis to  $\mathbf{Q}_k$ . We simply need to "rotate" by  $\mathbf{Q}_k^{-1} = \mathbf{Q}_k^T$  to do so. Multiplying (4.2) by  $\mathbf{Q}_k^T$  yields

$$\mathbf{Q}_k^T \mathbf{Q}_{k+1} \mathbf{R}_{k+1} = \mathbf{Q}_k^T \mathbf{M}_{k+1} = \mathbf{Q}_k^T \mathbf{A} \mathbf{Q}_k. \quad (4.3)$$

This equation says that  $\mathbf{Q}_k^T \mathbf{Q}_{k+1} \mathbf{R}_{k+1}$  is unitarily similar to  $\mathbf{A}$ . So  $\mathbf{Q}_k^T \mathbf{Q}_{k+1} \mathbf{R}_{k+1}$  has the same eigenvalues as  $\mathbf{A}$ . If the iteration converges, then  $\mathbf{Q}_k \rightarrow \mathbf{Q}_{k+1}$  and  $\mathbf{Q}_k^T \mathbf{Q}_{k+1} \rightarrow \mathbf{I}$ , 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

$$\hat{\mathbf{Q}}_{k+1}\mathbf{R}_{k+1} = \hat{\mathbf{M}}_{k+1} = \mathbf{Q}_k^T \mathbf{A} \mathbf{Q}_k$$

where  $\hat{\mathbf{M}}_{k+1} = \mathbf{Q}_k^T \mathbf{M}_{k+1}$ . From (4.3) we have  $\mathbf{R}_{k+1}\mathbf{Q}_k^T = \mathbf{Q}_{k+1}^T \mathbf{A}$ , and hence  $\mathbf{R}_k \mathbf{Q}_{k-1}^T = \mathbf{Q}_k^T \mathbf{A}$ . So,

$$\hat{\mathbf{Q}}_{k+1}\mathbf{R}_{k+1} = \hat{\mathbf{M}}_{k+1} = \mathbf{R}_k \mathbf{Q}_{k-1}^T \mathbf{Q}_k = \mathbf{R}_k \hat{\mathbf{Q}}_k$$

We are left with the simple implementation. Starting 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 the Schur decomposition of a matrix

$$\mathbf{R} = \mathbf{U}^H \mathbf{A} \mathbf{U}$$

where  $\mathbf{U}$  is unitary. At each iteration

$$\mathbf{T}_{k+1} = \mathbf{R}_k \mathbf{Q}_k = \mathbf{Q}_k^H \mathbf{Q}_k \mathbf{R}_k \mathbf{Q}_k = \mathbf{Q}_k^H \mathbf{T}_k \mathbf{Q}_k.$$

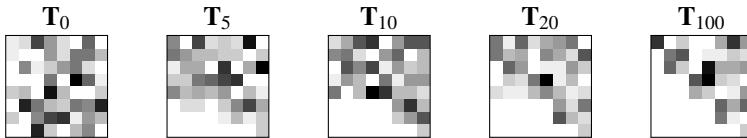
If we continue and unravel the terms all the of the terms,

$$\mathbf{T}_{k+1} = (\mathbf{Q}_0 \mathbf{Q}_1 \cdots \mathbf{Q}_k)^H \mathbf{T}_0 (\mathbf{Q}_0 \mathbf{Q}_1 \cdots \mathbf{Q}_k).$$

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 making a few simple modifications: 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 generally impossible to determine the Schur decomposition of a matrix  $\mathbf{A}$ , i.e., an upper triangular matrix 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. Each QR step takes  $O(n)$  operations for a Hermitian matrix because we are working on a tridiagonal matrix.

We can use Householder reflectors in a manner similar to QR decomposition to get an upper Hessenberg matrix. We first construct a Householder reflector  $\mathbf{Q}_1$  to zero out the elements below the first subdiagonal element. To do this,

$$\text{partition } \mathbf{A} \text{ as } \begin{bmatrix} a_{11} & \mathbf{c}^\top \\ \mathbf{b} & \hat{\mathbf{A}} \end{bmatrix} \text{ and let } \mathbf{Q}_1 \text{ equal } \begin{bmatrix} \mathbf{I} & \mathbf{0}^\top \\ \mathbf{0} & \hat{\mathbf{Q}}_1 \end{bmatrix}$$

where  $\hat{\mathbf{Q}}_1$  is the reflector that maps a vector  $\mathbf{b}$  to  $[-\|\mathbf{b}\|_2 \ 0 \ \cdots \ 0]^\top$ . Because  $\hat{\mathbf{Q}}_1$  is a Householder reflector,  $\hat{\mathbf{Q}}_1 = \hat{\mathbf{Q}}_1^{-1} = \hat{\mathbf{Q}}_1^\top$ . Note that  $\mathbf{Q}_1$  is a block matrix with the identity matrix in the upper-left block. 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

$$\begin{bmatrix} 1 & & \\ & \ddots & \\ & & \hat{\mathbf{Q}}_1 \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} 1 & & \\ & \ddots & \\ & & \hat{\mathbf{Q}}_1 \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} 1 & & \\ & \ddots & \\ & & \hat{\mathbf{Q}}_1 \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix}.$$

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

$$\begin{bmatrix} 1 & 1 & & \\ & \ddots & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \ddots \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ & \ddots & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \ddots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ & \ddots & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \ddots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix}.$$

Continuing this way, we will form an upper Hessenberg matrix unitarily similar to  $\mathbf{A}$ . Since it is similar, it has the same eigenvalues. If  $\mathbf{A}$  is a Hermitian matrix, the upper Hessenberg matrix is a tridiagonal matrix.

• The `LinearAlgebra.jl` function `hessenberg(A)` computes the unitarily similar upper Hessenberg form of a matrix  $\mathbf{A}$  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  $\mathbf{A}$  are  $\{\lambda_1, \lambda_2, \dots, \lambda_n\}$ , then the eigenvalues of  $\mathbf{A} - \rho\mathbf{I}$  are  $\{\lambda_1 - \rho, \lambda_2 - \rho, \dots, \lambda_n - \rho\}$ . The diagonal elements of  $\mathbf{A}_k$  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  $\mathbf{A} - \rho\mathbf{I}$  and choosing  $\rho$  to be close to an eigenvalue. This is the same idea that we used to pick the Rayleigh iteration shift. Since the diagonal elements of  $\mathbf{A}_k$  converge to the eigenvalues of  $\mathbf{A}$ , they are the natural choices for  $\rho$ . 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,

$$\begin{cases} \text{set} & \rho = a_{ii}^{(k)} \\ \text{factor} & (\mathbf{A}_k - \rho\mathbf{I}) \rightarrow \mathbf{Q}_{k+1}\mathbf{R}_{k+1} \\ \text{restor} & \mathbf{A}_{k+1} \leftarrow (\mathbf{R}_{k+1}\mathbf{Q}_{k+1} + \rho\mathbf{I}) \end{cases}.$$

Once  $a_{ii}^{(k)}$  has converged to an eigenvalue, move to the previous diagonal element ( $i \rightarrow i - 1$ ) and repeat the process. Note that

$$\mathbf{A}_{k+1} = \mathbf{R}_{k+1}\mathbf{Q}_{k+1} + \rho\mathbf{I} = \mathbf{Q}_{k+1}^\top(\mathbf{A}_k - \rho\mathbf{I})\mathbf{Q}_{k+1} + \rho\mathbf{I} = \mathbf{Q}_{k+1}^\top\mathbf{A}_k\mathbf{Q}_{k+1}.$$

So, the new  $\mathbf{A}_{k+1}$  has the same eigenvalues as the old  $\mathbf{A}_k$ .

The convergence with each shift is  $O(|\lambda_i - \rho|/|\lambda_{i-1} - \rho|)$  where  $\rho \approx \lambda_i$ . Overall, we get quadratic convergence. We get cubic convergence for Hermitian matrices, where the eigenvectors are orthogonal. This type of shifting is called *Rayleigh quotient shifting*. See the figure on the following page or the QR code below, which shows the convergence of the QR method and convergence of the shifted QR method to eigenvalues of a  $6 \times 6$  complex matrix. The standard 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

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix},$$

comparison of QR and  
shifted QR methods



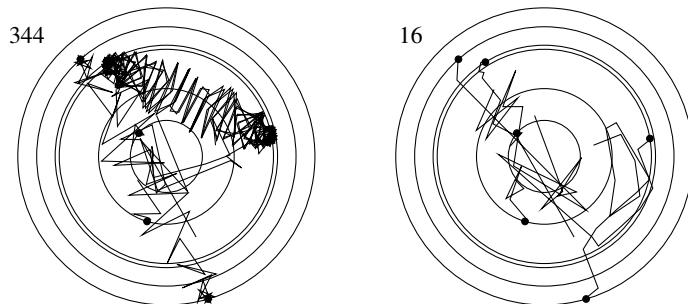


Figure 4.7: Convergence of the QR method (left) and the shifted QR method (right) to eigenvalues of a  $6 \times 6$  complex matrix. The eigenvectors are depicted by  $\bullet$  and line segments show the paths of convergence.

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  $\pm 1$ , each of equal magnitude. Each eigenvector pulls equally, and the algorithm cannot decide which eigenvalue to go to. One fix to this problem is to use the *Wilkinson shift*. The Wilkinson shift defines  $\rho$  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  $2 \times 2$  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  $\lambda_i$ . At this point set the shift  $\rho$  to equal  $a_{i-1,i-1}$  and recover  $\lambda_i$ . 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

$$(\mathbf{A} - \rho\mathbf{I}) = \mathbf{QR}, \quad \hat{\mathbf{A}} = \mathbf{R}\mathbf{Q} + \rho\mathbf{I} \quad (4.4)$$

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  $\rho$  is close to an eigenvalue (which is what we want it to be), the QR method is sensitive to round-off and 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  $\rho s$  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  $\mathbf{Q}_2^\top$  to zero out  $a_{32}$ . Exactly the same  $\mathbf{Q}_2^\top$  works for both  $\mathbf{A}$  and  $\mathbf{A} - \rho\mathbf{I}$ . The same is true for  $\mathbf{Q}_3^\top$  and so on.

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  $\mathbf{H} = \mathbf{Q}^\top \mathbf{A} \mathbf{Q}$  be the reduction of a matrix  $\mathbf{A}$  to Hessenberg form, and the elements in the lower diagonal of  $\mathbf{H}$  are nonzero. 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:

$$\mathbf{A} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}, \quad \text{and} \quad \mathbf{Q} = \begin{bmatrix} \bullet & \circ & \circ & \circ & \circ \\ \bullet & \bullet & \circ & \circ & \circ \\ \bullet & \bullet & \bullet & \circ & \circ \\ \bullet & \bullet & \bullet & \bullet & \circ \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}.$$

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  $\mathbf{Q}^\top$  that zeros out the  $(2, 1)$  element of  $\tilde{\mathbf{H}}$ .
3. Compute  $\tilde{\mathbf{H}} \leftarrow \mathbf{Q}^\top \tilde{\mathbf{H}} \mathbf{Q}$ .
4. Now, continue with 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  $\mathbf{A} \leftarrow \mathbf{Q}^\top \mathbf{H} \mathbf{Q}$ .

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{H} \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{Q}_1^T \mathbf{H} \mathbf{Q}_1 \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{Q}_2^T \cdots \mathbf{Q}_2 \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{Q}_3^T \cdots \mathbf{Q}_3 \end{bmatrix} \rightarrow \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \mathbf{Q}_4^T \cdots \mathbf{Q}_4 \end{bmatrix}$$

By this point, the QR method bears little resemblance to the underlying power method. But just as we could speed up the 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 we need to chase is size  $k$ .

Taking multiple steps is especially appealing when working with real matrices with 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, this approach is inefficient. Instead, computing the eigenvectors after finding the eigenvalues is typically a simple task. This section looks 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  $\rho$  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  $\mathbf{H}$ . Only one iteration is typically needed when  $\rho$  is a good approximation. The eigenvector to  $\mathbf{A}$  is then  $\mathbf{Q}^T \mathbf{q}$ .

We can use an *ultimate shift* strategy to get the eigenvectors using the Schur form  $\mathbf{Q}^T \mathbf{A} \mathbf{Q} = \mathbf{T}$  without accumulating the matrix  $\mathbf{Q}$ . 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,  $\mathbf{T}$  is diagonal, and  $\mathbf{Q}$  gives us the eigenvectors.

Let's look at the nonsymmetric case. Suppose that  $\lambda$  is the  $k$ th eigenvalue of the upper triangular matrix

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_{11} & \mathbf{v} & \mathbf{T}_{13} \\ \mathbf{0} & \lambda & \mathbf{w}^\top \\ \mathbf{0} & \mathbf{0} & \mathbf{T}_{33} \end{bmatrix},$$

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  $\lambda$  is a simple eigenvalue so that  $\lambda \notin \lambda(\mathbf{T}_{11}) \cup \lambda(\mathbf{T}_{33})$ . Then eigenvector problem  $(\mathbf{T} - \lambda \mathbf{I})\mathbf{y} = 0$  can be written as

$$\begin{aligned} (\mathbf{T}_{11} - \lambda \mathbf{I}_{k-1})\mathbf{y}_{k-1} + \mathbf{v}\mathbf{y} + \mathbf{T}_{13}\mathbf{y}_{n-k} &= \mathbf{0} \\ \mathbf{w}^\top \mathbf{y}_{n-k} &= 0, \\ (\mathbf{T}_{33} - \lambda \mathbf{I}_{n-k})\mathbf{y}_{n-k} &= \mathbf{0} \end{aligned}$$

where  $\mathbf{y} = [\mathbf{y}_{k-1} \quad y \quad \mathbf{y}_{n-k}]^\top$ . Because  $\lambda$  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  $\mathbf{y}_{k-1} = y(\mathbf{T}_{11} - \lambda \mathbf{I}_{k-1})^{-1}\mathbf{v}$ . Since  $y$  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. Using a subspace that is one-tenth the size of the original subspace, we'll get an algorithm 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  $\mathbf{R} = \mathbf{V}^\top \mathbf{A} \mathbf{V}$ , 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  $V$  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. The Arnoldi method combines the power method and the Gram–Schmidt process to get the first  $m$  eigenvectors approximations.

The power method preserves the sparsity of the matrix. Still, it only gives us one eigenvalue at a time because it throws away information about the other eigenvectors by only keeping the most recent vector. What if we were to keep a history of the vectors at each iteration? An initial guess  $\mathbf{q}$  can be expressed in terms of an eigenvector basis  $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ . Let's repeatedly multiply this initial guess by  $\mathbf{A}$ :

$$\begin{aligned}\mathbf{q} &= c_1 \mathbf{v}_1 + c_2 \mathbf{v}_2 + \cdots + c_n \mathbf{v}_n \\ \mathbf{A}\mathbf{q} &= c_1 \lambda_1 \mathbf{v}_1 + c_2 \lambda_2 \mathbf{v}_2 + \cdots + c_n \lambda_n \mathbf{v}_n \\ \mathbf{A}^2 \mathbf{q} &= c_1 \lambda_1^2 \mathbf{v}_1 + c_2 \lambda_2^2 \mathbf{v}_2 + \cdots + c_n \lambda_n^2 \mathbf{v}_n \\ &\vdots \\ \mathbf{A}^{k-1} \mathbf{q} &= c_1 \lambda_1^{k-1} \mathbf{v}_1 + c_2 \lambda_2^{k-1} \mathbf{v}_2 + \cdots + c_n \lambda_n^{k-1} \mathbf{v}_n,\end{aligned}$$

where  $k$  is bigger than  $m$  but much smaller than the  $n$ . 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

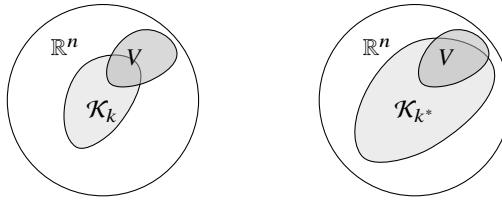
$$\text{span}\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \dots, \mathbf{v}_k\}.$$

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  $\mathbf{K}_k$ . This  $k$ -dimensional Krylov subspace is much smaller than the original  $n$ -dimensional subspace, so it is unlikely to 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  $k$ -dimensional Krylov subspace  $\mathcal{K}_k \subset \mathbb{R}^n$  does not contain the  $m$ -dimensional eigenspace  $V$ —only its projection into  $\mathcal{K}_k$ . If the starting vector  $\mathbf{q}$  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 any vector chosen at random 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  $\mathcal{K}_k(\mathbf{A}, \mathbf{q})$  is called the *Arnoldi method*. The Arnoldi method is similar to the Gram–Schmidt process, except we don't start with a set of  $m$  basis vectors already in hand. Instead, we only have  $\mathbf{q}$ .

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  $\mathbf{q}_2 \leftarrow \mathbf{q}_2 / \|\mathbf{q}_2\|$ . 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  $\mathbf{q}_{j+1} \leftarrow \mathbf{q}_{j+1} / h_{j+1,j}$  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  $\mathbf{Q}_k = [\mathbf{q}_1 \ \mathbf{q}_2 \ \cdots \ \mathbf{q}_k]$  and

$$\mathbf{H}_{k+1,k} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}$$

is a nonsquare upper Hessenberg matrix. Note we can complete the  $\mathbf{H}_{k+1,k}$  by adding a column  $\xi_1$  to get an upper triangular matrix

$$\mathbf{R} = \begin{bmatrix} 1 & & & \\ \bullet & \ddots & & \\ \bullet & \bullet & \ddots & \\ \bullet & \bullet & \bullet & \ddots \end{bmatrix}.$$

In this case,  $\mathbf{Q}_k \mathbf{R}$  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

$$\underbrace{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}}_{\mathbf{Q}_k} = \underbrace{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}}_{\mathbf{Q}_k} \underbrace{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}}_{\mathbf{H}_k} + \underbrace{\begin{bmatrix} \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \\ \bullet \end{bmatrix}}_{\mathbf{q}_{k+1} [0 \cdots 0 \ h_{k+1,k}]} \quad (4.6)$$

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  $(\mathbf{x}, \mu)$  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*. The Ritz estimate is small if  $(\mu, \mathbf{v})$  is a good approximation to an eigenpair of  $\mathbf{A}$ . See the figure on the next page and the QR code at the bottom of this page.

It is mathematically intuitive that the success of Arnoldi iteration depends on choosing a good starting vector  $\mathbf{q}$ . A vector chosen at random likely has significant components in all eigenvector directions, not just the first  $k$ . So the Krylov subspace is not a great match. In practice, how do we get a good starting vector  $\mathbf{q}$  for the Krylov subspace?

One solution is to run the Arnoldi method to get a good starting vector and 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, we run the Arnoldi method to get  $k$  Arnoldi vectors  $\mathbf{Q}_k$  and  $\mathbf{H}_k$ . Then, we 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.

Ritz values for  
increasing Krylov  
dimensions and  
eigenvalues



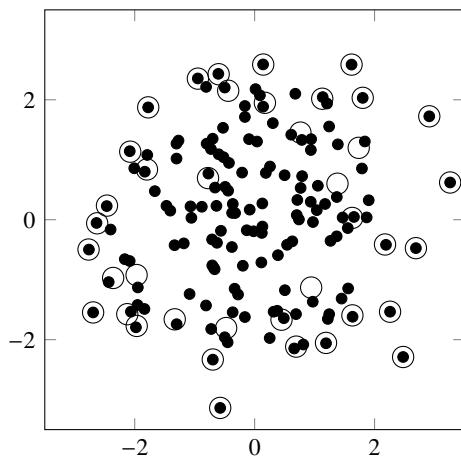


Figure 4.8: The location of the Ritz values  $\circ$  of  $\mathbf{K}_{40}(\mathbf{A}, \mathbf{q})$  and eigenvalues  $\bullet$  in the complex plane. The matrix  $\mathbf{A}$  is a  $144 \times 144$  complex-valued tridiagonal matrix and  $\mathbf{q}$  is a random initial guess. Notice that the Ritz values are closer to eigenvalues with 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 matrix is called the *Lanczos method*. Symmetric matrices have two clear advantages over nonsymmetric matrices. Their eigenvectors are orthogonal, so convergence is fast. And their upper Hessenberg form is tridiagonal, which is  $O(n)$  complex.

## 4.7 Singular value decomposition

Recall that the singular value decomposition of an  $m \times n$  matrix  $\mathbf{A}$  is  $\mathbf{U}\Sigma\mathbf{V}^T$  where  $\mathbf{U}$  is an  $m \times m$  unitary matrix of eigenvectors of  $\mathbf{A}\mathbf{A}^T$ ,  $\Sigma$  is an  $m \times n$  matrix of singular values  $\sigma(\mathbf{A}) = \sqrt{\mathbf{A}^T\mathbf{A}}$ , and  $\mathbf{V}$  is an  $n \times n$  unitary matrix of eigenvectors of  $\mathbf{A}^T\mathbf{A}$ . This section examines the Golub–Kahan–Reinisch algorithm for the SVD—there are several others. The algorithm consists of two steps: transforming the matrix to upper bidiagonal form and applying the QR algorithm.

*Step 1.* Compute

$$\tilde{\mathbf{U}}^T \mathbf{A} \tilde{\mathbf{V}} = \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix},$$

where  $\mathbf{B}$  is an upper bidiagonal matrix. We use a series of Householder reflections (or Givens rotations).

$$\begin{array}{c} \mathbf{A} \\ \left[ \begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right] \xrightarrow{\tilde{\mathbf{U}}_1^T} \left[ \begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right] \xrightarrow{\dots \tilde{\mathbf{V}}_1} \left[ \begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right] \\ \xrightarrow{\tilde{\mathbf{U}}_2^T} \left[ \begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right] \xrightarrow{\dots \tilde{\mathbf{V}}_2} \left[ \begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right] \\ \xrightarrow{\tilde{\mathbf{U}}_3^T} \left[ \begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right] \xrightarrow{\dots \tilde{\mathbf{V}}_3} \left[ \begin{array}{cccc} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{array} \right] \end{array}$$

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.* Use the QR method on  $\mathbf{B}$  to get the singular values  $\Sigma$ . One cycle of implicit QR on  $\mathbf{B}^T \mathbf{B}$  is

1. Take  $\rho = b_{n,n}^2 + b_{n-1,n}^2$  (Rayleigh shifting)

2. Compute  $\mathbf{A}\mathbf{Q}^T$  for Givens rotation

$$\mathbf{Q} : [b_{11}^2 - \rho \quad b_{11}b_{12}] \rightarrow [* \quad 0]$$

3. “Chase the bulge” using left and right Givens rotations

$$\mathbf{U} : \begin{bmatrix} b_{i,i} \\ b_{i+1,i} \end{bmatrix} \rightarrow \begin{bmatrix} * \\ 0 \end{bmatrix} \quad \mathbf{V} : [b_{i,i+1} \quad b_{i,i+2}] \rightarrow [* \quad 0].$$

4. Deflate when  $|b_{n-1,n}| < \varepsilon$  by using the upper left  $n - 1 \times n - 1$  matrix.

We can use the Lanczos method of finding eigenvalues to find the approximate singular values and singular vectors of a large, sparse matrix. 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 the eigenvectors of the symmetric block matrix

$$\underbrace{\begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{A}^T & \mathbf{0} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \lambda \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix},$$

where the eigenvalue  $\lambda$  is an eigenvalue of  $\mathbf{A}^T\mathbf{A}$ . To see this, note that

$$\left. \begin{array}{l} \mathbf{Av} = \lambda \mathbf{u} \\ \mathbf{A}^T \mathbf{u} = \lambda \mathbf{v} \end{array} \right\} \text{implies} \quad \left\{ \begin{array}{l} \mathbf{A}^T \mathbf{Av} = \lambda \mathbf{A}^T \mathbf{u} = \lambda^2 \mathbf{v} \\ \mathbf{AA}^T \mathbf{u} = \lambda \mathbf{Av} = \lambda^2 \mathbf{u} \end{array} \right..$$

The matrix  $\mathbf{M}$  is Hermitian, 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:  $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^T\mathbf{A}$ . 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  $\mathbf{Q}^T\mathbf{A}$  as usual. So,

$$\mathbf{A} \approx \mathbf{Q} (\mathbf{Q}^T \mathbf{A}) = \mathbf{Q} (\mathbf{W} \Sigma \mathbf{V}^T) = \mathbf{U} \Sigma \mathbf{V}^T$$

where  $\mathbf{U} = \mathbf{Q}\mathbf{W}$ . In practice, instead of computing the randomized SVD of  $\mathbf{A}$  we compute the randomized SVD of  $(\mathbf{A}\mathbf{A}^T)^r \mathbf{A}$ , 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  $\Omega$

$\mathbf{QR} \leftarrow \mathbf{A}\Omega$

for  $i = 1, \dots, r$

$$\left[ \begin{array}{l} \tilde{\mathbf{Q}}\tilde{\mathbf{R}} \leftarrow \mathbf{A}^T \mathbf{Q} \\ \mathbf{QR} \leftarrow \mathbf{A}\tilde{\mathbf{Q}} \end{array} \right]$$

Compute the SVD:  $\mathbf{W}\Sigma\mathbf{V}^T \leftarrow \mathbf{Q}^T\mathbf{A}$

$\mathbf{U} \leftarrow \mathbf{Q}\mathbf{W}$

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  $\lambda(\mathbf{A})$  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 magnitudes of all other elements in 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 get cubic 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  $\Lambda$  is a diagonal matrix of integers 1,2, ...,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 finds the actors at the center of Hollywood. Compute the eigenvector centrality of the actor-actor adjacency matrix. Explain in plain language what centrality means in the context of the six-degrees game. 

4.7. Implement the randomized SVD algorithm. Test your code using a grayscale image of your choosing. How does the randomized SVD compare with the full SVD in accuracy and performance? 

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  $\ell^2$ . 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 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  $1000 \times 1000$  matrix, Gaussian elimination takes less than a second on my exceptionally ordinary laptop. Now, suppose we want to solve a partial differential equation in three dimensions. We make a discrete approximation to the problem using 100 grid points in each dimension. Such a problem requires that we solve a system of a million equations. Now, Gaussian elimination would take over thirty years on my laptop. Fortunately, such a system is sparse—only 0.001 percent of the elements in the matrix are nonzero. Gaussian elimination is still fast if the matrix has a narrow bandwidth or can be reordered to have narrow bandwidth. But for a matrix with no redeeming features other than its sparsity, a better approach is an iterative method. Iterative methods use repeated matrix multiplications in the place of matrix inversion. Gaussian elimination takes  $O(n^3)$  operations. Matrix-vector multiplication takes only  $O(n^2)$  operations and as little as  $O(n)$  operations if the matrix 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 are other reasons to use an iterative method. While a direct method must always solve each problem from scratch with zero knowledge, we can give an iterative method a good initial guess. Take a time-dependent partial differential equation. At each new time step, the solution may not change much from the solution computed at the previous time step. Using a direct solver with Cholesky decomposition and Cuthill–McKee reordering to avoid fill in, we only need  $O(n^2)$  operations to compute the solution. On the other hand, an iterative method can use the solution from the previous step as a good initial guess for the new time step, and it automatically preserves sparsity.

Sometimes we may not need a solution as accurate as the one provided by a direct solver. For example, a finite difference approximation to a partial

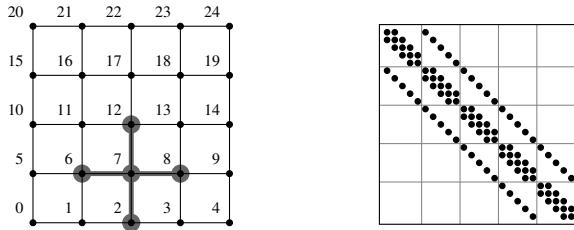


Figure 5.1: The finite difference stencil for the discrete two-dimensional Laplacian on a  $5 \times 5$  mesh. By labeling the points in lexicon fashion, we can construct a  $25 \times 25$  block tridiagonal matrix.

differential equation may already have a substantial 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 close-enough solution might be as good as finding a machine-precision solution, especially if the model contains other errors.

One more reason why iterative methods are helpful is that they reduce the error in the solution with each iteration. Using an iterative method as a corrector to a direct method can help us accurately solve ill-conditioned matrices.

**Example.** Consider the 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  $f(x, y) = 0$  on the unit square. The two-dimensional Poisson equation models several physical systems, such as the steady-state heat distribution  $u(x, y)$  given a source  $f(x, y)$  or the shape of an elastic membrane  $z = u(x, y)$  given a load  $f(x, y)$ .

The finite difference method is a simple yet effective numerical method for solving the Poisson equation. 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

$$\begin{aligned}\mathbf{x} &= [u_{11} \quad u_{21} \quad \cdots \quad u_{n1} \quad u_{12} \quad u_{22} \quad \cdots \quad u_{nn}]^\top \\ \mathbf{b} &= [f_{11} \quad f_{21} \quad \cdots \quad f_{n1} \quad f_{12} \quad f_{22} \quad \cdots \quad f_{nn}]^\top.\end{aligned}$$

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  $\mathbf{x}$ . 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 approach 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

for  $i = 1, 2, \dots, n$   

$$\left[ x_i \leftarrow \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^n a_{ij}x_j^{(k)} \right) / a_{ii} \right]$$

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 more generally. 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 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 choose  $\mathbf{P} \approx \mathbf{A}$ , 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 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:

$$\mathbf{A} = \mathbf{P} - \mathbf{N} = \mathbf{L} + \mathbf{D} + \mathbf{U} : \quad \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} = \begin{bmatrix} \bullet & & & \\ & \bullet & & \\ & & \bullet & \\ & & & \bullet \end{bmatrix} + \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet \\ & & \bullet & \bullet \\ & & & \bullet \end{bmatrix}$$

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 bound is true for *every* induced norm. From theorem 5, 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  $\lambda$  with  $|\lambda| \geq 1$ . Then if  $\mathbf{e}^{(0)}$  happens to be an eigenvector associated with  $\lambda$ , 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 are greater than the sum of the magnitudes 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 should 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:

$$\|\mathbf{e}^{(k+1)}\|_{\infty} \leq \frac{\sum_{j>i} |a_{ij}|}{|a_{ii}| - \sum_{j<i} |a_{ij}|} \|\mathbf{e}^{(k)}\|_{\infty} < \frac{|a_{ii}| - \sum_{j<i} |a_{ij}|}{|a_{ii}| - \sum_{j<i} |a_{ij}|} \|\mathbf{e}^{(k)}\|_{\infty} = \|\mathbf{e}^{(k)}\|_{\infty}$$

because the matrix is diagonally dominant.  $\square$

The *convergence factor*  $r_k$  of an iterative method is the ratio of errors at each iteration:  $r_k = \|\mathbf{e}^{(k)}\|/\|\mathbf{e}^{(k-1)}\|$ . The average convergence factor over  $m$  iterations is the geometric mean  $(\prod_{k=1}^m r_k)^{1/m}$ . From this expression, 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 rate of the Jacobi and Gauss–Seidel methods on the  $n \times n$  discrete Laplacian

$$\mathbf{A} = \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}. \quad (5.6)$$

Let's start with the Jacobi method. The spectral radius of  $\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$  is  $\cos \frac{\pi}{n+1}$ , which is approximately  $1 - \frac{1}{2}\pi^2(n+1)^{-2}$  when  $n$  is large. This value gives us the convergence factor of the Jacobi method. The asymptotic convergence rate (again for large  $n$ ) is about  $\frac{1}{2}\pi^2(n+1)^{-2}$ , using the approximation  $\log(1+z) \approx z - \frac{1}{2}z^2 + \dots$ .

The spectral radius of  $(\mathbf{L} + \mathbf{D})^{-1}\mathbf{U}$  is  $\cos^2 \frac{\pi}{n+1}$ . So, the convergence factor of the Gauss–Seidel method is approximately  $1 - \pi^2(n+1)^{-2}$  when  $n$  is large. The asymptotic convergence rate of the Gauss–Seidel method is  $\pi^2(n+1)^{-2}$ , twice that of the Jacobi method.

If  $\mathbf{A}$  is smallish  $20 \times 20$  matrix, then the convergence factor for the Jacobi method is 0.99. It is 0.98 for the Gauss–Seidel method. 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.<sup>1</sup> Pretty lousy, especially when naïve Gaussian elimination gives an exact solution in less than one-tenth of the time. 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 the Jacobi and Gauss–Seidel methods are easy to invert, but both approaches converge slowly. Our goal in this section will be find a

---

<sup>1</sup>The missing arithmetic is  $\log 0.01/\log 0.99 = 450$  and  $\log 0.01/\log 0.98 = 225$ .

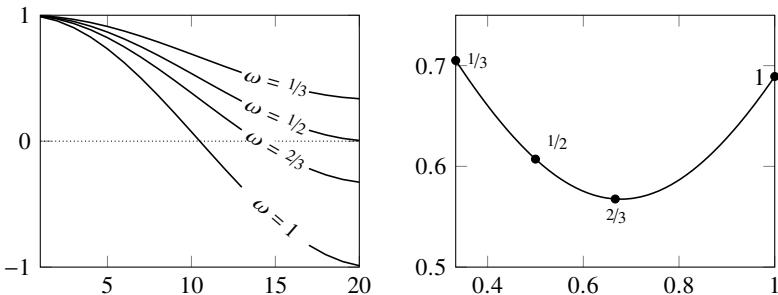


Figure 5.2: Left: eigenvalues of the weighted Jacobi method for the  $20 \times 20$  discrete Laplacian (5.6) for different weights  $\omega$ . Right: root mean squared values of the eigenvalues as a function of  $\omega$ .

splitting  $\mathbf{A} = \mathbf{P} - \mathbf{N}$  that quickly shrinks 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 iteration. A component with a small eigenvalue will swiftly fade away. A component with 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  $\mathbf{P}^{-1} = \omega \mathbf{D}^{-1}$ . This approach is called a *weighted Jacobi method* with the splitting

$$\mathbf{A} = \mathbf{P} - \mathbf{N} = (\omega^{-1}\mathbf{D}) + (\mathbf{A} - \omega^{-1}\mathbf{D}).$$

The eigenvalues  $\lambda_i(\mathbf{P}^{-1}\mathbf{N})$  of the discrete Laplacian (5.6) are

$$\lambda_i(\mathbf{P}^{-1}\mathbf{N}) = 1 + \omega \left( \cos \frac{i\pi}{n+1} - 1 \right),$$

or  $i = 1, 2, \dots, n$ . 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  $\omega$  might minimize the root mean squared values of the eigenvalues  $\lambda_i$ . For the discrete Laplacian (5.6), that choice for  $\omega$  happens to be about  $\frac{2}{3}$ . However, when we reduce the magnitude of some eigenvalues by tuning  $\omega$ , we are also increase the magnitude of other eigenvalues, including the dominant one.

Before modifying the Gauss–Seidel method, let's think about the meaning of  $\omega$ . The residual at the  $k$ th iteration is

$$\mathbf{r}^{(k)} = \mathbf{A}\mathbf{e}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}.$$

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. We can think of an iterative method 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 is equivalent to

$$\mathbf{u}^{(k+1)} - \mathbf{u}^{(k)} = \mathbf{P}^{-1}(\mathbf{b} - \mathbf{Au}^{(k)}).$$

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 a 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 are tuned by modifying the step size  $\Delta t$ . 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 *successive over-relaxation* (SOR) method speeds up convergence by choosing  $\omega > 1$  to go *beyond* the Gauss–Seidel correction. The naïve algorithm for the SOR method is

for  $i = 1, 2, \dots, n$

$x^* \leftarrow \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i+1}^n a_{ij}x_j \right) / a_{ii}$
$\delta \leftarrow x^* - x_i$
$x_i \leftarrow x_i + \omega\delta$

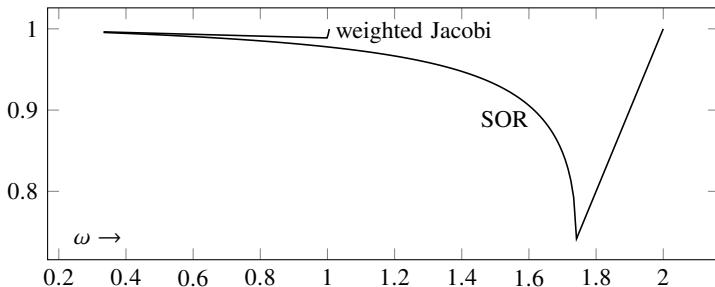


Figure 5.3: The spectral radius  $\rho(\mathbf{P}^{-1}\mathbf{N})$  of the weighted Jacobi and SOR methods for the  $20 \times 20$  discrete Laplacian as functions of the relaxation factor  $\omega$ .

The constant  $\omega$  is called the relaxation factor. The SOR algorithm simply computes the step size  $\delta$  determined by the Gauss–Seidel algorithm and scales that step size by  $\omega$ . When  $\omega = 1$ , the SOR method is simply the Gauss–Seidel method. In general, the SOR method has the splitting

$$\mathbf{A} = \mathbf{P} - \mathbf{N} = (\mathbf{L} + \omega^{-1}\mathbf{D}) + ((1 - \omega^{-1})\mathbf{D} + \mathbf{U}).$$

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 challenge is to choose an  $\omega$  so that the SOR method converges quickly, i.e., choose the  $\omega$  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 is  $\omega = 2/(1 + \sin \pi/n)$ , for which  $\rho(\mathbf{P}^{-1}\mathbf{N}) = 1 - 2\pi n^{-1} + O(n^{-2})$ . See Watkins [2014].

The convergence factor for a  $20 \times 20$  discrete Laplacian is computed in the figure above for several values of  $\omega$ . 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 discrete Laplacian, we can reduce the number of iterations from 225 down to about 15—a significant improvement.<sup>2</sup> Still, a  $20 \times 20$  matrix is tiny, 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

<sup>2</sup>The missing arithmetic is  $\log 0.01 / \log 0.74 \approx 15$ .

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. Information is shared across adjacent mesh points in the five-point stencil shown in Figure 5.1. A domain with  $n$  mesh points in each dimension requires  $n/2$  iterations to pass information from one side to the other. As a result, convergence is slow. Of course, we could use a larger stencil. Information would travel more quickly across the domain, but we are adding more nonzero elements to our sparse matrix.

Another idea is to use a coarser mesh. 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, the solution would not be as accurate with a coarser mesh.

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:

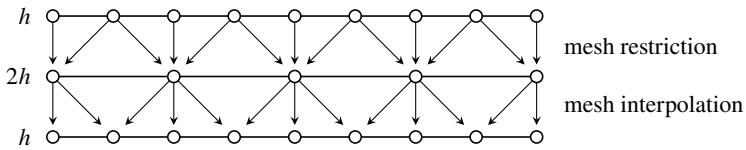
- solve the problem on a coarse mesh to kill off the residual from the low frequencies components;
- solve the problem on a fine mesh for high accuracy; and
- use the solution on the coarse mesh as a starting guess for the solution on the fine mesh (and vice versa).

Consider the one-dimensional Poisson equation. 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  $h$ ) has the following steps:

1. Iterate a few times on  $\mathbf{A}_h \mathbf{u}_h = \mathbf{b}_h$  using weighted Jacobi, Gauss–Seidel, or SOR. This step shrinks the residual  $\mathbf{r}_h = \mathbf{b}_h - \mathbf{A}_h \mathbf{u}_h$  from the high-frequency eigenvector components and smooths 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

$$\mathbf{R}_h^{2h} = \frac{1}{4} \begin{bmatrix} 3 & 1 & & & & \\ & 1 & 2 & 1 & & \\ & & & 1 & 2 & 1 \\ & & & & \ddots & \\ & & & & & 1 & 3 \end{bmatrix}$$

mapping a  $(2n + 1)$ -dimensional vector  $\mathbf{r}_h$  to an  $(n + 1)$ -vector  $\mathbf{r}_{2h}$ .



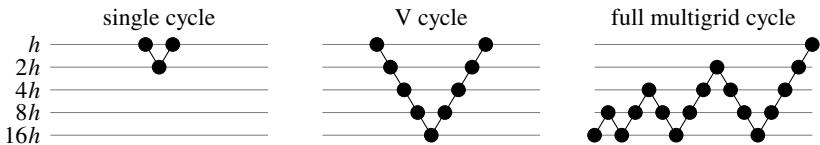
3. Solve (iterate a few times)  $\mathbf{A}_{2h}\mathbf{e}_{2h} = \mathbf{r}_{2h}$  where  $\mathbf{e}_{2h} = \mathbf{u}_h - \mathbf{u}_{2h}$ . Not surprisingly,  $\mathbf{A}_{2h} = \mathbf{R}_h^{2h} \mathbf{A}_h$ . Because the size of the matrix has changed, we may also need to adjust the relaxation factor  $\omega$  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

$$\mathbf{I}_{2h}^h = \frac{1}{2} \begin{bmatrix} 2 & 1 & & & & & & \\ & 1 & 2 & 1 & & & & \\ & & & 1 & 2 & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 & 2 \end{bmatrix}^T.$$

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, one typically uses multiple cycles, restricting to coarser and coarser meshes, before interpolating back. Or, they might start multigrid on a coarse grid. Here are different cycle designs used to restrict and interpolate between an 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. We'll take  $n = 129$  grid points and choose the relaxation factor  $\omega$  optimal for the SOR and multigrid methods. 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

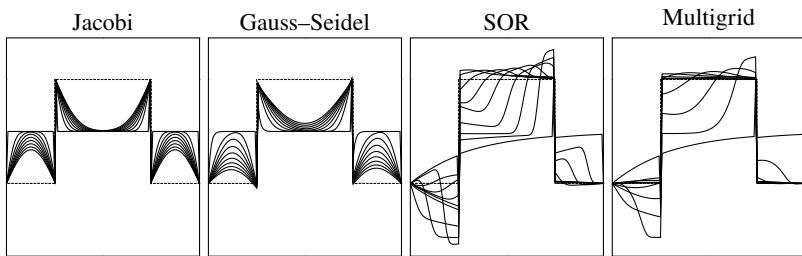


Figure 5.4: Iterative solutions to the Poisson equation converging to the exact solution, the rectangle function. Snapshots are taken every twentieth iteration for a total of 160 iterations. Also, see the QR code at the bottom of this page.

guess of  $u(x) = \frac{1}{2}$ . The figure on the current page plots the solution. Notice that the SOR and multigrid methods converge relatively quickly, while the Jacobi and Gauss–Seidel methods still have significant errors after many iterations and convergence slows down.  $\blacktriangleleft$

## 5.4 Solving the minimization problem

One of the difficulties of the SOR method is that you need to know the optimal relaxation parameter  $\omega$ . A better approach for symmetric, positive-definite matrices is recasting the problem as a minimization problem. In Chapter 3, we solved an overdetermined system as an equivalent minimization problem. And in the last chapter, we approximated the eigenvalues of sparse matrices by finding the solution that minimized the error in the Krylov subspace. The Poisson equation  $-\Delta u = f$  has an equivalent formulation of finding the solution  $u$  that minimizes the energy functional  $\int_{\Omega} \frac{1}{2} |\nabla u|^2 - uf \, dV$ . This section examines how to solve an iterative method as a minimization problem.

### ▷ Gradient descent method

Gradient descent is arguably the most commonly used approach to solving minimization problems.<sup>3</sup> Nonlinear functions may have no minimum, multiple minima, saddle-points, and local minima that can cause gradient descent to fail. These difficulties vanish for positive-definite quadratic forms.

---

<sup>3</sup>Augustin–Louis Cauchy proposed gradient descent in 1847—a mere three-page note among his thirteen thousand pages of mathematical writing.



comparison of Jacobi,  
Gauss–Seidel, SOR,  
and multigrid methods

**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*

$$\Phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Ax} - \mathbf{x}^T\mathbf{b} = \frac{1}{2}\|\mathbf{x}\|_{\mathbf{A}}^2 - (\mathbf{x}, \mathbf{b}).$$

*Proof.* Take  $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$  and consider any other vector  $\mathbf{y}$ . Then

$$\begin{aligned}\Phi(\mathbf{y}) - \Phi(\mathbf{x}) &= \frac{1}{2}\mathbf{y}^T\mathbf{Ay} - \mathbf{y}^T\mathbf{b} - \frac{1}{2}\mathbf{x}^T\mathbf{Ax} + \mathbf{x}^T\mathbf{b} \\ &= \frac{1}{2}\mathbf{y}^T\mathbf{Ay} - \mathbf{y}^T\mathbf{Ax} - \frac{1}{2}\mathbf{x}^T\mathbf{Ax} + \mathbf{x}^T\mathbf{Ax} \\ &= \frac{1}{2}\mathbf{y}^T\mathbf{Ay} - \frac{1}{2}\mathbf{y}^T\mathbf{Ax} - \frac{1}{2}\mathbf{y}^T\mathbf{Ax} + \frac{1}{2}\mathbf{x}^T\mathbf{Ax} \\ &= \frac{1}{2}\mathbf{y}^T\mathbf{Ay} - \frac{1}{2}\mathbf{y}^T\mathbf{Ax} - \frac{1}{2}\mathbf{x}^T\mathbf{Ay} + \frac{1}{2}\mathbf{x}^T\mathbf{Ax} \\ &= \frac{1}{2}(\mathbf{y} - \mathbf{x})^T\mathbf{A}(\mathbf{y} - \mathbf{x}) > 0\end{aligned}$$

because  $\mathbf{A}$  is symmetric, positive definite. So,  $\Phi(\mathbf{y}) > \Phi(\mathbf{x})$ .  $\square$

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

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \quad (5.7)$$

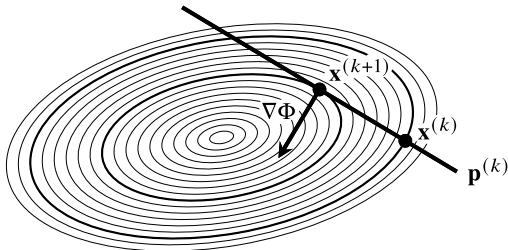
for some direction  $\mathbf{p}^{(k)}$ . The error at the  $k$ th step is  $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$ . To get to the bottom of the “bowl” knowing only the local topography at  $\mathbf{x}^{(k)}$ , the best path to take is the steepest descent, i.e., the direction of the negative gradient. The gradient of  $\Phi$  at  $\mathbf{x}^{(k)}$  is

$$\nabla\Phi(\mathbf{x}^{(k)}) = \mathbf{Ax}^{(k)} - \mathbf{b} = -\mathbf{r}^{(k)}.$$

The negative gradient is simply the residual. So we can rewrite the  $k$ th subsequent estimate (5.7) 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  $\alpha_k$ . Since the evaluation of  $\mathbf{Ax}^{(k)}$  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 nonzero 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  $\Phi$  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  $\Phi$  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 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  $\mathbf{r}^{(k)}$ .

Figure 5.5: The optimal vector  $\mathbf{x}^{(k+1)}$  along direction  $\mathbf{p}^{(k)}$ .

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  $\mathbf{p}^{(k)}$ .

$$\begin{aligned}\mathbf{r}^{(k+1)} &= \mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)} \\ &= \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}) \\ &= \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)} \\ &= \mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)}.\end{aligned}\quad (5.9)$$

When  $\mathbf{r}^{(k+1)} \perp \mathbf{p}^{(k)}$ :

$$0 = \mathbf{p}^{(k)\top} \mathbf{r}^{(k+1)} = \mathbf{p}^{(k)\top} \mathbf{r}^{(k)} - \alpha_k \mathbf{p}^{(k)\top} \mathbf{A}\mathbf{p}^{(k)}.$$

Therefore,  $\mathbf{x}^{(k+1)}$  is optimal with respect to  $\mathbf{r}^{(k)}$  when

$$\alpha_k = \frac{\mathbf{p}^{(k)\top} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)\top} \mathbf{A}\mathbf{p}^{(k)}}.$$

By taking  $\mathbf{p}^{(k)} = \mathbf{r}^{(k)}$  (for the gradient descent method), we have

$$\alpha_k = \frac{\mathbf{r}^{(k)\top} \mathbf{r}^{(k)}}{\mathbf{r}^{(k)\top} \mathbf{A}\mathbf{r}^{(k)}}.$$

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 x
for k = 0, 1, 2, ...
  [
    r ← b - Ax
    if ||r|| < tolerance, then we're done
    α ← rᵀ r / rᵀ Ar
    x ← x + αr
  ]

```

## ► 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

$$\alpha_k = \frac{\mathbf{p}^{(k)\top} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)\top} \mathbf{A} \mathbf{p}^{(k)}}.$$

Consider taking the direction  $\mathbf{p}^{(k)}$  successively along each coordinate axis  $\xi_i$ . For example, if  $\mathbf{A}$  is  $3 \times 3$ , 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 following page.

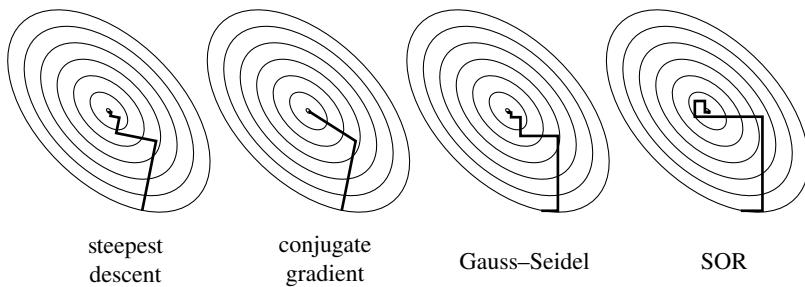
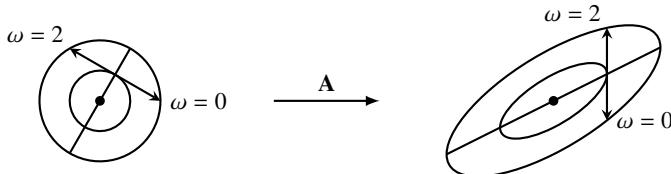


Figure 5.6: Directions taken by different iterative methods to get to the solution.

**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  $\Phi$  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 sheared and stretched along a coordinate axis. This interpretation follows directly from QR decomposition.



The value  $\omega = 2$  takes us in a direction parallel to a coordinate axis to the point at the same elevation on the opposite side of the bowl. Any value  $0 < \omega < 2$  positions us lower in the bowl, and  $\omega = 1$  positions us optimally along the direction.  $\square$

### ► Conjugate gradient method

Locally, gradient descent gives the best choice of directions. Globally, it doesn’t—especially when the bowl is very eccentric, i.e., when the matrix’s condition number is large. So the gradient descent method doesn’t always work well in practice. We can improve it by taking globally optimal directions, 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  $\mathbf{p}^{(k-1)}$ . Let  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}$  for some direction  $\mathbf{p}^{(k)}$ , and suppose that  $\mathbf{x}^{(k+1)}$  is optimal with respect to  $\mathbf{p}^{(k)}$ —just as we did in developing the gradient method. So  $\mathbf{p}^{(k)} \perp \nabla\Phi(\mathbf{x}^{(k+1)})$  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  $j = 0, 1, \dots, k$ .

What does this new optimality condition mean? 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  $\text{span}\{\mathbf{p}^{(k)}, \mathbf{p}^{(k-1)}, \dots, \mathbf{p}^{(0)}\}$  by the 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

$$\mathbf{r}^{(k+1)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)} = \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}) = \mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)},$$

and from our optimality condition, we have

$$0 = \mathbf{p}^{(j)\top} \mathbf{r}^{(k+1)} = \mathbf{p}^{(j)\top} \mathbf{r}^{(k)} - \alpha_k \mathbf{p}^{(j)\top} \mathbf{A}\mathbf{p}^{(k)}$$

for all  $j \leq k$ . For  $j = k$ , it follows that

$$\alpha_k = \frac{\mathbf{p}^{(k)\top} \mathbf{r}^{(k)}}{\mathbf{p}^{(k)\top} \mathbf{A}\mathbf{p}^{(k)}}.$$

And, because  $\mathbf{r}^{(k)} \perp \mathbf{p}^{(j)}$  for all  $j < k$ , we must have

$$0 = -\alpha_k \mathbf{p}^{(j)\top} \mathbf{A}\mathbf{p}^{(k)}.$$

In other words, to preserve optimality between iterates,  $\mathbf{p}^{(k)}$  and  $\mathbf{p}^{(j)}$  must be *A-conjugate* for all  $j < k$ :

$$(\mathbf{p}^{(j)}, \mathbf{p}^{(k)})_{\mathbf{A}} \equiv \mathbf{p}^{(j)\top} \mathbf{A}\mathbf{p}^{(k)} = 0 \quad \text{for } j = 0, 1, \dots, k-1. \quad (5.10)$$

As before, we'll take the first direction equal to the gradient at  $\mathbf{x}^{(0)}$ , i.e., we'll take  $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$ . Then each subsequent direction will be A-conjugate to this initial 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  $\beta_k$  is chosen so that

$$(\mathbf{p}^{(k+1)}, \mathbf{p}^{(j)})_{\mathbf{A}} = 0, \quad j = 0, 1, \dots, k. \quad (5.12)$$

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. (We'll better clarify this mathematical sleight-of-hand when discussing Krylov subspaces in the next section.) If we multiply (5.11) by  $\mathbf{A}\mathbf{p}^{(k)}$  and enforce (5.12) when  $j = k$ , then it follows that

$$\beta_k = \frac{\mathbf{r}^{(k+1)\top} \mathbf{A} \mathbf{p}^{(k)}}{\mathbf{p}^{(k)\top} \mathbf{A} \mathbf{p}^{(k)}} = \frac{(\mathbf{p}^{(k)}, \mathbf{r}^{(k+1)})_{\mathbf{A}}}{(\mathbf{p}^{(k)}, \mathbf{p}^{(k)})_{\mathbf{A}}}.$$

Now, let's show that this choice for  $\beta_k$  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  $\mathbf{x}$   
 $\mathbf{p} \leftarrow \mathbf{r} \leftarrow \mathbf{b} - \mathbf{Ax}$   
for  $k = 0, 1, 2, \dots$

$\alpha \leftarrow \mathbf{r}^T \mathbf{p} / \mathbf{p}^T \mathbf{Ap}$ $\mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{p}$ $\mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{Ap}$ if $\ \mathbf{r}\  <$ tolerance, then we're done $\beta \leftarrow \mathbf{r}^T \mathbf{Ap} / \mathbf{p}^T \mathbf{Ap}$ $\mathbf{p} \leftarrow \mathbf{r} - \beta \mathbf{p}$
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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 entire  $n$  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

$$\begin{aligned}\beta &\leftarrow \mathbf{r}^\top \mathbf{Ap} / \mathbf{p}^\top \mathbf{Ap} && \text{solve } \mathbf{Pz} = \mathbf{r} \\ \mathbf{p} &\leftarrow \mathbf{r} - \beta \mathbf{p} && \beta \leftarrow \mathbf{z}^\top \mathbf{Ap} / \mathbf{p}^\top \mathbf{Ap} \\ &&& \mathbf{p} \leftarrow \mathbf{z} - \beta \mathbf{p}\end{aligned}$$

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 expression 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 the general minimal residual method (GMRES). It is simply called 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

$$\underbrace{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix}}_{\mathbf{Q}_k} = \underbrace{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}}_{\mathbf{Q}_{k+1}} \underbrace{\begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix}}_{\mathbf{H}_{k+1,k}}. \quad (5.13)$$

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  $\mathbf{H}_k$  is tridiagonal if  $\mathbf{A}$  is Hermitian or real symmetric. 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

$$\|\mathbf{e}^{(k+1)}\|_{\mathbf{A}}^2 = \mathbf{e}^{(k+1)T} \mathbf{A} \mathbf{e}^{(k+1)}.$$

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

$$\|\mathbf{e}^{(0)} - \mathbf{Q}_k \mathbf{z}^{(k)}\|_{\mathbf{A}}^2 = (\mathbf{e}^{(0)} - \mathbf{Q}_k \mathbf{z}^{(k)})^T \mathbf{A} (\mathbf{e}^{(0)} - \mathbf{Q}_k \mathbf{z}^{(k)}),$$

which happens when the gradient is zero:

$$2(\mathbf{Q}_k^T \mathbf{A} \mathbf{Q}_k \mathbf{z}^{(k)} - \mathbf{Q}_k^T \mathbf{A} \mathbf{e}^{(0)}) = \mathbf{0}.$$

Hence,

$$\mathbf{Q}_k^T \mathbf{A} \mathbf{Q}_k \mathbf{z}^{(k)} = \mathbf{Q}_k^T \mathbf{A} \mathbf{e}^{(0)} = \mathbf{Q}_k^T \mathbf{r}^{(0)}.$$

From (5.14) it follows that  $\mathbf{Q}_k^T \mathbf{A} \mathbf{Q}_k = \mathbf{H}_k$  because  $\mathbf{Q}_k \perp \mathbf{q}^{(k+1)}$ . Also, because  $\mathbf{r}^{(0)} = \|\mathbf{r}^{(0)}\|_2 \mathbf{q}^{(1)}$  and  $\mathbf{Q}_k^T \mathbf{q}^{(1)} = \boldsymbol{\xi}_1$ , it follows that  $\mathbf{z}^{(k)}$  must solve the upper Hessenberg system

$$\mathbf{H}_k \mathbf{z}^{(k)} = \|\mathbf{r}^{(0)}\|_2 \boldsymbol{\xi}_1$$

where  $\xi_1 = (1, 0, \dots, 0)$ . Note that the norm of the residual

$$\|\mathbf{r}^{(k)}\|_2 = \|\mathbf{b} - \mathbf{Ax}^{(k)}\|_2 = |h_{k+1,k}| \cdot |\xi_k^\top \mathbf{z}^{(k)}| = |h_{k+1,k}| \cdot |z_k^{(k)}|,$$

which we can use as a stopping criterion when the value is less than  $\varepsilon \|\mathbf{r}^{(0)}\|_2$  for some tolerance  $\varepsilon$ .

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

$$\underbrace{\mathbf{P}_k^\top \mathbf{A} \mathbf{P}_k}_{\text{symmetric}} = \underbrace{\mathbf{U}_k^{-\top} \mathbf{Q}_k^\top \mathbf{A} \mathbf{Q}_k \mathbf{U}_k^{-1}}_{\mathbf{U}_k^\top \mathbf{H}_k \mathbf{U}_k^{-1}} = \underbrace{\mathbf{U}_k^\top \mathbf{H}_k \mathbf{U}_k^{-1}}_{\text{lower triangular}} = \mathbf{U}_k^{-\top} \mathbf{L}_k ..$$

Hence,  $\mathbf{P}_k^\top \mathbf{A} \mathbf{P}_k$  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  $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{Ax}^{(k)}$ , we have

$$\begin{aligned} \mathbf{r}^{(k+1)} &= \mathbf{r}^{(0)} - \mathbf{A} \mathbf{Q}_k \mathbf{z}^{(k)} \\ &= \mathbf{Q}_{k+1} \mathbf{Q}_{k+1}^\top \mathbf{r}^{(0)} - \mathbf{A} \mathbf{Q}_k \mathbf{z}^{(k)} \\ &= \mathbf{Q}_{k+1} \left( \|\mathbf{r}^{(0)}\|_2 \xi_1 - \mathbf{H}_{k+1,k} \mathbf{z}^{(k)} \right) \end{aligned}$$

where  $\xi_1 = (1, 0, \dots, 0)$ . We now have the problem of finding  $\mathbf{z}^{(k)}$  to minimize

$$\|\mathbf{r}^{(k+1)}\|_2 = \| \|\mathbf{r}^{(0)}\|_2 \xi_1 - \mathbf{H}_{k+1,k} \mathbf{z}^{(k)} \|_2.$$

The solution to this problem is the solution to the overdetermined system

$$\mathbf{H}_{k+1,k} \mathbf{z}^{(k)} = \|\mathbf{r}^{(0)}\|_2 \xi_1,$$

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

$$\hat{\mathbf{R}}\mathbf{z}^{(k)} = \hat{\mathbf{Q}}^T \|\mathbf{r}^{(0)}\|_2 \xi_1,$$

where  $\hat{\mathbf{Q}}$  is an orthogonal matrix and  $\hat{\mathbf{H}}$  is an upper triangular matrix. The residual  $\|\mathbf{r}^{(k+1)}\|_2$ , given by the bottom element of this expression, is

$$\|\mathbf{r}^{(0)}\|_2 (\xi_1^T \hat{\mathbf{Q}} \xi_{k+1}) = \|\mathbf{r}^{(0)}\|_2 \hat{q}_{1,k+1}.$$

We stop when  $\|\mathbf{r}^{(k)}\|_2 < \varepsilon \|\mathbf{r}^{(0)}\|_2$  for some tolerance  $\varepsilon$  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. Confirm that the spectral radius of the Jacobi method is  $\cos \frac{\pi}{n+1}$  and the spectral radius of the Gauss–Seidel method is  $\cos^2 \frac{\pi}{n+1}$  for the discrete Laplacian (5.6). 

5.2. 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  $\omega$  will the SOR method converge? What is the optimal choice of  $\omega$ ? 

5.3. 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.

5.4. 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. 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  $125000 \times 125000$  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

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

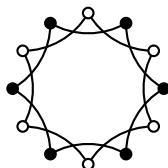
The seven-point finite difference operator is  $\mathbf{D} \otimes \mathbf{I} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{D} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{I} \otimes \mathbf{D}$  where  $\mathbf{D}$  is a discrete Laplacian (5.6) and  $\mathbf{I}$  is an identity matrix. 

5.5. Let  $\mathbf{A}$  be the  $20000 \times 20000$  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 typically 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  $y = c_0 + c_1 z + c_2 z^2 + \cdots + c_{n-1} z^{n-1}$  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,  $\omega_n^k$  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  $\omega$ .

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<sup>1</sup> generated by  $\omega_n^k$ . That

---

<sup>1</sup>A group is a set with an operation that possesses closure, associativity, an identity, and an inverse. A subgroup is any subset of a group, which is itself a group under the same operation.

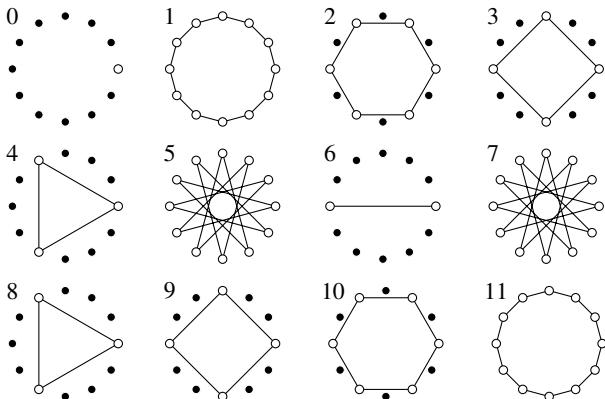


Figure 6.1: Subgroups using the generator  $\omega_{12}^j$  for  $j = 0, 1, \dots, 11$ .

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

$$\{2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24\} \pmod{12} \equiv \{2, 4, 6, 8, 10, 0\}.$$

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  $\omega_{12}^2$  equals the group generated by  $\omega_6^1$ .

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* formed by taking each element of the group and adding one. So,

$$\{0, 2, 4, 6, 8, 10\} \quad \text{has the coset} \quad \{1, 3, 5, 7, 9, 11\}.$$

Equivalently, for  $\omega \equiv \omega_{12}$

$$\{1, \omega^2, \omega^4, \omega^6, \omega^8, \omega^{10}\} \quad \text{times } \omega \text{ equals} \quad \{\omega, \omega^3, \omega^5, \omega^7, \omega^9, \omega^{11}\}.$$

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 the 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—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 with 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  $\omega_n^{m+j} = \omega_m^m \omega_n^j = -\omega_n^j$ . So

$$y_{m+j} = y'_j - \omega_n^j y''_j.$$

Therefore,

$$\begin{cases} y_j = y'_j + \omega_n^j y''_j & \text{for } j = 0, 1, \dots, m - 1. \\ y_{m+j} = y'_j - \omega_n^j y''_j & \end{cases} \quad (6.2)$$

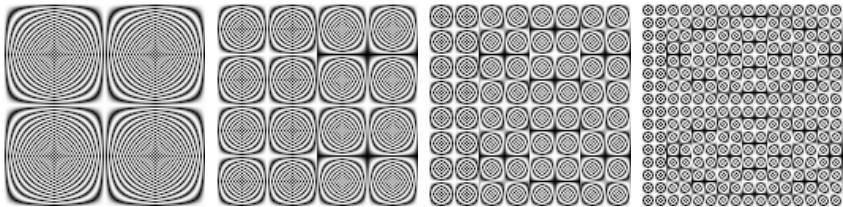


Figure 6.2:  $\mathbf{F}_{128}$  (left), after one permutation, after a second permutation, and after a third permutation (right) of the columns.

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

$$\mathbf{F}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & -i \end{bmatrix}.$$

Consider the even-odd permutation matrix (a reverse perfect shuffle<sup>2</sup>)

$$\mathbf{P}_4^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

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

$$\mathbf{B}_4 = \begin{bmatrix} \mathbf{I}_2 & \mathbf{\Omega}_2 \\ \mathbf{I}_2 & -\mathbf{\Omega}_2 \end{bmatrix}$$

---

<sup>2</sup>To 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.

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 on the preceding page, which 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

$$M_p = 2M_{p-1} + 2^{p-1}$$

$$A_p = 2A_{p-1} + 2^p.$$

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  $O\left(\frac{3}{2}n \log_2 n\right)$ . 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  $c$ :

```
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
```

We can adapt the FFT using the identity  $\mathbf{F}_n^{-1} = \bar{\mathbf{F}}_n/n$  to get the IFFT.

```
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 constant along each of its 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

$$\mathbf{R} = [\xi_2 \quad \xi_2 \quad \cdots \quad \xi_n \quad \xi_1] = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

That is,

$$\mathbf{C} = [\mathbf{c} \quad \mathbf{R}\mathbf{c} \quad \mathbf{R}^2\mathbf{c} \quad \cdots \quad \mathbf{R}^{n-1}\mathbf{c}] \text{ with } \mathbf{c} = [c_0 \quad c_1 \quad \cdots \quad c_{n-1}]^\top.$$

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  $\mathbf{W} = \text{diag}(1, \omega, \omega^2, \dots, \omega^{n-1})$ . Then,  $\mathbf{FR} = [\omega^{i(j+1)}] = [\omega^{ij}\omega^j]$  and  $\mathbf{WF} = [\omega^i\omega^{ij}]$ . So,  $\mathbf{R} = \mathbf{F}^{-1}\mathbf{WF}$ . For  $\mathbf{c} = (c_0, c_1, \dots, c_{n-1})$ ,

$$\mathbf{C} = c_0\mathbf{I} + c_1\mathbf{R} + c_2\mathbf{R} + \dots + c_{n-1}\mathbf{R}^{n-1}$$

and it follows that

$$\begin{aligned}\mathbf{C} &= \mathbf{F}^{-1} \left( c_0\mathbf{I} + c_1\mathbf{W} + c_2\mathbf{W}^2 + \dots + c_{n-1}\mathbf{W}^{n-1} \right) \mathbf{F} \\ &= \mathbf{F}^{-1} \text{diag} \left( \sum_{j=0}^{n-1} c_j, \sum_{j=0}^{n-1} c_j \omega^j, \dots, \sum_{j=0}^{n-1} c_j \omega^{j(n-1)} \right) \mathbf{F}\end{aligned}$$

Hence,  $\mathbf{C} = \mathbf{F}^{-1} \text{diag}(\mathbf{Fc}) \mathbf{F}$ . □

### ► Fast convolution

We can think of the convolution of two functions or arrays 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 definition of a convolution may at first seem foreign to many, we are 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  $\mathbf{u} * \mathbf{v} = \mathbf{C}_\mathbf{u}\mathbf{v}$  where  $\mathbf{C}_\mathbf{u}$  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, the convolution  $\mathbf{u} * \mathbf{v} = \mathbf{C}_\mathbf{u}\mathbf{v} = \mathbf{F}^{-1} \text{diag}(\mathbf{F}\mathbf{u})\mathbf{F}\mathbf{v}$ . 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 (or componentwise) product. Let me repeat—we compute the convolution by taking the IDFT of the component-wise product of the DFTs of  $\mathbf{u}$  and  $\mathbf{v}$ . This evaluation requires three FFTs. Instead of the  $2n^2$  operations needed 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

The previous section explained how to quickly evaluate circulant matrix multiplication using FFTs. We can do the same for a general Toeplitz matrix by padding it out to become a circulant matrix. For example,

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} a & d & e \\ b & a & d \\ c & b & a \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad \longrightarrow \quad \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ * \\ * \end{bmatrix} = \begin{bmatrix} a & d & e & c & b \\ b & a & d & e & c \\ c & b & a & d & e \\ e & c & b & a & d \\ d & e & c & b & a \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ 0 \\ 0 \end{bmatrix}$$

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,

$$\mathbf{T} = \begin{bmatrix} a & d & e \\ b & a & d \\ c & b & a \end{bmatrix} \quad \longrightarrow \quad \mathbf{C} = \begin{bmatrix} a & d & e & 0 & 0 & 0 & c & b \\ b & a & d & e & 0 & 0 & 0 & c \\ c & b & a & d & e & 0 & 0 & 0 \\ 0 & c & b & a & d & e & 0 & 0 \\ 0 & 0 & c & b & a & d & e & 0 \\ 0 & 0 & 0 & c & b & a & d & e \\ e & 0 & 0 & 0 & c & b & a & d \\ d & e & 0 & 0 & 0 & c & b & a \end{bmatrix}.$$

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 an  $n$ -long array  $\mathbf{c}$  and whose first row is 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(ifftx2(x₁).*ifftx2(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—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")
```

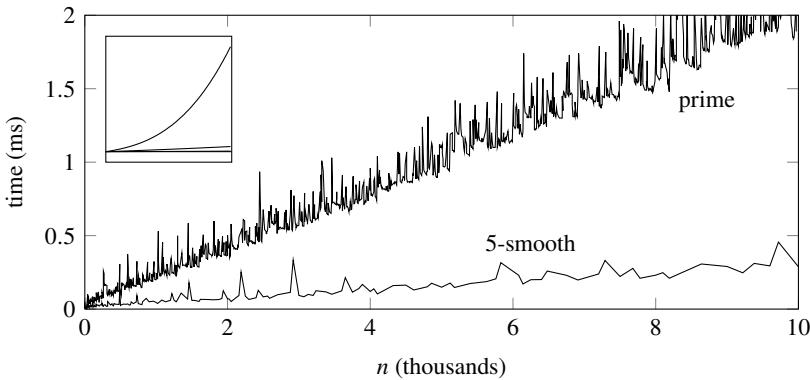


Figure 6.3: FFT run times on vectors of length  $n$ . The lower curve shows 5-smooth numbers and the upper curve shows prime numbers. The inset shows both curves relative to the run time of directly computing a DFT.

The figure above 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  $(j^2 - (j - k)^2 + k^2)/2$ :

$$y_j = \sum_{k=0}^{n-1} c_k \omega_n^{jk} = \omega_n^{j^2/2} \sum_{k=0}^{n-1} c_k \omega_n^{-(j-k)^2/2} \omega_n^{k^2/2}.$$

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

$$\begin{bmatrix} 1 & & & & \\ \omega & & & & \\ & \omega^4 & & & \\ & & \omega^9 & & \\ & & & \omega^6 & \end{bmatrix} \begin{bmatrix} 1 & \omega^9 & \omega^6 & \omega^1 & \omega^4 \\ \omega^9 & 1 & \omega^9 & \omega^6 & \omega^1 \\ \omega^6 & \omega^9 & 1 & \omega^9 & \omega^6 \\ \omega^1 & \omega^6 & \omega^9 & 1 & \omega^9 \\ \omega^4 & \omega^1 & \omega^6 & \omega^9 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & \\ \omega & & & & \\ & \omega^4 & & & \\ & & \omega^9 & & \\ & & & \omega^6 & \end{bmatrix}.$$

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 153:

```
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 kryptonite to the Cooley–Tukey algorithm. Luckily, there is an algorithm called Rader factorization whose superpower is prime numbers—and 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. We can replace the circulant matrix with 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 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  $r^{-1}$ .*

**Example.** The figure on the next page shows the finite cyclic groups of order  $n = 13$ . 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.  $\blacktriangleleft$

Suppose 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

$$y_0 = \sum_{j=0}^{n-1} c_j \quad \text{and} \quad y_{(r^{-k})} = c_0 + \sum_{j=1}^{n-1} c_{(r^j)} \omega_n^{(r^{j-k})}.$$

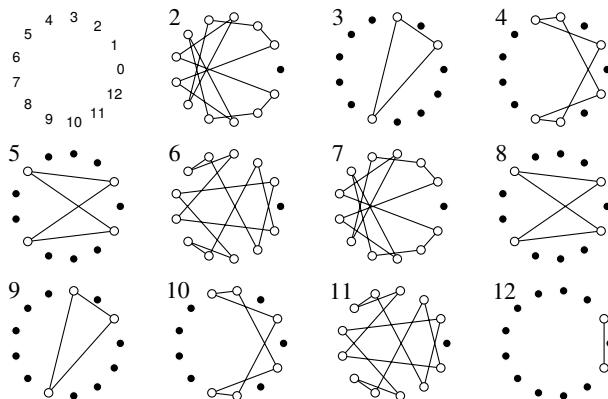


Figure 6.4: Finite cyclic groups of order 13.

We can rewrite the sum 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

$$[\omega_n^r \quad \omega_n^{r^2} \quad \cdots \quad \omega_n^{r^{n-1}}],$$

the matrix  $\mathbf{P}_r$  is the permutation matrix generated by  $r$

$$\{r, r^2 \pmod{n}, r^3 \pmod{n}, \dots, r^{n-1} \pmod{n}\},$$

and the matrix  $\mathbf{P}_s$  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 straightforward. We check each integer  $r$  until we find one that works. Let  $p$  be the factors of  $\varphi(n) = n - 1$ . If  $r^{p_i} \equiv 1 \pmod{n}$  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÷p_i, n) for p_i ∈ 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 constructing the frequency components of a time-varying or space-varying function, such as an audio signal. We examine several other applications for the DFT in this section. Before doing so, it's helpful to understand a little more about practical FFT implementation.

### ► 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 the library is called, it may try several different FFT algorithms for the same problem and use the fastest after that. The FFTW library decomposes the problem using the composite Cooley–Tukey algorithm recursively until the problem can be solved using fixed-size codelets.<sup>3</sup> 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.

The FFTW library saves the fastest approach for a given array size as the so-called *wisdom*, which it uses when for FFTs on other same-sized arrays. If a problem requires several repeated FFTs, it may be advantageous for FFTW to initially try all of the possible algorithms to see which is fastest. On the other hand, if the FFT is only needed once or twice, it may be better to let FFTW estimate the quickest 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.

Julia's `fft` and `ifft` functions are all multi-dimensional transforms unless an

---

<sup>3</sup>Codelets are small-sized (typically  $n \leq 64$ ), hard-coded transforms that use split-radix, Cooley–Tukey, and a prime factor algorithm.

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 put the zero frequency in the first position output array, it may be more mathematically meaningful or convenient to have the zero frequency in the middle position of the output array. We can do this by swapping the left and right halves of the array.

- The FFTW.jl functions `fftshift` and `ifftshift` shift the zero-frequency component to the middle of the array.

Discrete cosine transforms (DCTs) and discrete sine transforms (DSTs) 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, we can reuse the same functions 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 5.4. This problem  $\mathbf{Au} = \mathbf{f}$  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

$$\mathbf{x}_k = \begin{bmatrix} \sin\left(\frac{k\pi}{n+1}\right) & \sin\left(\frac{2k\pi}{n+1}\right) & \sin\left(\frac{3k\pi}{n+1}\right) & \cdots & \sin\left(\frac{nk\pi}{n+1}\right) \end{bmatrix}^T.$$

Take two arbitrary  $n \times n$  matrices  $\mathbf{M}$  and  $\mathbf{N}$  and an  $n \times n$  identity matrix  $\mathbf{I}$ . If  $\mathbf{Mx}_i = \lambda_i \mathbf{x}_i$  and  $\mathbf{My}_i = \mu_i \mathbf{y}_i$ , 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

$$\lambda_{ijk} = \left(6 - 2 \cos \frac{i\pi}{n+1} - 2 \cos \frac{j\pi}{n+1} - 2 \cos \frac{k\pi}{n+1}\right) / (4x)^2$$

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  $\mathbf{Au} = \mathbf{f}$  is  $\mathbf{SAS}^{-1}\mathbf{u} = \mathbf{f}$ . And the diagonalization of its solution  $\mathbf{u} = \mathbf{A}^{-1}\mathbf{f}$  is  $\mathbf{u} = \mathbf{SA}^{-1}\mathbf{S}^{-1}\mathbf{f}$ . 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. We can compute the discrete sine transform of  $(f_1, f_2, \dots, f_n)$  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 5.4 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);
```

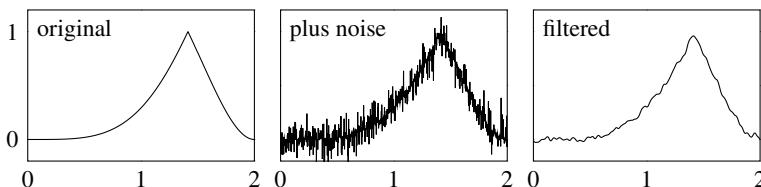
Comparing the numerical solution with the analytic solution,

```
norm(u - [(x-x^2)*(y-y^2)*(z-z^2)/2 for x∈x,y∈x,z∈x])
```

we find an error of  $1.5 \times 10^{-14}$ . 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. We can filter out the high frequencies and regularize the data 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 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 convolution kernel 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  $8 \times 8$  pixel blocks and then takes discrete cosine transforms (DCTs) of each of these blocks. Similarly, audio is often compressed as an MP3, using DCT compression of blocks. Using the Fourier series to deconstruct the natural world should be, well, natural. Joseph Fourier introduced



Figure 6.5: Discrete cosine transform (right) of the image (left).

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:  $\|\mathbf{e}\|_2 = \|\mathbf{Fe}\|_2$  and  $\|\mathbf{e}\|_F = \|\mathbf{Fe}\|_F$ . The pixelwise root mean squared error of an image equals the pixelwise root mean squared error of the DCT of an image.

Figure 6.5 above shows the magnitudes of the discrete cosine transform of a cartoon image. The image isn't an exceptionally smooth one—it has several sharp edges, particularly the eyes, mouth, and chin. The most significant Fourier coefficients are associated with low-frequency components in the upper left-hand corner, and in general, the Fourier coefficients decay as their frequencies increase.

By cropping out the high-frequency components in the Fourier domain, we can keep relevant information with only a marginal increase in the error if the data is smooth. However, if the data is not smooth, important information resides in the higher frequency components and discarding it can result in substantial error. Notice in Figure 6.5 that several slowly decaying rays corresponding to sharp edges and lines in the cartoon emanate from the upper left-hand corner. The so-called Gibbs phenomenon associated with lossy compression of data with discontinuities can result in noticeable JPEG ringing artifacts at sharp edges. As an alternative approach, we can zero out only those Fourier components whose magnitudes are less than some given tolerance regardless of their frequencies. We can store the information efficiently as a sparse array by zeroing out enough components. We'll leave the first approach as an exercise and consider the second approach below.

Consider the following Julia function that compresses a grayscale image  $A$  to a factor  $d$  from its original array size. We could consider each RGB or HSV channel independently for a color image. The function first computes the DCT  $B$  of an image  $A$ . It determines the threshold tolerance  $\text{tol}$  based on the desired density  $d$  and then drops the smallest values to get a sparse matrix  $B_0$ . Finally, the function returns a reconstruction of the compressed image.

```
using Images, FFTW, SparseArrays, FileIO
function dctcompress(A,d)
    B = dct(Float64.(A))
    idx = d*prod(size(A)) |> floor |> Int
    tol = sort!(abs.(B[:]),rev=true)[idx]
    B₀ = droptol!(sparse(B),tol)
    idct(B₀) |> clamp01! .|> Gray
end
```

Let's compare an original image and a 95-percent compressed image.

```
A = Gray.(load(download(bucket*"laura_square.png")))
[A dctcompress(A,0.05)]
```

To be fair,  $B_0$  is a 64-bit floating-point array and would still need to be quantized to 256 discrete values, and the sparse array requires additional overhead for the row-column indices. We can compute the relative compression and error using

```
Base.summarysize(B₀)/sizeof(Matrix(B₀)), norm(B - B₀)/norm(B)
```

By computing the errors from several values of  $d$ , we can determine the relative efficiency of DCT compression. See the figure on the next page and the QR code below.

## 6.6 Exercises

6.1. Modify the function `fftx2` on page 149 for a radix-3 FFT algorithm. Verify your algorithm using a vector of size  $3^4 = 81$ .



6.2. *Fast multiplication of large integers.* Consider two polynomials

$$p(x) = p_n x^n + \cdots + p_1 x + p_0 \quad \text{and} \quad q(x) = q_n x^n + \cdots + q_1 x + q_0.$$

When  $x = 10$ , the polynomials  $p(x)$  and  $q(x)$  return the decimal numbers  $p(10)$  and  $q(10)$ , respectively. We can represent the polynomials as coefficient vectors by using the basis  $\{1, x, x^2, \dots, x^{2n}\}$ :

$$\mathbf{p} = (p_0, p_1, \dots, p_n, 0, \dots, 0) \quad \text{and} \quad \mathbf{q} = (q_0, q_1, \dots, q_n, 0, \dots, 0).$$

an image with  
increasing levels of  
DCT compressed



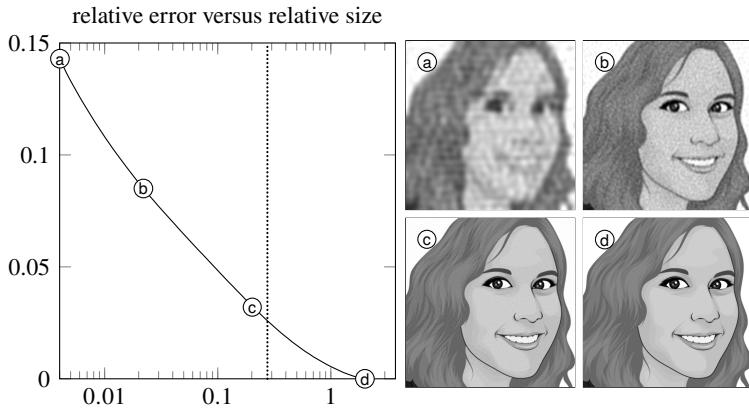


Figure 6.6: The relative error of a DCT-compressed image. The dotted line shows equivalent lossless PNG compression.

The product  $p(x)q(x)$  is represented by

$$(p_0q_0, p_0q_1 + p_1q_0, \dots, p_{n-1}q_n + p_nq_{n-1}, p_0q_0), \quad (6.5)$$

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  $\sum_{i=0}^{2n} p_i q_{j-i} \pmod{2n}$ . 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 is then adjusted 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. We can compute the fast discrete cosine transform using an FFT by 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 mesh points  $\{1, 2, \dots, n - 1\}$ , we can arrange the nodes between the mesh points  $\{\frac{1}{2}, \frac{3}{2}, \dots, n - \frac{1}{2}\}$ , counting over every second node and then reflecting the signal back at the boundary. For example, with eight nodes  $\textcircled{0} \textcircled{1} \textcircled{2} \textcircled{3} \textcircled{4} \textcircled{5} \textcircled{6} \textcircled{7} \textcircled{7} \textcircled{6} \textcircled{5} \textcircled{4} \textcircled{3} \textcircled{2} \textcircled{1} \textcircled{0}$  would have the reordering  $\textcircled{0} \textcircled{2} \textcircled{4} \textcircled{6} \textcircled{7} \textcircled{5} \textcircled{3} \textcircled{1}$ . 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. 

6.4. Compare two approaches to image compression using a DCT—cropping high-frequency components and removing low magnitude components—discussed on page 162. 



# Numerical Methods for Analysis



Chapter 7

## Preliminaries



This chapter briefly introduces 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 handwave 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 of 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).

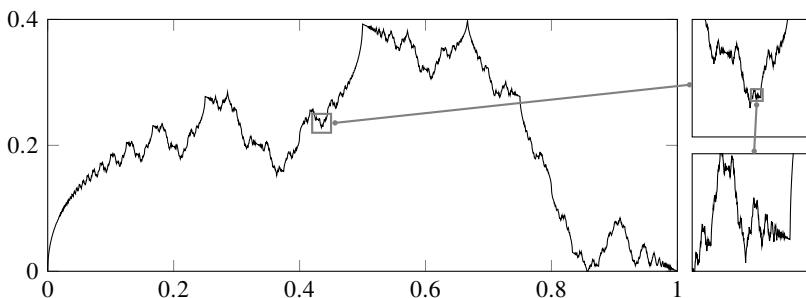


Figure 7.1: The nowhere differentiable Weierstrass function. The callouts depict the fractal nature of this pathological function.

[1916]). The Weierstrass function is also one of the earliest examples of a fractal, a set with repeating self-similarity at every scale and noninteger 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, the addition of two numbers  $x + y$ , which is the succession of  $x$  applied  $y$  times. Then, the multiplication of two numbers  $xy$ , which is the addition of  $x$  with itself  $y$  times. Followed by the exponentiation of two numbers  $x^y$ , which is the 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  ${}^4 3 = 1258 \dots 9387$  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, its spine would be over 38 miles thick. ▶



the Weierstrass  
function

To dig deeper into pathological functions, see Gelbaum and Olmstead's *Counterexamples in Analysis*. While mathematically, almost all functions are pathological, the functions important for applications are often quite nice. The typical naughty functions we encounter might have a cusp, a discontinuity, or a blowup. One feature of nice functions is a 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 simplify the 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

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + \frac{1}{6}h^2f'''(x) + \dots \\ &= f(x) + hf'(x) + O(h^2). \end{aligned}$$

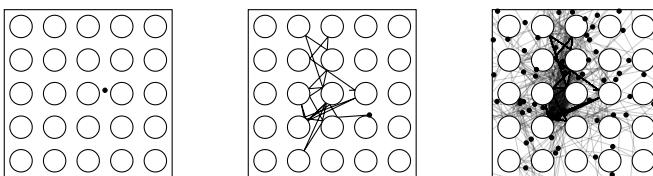
From this approximation, it follows that

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h),$$

where we've taken  $O(h^2)/h = O(h)$ . So forward differencing provides an  $OO[h]1$  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  $t = 10$ ?”. We can solve the problem using the geometry of intersections of lines and circles. At each intersection, we need to evaluate a square root. And each numerical evaluation of a square root adds round-off error to the solution, repeatedly compounding. The solution loses about a digit of precision with every reflection. While the original problem asked for the solution at  $t = 10$  to 100-digits of accuracy, we could also think about what happens at  $t = 20$ . The figure on the left below shows the starting position.

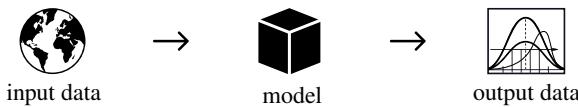


The middle figure shows the solution for a photon at  $t = 20$ . Instead of just one photon, let’s solve the problem using 100 photons each with a minuscule initial velocity angle spread of  $10^{-16}$  for each photon (machine floating-point precision). To put it into perspective, if these photons left 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 code at the bottom of this page. It’s hard to tell which solution is the right one or how accurate our original solution in the middle actually was. ▶

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.



photons in a room of  
circular mirrors



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. 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 *ill-posed*.

## ► Condition number

Existence and uniqueness are straightforward. Let's dig into stability. We say that the output  $y$  depends continuously on the input  $x$  if small perturbations in the input cause 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  $\kappa$  such that  $\|\delta y\| \leq \kappa \|\delta x\|$  for any  $\delta x$ . We call  $\kappa$  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  $\ell^1$ -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 the cancellation of significant digits.

The quadratic formula

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

is a well-known and frequently memorized solution to the quadratic equation  $ax^2 + bx + c = 0$ . However, 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:<sup>1</sup>

$$x_{\pm} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

The combined formula is

$$x_- = \frac{-b - \text{sign}(b)\sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_+ = \frac{c}{ax_-}. \quad \blacktriangleleft$$

### ► 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

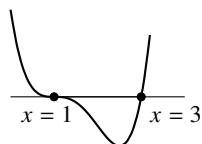
$$\kappa_{\text{abs}}(x) = \|(\varphi^{-1})'(y)\| = \|[\varphi'(x)]^{-1}\|,$$

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  $\kappa(3) = 8$ . 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.

---

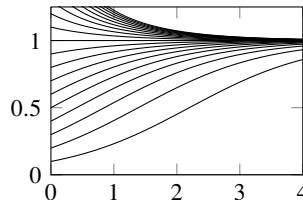
<sup>1</sup>As you may have deduced, *citardauq* is simply *quadratic* spelled backward.

**Example.** The logistic<sup>2</sup> 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  $s = 1$ , 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.

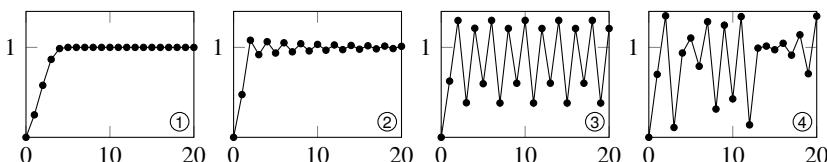
The simplest numerical method of solving the differential equation is to use the forward Euler approximation to the derivative and then iteratively solve the resulting difference equation. In this case, we have

$$\frac{s^{(k+1)} - s^{(k)}}{\Delta t} = s^{(k)}(1 - s^{(k)}), \quad (7.3)$$

where  $\Delta t$  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  $\Delta t$  given by 1, 1.9, 2.5, and 2.8:




---

<sup>2</sup>The term *logistic* was coined by Belgian mathematician Pierre François Verhulst as *logistique* in his 1845 article on population growth. The name refers to 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.

- ① When  $\Delta t = 1$ , the solution to (7.3) closely matches the solution to (7.2).  
 ② When  $\Delta t = 1.9$ , the numerical solution exhibits transient oscillations and converges to the equilibrium solution to (7.2). ③ When  $\Delta t = 2.5$ , the solution converges to a limit cycle with period 4. As  $\Delta t$  increases, so does the periodicity of the limit cycle. ④ When  $\Delta t = 2.8$ , the numerical solution is completely chaotic. As  $\Delta t$  continues to increase, the solution eventually becomes unstable and blows up. We'll return to this example when we study dynamical systems in section 8.4.

◀

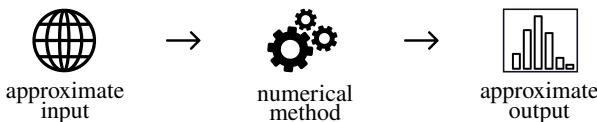
Again, consider the problem statement:

$$F(y, x) = 0 \text{ for some input data } x \text{ and some output data } y.$$

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, such as 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  $\delta x_n$ , and we let  $\delta y_n$  be the corresponding perturbation in the solution determined by the numerical method  $F_n(y + \delta y_n, x + \delta x_n) = 0$ . A 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  $\kappa$ , whenever  $\|\delta x_n\| \leq \eta$  for some  $\eta$ . 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. 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,

$$y(x) - y_n(x) = \left( \frac{\partial F_n}{\partial y} \right)^{-1} \left( F_n(y(x), x) - F_n(y_n(x), x) \right).$$

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,

$$y(x) - y_n(x) = \left( \frac{\partial F_n}{\partial y} \right)^{-1} \left( F_n(y(x), x) - F(y(x), x) \right),$$

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  $\varepsilon'$  comes from convergence. Choosing  $\varepsilon$  and  $\varepsilon'$  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.

$$\begin{aligned}\|y(x) - y_n(x + \delta x_n)\| &= \|y(x) - y_n(x) + y_n(x) - y_n(x + \delta x_n)\| \\ &\leq \|y(x) - y_n(x)\| + \|y_n(x) - y_n(x + \delta x_n)\| \\ &\leq M\varepsilon + \kappa\|\delta x_n\|,\end{aligned}$$

where the bound  $M\varepsilon$  comes from consistency and the bound  $\varepsilon'$  comes from stability. By choosing  $\|\delta x_n\| \leq \varepsilon/\kappa$ , we have

$$\|y(x) - y_n(x + \delta x_n)\| \leq (M + 1)\varepsilon.$$

So the method is convergent. □

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 $\kappa$
convergence	for any $\varepsilon$ there are $\eta$ and $N$ such that if $n > N$ and $\ \delta x_n\  \leq \eta$ , then $\ y(x) - y_n(x + \delta x_n)\  \leq \varepsilon$
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), 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)` returns the first ( $p/\log_2 10$ ) digits of  $\pi$ . The command `precision(x)` returns the precision of a variable  $x$ .

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, you should be caution 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: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

$$0 \boxed{10000011} \boxed{000101000000000000000000} \text{ equals}$$

$$1.000101_2 \times 2^4 = 10001.01_2 = 2^4 + 2^0 + 2^{-2} = 17.25$$

$$-1.11_2 \times 2^{-1} = -0.111_2 = -(2^{-1} + 2^{-2} + 2^{-3}) = -0.875$$

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 round-off* 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. The cryptic and seemingly obfuscated C code is translated to Julia code below.<sup>3</sup>

```
function Q_rsqrt(x::Float32)
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.,  $1/\sqrt{x}$ . Video game developers use the reciprocal square root to normalize vectors for shading and reflection in 3D generated scenes. In the late 1990s, `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 for 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

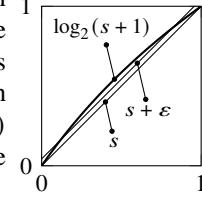
$$\log_2 x = \log_2 \left( 2^{e-127} (1 + 2^{-23}m) \right) = (e - 127) + \log_2 (1 + 2^{-23}m). \quad (7.7)$$

---

<sup>3</sup>The 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.

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  $\varepsilon$  is some offset. Let's choose an  $\varepsilon$  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  $m_x$

$$\begin{aligned} (e_y - 127) + (2^{-23}m_y + \varepsilon) \\ = -\frac{1}{2}((e_x - 127) + (2^{-23}m_x + \varepsilon)) \\ = \left((- \frac{1}{2}e_x) - 127\right) + \left(2^{-23}\left(-\frac{1}{2}m_x\right) + \varepsilon\right) + \frac{3}{2}(127 - \varepsilon) \\ = \left((- \frac{1}{2}e_x) - 127\right) + \left(2^{-23}\left(-\frac{1}{2}m_x\right) + \varepsilon\right) + 2^{-23}\left(2^{23} \cdot \frac{3}{2}(127 - \varepsilon)\right). \end{aligned}$$

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 doesn't match the magic number used in the Q\_rsqrt function. How the original value was chosen is not known.<sup>4</sup>

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  $i \gg 1$ .

To summarize the Q\_rqsrt 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. We'll benchmark it using random input data to prevent Julia from optimizing the function output to a constant during compilation.

<sup>4</sup>Jean-Michel Muller argues that the optimal choice of magic numbers is one that minimizes the relative error after one Newton iteration, not the error going to the Newton iteration as is done here. Moroz et. al find the optimal magic number 5f37642f<sub>16</sub> using an exhaustive search.

```
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  $2 \times 10^{-2}$  after the first approximation. It is  $1 \times 10^{-3}$  after the Newton iteration—good enough for a late 1990s shooter game. Were we to add another Newton iteration, the relative error would drop to  $2 \times 10^{-6}$ .  $\blacktriangleleft$

## ► 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  $1.641 \times 10^{21}$ . Matlab and R both return about  $-1.181 \times 10^{21}$ , 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.  $\blacktriangleleft$

**Example.** The U.S. Army used Patriot surface-to-air missiles to intercept Iraqi Scud missiles during the Gulf War. In February 1991, a missile battery in Dhahran, Saudi Arabia, failed to track an incoming Scud missile that subsequently hit a 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. 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  $9.5 \times 10^{-8}$ . So, every second the clock lost  $9.5 \times 10^{-7}$  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 the predicted range-gate because of the round-off error, so the system deemed the initial detection spurious. ▶

#### ► Underflow, overflow, and NaN

Because the number of bits reserved for the exponent of a floating-point number is limited, floating-point arithmetic has an upper bound (closest to infinity) and a lower bound (closest to zero). Any computation outside 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  $1.8 \times 10^{308}$ . The command `floatmin(Float64)` returns the smallest normalized floating-point number— $2^{-1022}$  or about  $2.2 \times 10^{-308}$ .

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. Numbers can be as small as  $\text{eps}(0.0) = \text{floatmin}(\text{Float64}) * \text{eps}(1.0)$  before complete underflow.

**Example.** A spectacular example of overflow happened in 1996 when the European Space Agency (after spending ten years and \$7 billion) launched the first Ariane 5 rocket. The rocket crashed within 40 seconds. An error occurred converting a 64-bit floating-point number to 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 nonzero 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  $\varepsilon$  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  $\varepsilon_i$  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  $\varepsilon$  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  $\varepsilon$  can be substantially larger than machine epsilon leading to catastrophic cancellation.

We can extend our analysis to a sum or arbitrarily many numbers. To add  $n$  numbers sequentially  $x_1 + x_2 + \dots + x_n$  requires  $2n - 1$  rounding operations. We have 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. We see from this expression that we should add smaller terms first to minimize the propagation of rounding error.

**Example.** In 1982 the Vancouver Stock Exchange<sup>5</sup> 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 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.

Suppose the algorithm had instead used proper rounding to avoid bias. In that case, we should expect the sum of the errors to be normally distributed with zero mean and standard deviation  $\sigma\sqrt{n}$ , where the  $\sigma$  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 of those random variables.) The standard deviation for the discrete uniform distribution is  $\sigma = 0.0003$ . After 1.2 million trades, we should 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 from 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

$$f(x + h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(\xi)$$

---

<sup>5</sup>The building of the defunct Vancouver Stock Exchange now has a swanky cocktail bar.

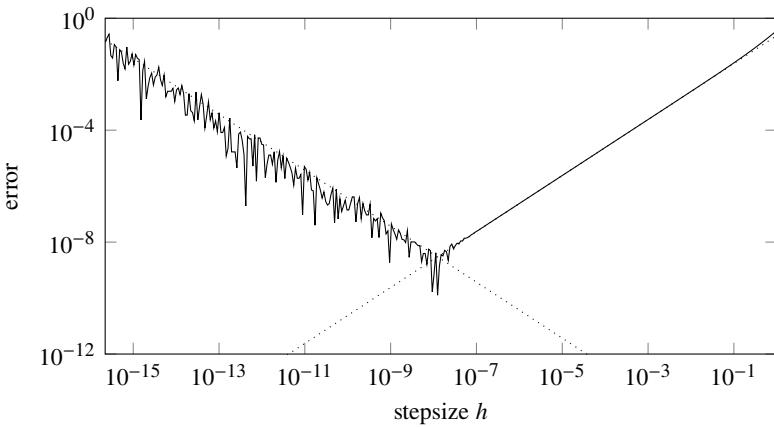


Figure 7.2: The total error for a first-order finite difference approximation. The truncation error decreases as  $h$  gets smaller and the round-error increases as  $h$  gets smaller. The minimum is near  $h = 10^{-8}$ .

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  $h = 2\sqrt{\text{eps}/m}$ .

**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  $\varepsilon_{\text{trunc}} = \frac{1}{3}h^2m$  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. 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 == 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  $h$ .

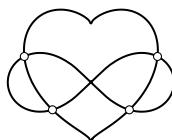
7.4. In deriving the `Q_sqrt` magic number, we chose  $\varepsilon$  to minimize the  $L^\infty$ -norm of  $\log(x+1) - (x+\varepsilon)$ . Instead, we could have chosen  $\varepsilon$  to minimize the  $L^2$ -norm, i.e., to minimize the mean error for a number selected at random. What is the magic number in this case? What is the magic number if we apply the `Q_sqrt` 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 get 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  $f(x)$  that is continuous but not necessarily differentiable. The bisection method is perhaps the simplest way of finding a zero  $x^*$  of  $f(x)$ . 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. The function  $f$  has a zero in the interval  $[a, b]$  by the intermediate value theorem. 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
```

<i>convergence</i>	<i>sequence</i>	<i>convergence rate</i>	$k \rightarrow \infty$
sublinear	$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots, k^{-1}, \dots$	$\frac{(k+1)^{-1}}{k^{-1}}$	1
linear	$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots, 2^{-k}, \dots$	$\frac{2^{-(k+1)}}{2^{-k}}$	$\frac{1}{2}$
superlinear	$1, \frac{1}{4}, \frac{1}{27}, \frac{1}{256}, \dots, k^{-2}, \dots$	$\frac{(k+1)^{-k-1}}{k^{-k}}$	0
quadratic	$1, \frac{1}{16}, \frac{1}{256}, \frac{1}{65536}, \dots, 2^{-k^2}, \dots$	$\frac{2^{-(k+1)^2}}{(2^{-k^2})^{p=2}}$	1

Figure 8.1: Some sequences that converge to zero.

```

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? When does the method work? And when it works, how quickly should we expect the method to converge to a solution? We'll need to define the order and rate of convergence to answer these questions.

## ► Convergence

A sequence  $\{x^{(0)}, x^{(1)}, x^{(2)}, \dots\}$  converges to  $x^*$  with order  $p$  if there is a positive constant  $r$  such that the subsequent error  $|x^{(k+1)} - x^*| \leq r|x^{(k)} - x^*|^p$  for all  $k$  sufficiently large. Likewise, a numerical method that generates such a sequence is said to be of order  $p$ . The value  $p$  is the *order of convergence*, and the value  $r$  is the *rate of convergence*.

A method converges linearly if  $p = 1$  and the convergence rate  $r$  is strictly less than 1. A method is superlinear if the convergence rate  $r_k$  varies with each iteration and  $r_k \rightarrow 0$  as  $k \rightarrow \infty$ . And a method is sublinear if  $r_k \rightarrow 1$  as  $k \rightarrow \infty$ . Convergence is quadratic if  $p = 2$  Convergence is cubic if  $p = 3$ . And so on. See Figure 8.1.

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  $e^{(k)} = x^{(k)} - x^*$ . So

$$e^{(k)} \leq |I^{(k)}| = \frac{1}{2}|I^{(k-1)}| = \frac{1}{4}|I^{(k-2)}| = \dots = 2^{-k}|b-a|.$$

We see that convergence is linear with a convergence rate  $r = \frac{1}{2}$ . We add one bit of precision to the solution with each iteration. We will need to take about  $\log_2(10) \approx 3.3$  iterations to gain a digit of precision in the solution. 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. The bisection method can help us get close enough to a solution to 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

$$0 = f(x^*) = f(x) + (x^* - x)f'(x^*) + \frac{1}{2}(x^* - x)^2f''(\xi)$$

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 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  $e^{(k+1)}$ :

$$\begin{aligned}
 e^{(k+1)} &= e^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \\
 &= \frac{f'(x^{(k)})e^{(k)} - f(x^{(k)})}{f'(x^{(k)})} \\
 &= \frac{f'(x^* + e^{(k)})e^{(k)} - f(x^* + e^{(k)})}{f'(x^* + e^{(k)})} \\
 &= \frac{\frac{1}{2}f''(x^*)(e^{(k)})^2 + \frac{1}{3}f^{(3)}(x^*)(e^{(k)})^3 + o((e^{(k)})^3)}{f'(x^*) + f''(x^*)e^{(k)} + o((e^{(k)}))} \\
 &= \frac{f''(x^*)}{2f'(x^*)}(e^{(k)})^2 + o((e^{(k)})^2)
 \end{aligned} \tag{8.2}$$

when  $f'(x^*) \neq 0$ . Therefore, Newton's method converges quadratically. The number of correct digits in the solution more or less doubles with every iteration. With a good starting guess, we should get an answer 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:

```

1.000000000000000000000000000000000000000000000000000000000000000000...
1.500000000000000000000000000000000000000000000000000000000000000000...
1.416666666666666666666666666666666666666666666666666666666666666666...
1.41421568627450980392156862745098039215686274509803...
1.41421356237468991062629557889013491011655962211574...
1.41421356237309504880168962350253024361498192577619...
1.41421356237309504880168872420969807856967187537723...

```

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. As we should expect for a quadratic method, the number of correct decimals doubles with each iteration. ▶

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—the

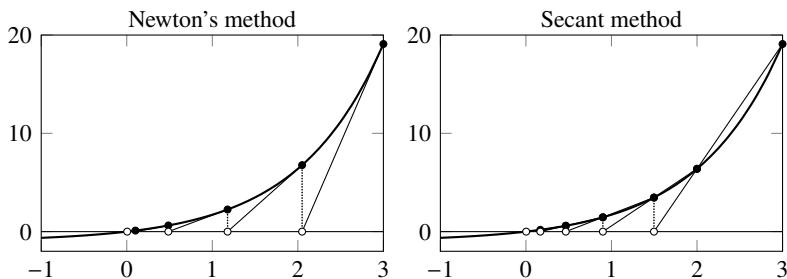


Figure 8.2: Comparison of two methods for finding the zeros of  $e^x - 1$ .

same as the bisection method. Moreover, if  $x^*$  is a root with multiplicity  $n$ , then Newton's method converges linearly with convergence rate  $r = 1 - \frac{1}{n}$ . 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 8.2 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

$$q^{(k)} = \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}, \quad (8.4)$$

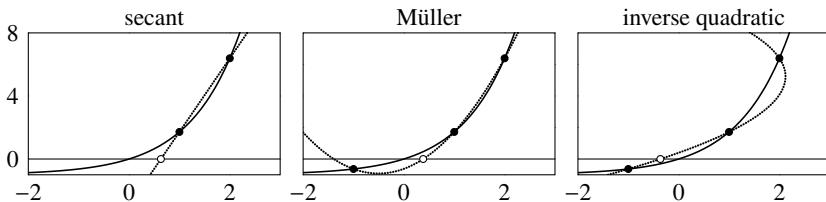
we have the *secant method*. Note that the method requires two initial values. The secant method's order of convergence is about 1.63—a bit lower than Newton's method.

### ► 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, it's often unnecessary to go through the trouble of implementing higher-order methods because we can achieve the same net result with multiple iterations of a superlinear method. 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

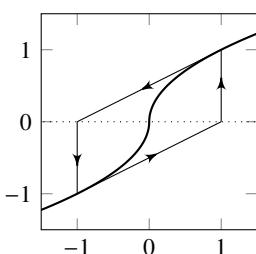
parabola  $y = ax^2 + bx + c$  which passes through the points  $(x^{(k-2)}, f(x^{(k-2)}))$ ,  $(x^{(k-1)}, f(x^{(k-1)}))$  and  $(x^{(k)}, f(x^{(k)}))$ . Because the parabola likely crosses the  $x$ -axis at two points, we'll need to choose the best one. This method, called *Müller's method*, has an order of convergence 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 the two needed for the secant method or the one needed for Newton's method.



### ► Dekker-Brent method

Newton's method and the secant method are not globally convergent. An iterated initial guess can converge to a different root, go off to infinity, or 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 does not converge. Instead, it oscillates between  $x^{(0)}$  and  $-x^{(0)}$  for any nonzero 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 nonzero guess, such as  $x^{(0)} = 1$ , sends us off to infinity  $\{1, -2, 4, -8, \dots\}$ . ◀

The bisection method is robust, but it is only linearly convergent. The secant method has an order of convergence approaching 1.63, but it is only locally convergent. The Dekker-Brent method, a hybrid of the bisection and secant methods, takes advantage of the strengths of both.

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.

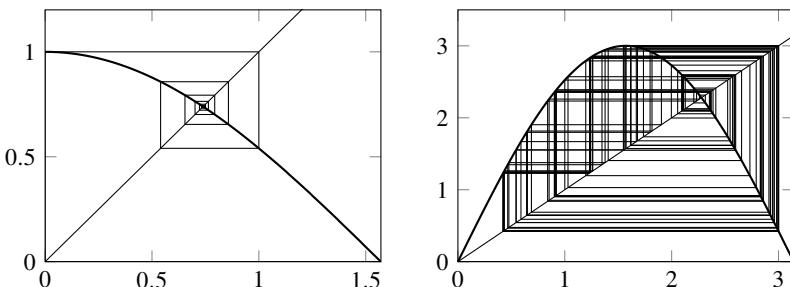


Figure 8.3: The fixed point iterations for  $\phi(x) = \cos x$ , which converges, and for  $\phi(x) = 3 \sin x$ , which does not converge.

• 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 procedures, such as `fzero(f, x0)` for finding roots.

### 8.3 Fixed-point iterations

If you type any number into a calculator and repeatedly hit the cosine button, the display eventually stops changing and gives you 0.7390851s. 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  $y = \phi(x)$ . We can study Newton's method as a special case of a broader class of numerical methods—fixed-point iterations:

$$\text{Given } x^{(0)}, \quad x^{(k+1)} = \phi(x^{(k)}). \quad (8.5)$$

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 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 precisely at these three points will never converge. Instead, it will bounce around in the neighborhood of either of the two nonzero fixed points. See the figure above. Let's identify the conditions that do guarantee convergence.

We say that  $\phi(x)$  is a *contraction mapping* over the interval  $[a, b]$  if  $\phi$  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  $\phi$  is a contraction map over  $[a, b]$ , then  $\phi$  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  $\phi$  is a continuous map from  $[a, b]$  to  $[a, b]$ , so  $\phi$  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  $\phi$  expanded about  $x^*$  and evaluated at  $y^*$  is

$$\phi(y^*) = \phi(x^*) + (y^* - x^*)\phi'(x^*)$$

for some  $\xi \in [x^*, y^*]$ . Therefore,

$$|y^* - x^*| = |\phi(y^*) - \phi(x^*)| = |(y^* - x^*)\phi'(\xi)| \leq K|y^* - x^*|.$$

Since  $K < 1$ , it must follow that  $y^* = x^*$ .

Now, we'll prove convergence. The first-order Taylor approximation of  $\phi$  expanded about  $x^*$  and evaluated at  $x^{(k)}$  is

$$\phi(x^{(k)}) = \phi(x^*) + (x^{(k)} - x^*)\phi'(x^*)$$

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  $\phi$  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  $x^*$ . 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  $\phi$  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 a linearly convergent method

Linearly convergent methods can be excruciatingly slow. Fortunately, there are some techniques to accelerate convergence. This section introduces Aitken's  $\Delta^2$  process or *Aitken's extrapolation*.<sup>1,2</sup>

For any sequence  $x^{(0)}, x^{(1)}, x^{(2)}, \dots$  that converges linearly to a value  $x^*$ , the ratio in errors is more or less constant

$$\frac{x^* - x^{(k+1)}}{x^* - x^{(k)}} \approx \frac{x^* - x^{(k)}}{x^* - x^{(k-1)}}.$$

Solving this expression for  $x^*$  yields

$$x^* \approx \frac{x^{(k+1)}x^{(k-1)} - (x^{(k)})^2}{x^{(k+1)} - 2x^{(k)} + x^{(k-1)}}, \quad (8.6)$$

which we can rewrite as a correction to  $x^{(k+1)}$  as

$$x^* \approx x^{(k+1)} - \frac{(x^{(k+1)} - x^{(k)})^2}{x^{(k+1)} - 2x^{(k)} + x^{(k-1)}}. \quad (8.7)$$

The difference operator  $\Delta x^{(k)} = x^{(k)} - x^{(k-1)}$  simplifies the notation

$$x^* \approx x^{(k+1)} - \frac{(\Delta x^{(k+1)})^2}{\Delta^2 x^{(k+1)}}.$$

---

<sup>1</sup>New Zealand mathematician Alexander Aitken published his namesake process in 1926. The method had earlier been described by the Japanese mathematician Seki Kōwa in his 1642 book *Hatubi Sanpō*. Kōwa used *enri* (円理, circle theory) to approximate  $\pi$ , taking  $c_{16} + \frac{(c_{16}-c_{15})(c_{17}-c_{16})}{(c_{16}-c_{15})-(c_{17}-c_{16})}$ , where  $c_n$  is the perimeter of a  $2^n$ -gon inscribed in a circle of unit diameter.

<sup>2</sup>Alexander Aitken had a prodigious memory. He would recall numbers and long text several decades later. Aitken would also relive the war trauma from his experiences as an infantry soldier. He coped with recurrent psychological breakdowns by writing a memoir *Gallipoli to The Somme*. The acclaimed book was the inspiration behind Anthony Ritchie's 2016 oratorio.

While (8.6) is mathematically identical to (8.7), it is numerically less stable. (Why?) So it's advisable to use (8.7).

By feeding the Aitken's extrapolation to  $x^{(k+1)}$  back into the fixed point method, we have an accelerated method

$$u_0 = x^{(k)}, \quad u_1 = \phi(u_0), \quad u_2 = \phi(u_1), \quad x^{(k+1)} = u_2 - \frac{(\Delta u_2)^2}{\Delta^2 u_2}. \quad (8.8)$$

This method is called *Steffenson's method*, and it is quadratically convergent under suitable conditions. Aitken's extrapolation also improves stability. Aitken's method may still converge even if the fixed-point method does not converge.

### ► Stopping criteria

We can stop an iterative method 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 as a proxy for the error and stop when one of them falls below a prescribed tolerance. The residual  $\phi(x^{(k)}) - x^{(k)}$  is the same as the increment  $\Delta x^{(k+1)} = x^{(k+1)} - x^{(k)}$  for a fixed-point method. 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 relative to  $\varepsilon$ . In this case, using the increment will not be a reliable stopping criterion.

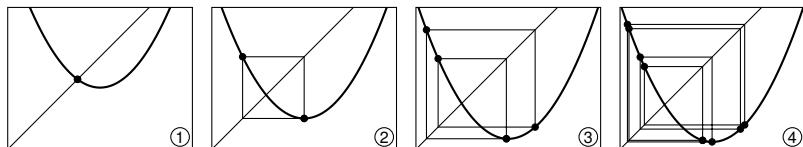
Designing robust stopping criteria is not trivial. For example, consider Newton's method applied to the function  $f(x) = e^{-\lambda x}$  for a positive value  $\lambda$ . While there are no solutions, the method will evaluate  $x^{(k+1)} = x^{(k)} + \lambda^{-1}$  at each step. In this case, the increment is always  $\lambda^{-1}$ , and using the increment as a stopping criterion will result in an endless loop. Using the residual  $e^{-\lambda x}$  as a stopping criterion will eventually terminate the method. Still, the solution will hardly be correct, and if  $\lambda$  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 handling 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.<sup>3</sup> 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 176 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.



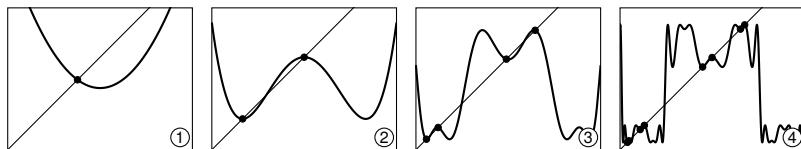
Why does the solution have this periodic limiting behavior? Take the iterated function compositions  $\phi_c^2 = \phi_c \circ \phi_c$ ,  $\phi_c^3 = \phi_c \circ \phi_c \circ \phi_c$ , and so on. The forward orbit  $\{x^{(0)}, \phi_c(x^{(0)}), \phi_c^2(x^{(0)}), \phi_c^3(x^{(0)}), \dots\}$  can be decomposed 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  $y = x$ .

---

<sup>3</sup>Robert May's influential 1976 review article "Simple mathematical models with very complicated dynamics," which 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  $\phi_c^2$ , 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 *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 code 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 determine 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  $\heartsuit$ , the shape traced by a point on a circle rolling around another circle of the same diameter.<sup>4</sup> For values  $c$  inside the cardioid, the orbit  $\{0, \phi_c(0), \phi_c^2(0), \dots\}$  limits a fixed point.

<sup>4</sup>There is an interesting related puzzle: “A coin is rolled around a similar coin without slipping.



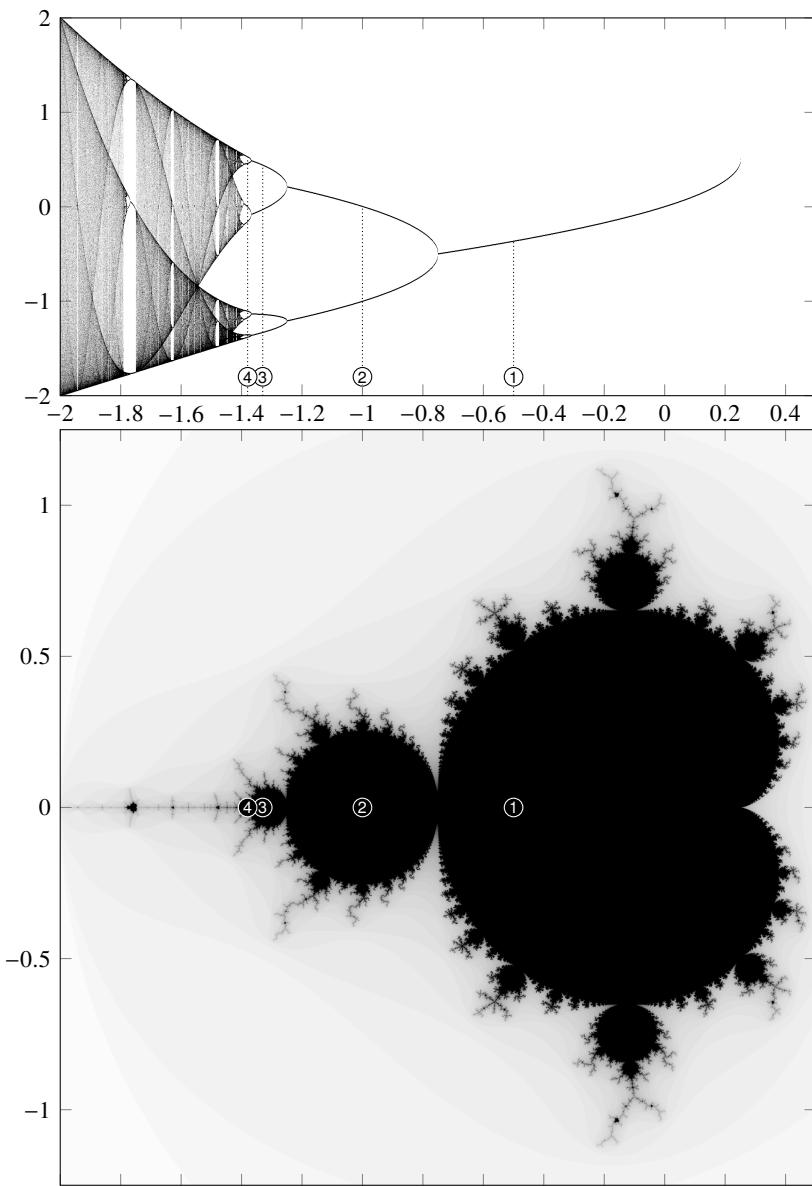


Figure 8.4: Top: The bifurcation diagram showing the limit cycle  $\{\phi_c^\infty(0)\}$  of the quadratic map as a function of  $c$ . Bottom: The Mandelbrot set is the set of complex values  $c$  for which  $\phi_c(0)$  has a limit cycle. The cases ①–④ from page 201 are indicated. Pay attention to changes in the period-doubling, bifurcation points, and matching cusps along the real axis of the Mandelbrot set.

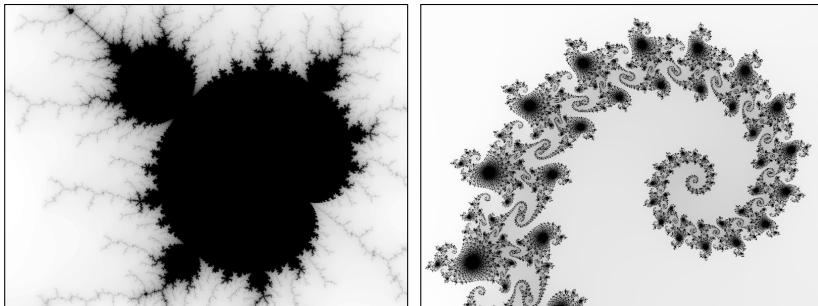


Figure 8.5: Snapshots of the Mandelbrot set showing a self-similar “minibrot” near  $c = -0.1602 + 1.0335i$  (left) and a spiral around a Misiurewicz point near  $c = -0.7452 + 0.1126i$  (right).

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 preceding 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 8.6 on the facing page.

Imaging the Mandelbrot set is straightforward. The following functions takes the parameters  $bb$  for the lower-left and upper-right corners of a bounding box,  $xn$  for the number of horizontal pixels, and  $n$  for the maximum number of iterations. The function `escape` counts the iterations  $k$  until  $|z^{(k)}| > 2$  for a given  $c$ .

```
function escape(n, c=0, z=0)
    for k in 1:n
        z = z^2 + c
        abs2(z) > 4 && return k
    end
    return n
```

---

What is the coin’s orientation when it is halfway around the stationary coin?” The answer is counter-intuitive, but it is evident in the equation of the cardioid.

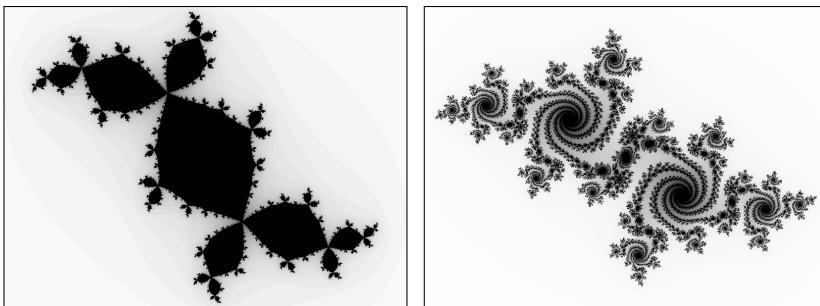


Figure 8.6: Julia sets for  $-0.123 + 0.745i$  (left) and  $-0.513 + 0.521i$  (right).

```
end
```

The function `mandelbrot` returns an array of escape iterations for an array  $c$ .

```
function mandelbrot(bb, xn=800, n=200, z=0)
    yn = (xn*(bb[4]-bb[2]))÷(bb[3]-bb[1]) |> Int
    c = LinRange(bb[1],bb[3],xn)' .+ im*LinRange(bb[4],bb[2],yn)
    [escape(n,c,z) for c in c]
end
```

The following produces the second image of Figure 8.5.

```
using Images
M = mandelbrot([-0.172, 1.0228, -0.1494, 1.0443])
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 variables  $z$  and  $c$ .

**Example.** Orbits that are chaotic, erratic, and unpredictable sometimes have applications. Such behavior makes elliptic curves especially useful in Diffie–Hellman public-key encryption<sup>5</sup> and pseudorandom number generators. An

---

<sup>5</sup>Suppose 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 the 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.

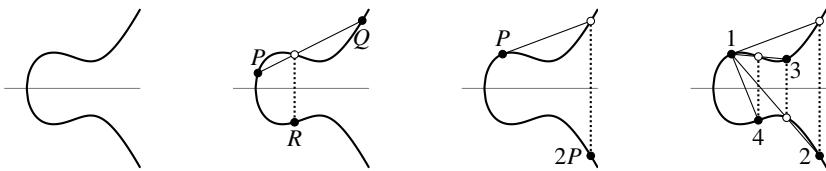


Figure 8.7: An elliptic curve, such as  $y^2 = x^3 - 2x + 4$ , is symmetric about the  $x$ -axis. A secant line through points  $P$  and  $Q$  passes through another point on the curve, the reflection of which about the  $x$ -axis is a point  $R = P \oplus Q$ . A tangent line through a point  $P$  passes 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

$$y^2 = x^3 + ax + b \quad (8.9)$$

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 nonvertical 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  $R = P \oplus Q$ . Points generated in this way using  $\oplus$  (the reflection of the third colinear point on the curve) form a group. The point  $O$  at infinity is the identity element in this group. We define  $2P = P \oplus P$ ,  $3P = P \oplus 2P$ , and so on. See Figure 8.7 above.

The arithmetic is straightforward. To compute  $2P$ , start by implicitly differentiating (8.9). We have  $2y \frac{dy}{dx} = (3x^2 + a) dx$  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 (8.9) gives

$$x^3 - (\lambda(x - x_0) + y_0)^2 + ax + b = 0.$$

The coefficient of the  $x^2$  term of the resulting monic cubic equation is  $-\lambda^2$ . The coefficient of a monic polynomial's second term is the negative sum of the polynomial's roots. Because the polynomial has a double root at  $x_0$ , 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 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 to the cubic equation.<sup>6</sup> At the time, a mathematician's fame (and fortune?) was tied to winning competitions with other mathematicians, duels of a sort to prove who was the better mathematician. So, while Tartaglia knew a formula for solving a cubic equation, he kept his knowledge of it secret and even obfuscated his formula in a poem, “*Quando chel cubo con le cose appreso. . .*”

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 counterexample 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.<sup>7</sup> To dig deeper, see Mario Livio’s book *The Equation That Couldn’t Be Solved*.

---

<sup>6</sup>As 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.”

<sup>7</sup>Évariste 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 be 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 method discussed in the previous section would work, a better design 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 we can write the polynomial  $p(x) = x^3 - 6x^2 + 11x - 6$  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 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,

$$p(x) = (x - z)q(x) + b_0 \quad (8.10)$$

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 (8.10), 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  $p'(x^{(0)})$ :

$$\begin{array}{c} 4 \left| \begin{array}{cccc} 1 & -6 & 11 & -6 \\ & 4 & -8 & 12 \\ \hline 1 & -2 & 3 & 6 \end{array} \right. \\[10pt] 4 \left| \begin{array}{ccc} 1 & -2 & 3 \\ & 4 & 8 \\ \hline 1 & 2 & 11 \end{array} \right. \end{array}$$

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. 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.<sup>8</sup>

## ► Other methods

Other popular methods of finding roots of polynomials include the Jenkins–Traub method, used in Mathematica’s `NSolve` function, and Laguerre’s method. These methods typically find a root and then deflate using Horner deflation.

## 8.6 Systems of nonlinear equations

Systems of equations often arise when finding a nonlinear least squares fit or implementing an implicit solver for a partial differential equation. Adding dimensions adds complexity, and 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. Furthermore, consider  $\mathbf{x}$  to be sufficiently close to 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

$$0 = f(\mathbf{x}^*) = f(\mathbf{x}) + \mathbf{J}_f(\mathbf{x})\Delta\mathbf{x} + O(\|\Delta\mathbf{x}\|^2)$$

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

$$\Delta\mathbf{x} = -(\mathbf{J}_f(\mathbf{x}))^{-1} f(\mathbf{x}) + O(\|\Delta\mathbf{x}\|^2)$$

---

<sup>8</sup>The `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  $O(n^3)$ .

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:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - (\mathbf{J}_f(\mathbf{x}^{(k)}))^{-1} f(\mathbf{x}^{(k)}). \quad (8.11)$$

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

$$\text{evaluate} \quad \mathbf{J}_f(\mathbf{x}^{(k)}), \quad (8.12a)$$

$$\text{solve} \quad \mathbf{J}_f(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)} = -f(\mathbf{x}^{(k)}), \quad (8.12b)$$

$$\text{update} \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}. \quad (8.12c)$$

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 a few techniques we can do to speed things up.

First, we could reuse the Jacobian. It's not necessary to compute the Jacobian matrix at every iteration. Instead, we can evaluate it every few steps. We can also reuse the upper and lower triangular factors of the LU decomposition of (8.12b), so that each subsequent iteration takes  $O(n^2)$  operations.

Alternatively, we could use an iterative method like the SOR or conjugate gradient method to solve the linear system (8.12b). Iterative methods multiply instead of factor, requiring  $O(n^2)$  operations per iteration for dense matrices and only  $O(n)$  operations per iteration for sparse matrices. They may take many iterations to converge, but because we only want an approximate solution  $\mathbf{x}^{(k+1)}$  with each step, we don't need full convergence for the iterative method to be effective. Chapter 5 discusses these methods.

Another way to speed up Newton's method is using a finite difference approximation of the Jacobian matrix. The  $ij$ -element of the Jacobi matrix is  $J_{ij} = \partial f_i / \partial x_j$ . So the  $j$ th column of the Jacobian matrix is the derivative of  $f(\mathbf{x})$  with respect to  $x_j$ . A second-order approximation of the  $j$ th column of  $\mathbf{J}_f(\mathbf{x})$  is

$$[\mathbf{J}_f(\mathbf{x})]_j = \frac{f(\mathbf{x} + h\xi_j) - f(\mathbf{x} - h\xi_j)}{2h}, +O(h^2) \quad (8.13)$$

where  $h$  is the step size and  $\xi_j$  is the  $j$ th vector in the standard basis of  $\mathbb{R}^n$ . We can typically take the step size  $h$  roughly equal to the cube root of machine epsilon For a well-conditioned problem.<sup>9</sup> If  $f(\mathbf{x})$  is a real-valued function, we can also approximate the Jacobian using a complex-step derivative

$$[\mathbf{J}_f(\mathbf{x})]_j = \frac{\operatorname{Im} f(\mathbf{x} + ih\xi_j)}{h} + O(h^2),$$

---

<sup>9</sup>See exercise 7.1.

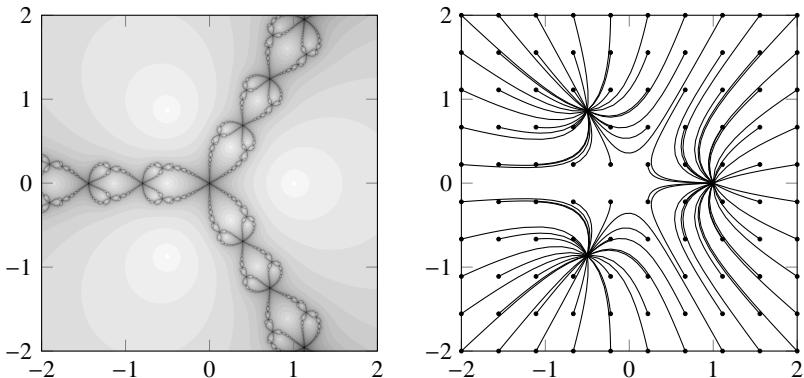


Figure 8.8: Newton fractal (left) for  $z^3 - 1$ . Darker shades of gray require more iterations to converge. Homotopy continuation (right) for the same problem.

where  $h$  is 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. A root's *basin of attraction* is the set of starting points for which Newton's method eventually converges to that specific root. A Julia set is the union of the boundaries of each basin of attraction. See Figure 8.8 above.  $\blacktriangleleft$

### ► Secant-like methods

The secant method used two subsequent iterates  $x^{(k)}$  and  $x^{(k-1)}$  to approximate 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. We have

$$\mathbf{J}^{(k)} \Delta \mathbf{x}^{(k)} = \Delta \mathbf{f}^{(k)} \quad (8.14)$$

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  $\Delta \mathbf{f}^{(k)} = f(\mathbf{x}^{(k)}) - f(\mathbf{x}^{(k-1)})$ . We just need to choose a suitable  $\mathbf{J}^{(k)}$ .

One way to interpret is, “Given an output vector  $\Delta \mathbf{f}^{(k)}$  and input vector  $\Delta \mathbf{x}^{(k)}$ , 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 (8.6) as a constraint. In other words, let’s 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? The Frobenius norm is a smart choice because it is an extension of the  $\ell^2$ -norm for vectors, and minimizing the  $\ell^2$ -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  $\lambda$ . 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

$$\mathbf{J}^{(k)} = \mathbf{J}^{(k-1)} + \frac{\Delta \mathbf{f}^{(k)} - \mathbf{J}^{(k-1)} \Delta \mathbf{x}^{(k)}}{\|\Delta \mathbf{x}^{(k)}\|^2} \Delta \mathbf{x}^{(k)\top}. \quad (8.15)$$

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 (8.15)
    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*

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^\top)^{-1} = \mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^\top\mathbf{A}^{-1}}{1 + \mathbf{v}^\top\mathbf{A}^{-1}\mathbf{u}}$$

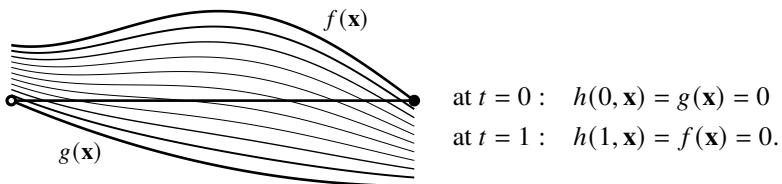
to update  $(\mathbf{J}^{(k)})^{-1}$  given  $(\mathbf{J}^{(k-1)})^{-1}$ :

$$(\mathbf{J}^{(k)})^{-1} = (\mathbf{J}^{(k-1)})^{-1} + \frac{\Delta \mathbf{x}^{(k)} - (\mathbf{J}^{(k-1)})^{-1} \Delta \mathbf{f}^{(k)}}{\|\Delta \mathbf{f}^{(k)}\|^2} \Delta \mathbf{f}^{(k)\top}.$$

## 8.7 Homotopy continuation

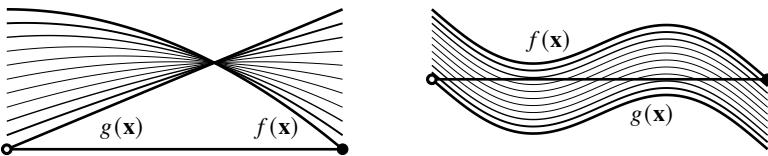
One might wonder if we could solve a simpler problem  $g(\mathbf{x}) = 0$  as a means to solve  $f(\mathbf{x}) = 0$ . Well, we can. One way is to use a homotopy continuation. Two functions are *homotopic* if one can be continuously deformed into the other. The deformation is called a homotopy.

Suppose that we have a system of equations  $f(\mathbf{x}) = 0$  and a system  $g(\mathbf{x}) = 0$ , whose roots we know or can easily determine. 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 assumption is not trivial—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  $\mathbf{x}(1)$ .

We can make a simple homotopy 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

$$\begin{aligned} h(t, \mathbf{x}) &= t f(\mathbf{x}) + (1-t)[f(\mathbf{x}) - f(\mathbf{x}_0)] \\ &= f(\mathbf{x}) - (1-t)f(\mathbf{x}_0). \end{aligned} \tag{8.16}$$

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

$$\frac{d\mathbf{x}}{dt} = -\mathbf{J}_f^{-1}(\mathbf{x}) f(\mathbf{x}_0)$$

with the initial conditions  $\mathbf{x}_0$ . Solving such a differential equation 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 (8.16), 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 8.8 on page 213.  $\blacktriangleleft$

The homotopy continuation method looks similar to Newton's method (8.13). In fact, we can derive Newton's method as a special case of homotopy continuation. Let's define the homotopy

$$h(t, \mathbf{x}) = f(\mathbf{x}) - e^{-t} f(\mathbf{x}_0). \quad (8.17)$$

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  $t = \infty$ . We just need to find a solution for a large enough  $t$  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 (8.17) 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  $\Delta t$ :

$$\frac{\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}}{\Delta t} = -\mathbf{J}_f^{-1}(\mathbf{x}^{(k)}) f(\mathbf{x}^{(k)}).$$

By taking  $\Delta t = 1$ , we simply have Newton's method

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{J}_f^{-1}(\mathbf{x}^{(k)}) f(\mathbf{x}^{(k)}).$$

Like Newton's method and other fixed-point methods, the Euler method has stability restrictions. One way to handle the stability restrictions is by taking a

smaller step size  $\Delta t$ . Smaller steps, of course, mean more iterations. Homotopy continuation, too, effectively reduces the step size to maintain stability. In Chapter 12, we examine the forward Euler method and other differential equation solvers, many of which 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)

We find the relative minima and maxima of a function by determining where its derivatives are zero. It shouldn’t be surprising that numerical methods for finding a function’s extrema are similar to those for finding a function’s zeros.

### ► Derivative-free methods

The bisection method, introduced at 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 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 the 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 the 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  $x$ . The relation  $1 : (1 - x) = (1 - x) : x$  maintains equal scaling at every iteration. 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 vertices of the polytope. 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 (8.11) 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  $\alpha$ .

Computing the Hessian can be costly, particularly when working with many 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 entirely by using a general class of gradient descent methods. Gradient descent methods are standard in machine learning and neural nets, where the number of variables can be pretty 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  $\alpha$ , 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  $\alpha$  to be too small, you'll frequently check your gradient and your progress will be slow. If  $\alpha$  is too

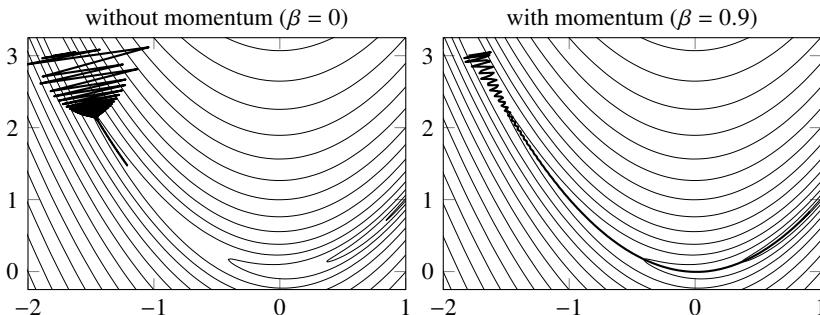


Figure 8.9: The gradient descent method with a learning rate  $\alpha = 10^{-3}$  on the Rosenbrock function for 500 iterations without and with momentum.

large and the valley is relatively narrow, you'll likely 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  $\beta$  is a parameter usually between zero and one. 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 valley floor, these tiny steps make for excruciatingly slow progress.

The following code uses the gradient descent method to find the minimum of the Rosenbrock function:

```

∇f = x -> [-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

```

Figure on the facing page shows the trace 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)

```

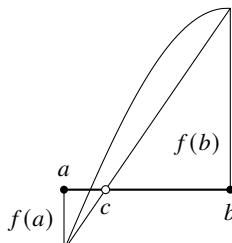
Methods include NelderMead (the default when no derivatives are provided), Newton, BFGS, GradientDescent, and GoldenSection among others. ▶

•• 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. A straightforward way to do this is by linearly interpolating between  $a$  and  $b$  rather than by averaging  $a$  and  $b$ :

$$c = \frac{f(a)b - f(b)a}{f(a) - f(b)}.$$



The *regula falsi* method subdivides brackets the same way. 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. Newton's method converges linearly at a root of multiplicity  $n$ . Modify it so that it converges quadratically. ↗

8.4. 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. 

8.5. 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.

8.6. Prove that, for sufficiently smooth functions, the order of convergence for the secant method is the golden ratio  $p = (1 + \sqrt{5})/2$ . 

8.7. The Leibniz series, sometimes called the Gregory–Leibniz series or the Madhava–Leibniz series,  $4 \cdot (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots)$  converges sublinearly to  $\pi$ . It's slow. You would need a trillion terms to accurately compute the first 13 digits of  $\pi$ . Compare the Leibniz series and Aitken's extrapolation (8.7) of it.<sup>10</sup> What happens if you use the formulation (8.6) of Aitken's extrapolation? 

8.8. Show that Steffensen's method (8.8) is a quasi-Newton method. Show that this method is quadratically convergent under suitable conditions. 

8.9. Use the Newton–Horner method to compute  $x^{(1)}$  given  $x^{(0)} = 4$  for the polynomial:  $p(x) = 3x^5 - 7x^4 - 5x^3 + x^2 - 8x + 2$ .

8.10. Suppose that we want to solve  $f(x) = 0$ . Taking  $x = x - \alpha f(x)$  for some  $\alpha$ , we get the fixed-point method  $x^{(k+1)} = \phi(x^{(k)})$  where  $\phi(x) = x - \alpha f(x)$ . How does  $\alpha$  affect the stability and convergence of the method? 

8.11. Derive the Newton iteration step of the Q\_rsqrt algorithm on page 182.

8.12. Use Newton's method, Broyden's method, and homotopy continuation to find the intersection of

---

<sup>10</sup>Aitken's extrapolation applied to a sequence of partial sums is known as Shanks' transformation.

$$(x^2 + y^2)^2 - 2(x^2 - y^2) = 0$$
$$(x^2 + y^2 - 1)^3 - x^2 y^3 = 0$$



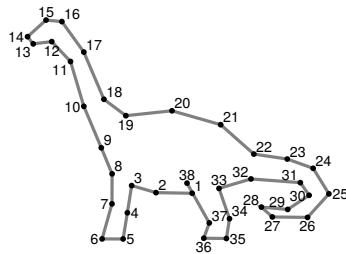
8.13. The elliptic curve  $\text{secp256k1}$   $y^2 = x^3 + 7 \pmod{r}$  was popularized when Satoshi Nakamoto used it in Bitcoin's public-key digital signature algorithm. Implement the elliptic curve Diffie–Hellman (ECDH) algorithm using  $\text{secp256k1}$ . The order of the field is the prime number  $r = 2^{256} - 2^{32} - 977$ , 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>.

8.14. 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 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 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 significant 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

$$y_k = p_{k-1}(x_k) + c(x_k - x_0)(x_k - x_1) \cdots (x_k - x_{k-1}).$$

Canonical:  $\varphi_k(x) = x^k$

Langrange:  $\ell_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^n \frac{x - x_j}{x_k - x_j}$

Newton:  $v_k(x) = \prod_{j=0}^{k-1} (x - x_j)$

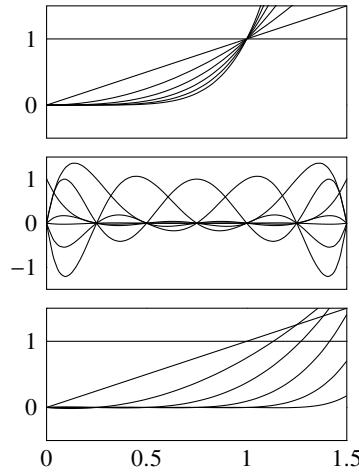


Figure 9.1: Polynomial basis functions for  $k = 0, \dots, 6$  with nodes at  $x_j = j/4$ .

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. Moreover, the Vandermonde matrix is increasingly ill-conditioned as it gets larger.

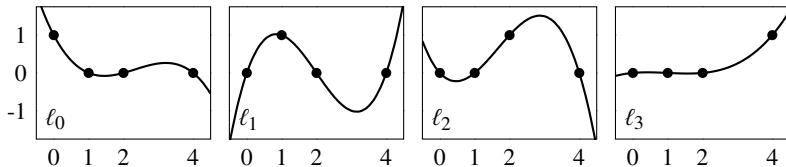
It's easy to construct a Vandermonde matrix using broadcasting:  $x.^{[0:n]}'$ . For something with more features, try the SpecialMatrices.jl function `Vandermonde` that overloads the \ operator with the  $O(n^2)$ -time Björck and Pereyra algorithm.

## ► 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 from scratch. And the computations can be numerically unstable.

## ► Newton polynomial basis

The Newton polynomial basis fixes 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  $x_0, x_1, \dots, x_i$ . We will use the notation  $c_i = f[x_0, x_1, \dots, x_i]$  to denote this dependency. For example, using forward substitution to find  $\{c_0, c_1, c_2\}$  for three nodes

$$\begin{aligned} f[x_0] &= c_0 = f(x_0) \\ f[x_0, x_1] &= c_1 = \frac{f(x_1) - c_0}{x_1 - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\ f[x_0, x_1, x_2] &= c_2 = \frac{f(x_2) - c_0 - c_1(x_2 - x_0)}{(x_2 - x_1)(x_2 - x_0)} \\ &\quad = \frac{\frac{f(x_2) - f(x_0)}{x_2 - x_0} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_1}. \end{aligned}$$

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 calculation<sup>1</sup>

$x_0$	$f[x_0]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, x_1, x_2, x_3]$
$x_1$	$f[x_1]$	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$	
$x_2$	$f[x_2]$	$f[x_2, x_3]$		
$x_3$	$f[x_3]$			

<sup>1</sup>Nineteenth-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.

$$\text{From } \begin{array}{r|rrrr} 1 & 0 & \square & \square & \square \\ 3 & 4 & \square & \square & \\ 5 & 5 & \square & & \\ 6 & 7 & & & \end{array} \quad \text{we have } \begin{array}{r|rrr} 1 & 0 & 2 & -\frac{3}{8} \\ 3 & 4 & \frac{1}{2} & \frac{1}{2} \\ 5 & 5 & 2 & \\ 6 & 7 & & \end{array}.$$

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  $\frac{1}{2}$ . Now, we can use these new values to compute to unknown value to their right:  $(\frac{1}{2} - 2)/(5 - 1) = -\frac{3}{8}$ . The rest of the table is filled out similarly. From the first row, we have the coefficients of the interpolating polynomial

$$p_n(x) = 0 + 2(x - 1) - \frac{3}{8}(x - 1)(x - 3) + \frac{7}{40}(x - 1)(x - 3)(x - 5),$$

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

Let's continue our work from the previous section by constructing interpolating polynomials that match a function's values *and* derivatives at a set of 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  $\xi$  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  $\xi$  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	$ 123\square\square\square 12\square\square\square 2674 26\square 26 $	1	2	3	$\square$	$\square$	$\square$	1	2	$\square$	$\square$	$\square$		2	6	7	4			2	6	$\square$				2	6					we have	$ 12312-1 12431 2674 26\square 26 $	1	2	3	1	2	-1	1	2	4	3	1		2	6	7	4			2	6	$\square$				2	6					
1	2	3	$\square$	$\square$	$\square$																																																											
1	2	$\square$	$\square$	$\square$																																																												
2	6	7	4																																																													
2	6	$\square$																																																														
2	6																																																															
1	2	3	1	2	-1																																																											
1	2	4	3	1																																																												
2	6	7	4																																																													
2	6	$\square$																																																														
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  $\xi$ . 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

$$f(x) - p_n(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \prod_{i=0}^n (x - x_i)^2.$$

From the expression for pointwise error, we can see 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:



The value  $|\prod_{i=0}^8 (x - x_i)|$  is quite large near both ends of the interval. 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 norm<sup>2</sup>  $\|\prod_{i=0}^8 (x - s_i)\|_\infty$ . We do exactly this 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  $s_i$  overlays the lighter gray curve of  $\prod_{i=0}^8 (x - x_i)$  with uniformly spaced nodes  $x_i$ . In exchange for moderately increasing the error near the center of the interval, using Chebyshev nodes significantly decreases the error near its ends.

Let's 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, Forman Acton interprets it in a parenthetical comment in 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:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x),$$

---

<sup>2</sup>Also called the Chebyshev norm, the infinity norm, and the supremum norm.

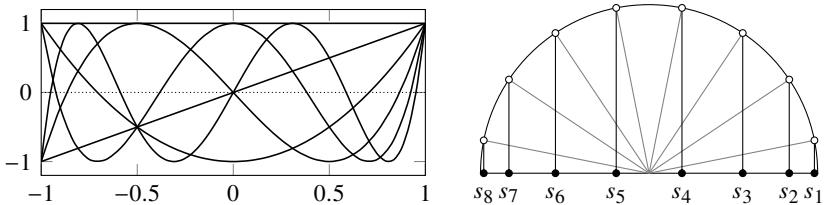


Figure 9.2: Chebyshev polynomials  $T_0(x), T_1(x), \dots, T_5(x)$  (left). Chebyshev nodes of  $T_8(x)$  are abscissas of eight equispaced nodes on the unit circle (right).

where  $T_0(x) = 1$  and  $T_1(x) = x$ . The roots of the Chebyshev polynomial, called Chebyshev nodes, occur at

$$s_i = \cos \frac{(i - \frac{1}{2})\pi}{n} \quad \text{for } i = 1, 2, \dots, n.$$

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

$$\hat{s}_i = \cos \frac{i\pi}{n} \quad \text{for } i = 0, 1, \dots, n.$$

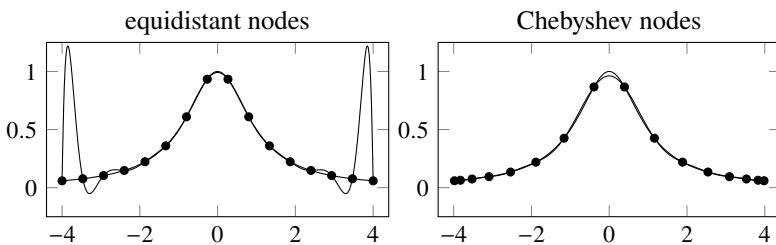
and take the values  $T_n(\hat{s}_i) = (-1)^{i+1}$ . It follows that Chebyshev polynomials are bounded below by  $-1$  and above by  $+1$  for  $x \in [-1, 1]$ . Finally, the leading coefficient of  $T_n(x)$  is  $2^{n-1}$ .

**Theorem 28.** *The maximum magnitude of any degree- $n$  polynomial with a leading coefficient  $2^{n-1}$  must be at least one over the interval  $[-1, 1]$ .*

*Proof.* Let  $p_n$  be a degree- $n$  polynomial with a leading coefficient  $2^{n-1}$ . Then  $T_n - p_n$  is a degree- $(n-1)$  or smaller polynomial. 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  $T_n(\hat{s}_i) = +1$  for odd  $i$ , 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  $T_n(\hat{s}_i) = -1$  for even  $i$ , 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  $n+1$  Chebyshev extrema. 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  $x \in [-1, 1]$  where  $|p_n(x)| \geq 1$ .  $\square$

**Example.** In 1885, at the age of seventy, Karl Weierstrass proved his famous theorem that any continuous function could 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

positive  $\varepsilon$ , 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 increase the pointwise error around the origin. The function  $(x^2 + 1)^{-1}$  is called the Runge function or the witch of Agnesi.<sup>3</sup> ▶

### 9.3 Splines

One problem with polynomial interpolation is that polynomials basis elements are smooth and unbounded, so they are ill-suited to approximate functions with predominant local features like discontinuities. We've seen how polynomials lead to Runge's phenomenon. Radial basis functions, examined in exercise 9.3, provide an alternative to polynomial basis functions. Piecewise polynomial functions, called *splines*, provide another. This section 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

---

<sup>3</sup>The witch of Agnesi was named after 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* ("versed sine curve") as *aversiera* ("witch"). The versine (or versed sine) function is  $\text{versin } \theta = 1 - \cos \theta$ . It's one of several oddball trigonometric functions rarely used in modern mathematics: versine, coversine, haversine, archacovercosine, . . . .

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. We have two degrees of freedom remaining.

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 a shape that minimizes the energy, which 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 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

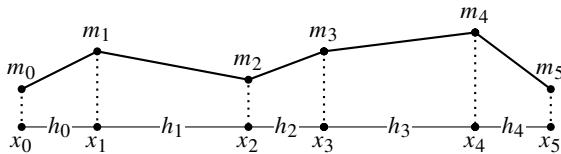
$$\begin{aligned} \int_a^b s'''(y' - s') \, dx &= \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} d_i(y' - s'_i) \, dx \\ &= \sum_{i=0}^{n-1} d_i [(y(x_{i+1}) - s_i(x_{i+1})) - (y(x_i) - s_i(x_i))] = 0. \quad \square \end{aligned}$$

## ► 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  $s_i(x_{i+1}) = y_{i+1}$ :

$$y_i = \frac{1}{6} \frac{m_i}{h_i} h_i^3 + B_i \quad \text{and} \quad y_{i+1} = \frac{1}{6} \frac{m_{i+1}}{h_i} h_i^3 + A h_i + B_i.$$

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

$$h_{i-1}m_{i-1} + 2(h_i + h_{i-1})m_i + h_im_{i+1} = 6 \left( \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right)$$

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

$$\alpha_i = h_i, \quad \beta_i = 2(h_i + h_{i-1}), \quad \text{and} \quad \gamma_i = 6 \left( \frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}} \right). \quad (9.5)$$

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  $t_i = \sum_{j=1}^i \ell_j$ . 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)
```

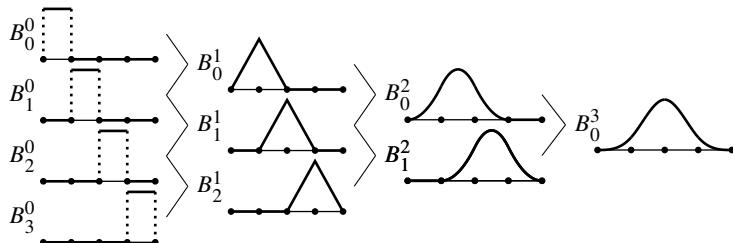


Figure 9.3: B-spline bases generated recursively using de Boor's algorithm.

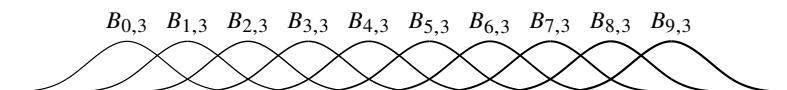
```
x_i = 0:5; y_i = g(x_i)
x = LinRange(0,5,101);
```

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)
```

## ► 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 recursively generate splines of any degree, 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:

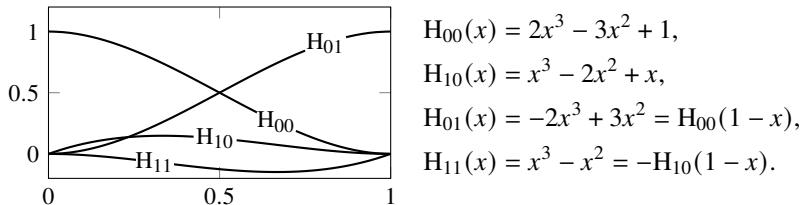
```
using Interpolations
method = BSpline(Cubic(Natural(OnGrid())))
spline = scale(interpolate(yi, method), xi)
plot(x,spline(x)); plot!(x,g(x)); scatter!(xi,yi)
```

### ► Cubic Hermite spline

Another type of cubic spline is a piecewise cubic Hermite interpolating polynomial or PCHIP. Cubic Hermite polynomials come from a linear combination of four basis functions, each one having 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



$$\begin{aligned} H_{00}(x) &= 2x^3 - 3x^2 + 1, \\ H_{10}(x) &= x^3 - 2x^2 + x, \\ H_{01}(x) &= -2x^3 + 3x^2 = H_{00}(1-x), \\ H_{11}(x) &= x^3 - x^2 = -H_{10}(1-x). \end{aligned}$$

We can modify these bases to fit an arbitrary interval  $[x_i, x_{i+1}]$  by taking

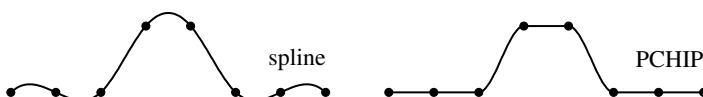
$$p(x) = p_i H_{00}(t) + m_i h_i H_{10}(t) + p_{i+1} H_{01}(t) + m_{i+1} h_i H_{11}(t)$$

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

$$\frac{p_{i+1} - p_{i-1}}{x_{i+1} - x_{i-1}} \quad \text{or} \quad \frac{1}{2} \left( \frac{p_{i+1} - p_i}{x_{i+1} - x_i} + \frac{p_i - p_{i-1}}{x_i - x_{i-1}} \right),$$

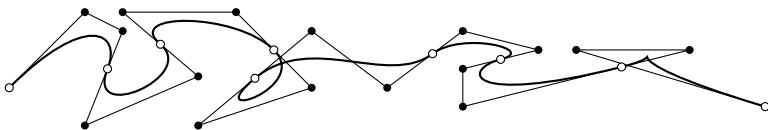
using a one-sided derivative at the boundaries.

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. A Bézier curve doesn't go through control points except at the endpoints. A Bézier spline is built by matching the position and slopes of several Bézier curves at their terminal control points (○):



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  $\mathbf{q}_{10}(1) = \mathbf{p}_1$ . With 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 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

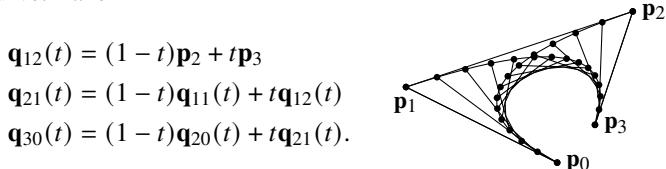
$$\mathbf{q}_{20}(t) = (1-t)^2 \mathbf{p}_0 + 2t(1-t) \mathbf{p}_1 + t^2 \mathbf{p}_2$$

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



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 code 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

$$\mathbf{q}_{n0}(t) = \sum_{k=0}^n \binom{n}{k} (1-t)^{n-k} t^k \mathbf{p}_k = \sum_{k=0}^n B_{nk}(t) \mathbf{p}_k,$$

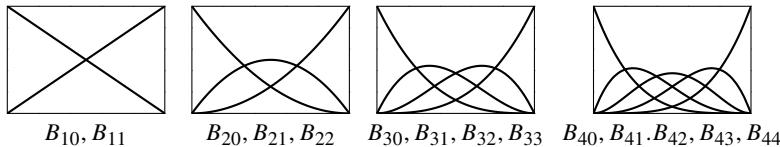
or simply  $\mathbf{q}_{n0}(t) = \mathbf{B}_n(t) \mathbf{p}$ , where  $\mathbf{B}_n(t)$  is an  $m \times (n+1)$  Bernstein matrix constructed by taking  $m$  points along  $[0, 1]$  and  $\mathbf{p}$  is an  $(n+1) \times 2$  vector of control points. We can build a Bernstein matrix using the following function, which takes a column vector such as  $\mathbf{t} = \text{LinRange}(0, 1, 20)$ :

```
bernstein(n, t) = @. binomial(n, 0:n)' * t^(0:n)' * (1-t)^(n:-1:0)'
```



animation of the cubic  
Bézier curve

The bases  $\{B_{n0}(t), B_{n1}(t), \dots, B_{nn}(t)\}$  form a partition of unit, meaning 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

Rénault engineer Pierre Bézier popularized Bézier curves 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 of the Weierstrass theorem. We'll conclude the chapter with Bernstein's proof. The proof is elegant, and it provides a nice segue to the material in the next chapter.

It's no coincidence that the terms of the Bernstein polynomial bear a 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 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  $\delta$  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 the 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. 

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  $\phi$  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 by considering candidate solutions from a finite-dimensional space, determined by a set of points called collocation points. The finite-dimensional problem is 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 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?

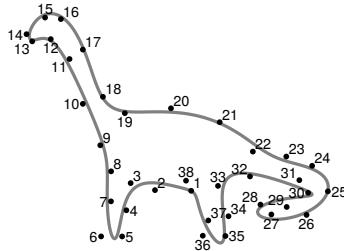




## Chapter 10

---

# Approximating Functions



The last chapter used interpolation to find polynomials that coincide with a function at a set of nodes or knots. These methods allowed us to model data and functions using smooth curves. This chapter considers *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  $L^\infty$ -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  $\alpha$  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  $(f, g) = (g, f)$  for a real vector space and for  $(f, g) = \overline{(g, f)}$  for a complex one.

A vector space with an assigned inner product is 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.

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  $e^{-x^2}$  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  $g \in U$

$$\begin{aligned}\|f - g\|^2 &= \|(f - u) + (u - g)\|^2 \\ &= \|f - u\|^2 + 2\underbrace{(f - u, f - g)}_0 + \|u - g\|^2 \geq \|f - u\|^2.\end{aligned}$$

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  $2(f - u, g) + \lambda\|g\|^2 \geq 0$ . Letting  $\lambda$  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  $i = 1, 2, \dots, n$ :

$$(\varphi_j, \varphi_i) = (x^j, x^i) = \int_0^1 x^{i+j} dx = \frac{1}{i+j+1}.$$

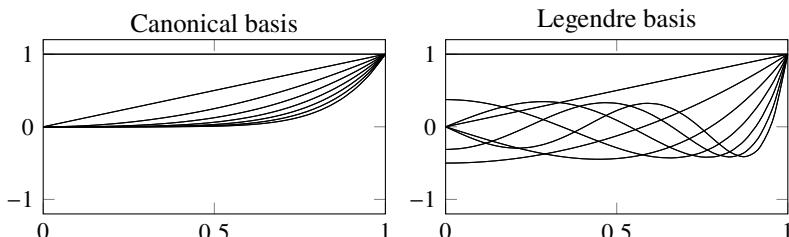
We now must solve the normal equation

$$\begin{bmatrix} 1 & 1/2 & 1/3 & \cdots \\ 1/2 & 1/3 & 1/4 & \cdots \\ 1/3 & 1/4 & 1/5 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (f, 1) \\ (f, x) \\ \vdots \\ (f, x^n) \end{bmatrix}$$

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  $7 \times 7$  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  $12 \times 12$  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 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. ◀

## 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 major tool for 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  $F$ . Directly, applying the Gram–Schmidt process can be a lot of work. Luckily, there's a shortcut to generate orthogonal polynomials. We can use a three-term recurrence relation known as Favard's theorem.<sup>1</sup>

**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*

$$\begin{cases} p_{n+1}(x) = (x + a_n) p_n(x) + b_n p_{n-1}(x) \\ p_0(x) = 1, \quad p_1(x) = x + a_1 \end{cases}$$

---

<sup>1</sup>Jean 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 whose degree is 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

$$a_n = c_n = -\frac{(xp_n, p_n)}{(p_n, p_n)}, \quad b_n = c_{n-1} = -\frac{(p_n, p_n)}{(p_{n-1}, p_{n-1})}. \quad \square$$

**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  $b_n = -(n^2 - 1)/(4n^2 - 1)$  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 inner product  $(f, g) = \frac{1}{2\pi} \int_0^{2\pi} f(x) \overline{g(x)} dx$ .*

*Proof.* We'll show that  $(\phi_k, \phi_m) = \delta_{km}$ .

$$\begin{aligned} (\phi_k, \phi_m) &= \frac{1}{2\pi} \int_0^{2\pi} e^{ikx} e^{-imx} dx = \frac{1}{2\pi} \int_0^{2\pi} e^{i(k-m)x} dx \\ &= \begin{cases} \frac{1}{2\pi} \int_0^{2\pi} 1 dx = 1, & \text{if } k = m \\ \frac{1}{2\pi} \left[ \frac{e^{i(k-m)x}}{i(k-m)} \right]_0^{2\pi} = \frac{1}{2\pi} \frac{1}{i(k-m)} [1 - 1] = 0, & \text{if } k \neq m \end{cases} \end{aligned}$$

So,  $(e^{ikm}, e^{imx}) = \delta_{km}$ . □

Because the Fourier polynomials are orthogonal, there is a projection operator

$$P_n f = \sum_{k=-n}^n c_k e^{ikx} \quad \text{with} \quad c_k = (f, e^{ikx}) = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-ikx} dx. \quad (10.2)$$

The projection  $P_n f$  is called the truncated Fourier series, the set of coefficients  $\{c_k\}$  is called the Fourier coefficients, and the inner product  $(f, \phi_k)$  is 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<sup>2</sup>  $(f, g) = \frac{1}{n} \sum_{j=0}^{n-1} f(x_j) \overline{g(x_j)}$ .*

---

<sup>2</sup>The bilinear form  $(f, g) = \frac{1}{n} \sum_{j=0}^{n-1} f(x_j) \overline{g(x_j)}$  is technically a *pseudo-inner product*. A pseudo-inner product satisfies all requirements to be an inner product except definiteness (nondegeneracy). The bilinear form  $(f, g) = \frac{1}{n} \sum_{j=0}^{n-1} f(x_j) \overline{g(x_j)}$  is degenerate because  $(f, g) = 0$  for any  $f(x)$  that is zero at all nodes  $2\pi j/n$ , like  $\sin nx$ . The corresponding “norm” is a *semi-norm*.

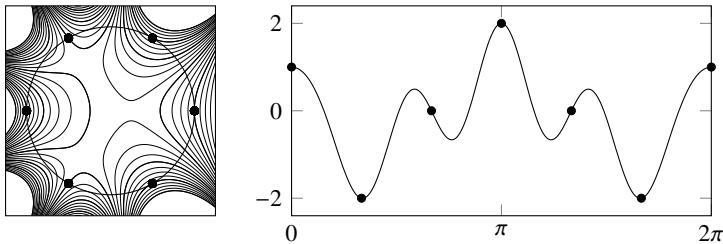


Figure 10.1: A polynomial over the complex plane  $\sum_{k=0}^{n-1} c_k z^k$  restricted to the unit circle is a Fourier polynomial  $\sum_{k=0}^{n-1} c_k e^{2\pi i k/n}$ .

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

$$P_n f(x_j) = \sum_{k=0}^{n-1} c_k e^{ijk} \quad \text{with} \quad c_k = (f, e^{ikx}) = \frac{1}{n} \sum_{j=0}^n f(x_j) e^{-ijk}. \quad (10.3)$$

We call this the *inverse discrete Fourier transform*.

### ► 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  $j = 0, 1, \dots, n-1$ . Take the exponential basis functions  $\phi_k(x) = e^{ikx}$ . Then the interpolating Fourier polynomial satisfies the system of equations

$$f(x_j) = \sum_{j=0}^{n-1} c_k \phi_k(x_j) = \sum_{j=0}^{n-1} c_k e^{i2\pi jk/n} = \sum_{j=0}^{n-1} c_k \omega^{jk} \quad (10.4)$$

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

$$p(z) = c_0 + c_1 z + c_2 z^2 + \dots + c_{n-1} z^{n-1}$$

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 around the unit circle—i.e., take  $z_k = \omega^k = e^{i2\pi k/n}$ , running 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 the preceding page.

### ► Approximation error

Let's finally examine the behavior of the Fourier coefficients for the continuous inner product

$$c_k = (f, e^{ikx}) = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-ikx} dx,$$

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

$$(f', e^{ikx}) = \frac{1}{2\pi} \int_0^{2\pi} f'(x) e^{-ikx} dx = \frac{ik}{2\pi} \int_0^{2\pi} f(x) e^{-ikx} dx = ik c_k$$

because  $f(x)$  is periodic. In general, we have  $(f^{(p)}, e^{ikx}) = (ik)^p c_k$  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,

$$\frac{1}{2\pi} \int_0^{2\pi} |f^{(p)}(x)|^2 dx = \sum_{k=1}^{\infty} |k^p c_k|^2.$$

The series converges only if  $|k^p c_k|^2 = o(k^{-1})$  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

$$\|f - P_n f\|^2 = \|P_\infty f - P_n f\|^2 = \sum_{k>|n|} |c_k|^2 = \sum_{k>|n|} o(k^{-2p-1}) = o(n^{-2p}).$$

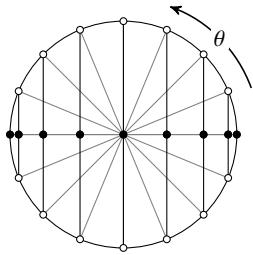
So the error is  $o(n^{-p})$ . □

Note that if  $f$  is smooth (with infinitely many continuous derivatives), then the error is  $o(n^{-p})$  for all integers  $p$ . The convergence of the Fourier series will be faster than any polynomial order. In other words, the Fourier series is exponentially convergent.

## 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. We can efficiently do all of this 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.<sup>3</sup>

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  $i \neq j$ ,

$$d_{ij} = \ell'_i(x_j) = \left( \prod_{\substack{k=0 \\ k \neq i}}^n \frac{x_j - x_k}{x_i - x_k} \right) \left( \sum_{\substack{k=0 \\ k \neq i}}^n \frac{1}{x_i - x_k} \right) = \frac{1}{x_i - x_j} \prod_{\substack{k=0 \\ k \neq i, j}}^n \frac{x_j - x_k}{x_i - x_k}.$$

We'll calculate the numerator—the denominator is almost the same:

$$\begin{aligned} \prod_{\substack{k=0 \\ k \neq i, j}}^n (x - x_k) &= \frac{\prod_{k=0}^n (x - x_k)}{(x - x_i)(x - x_j)} \\ &= \frac{(x^2 - 1) \prod_{k=1}^{n-1} (x - x_k)}{(x - x_i)(x - x_j)} \\ &= \frac{(x^2 - 1) T'_n(x)}{(x - x_i)(x - x_j) 2^{n-1}}. \end{aligned}$$

---

<sup>3</sup>The book is also available at <http://www.chebfun.org/ATAP> as a set of m-files consisting of  $\text{\LaTeX}$  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 take the  $n$ th derivative of 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  $\mathbf{u}$ .

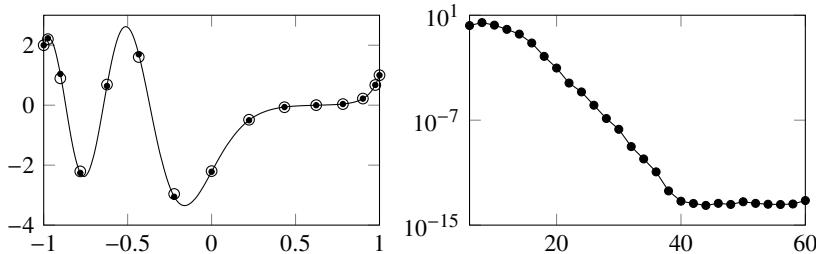


Figure 10.2: Left: Numerical solution ( $\circ$ ) and exact solution ( $\cdot$ ) to the Airy equation (10.7). Right:  $\ell^\infty$ -error as a function of the number of Chebyshev points.

```
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,  $D^2\mathbf{u}$  is the second derivative of  $u(x)$ . We can compute the antiderivative of  $u(x)$  by taking  $D^{-1}\mathbf{u}$  with a constraint. 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=:o)
```

How accurate is the solution? Figure 10.2 above shows a semi-log plot of the  $\ell^\infty$ -error as a function of the number of nodes. The solution is already quite good with just 15 points. After that, the solution gains one digit of accuracy for every two points added. 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 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 with operators using Chebyshev polynomial approximations, such as integration (using Clenshaw–Curtis quadrature) and differentiation. The backslash operator is overloaded to solve differential equations. Zeros are computed using a “colleague” matrix, the analog to the companion matrix for Chebyshev polynomial roots.<sup>4</sup>

• 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 principal difference is that while Fourier polynomials are only localized in frequency, wavelets are localized in both space and frequency. While Fourier polynomials achieve orthogonality by scaling 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.<sup>5</sup> 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 Gilbert Strang’s “Wavelets and

---

<sup>4</sup>The matrix analog to the companion matrix for finding the roots of a general orthogonal polynomial is called a “comrade” matrix.

<sup>5</sup>Mathematician Laurent Duval has a compendium of over one hundred “starlets,” or wavelets ending in “let.”

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.<sup>6</sup> 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.<sup>7</sup>

What is multiscale or multiresolution approximation intuitively? 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 another 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 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:

$$\begin{array}{c} 4, 20 \\ 12 \mp 8 \end{array}$$

---

<sup>6</sup>Wavelets were originally called *ondelettes* by the French pioneers Morlet and Grossman after a term frequently used in geophysics for an oscillating localized function.

<sup>7</sup>Electrical 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, we can write the scaled function  $\phi(x/2) \in V_{-1} \subset V_0$  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

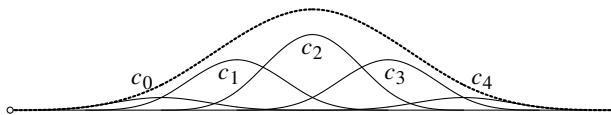
the dilation equation, then

$$\int_{-\infty}^{\infty} \phi(x) dx = \int_{-\infty}^{\infty} \sum_{k=-\infty}^{\infty} c_k \phi(2x - k) dx = \sum_{k=-\infty}^{\infty} c_k \int_{-\infty}^{\infty} \frac{1}{2} \phi(s) ds,$$

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 the 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, limiting 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  $\sum_{k=-\infty}^{\infty} c_k = 2$ .

**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  $2\pi$ -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, leaving us 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  $\xi$ , 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 another 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.  $\blacktriangleleft$

In practice, we don't ever need to construct the scaling functions explicitly, 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, eighth-integer values, and so on to get  $\phi(x)$  at dyadic rational values  $x = i/2^j$ . Here's a naïve implementation:

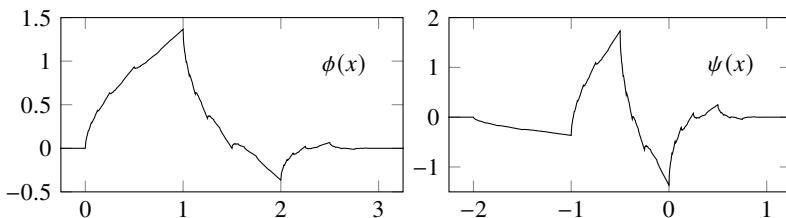
```
using OffsetArrays
function scaling(c,ϕₖ,n)
    m = length(c)-1; ℓ = 2^n
    ϕ = OffsetVector(zeros(3*m*ℓ), -m*ℓ)
    k = (0:m)*ℓ
    ϕ[k] = ϕₖ
    for j = 1:n
        for i = 1:m*2^(j-1)
            x = (2i-1)*2^(n-j)
            ϕ[x] = c .+ ϕ[2x .- k]
        end
    end
    ((0:m*ℓ-1)/ℓ, ϕ[0:m*ℓ-1])
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

Figure 10.3: Daubechies  $D_4$  scaling function  $\phi$  and wavelet function  $\psi$ .

right choice. Instead, 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+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)
```

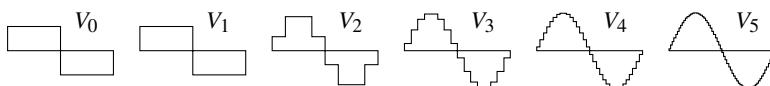
We can define normalized bases functions in  $V_n$  as

$$\phi_{nk}(x) = 2^{n/2}\phi(2^n x - k).$$

Furthermore, we can define an orthogonal projection of a function  $f \in L^2$ :

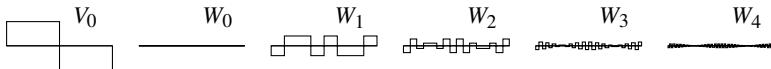
$$P_n f = \sum_{k=-\infty}^{\infty} (f, \phi_{nk}) \phi_{nk} = \sum_{k=-\infty}^{\infty} a_{nk} \phi_{nk} \quad (10.12)$$

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  $f_{n+1} \in V_{n+1}$ , 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

$$Q_n f = \sum_{k=-\infty}^{\infty} (f, \psi_{nk}) \psi_{nk} = \sum_{k=-\infty}^{\infty} d_{nk} \psi_{nk}$$

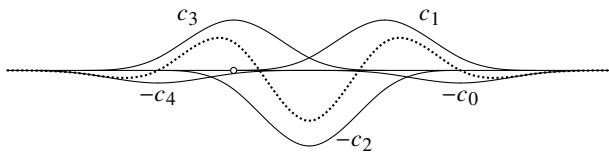
for  $d_{nk} = (f, \psi_{nk})$  and basis elements  $\psi_{nk} \in W_n$ :

$$\psi_{nk}(x) = 2^{n/2} \psi(2^n x - k).$$

Specifically, we have  $Q_0 f = (P_1 - P_0)f$ . 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 of the 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 266:



Using the scaling function  $\phi$  from the code on page 269, we can compute the wavelet function  $\psi$  with

```

 $\psi = \text{zero}(\phi); n = \text{length}(\phi)-1; \ell = \text{length}(\psi)/2n$ 
 $\text{for } k=0:n$ 
 $\quad \psi[(k*\ell+1):(k+n)*\ell] += (-1)^k * c[n-k+1] * \phi[1:2:end]$ 
 $\text{end}$ 

```

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

$$\phi_{nk}(x) = 2^{n/2} \sum_{i=-\infty}^{\infty} c_i \phi(2^n x - 2^n k - i) = \frac{1}{\sqrt{2}} \sum_{j=-\infty}^{\infty} c_{j-2k} \phi_{n+1,j}(x),$$

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

$$d_{n+1,k} = \frac{1}{\sqrt{2}} \sum_{i=-\infty}^{\infty} c_{k-2i} a_{ni} - \frac{1}{\sqrt{2}} \sum_{i=-\infty}^{\infty} (-1)^{1-k+2i} c_{1-k+2i} d_{ni}.$$

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

$$\mathbf{a}_{n+1} = [\mathbf{H}^\top \quad \mathbf{G}^\top] \begin{bmatrix} \mathbf{a}_n \\ \mathbf{d}_n \end{bmatrix} \quad \text{or alternatively} \quad [\mathbf{H} \quad \mathbf{G}] \mathbf{a}_{n+1} = \begin{bmatrix} \mathbf{a}_n \\ \mathbf{d}_n \end{bmatrix}.$$

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. 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  $\frac{1}{2}(c_0^2 + c_1^2 + c_2^2 + c_3^2) = 1$  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.

## ► Image and data compression

One application of discrete wavelet transforms is image compression. The JPEG 2000 format was developed in the late 1990s as a replacement for 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



Figure 10.4: Discrete wavelet transform (right) of the image (left).

low edge-compression artifacts are essential. ICER, a similar wavelet-based 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 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 because the JPEG format already worked quite well.

Let's examine the two-dimensional DWT of a  $512 \times 512$ -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, FileIO
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 265

$$\{12\} \mp \{8\} \mp \{4, 2\} \mp \{-2, 2, -2, 2\} \mp \{1, 1, 1, 1, 1, -1, 0, 1\}.$$

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

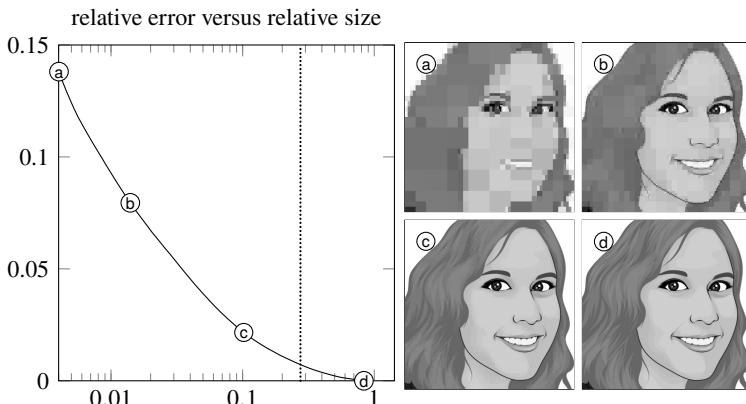


Figure 10.5: The relative error of a DWT (Haar) compressed image as a function of the file size. The dotted line shows equivalent lossless PNG compression.

finer detail with each subsequent generation of daughter wavelets. Figure 10.4 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  $2 \times 2$  pixels details. Then  $4 \times 4$ , and so on until we get to  $256 \times 256$ -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, FileIO
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 nonzero elements, is plotted in Figure 10.4 on the previous page. Also, see the QR code at the bottom of this page. Compare the DWT image compression with the DCT image compression in Figure 6.6 on page 164. DWT image compression maintains sharp edges without the ringing or nonlocal artifacts characteristic of DCT image compression. Even at high levels of compression, sharp details are well preserved. This feature, in particular, makes DWTs 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.<sup>8</sup> 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 do to birds.<sup>9</sup>

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

---

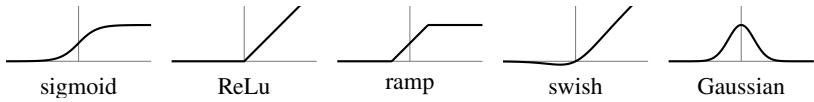
<sup>8</sup>Bullied as a boy, Walter Pitts spent hours hiding in the library, reading books like Whitehead and Russell's *Principia Mathematica*. When he was 15, he ran away from home and showed up at the University of Chicago, where Bertrand Russell was teaching. He remained at the university for several 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*.

<sup>9</sup>Biomimicry 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 Orville and Wilbur Wright engineered the first successful powered aircraft, it looked very little like a bird other than having wings.



an image compressed  
using Haar wavelets

input values by stretching and sliding the activation function along the  $x$ -axis. The second affine transformation affects the output values by stretching and sliding the activation function in the  $y$ -direction. An activation function 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  $\lfloor \cdot \rfloor$  for an activation function to model neurons as logical operators. Sigmoid activation functions  $\sim$ , which naturally arise from logistic regression, were the most popular until fairly recently. Now, rectified linear units or ReLUs  $\swarrow$  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  $x^+ = \text{ReLU}(x) = \max(0, x)$   $\swarrow$ . Combining two ReLUs  $(x+1)^+ - x^+$  will make a ramp function  $\overbrace{\phantom{...}}$ , 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  $\ell$  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  $\ell/n$ . 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 232, regarding polynomial interpolation error, on each subinterval

$$|f_n(x) - f(x)| = \frac{1}{2} f''(\xi_i)(x - x_i)(x_{i+1} - x) \leq \frac{1}{8} |f''(\xi)| h^2$$

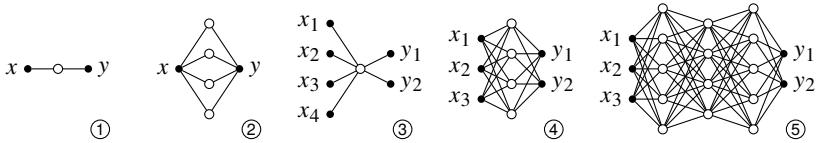


Figure 10.6: Neural networks: ① a simple artificial neuron; ② a simple neural net; ③ a multiple-input/multiple-output neuron; ④ a single layer neural net; and ⑤ a deep neural net.

for some  $\xi_i$  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  $n \geq \ell\sqrt{|f''(\xi)|/8\varepsilon}$ .  $\square$

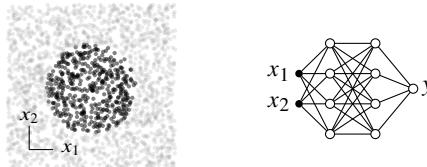
The function  $f(x)$  need not be smooth—the proof of a similar result for Lipschitz functions is left as an exercise. George Cybenko’s influential 1989 paper “Approximation by superpositions of a sigmoidal function” generalizes the universal approximation theorem for an arbitrary sigmoid function. From 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 denote 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

$$\mathbf{y} = \mathbf{W}_k\phi(\mathbf{W}_{k-1}\phi(\cdots\phi(\mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x}))\cdots)).$$

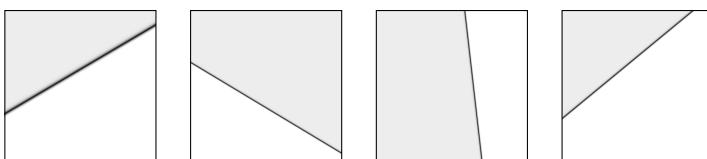
While McCulloch and Pitts forged their neural nets out of the mathematical certainty of formal logic, modern neural nets reign as tools of statistical uncertainty and inference. Deep neural networks are able to classify immense and complicated data sets, thanks in large part to specialized computing hardware and stochastic

optimization techniques that allow them to 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.

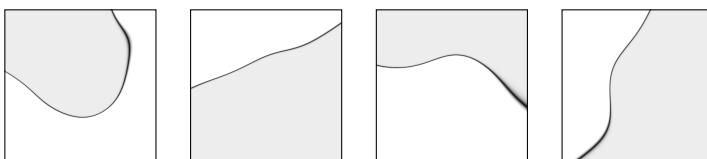


Let's partition the points using the neural net  $y = \phi(\mathbf{W}_3\phi(\mathbf{W}_2\phi(\mathbf{W}_1\mathbf{x})))$ , with two inputs, two hidden layers (each with four neurons), and one output for the label. We'll use  $\phi(x) = \tanh x$  as the activation function.

A single neuron partitions the plane with a line, like a crisp fold in a sheet of paper, called a decision boundary.<sup>10</sup> Each of the four neurons in the first hidden layer produces different partitions in the plane—four separate folds in the paper.



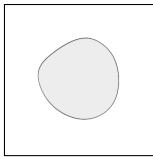
When combined linearly together, these decision boundaries partition a space into a simple, possibly unbound polytope. In a deep neural net, these initial partitions are the building blocks for more complicated geometries with each subsequent layer. The second hidden layer forms curved boundaries by 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 longer, the blob would have been more circular.

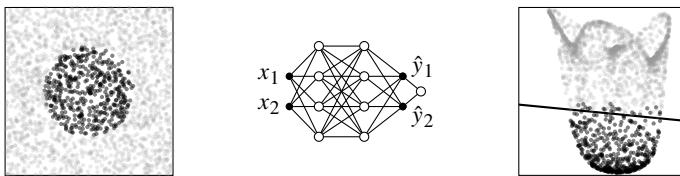
---

<sup>10</sup>Not 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 the output into the 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.<sup>11</sup> 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  $1 \times 2$  and  $2 \times 4$  matrices. The rightmost figure above shows how the neural net has

---

<sup>11</sup>For 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 than 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.

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  $1 \times 2$  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.

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. Still, 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. Fortunately, we can handle the problem 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  $\mathbf{A}^\top \mathbf{A}\mathbf{c} = \mathbf{A}^\top \mathbf{y}$ , 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 compute the pseudoinverse explicitly. 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 specifying 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 variables and parameters and define  $f_i(c) = f(c_1, c_2, \dots, c_n; x_i)$ . We can 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  $\mathbf{r} = \mathbf{y} - f(\mathbf{c})$ . 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

$$\mathbf{J}_r(\mathbf{c}^{(k)}) \Delta \mathbf{c}^{(k+1)} = -\mathbf{r}^{(k)},$$

where  $\Delta \mathbf{c}^{(k+1)} = \mathbf{c}^{(k+1)} - \mathbf{c}^{(k)}$  and  $\mathbf{J}_r(\mathbf{c})$  is the  $m \times n$  Jacobian matrix whose  $(i, j)$ -component is  $\partial r_i / \partial c_j$ . The Jacobian matrix  $\mathbf{J}_r(\mathbf{c}) = -\mathbf{J}_f(\mathbf{c})$ , so we equivalently have

$$\mathbf{J}_f(\mathbf{c}^{(k)}) \Delta \mathbf{c}^{(k+1)} = \mathbf{r}^{(k)} \tag{10.15}$$

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 using QR factorization to update  $\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + \Delta\mathbf{c}^{(k+1)}$  at each iteration. The method is called the *Gauss–Newton method*.

By multiplying both sides of (10.15) 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:

$$\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + \mathbf{J}_f^+(\mathbf{c}^{(k)})\mathbf{r}^{(k)}, \quad (10.16)$$

where the pseudoinverse  $\mathbf{J}_f^+$  is  $(\mathbf{J}_f^\top \mathbf{J}_f)^{-1} \mathbf{J}_f^\top$ . The Gauss–Newton method may be unstable. We can regularize it by modifying the  $\mathbf{J}_f^\top \mathbf{J}_f$  term of the pseudoinverse to be diagonally dominant. This regularization has the effect of weakly decoupling the system of equations. The *Levenberg–Marquardt method* is given by

$$\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + (\mathbf{J}^\top \mathbf{J} + \lambda \mathbf{D})^{-1} \mathbf{J}^\top \mathbf{r}^{(k)}, \quad (10.17)$$

where  $\mathbf{J}$  is  $\mathbf{J}_f(\mathbf{c}^{(k)})$ , the matrix  $\mathbf{D}$  is a nonnegative diagonal matrix, and the damping parameter  $\lambda$  is positive. Typical choices for  $\mathbf{D}$  include the identity matrix  $\mathbf{I}$  or the diagonal matrix  $\text{diag}(\mathbf{J}^\top \mathbf{J})$ . 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 construct the pseudoinverse to implement the method explicitly—we can use a stacked matrix instead. But the explicit formulation (10.17) 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

$$f(x; \mathbf{c}) = c_1 e^{-c_2(x-c_3)^2} + c_4 e^{-c_5(x-c_6)^2} \quad (10.18)$$

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
```

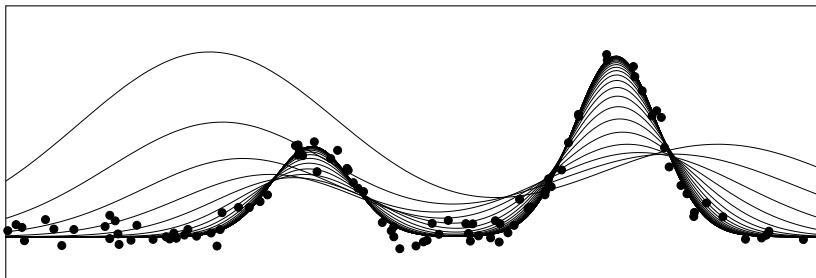


Figure 10.7: The least squares solution for (10.18). Solutions are depicted at intermediate iterations. The input data is represented by •.

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.")
end
```

For 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(x,[1, 3, 3, 2, 3, 6]) + 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  $c = \{0.98, 3.25, 98, 3.00, 1.96, 2.88, 5.99\}$ .

We can also control the stability of Newton's method by shortening the step size by a factor  $\alpha$ , called the *learning rate*. For example, we could substitute a Newton step  $c += \alpha * G \setminus r$  with  $\alpha=0.5$  in the loop in place of the Levenberg–Marquardt step. Alternatively, we could combine a faster learning rate with the Levenberg–Marquardt method. Because the Levenberg–Marquardt method is

more stable than the Gauss–Newton method, we can take a learning rate as large as 2.

In practice, we might use the LsqFit.jl package, which solves nonlinear least squares problems:

```
using LsqFit
cf = curve_fit(f, x, y, c₀)
```

The parameters are given by `cf.param`. The residuals are given by `cf.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. We determine parameters that make a distribution most likely to fit observed data by maximizing the likelihood function. We call such values the maximum likelihood estimate.

Consider an experiment with 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:

$$\ell(\mathbf{w}) = \sum_{i=1}^n y_i \log p + (1 - y_i) \log(1 - p). \quad (10.19)$$

Because the logarithm is a monotonically increasing function, the maximum likelihood estimate is the same as the maximum of the log-likelihood. So we just need to find the parameters  $\mathbf{w}$  that maximize  $\ell(\mathbf{w})$ . We'll first need a model for the probabilities  $p(\mathbf{x}_i, \mathbf{w})$  to do this.

A generalized linear model (GLM) 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, that is monotonically increasing, and that 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,<sup>12</sup> the logit function is easy to manipulate, it has the nice interpretation of being the log odds ratio, and it is the inverse of the logistic function  $\sigma(\eta) = 1/(1 + e^{-\eta})$ . Using the inverse logit (the logistic function) in the log-likelihood function (10.19) 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

$$\begin{aligned} \mathbf{w}^{(k+1)} &= \mathbf{w}^{(k)} - (\mathbf{H}^{(k)})^{-1} \mathbf{g}^{(k)} \\ &= \mathbf{w}^{(k)} + (\mathbf{X}^\top \mathbf{S}^{(k)} \mathbf{X})^{-1} \mathbf{X}^\top (\mathbf{y} - \sigma(\mathbf{w}^\top \mathbf{X}^{(k)})). \end{aligned}$$

The terms on the right-hand side are often regrouped as a formula called *iteratively reweighted least squares* (IRLS):

$$\mathbf{w}^{(k+1)} = (\mathbf{X}^\top \mathbf{S}^{(k)} \mathbf{X})^{-1} \mathbf{X}^\top (\mathbf{S}^{(k)} \mathbf{X} \mathbf{w}^{(k)} + \mathbf{y} - \sigma(\mathbf{w}^\top \mathbf{X}^{(k)})).$$

Let's see what the method looks like in Julia. We'll first define the logistic function and generate some synthetic data.

---

<sup>12</sup>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.

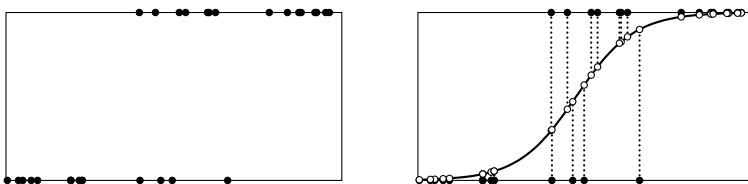


Figure 10.8: Classification data (left). The function fitting the data (right).

```
 $\sigma = x \rightarrow \text{@. } 1/(1+\exp(-x))$ 
 $x = \text{rand}(30); y = (\text{rand}(30) .< \sigma(10x.-5) );$ 
```

Now, we apply Newton's method.

```
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 magnitude of the 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 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  $\mathbf{y}$  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  $\ell^2$ -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. We'll use the notation  $[\cdot]$  to emphasize that  $\partial L/\partial \mathbf{y}$  is a matrix or a vector.

We can solve the minimization problem using the gradient descent method. The gradient descent method iteratively marches 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  $\alpha_1$  and  $\alpha_2$ , called learning rates, are chosen to ensure the 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

$$[\partial f/\partial \mathbf{A}] = [\partial f/\partial \mathbf{C}] [\mathbf{B}]^\top \text{ and } [\partial f/\partial \mathbf{B}] = [\mathbf{A}]^\top [\partial f/\partial \mathbf{C}],$$

If  $\mathbf{C} = g(\mathbf{A})$ , then we the chain rule for  $f(g(\mathbf{A}))$  is

$$[\partial f/\partial \mathbf{A}] = [g'(\mathbf{A})]^\top [\partial f/\partial \mathbf{C}],$$

where  $[g'(\mathbf{A})]$  is a Jacobian matrix. So,

$$[\partial L/\partial \mathbf{W}_2] = [\partial L/\partial \mathbf{y}] [\phi(\mathbf{W}_1\mathbf{x}_*)]^\top$$

and

$$[\partial L/\partial \mathbf{W}_1] = [\mathbf{W}_2\phi'(\mathbf{W}_1\mathbf{x}_*)]^\top [\partial L/\partial \mathbf{y}] [\mathbf{x}_*]^\top.$$

The Jacobian matrix  $[\phi']$  is diagonal. 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.<sup>13</sup> We'll pick 100 points.

```
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 two 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
    ī = 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 figure on the next page and the QR code at the bottom of this page show the solution. We can also solve the problem using a sigmoid activation function

```
ϕ = x -> @. 1 / (1 + exp(-x))
dϕ = x -> @. ϕ(x)*(1-ϕ(x))
```

---

<sup>13</sup>A neural network on a one-dimensional problem with a small number of 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.



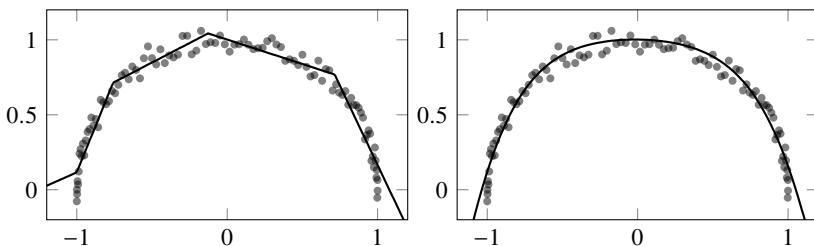


Figure 10.9: The neural network approximation for the data  $\bullet$  using ReLU activation functions (left) and sigmoid activation functions (right).

along with alternative loss function  $L(\mathbf{y}) = \|\mathbf{y} - \mathbf{y}_*\|_2$ . The gradient 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 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 = Flux.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`. ◀

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 thousands of

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 fully connected layers. Convolutional neural nets (CNNs), frequently used in image recognition, use convolution matrices with relatively small kernels. Such kernels—typical 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 to update 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.

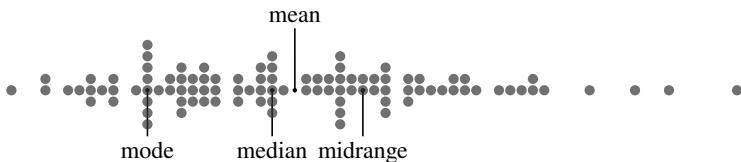
We can modify gradient descent to overcome its shortcomings, such as adding momentum to overcome shallow local minimum and saddle points or using adaptive learning rates such as AdaGrad and Adam to speed up convergence. Finally, to speed up the computation of gradients and avoid numerical stability, we can use automatic differentiation, which we discuss in the next chapter.

## ► Statistical best estimates

While the focus of this chapter has been on finding the “best” estimate in the  $\ell^2$ -norm, for completeness, let’s conclude with a comment on the “best” estimate of sample data in a general  $\ell^p$ -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  $\ell^p$ -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  $\ell^0$ -“norm” is simply the count of nonzero elements in the set.<sup>14</sup> Quotation marks are used to emphasize that the  $\ell^0$ -“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  $\ell^0$ ,  $\ell^1$ ,  $\ell^2$ , and  $\ell^\infty$ -norms is one of the four statistical averages—the mode, median, mean, and midrange, respectively. The mode is the most frequently occurring value 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 be located 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  $\{1, x, x^2, \dots, x^n\}$ .

---

<sup>14</sup>Not 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  $x \in [-\pi, \pi]$
- $-\frac{1}{4} \text{sign}(x) \cdot ((2|x| - 1)^2 - 1)$  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 is smooth. Similarly, the piecewise-linear and piecewise-quadratic functions are not smooth. Still, 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 of 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

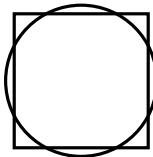
implement LeNet-5 to classify the digits in the MNIST dataset. The dataset is available through the `MLDatasets.jl` package.



## 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 the others by examining numerical and automatic differentiation as well as Newton–Cotes, Gaussian, and Monte Carlo integration.

### 11.1 Numerical differentiation

The previous chapter 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

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + h^2f''(x) + O(h^3) \\ f(x) &= f(x). \end{aligned}$$

Taking the difference of these equations gives us

$$f(x+h) - f(x) = hf'(x) + h^2f''(x) + O(h^3),$$

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,

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{1}{2}h^2f''(x) + O(h^3) \\ f(x) &= f(x) \\ f(x-h) &= f(x) - hf'(x) + \frac{1}{2}h^2f''(x) + O(h^3). \end{aligned}$$

We want to eliminate as many terms that could contribute to error. Consider the unknowns  $a, b, c$

$$\begin{aligned} af(x+h) &= af(x) + ahf'(x) + a\frac{1}{2}h^2f''(x) + a\frac{1}{6}h^3f'''(x) + O(h^4) \\ bf(x) &= bf(x) \\ cf(x-h) &= cf(x) - chf'(x) + c\frac{1}{2}h^2f''(x) - c\frac{1}{6}h^3f'''(x) + O(h^4). \end{aligned}$$

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

$$\begin{aligned} a + b + c &= 0 \\ a - c &= 1 \\ \frac{1}{2}a + \frac{1}{2}c &= 0 \end{aligned}$$

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 reach a fourth-order method, etc. And we could use nonequally spaced nodes. Let's use the Vandermonde matrix to automate the process of computing the coefficients. The system of Taylor polynomial approximations

$$f(x + \delta_i h) = \sum_{j=0}^{n-1} c_{ij} \frac{1}{j!} (\delta_i h)^j f^{(j)}(x) + O(h^n),$$

where  $\delta_i$  scales the step size  $h$ , can be written in matrix form

$$[f(x + \delta_i h)] = \mathbf{VC} [h^j f^{(j)}(x)] + O(h^{n+1}),$$

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 an  $O(h^n)$  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 are 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  $C * d.^n ./ factorial(n)$ :

$$\begin{array}{rrrrr} 0 & 1 & 0 & 0 & 0 \\ -\frac{1}{3} & -\frac{1}{2} & 1 & -\frac{1}{6} & -\frac{1}{12} \\ 1 & -2 & 1 & 0 & \frac{1}{12} \\ -1 & 3 & -3 & 1 & \frac{1}{2} \end{array}$$

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 `Rational` that expresses the ratio of integers. For example,  $3/4$  is the Julia representation for  $\frac{3}{4}$ . You can also convert the floating-point number 0.75 to a rational using `rationalize(0.75)`.

## ► Richardson extrapolation

We can improve the accuracy of a polynomial interpolant by adding more nodes—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 develop a way to add nodes that update 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

$$\phi(\delta h) = a_0 + a_1 \delta^p h^p + a_2 \delta^{2p} h^{2p} + a_3 \delta^{3p} h^{3p} + \dots.$$

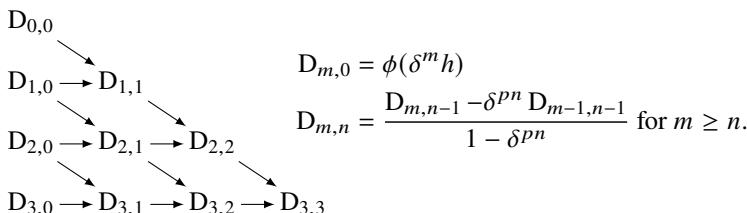
where  $0 < \delta < 1$ . Typically,  $\delta = \frac{1}{2}$ . Then

$$\phi(\delta h) - \delta^p \phi(h) = a_0(1 - \delta^p) + 0 + a_2(\delta^{2p} - \delta^p)h^2 + \dots,$$

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. We now have 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  $2.4 \times 10^{-14}$ .

British mathematician Lewis Fry Richardson made several contributions relating to the scales of things, from weather prediction to violent conflict. The success of Richardson's extrapolation is due to smooth functions having a rectifiable length. But not all curves are rectifiable. Richardson observed that measurements of things like seacoasts don't converge by taking progressively smaller scales. Capes and fjords, absent on the large scale, emerge on the small scale. Spits and coves appear at still smaller scales. Richardson found that by dividing a coastline into segments of length  $h$ , its total length could be approximated using  $mh^{1-d}$ , for some parameters  $m$  and  $d$ . The Australian coast has a dimensionality  $d = 1.13$ , and the west coast of Great Britain has  $d = 1.25$ . Richardson's empirical finding went largely unnoticed until Benoit Mandelbrot wrote about it in 1967. Several years later, Mandelbrot coined the word fractal (from the Latin *fractus* for shattered) and the "Richardson effect" to describe the fractal nature of infinite coastlines.

## 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 of that function or by computing the actual derivative of a simpler approximation of the original function. Because numerical differentiation takes the 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 the expression size increases. 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 are the adjunction of a real or complex number  $a$  with an infinitesimal number  $b\epsilon$ . Just as we could represent a complex number  $a + bi$  as an ordered pair of real numbers  $(a, b)$  with multiplication and other operations defined in a special way, we can represent a dual number  $a + b\epsilon$  as the ordered pair  $(a, b)$  with special rules of arithmetic. First, consider Taylor series expansion:

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x) + \dots = f(x) + \epsilon f'(x) \equiv (f, f'),$$

where all  $\epsilon^2$  and higher terms are zero. This expression is the formal infinitesimal definition of the derivative using nonstandard analysis

$$f'(x) = \frac{f(x + \epsilon) - f(x)}{\epsilon}.$$

Note that the derivative is not defined as the limit as a real number  $\epsilon$  goes to zero. Instead, it is defined entirely with respect to an infinitesimal  $\epsilon$ . Dual numbers follow the same addition and multiplication rules with the additional rule that  $\epsilon^2 = 0$ . For example, for addition,

$$\begin{aligned} (f, f') + (g, g') &\equiv (f + f'\epsilon) + (g + g'\epsilon) = (f + g) + (f' + g')\epsilon \\ &\equiv (f + g, f' + g'); \end{aligned}$$

for multiplication,

$$\begin{aligned} (f, f') \cdot (g, g') &\equiv (f + f'\epsilon) \cdot (g + g'\epsilon) = fg + (fg' + g'f)\epsilon \\ &\equiv (fg, fg' + g'f); \end{aligned}$$

and for division,

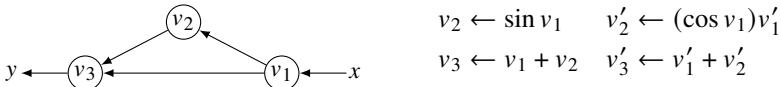
$$\begin{aligned}\frac{(f, f')}{(g, g')} &\equiv \frac{f + f'\varepsilon}{g + g'\varepsilon} = \frac{(f + f'\varepsilon)}{(g + g'\varepsilon)} \frac{(g - g'\varepsilon)}{(g - g'\varepsilon)} = \frac{fg + (f'g - g'f)\varepsilon}{g^2} \\ &\equiv \left( \frac{f}{g}, \frac{f'g - g'f}{g^2} \right).\end{aligned}$$

The results are the familiar addition, product, and quotient rules of calculus. We can compute the values of the derivatives at the same time that the values of the functions are evaluated with only a few more operations. When implemented in a programming language such as Julia or Matlab, each function can be overloaded to return its value and 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 compute either from the inside out by taking  $f \circ (g \circ h)$ —a process called forward accumulation—or 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)
```

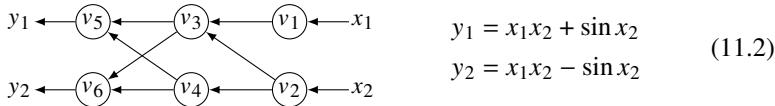
which returns  $\text{Dual}(0.0, 2.0)$  as expected. ◀

For a system  $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$  we have the dual form

$$(f(\mathbf{x}), f'(\mathbf{x})) = f(\mathbf{x} + \varepsilon) = f(\mathbf{x}) + \mathbf{J}_f(\mathbf{x})\varepsilon = (f(\mathbf{x}), \mathbf{J}_f(\mathbf{x})),$$

where  $\mathbf{J}_f(\mathbf{x})$  is the  $m \times n$  Jacobian matrix evaluated at  $\mathbf{x}$ . We can choose between forward accumulation and reverse accumulation in constructing the Jacobian matrix. We'll examine each in the following example.

**Example.** Consider the system of equations and its accompanying computational graph



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  $\xi_i$  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)
```

variables	$v$		$v'$		
$(v_1, v'_1)$	$x_1$	2	$x'_1$	1	0
$(v_2, v'_2)$	$x_2$	$\pi$	$x'_2$	0	1
$(v_3, v'_3)$	$v_1 v_2$	$2\pi$	$D_{31}v'_1 + D_{32}v'_2 \rightarrow v_2 v'_1 + v_1 v'_2$	$\pi$	2
$(v_4, v'_4)$	$\sin v_2$	0	$D_{42}v'_2 \rightarrow v'_2 \cos v_2$	0	-1
$(v_5, v'_5)$	$v_4 + v_3$	$2\pi$	$D_{53}v'_3 + D_{54}v'_4 \rightarrow v'_3 + v'_4$	$\pi$	1
$(v_6, v'_6)$	$v_4 - v_3$	$2\pi$	$D_{63}v'_3 + D_{64}v'_4 \rightarrow v'_3 - v'_4$	$\pi$	3

Figure 11.1: Forward accumulation automatic differentiation of the system (11.2) evaluated at  $(x_1, x_2) = (2, \pi)$ .

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

$$\begin{bmatrix} v_2 & 0 \\ v_1 & \cos v_2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \mathbf{J}^T \bar{\mathbf{x}},$$

where  $\bar{\mathbf{x}}$  is called the adjoint value. Using a standard unit vector  $\xi_i$  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, 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 the following 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 seems archaic like many others in mathematics. Historically, the quadrature of a figure described the

variables	$v$	$\bar{v}$		
$(v_1, \bar{v}_1)$	$x_1$	2	$D_{31}\bar{v}_3 \rightarrow v_2\bar{v}_3$	$\pi$
$(v_2, \bar{v}_2)$	$x_2$	$\pi$	$D_{32}\bar{v}_3 + D_{42}\bar{v}_4 \rightarrow v_1\bar{v}_3 + \bar{v}_4 \cos v_2$	1 3
$(v_3, \bar{v}_3)$	$v_1 v_2$	$2\pi$	$D_{53}\bar{v}_5 + D_{63}\bar{v}_6 \rightarrow \bar{v}_5 + \bar{v}_6$	1 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 -1
$(v_5, \bar{v}_5)$	$v_4 + v_3$	$2\pi$	$y'_1$	1 0
$(v_6, \bar{v}_6)$	$v_4 - v_3$	$2\pi$	$y'_2$	0 1

Figure 11.2: Reverse accumulation automatic differentiation of the system (11.2) evaluated at  $(x_1, x_2) = (2, \pi)$ .

process of constructing a square with the same areas as some given figure. The English 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  $\pi$  is a transcendental number. The name quadrature stuck with us, and most scientific programming languages now use some variant of the term *quad* for numerical integration.<sup>1</sup> 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 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

$$\begin{aligned} \int_a^b p_n(x) dx &= \int_a^b \left( \sum_{i=0}^n f(x_i) \ell_i(x) \right) dx \\ &= \sum_{i=0}^n f(x_i) \int_a^b \ell_i(x) dx \\ &= \sum_{i=0}^n w_i f(x_i) \quad \text{where} \quad w_i = \int_a^b \ell_i(x) dx. \end{aligned} \quad (11.3)$$

<sup>1</sup>The even more archaic-sounding term *cubature* (finding the volume of solids) is sometimes used in reference to multiple integrals.

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  $x = \{0, \frac{1}{2}, 1\}$ . This time the polynomial interpolant will be quadratic. Then

$$\begin{aligned} w_0 &= \int_0^1 \ell_0(x) dx = \int_0^1 2(1-x)\left(\frac{1}{2}-x\right) dx = \frac{1}{6}, \\ w_1 &= \int_0^1 \ell_1(x) dx = \int_0^1 4x(1-x) dx = \frac{2}{3}, \\ w_2 &= \int_0^1 \ell_2(x) dx = \int_0^1 2x\left(x-\frac{1}{2}\right) dx = \frac{1}{6}. \end{aligned}$$

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  $[\frac{1}{2}, 1]$  in half, giving us five equally spaced nodes. The polynomial interpolant is quartic this time, 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 232, 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  $\xi$  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 higher-order polynomials such as Boole's rule, such an approach is not generally good because of the Runge phenomenon. Instead, we can either use a piecewise polynomial approximation (a spline) or pick nodes to minimize the uniform error (such as using Chebyshev nodes). We'll look at both of these options in turn.

## ► Composite methods

Composite methods use splines to approximate the integrand. The simplest type of spline is a constant spline. Using these, we get Riemann sums. Linear splines give us the *composite trapezoidal rule*, a more accurate method that applies the trapezoidal rule to each subinterval. For  $n + 1$  equally-spaced 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  $\xi$  in a subinterval. The error for the composite trapezoidal rule is bounded by  $\frac{1}{12} (b-a)^3 |f^{(2)}(\xi)| / n^2$  for some  $\xi \in (a, b)$ . We can derive a much better error estimate with a bit of extra work, which we'll do in a couple of pages. 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.

With 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  $\frac{1}{90}h^5 f^{(4)}(\xi)$ , so the error of the composite Simpson’s rule is bounded by  $\frac{1}{180}(b-a)^5 f^{(4)}(\xi)/n^4$  for some  $\xi$  over an 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 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,  $D_{m,1}$  extrapolation is equivalent to the composite Simpson’s rule, and  $D_{m,2}$  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  $\phi$  in the Julia code on page 298 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  $1.98 \times 10^{-12}$ .

**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)
        S_n = trapezoidal(x->f(x,p),[0,2],n[i])
```

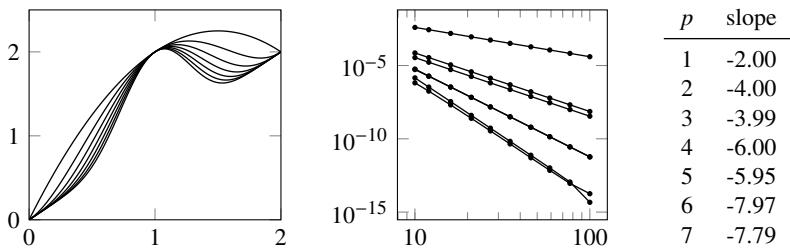


Figure 11.3: Left: Plot of  $f_p(x) = x + x^p(2-x)^p$  for  $p = 1, 2, \dots, 7$ . Middle: Error using composite trapezoidal rule for  $f_p(x)$  as a function of the number of nodes. Right: Slopes of the curves.

```

error[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 above. We can determine the convergence rate by computing  $[x \ 1]^{-1} y$  where  $x$  is a vector of logarithms of the number of nodes,  $y$  is a vector of logarithms of the errors, and  $1$  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 find that the error of the trapezoidal rule is  $O(n^2)$  when  $p = 1$ . 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. What's going on? ◀

### ► 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.<sup>2</sup> 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.

---

<sup>2</sup>The Euler–Maclaurin formula, which relates a sum to an integral, provides an alternative method for estimating the error of the composite trapezoidal rule. The Euler–Maclaurin formula

## ► 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

$$f(\cos \theta) = \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \cos(k\theta) \quad \text{where} \quad a_k = \frac{2}{\pi} \int_0^\pi f(\cos \theta) \cos k\theta d\theta$$

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

$$a_{2k} = \frac{2}{n} \left[ \frac{f_0}{2} + \sum_{j=1}^{n-1} f_j \cos \left( \frac{2\pi j k}{n} \right) + \frac{f_n}{2} \right] \quad \text{where} \quad f_j = \cos \left( \frac{\pi j}{n} \right)$$

Extend the function  $f(\cos \theta)$  as an even function at  $\pi$ :  $f_{n+j} = f_{n-j}$ . 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

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.

```
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 last 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  $f(x)$  using Chebyshev polynomials. The next section will discuss the expansion of functions in orthogonal polynomial bases.

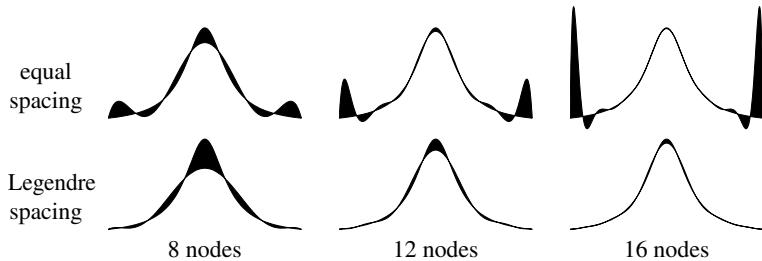
## 11.4 Gaussian quadrature

Let's return to the Newton–Cotes formula

$$\int_a^b p(x) dx = \sum_{i=0}^n w_i f(x_i) \quad \text{where} \quad w_i = \int_a^b \ell_i(x) dx. \quad (11.5)$$

We can construct higher-order polynomials to approximate the integrand more closely by adding more nodes. The equally spaced nodes of Newton–Cotes quadrature are problematic, specifically because of Runge's phenomenon. Chapter 9 used Chebyshev nodes (the zeros of the Chebyshev polynomial) to minimize

Runge's oscillations. This section considers 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 mitigate Runge's phenomenon. Remarkably, using these carefully selected nodes 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?” 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  $2n + 1$ . A quadrature that carefully chooses nodes to construct a maximal polynomial is called *Gaussian quadrature* (and sometimes Gauss–Christoffel quadrature).

From Chapter 10, 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 on the facing page. 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  $f \in \mathbb{P}_{2n+1}$*

$$\int_a^b f(x)\omega(x) dx = \sum_{i=0}^n w_i f(x_i) \quad \text{where} \quad w_i = \int_a^b \ell_i(x)\omega(x) dx$$

polynomial	interval	weight	$\ 1\ ^2$	$a_k$	$b_k$
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.

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,

$$\int_a^b f(x)\omega(x) dx = \int_a^b q(x)p(x)\omega(x) dx + \int_a^b r(x)\omega(x) dx = \sum_{i=0}^n w_i r(x_i)$$

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  $\xi$  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 on page 254. It 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  $\mathbf{J}_n \mathbf{p} = \mathbf{x} \mathbf{p} + \mathbf{r}$ :

$$\begin{bmatrix} a_0 & \sqrt{b_1} & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & \\ & \ddots & \ddots & \ddots \\ & & \sqrt{b_{n-1}} & a_{n-1} & \sqrt{b_n} \\ & & & \sqrt{b_n} & a_n \end{bmatrix} \begin{bmatrix} \hat{p}_0(x) \\ \hat{p}_1(x) \\ \vdots \\ \hat{p}_{n-1}(x) \\ \hat{p}_n(x) \end{bmatrix} = x \begin{bmatrix} \hat{p}_0(x) \\ \hat{p}_1(x) \\ \vdots \\ \hat{p}_{n-1}(x) \\ \hat{p}_n(x) \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \sqrt{b_{n+1}} \hat{p}_{n+1}(x) \end{bmatrix}.$$

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  $\mathbf{J}_n \mathbf{p} = x \mathbf{p}$ . In other words, 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  $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}$  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,

$$\sum_{k=0}^n \sqrt{w_i w_j} \hat{p}_k(x_i) \hat{p}_k(x_j) = \delta_{ij}$$

which says (for  $i = j$ ) that

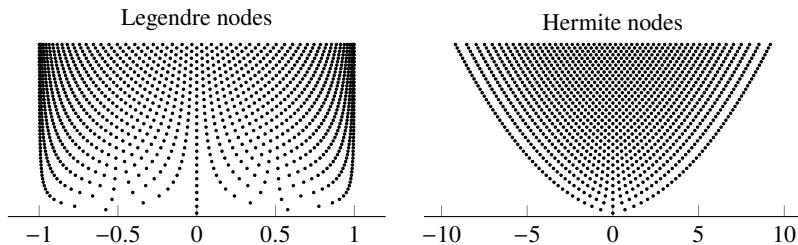
$$w_j \sum_{k=0}^n \hat{p}_k^2(x_j) = 1.$$

So  $\sqrt{w_j} [\hat{p}_0(x_j), \hat{p}_1(x_j), \dots, \hat{p}_n(x_j)]^\top$  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 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  $\|1\|^2$  times the square of the first element of the unit eigenvectors.*

Figure 11.5: Location of Legendre and Hermite nodes for  $n = 1, 2, \dots, 50$ .

Parameters for common Gaussian quadrature rules are summarized in Figure 11.4 on page 313. The position of the nodes of Gauss–Legendre and Gauss–Hermite quadrature are shown in the figure above. We can directly compute the nodes and weights of Gauss–Chebyshev quadrature using  $x_k = \cos((2k-1)\pi/2n)$  and  $w_k = \pi/n$ , so we don't need the Golub–Welsch algorithm in this case. 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.<sup>3</sup>

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` provides a fast, accurate method of computing the nodes and weights

```
using FastGaussQuadrature
nodes, weights = gausslegendre(n)
```

<sup>3</sup>Algorithms 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]

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  $d\xi = e^{-x} dx$ .

## ► 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 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

$$\int_a^b p(x)q(x)x^k \omega(x) dx = 0 \quad \text{for each } k = 0, 1, \dots, n \tag{11.6}$$

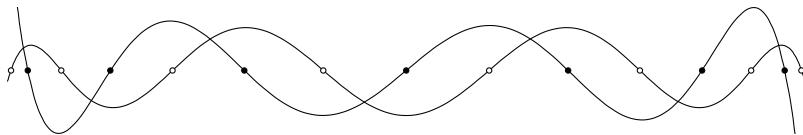
for some weight function  $\omega(x)$ , then

$$\int_a^b f(x)\omega(x) dx = \int_a^b r(x)\omega(x) dx.$$

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 with 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 have both real zeros and 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.<sup>4</sup> 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 uses 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 the error is estimated by subtracting the two rules. The weights and nodes are simply hard-coded into the algorithm.

---

<sup>4</sup>Another 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  $T_n(x)$ .

## ► 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 by 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  $\pm 1$ . 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  $\pm 1$ . Because Gauss–Radau quadrature fixes a node to be an endpoint, it removes one degree of freedom from the set of nodes and weights. By 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.

## ► 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 quad1, 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  $\ell$  be the length of a needle,  $d$  be the width of the strip of wood,  $\theta$  be the angle of the needle relative 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 to be 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  $\theta$  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  $\pi$ , 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  $\pi$ . The more needles we use, the better the approximation. ◀

We could apply Fubini's rule to repeated one-dimensional integrals to compute multidimensional 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 three 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—more than two months using the same computer. The problem is afflicted with the so-called *curse of dimensionality*.

Nuclear physicist Stanislaw Ulam developed the Monte Carlo method in 1946. While convalescing and playing solitaire, Ulam asked himself the likelihood that a Canfield game 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.<sup>5</sup>

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 nonnegative function that tells us the 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

$$\langle F^N \rangle = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(X_i)}{p_X(X_i)}.$$

---

<sup>5</sup>The name Monte Carlo, proposed by Ulam's colleague Nicholas Metropolis, was inspired by 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.

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 the sample average converges to the expected value as the sample size approaches infinity. For arbitrarily small  $\varepsilon$ , the probability

$$\lim_{N \rightarrow \infty} P \left( |\langle F^N \rangle - \mathbb{E} [\langle F^N \rangle]| > \varepsilon \right) = 0.$$

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  $\Omega$  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.

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 relatively poor compared to 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 more 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  $f(\mathbf{x})$ .

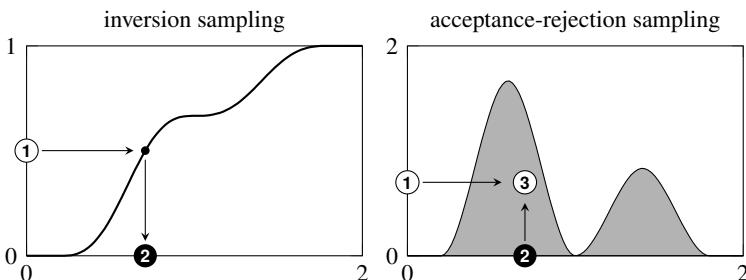


Figure 11.6: Techniques for sampling a variable ② from a distribution. Using inversion sampling, a value ① is sampled from  $U(0, 1)$  and mapped to ② using the inverse cumulative probability function. Using acceptance-rejection sampling, ② is uniformly sampled from the domain and ① is sampled from another 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.

## ► 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 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 using 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 by 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 remains in place or moves according to the acceptance ratio  $\alpha$ . 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  $\alpha$ .

**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.<sup>6</sup> 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

$$p(x)p(y) dx dy = \frac{1}{2\pi}e^{-r^2/2}r dr d\theta = \left(re^{-r^2/2} dr\right) \left(\frac{1}{2\pi} d\theta\right).$$

---

<sup>6</sup>The 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.

We want to sample  $r$  from the density  $re^{-r^2/2}$  and  $\theta$  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  $\theta$  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. ◀

••• The Distrtributions.jl package has an extensive collection of distributions. 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.” Still, he conceded that it was far faster than feeding a computer ran dom 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 as 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 and Xoshiro<sup>7</sup> developed in 2018 by David Blackman and Sebastiano Vigna.

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 156, 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

---

<sup>7</sup>Xoshiro is a portmanteau of XOR, shift, and rotate.

$$5 \rightarrow 10 \rightarrow 7 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 11 \rightarrow 9 \rightarrow$$

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  $a$  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 model 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 298 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(Dual(Dual(x)).deriv.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  $\gamma$  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. S674.5.6 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

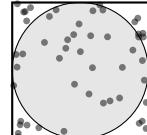
$$u(t, x) = \frac{1}{\sqrt{4\pi t}} \int_{-\infty}^{\infty} u_0(s) e^{-(x-s)^2/4t} ds.$$

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  $\pi$  by noting that the area of a circle of unit radius is  $\pi$ . 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

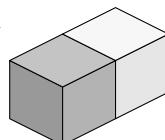
$\pi \approx \frac{m}{n}$ . Use Monte Carlo integration to compute an approximation for  $\pi$ .

2. Use a log-log plot of the error to confirm the order of convergence.
3. How many samples would be needed to accurately compute 13 digits of  $\pi$ ?
4. 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 *method of lines*. The method of lines discretizes the spatial component and solves the resulting semi-discrete problem 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  $\Delta$  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 numerical schemes for ordinary differential equations will help understand stability, consistency, and convergence of numerical schemes for partial differential equations. To dig even deeper, see Hairer, Nørsett, and Wanner's two-volume compendium *Solving Ordinary 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 the method of 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  $u_0$  or the function  $f(t, u)$  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  $\varepsilon$  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  $\varepsilon$  be the maximum of  $\varepsilon_0$  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.  $\square$

Remember the following takeaways over the rest of this chapter and the remainder of this book. 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.

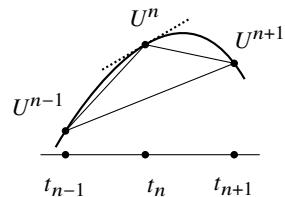
## 12.2 Single-step methods

Let  $u \equiv u(t)$ . Suppose we have 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  $\Delta t$  and  $h$  in place of  $\Delta x$  will help tidy some otherwise messier expressions. The derivative  $u'$  can be approximated by difference operators:

forward difference	$\delta_+ U^n = \frac{U^{n+1} - U^n}{k}$
backward difference	$\delta_- U^n = \frac{U^n - U^{n-1}}{k}$
central difference	$\delta_0 U^n = \frac{U^{n+1} - U^{n-1}}{2k}$



where  $U^n$  is the numerical approximation to  $u(t_n)$ . Using these operators, we can formulate several finite difference schemes to 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 has a graphical representation, called a *stencil*, that indicates which terms are employed. We'll use to depict the terms used on the left hand side to discretize the time derivative and 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 more intuition about their behavior. An *integral curve* is a parametric curve representing 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 nonintersecting, although they might get 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.

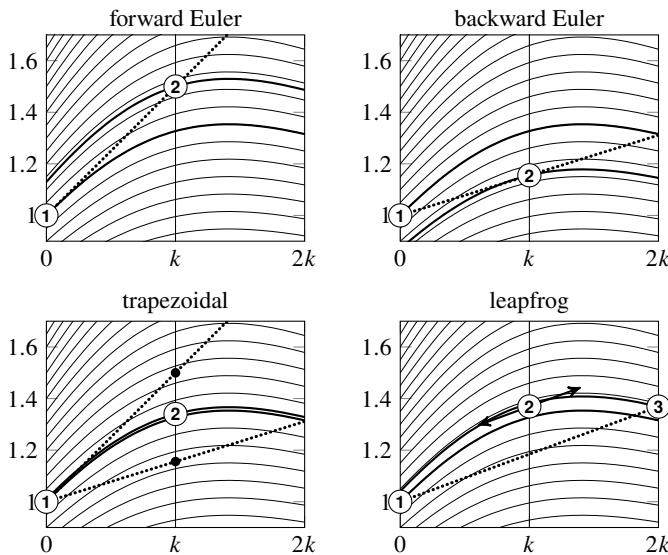


Figure 12.1: Integral curves and numerical solutions using the forward Euler, backward Euler, leapfrog, and trapezoidal methods.

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 the value of the integral curve at ② is not trivial. Like the forward Euler method, we also find ourselves bumped off the integral curve when using the backward Euler method. The trapezoidal method averages the slopes of the forward Euler and backward Euler to take a step forward. We still get bumped off the original integral curve, but it's not as far this time. 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 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  $\tau_k$ —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 of 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 expressions yields

$$u(t_{n+1}) - u(t_{n-1}) = 2ku'(t_n) + \frac{1}{3}k^3u'''(t_n) + O(k^4).$$

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  $e^{\lambda t}$ :

<i>exact solution</i>	<i>forward Euler</i>	<i>backward Euler</i>	<i>trapezoidal</i>	<i>leapfrog</i>
$e^{\lambda k}$	$1 + \lambda k$	$\frac{1}{1 - \lambda k}$	$\frac{1 + \frac{1}{2}\lambda k}{1 - \frac{1}{2}\lambda k}$	$\lambda k \pm \sqrt{(\lambda k)^2 + 1}$

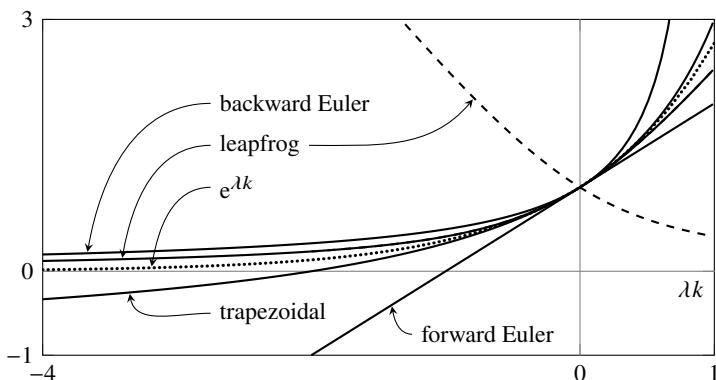


Figure 12.2: Plot of the multiplier  $r(\lambda k)$  for several numerical schemes along the real  $\lambda k$ -axis compared to the exact solution multiplier  $e^{\lambda k}$ .

The exact solution  $e^{\lambda k}$  is depicted as a dotted curve. Approximations are good when  $\lambda k$  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 that matches the exact solution fairly well and another one that 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  $\lambda k$  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 discussing 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 a 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  $\delta$  is not larger than  $\delta$  at any of the subsequent time steps. In other words, a method is stable if perturbations are bounded over a 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

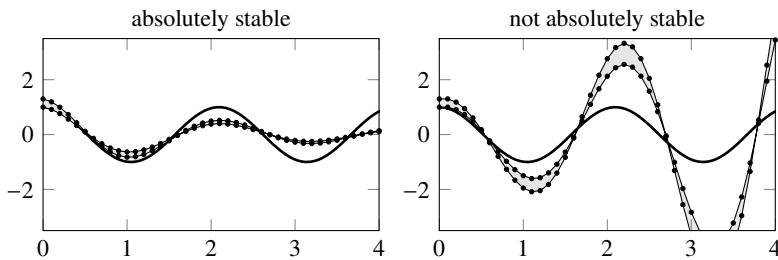


Figure 12.3: A perturbation decreases over time using an absolutely stable method and increases using an absolutely unstable one. Neither one is particularly accurate.

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  $\lambda$ . 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  $\lambda k$ -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  $\lambda$ . 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  $\lambda k$ -plane where  $|r| \leq 1$  is called the *region of absolute stability*. We'll use it to identify 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 to say, the region of absolute stability is bounded by the unit circle centered at  $\lambda k = -1$ . See Figure 12.4 on the next page. For a given value  $\lambda$ , we may need to limit the size of our step  $k$  to ensure that  $\lambda k$  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  $\lambda$ .

**Backward Euler.** The backward Euler method for  $u' = \lambda u$  is

$$\frac{U^{n+1} - U^n}{k} = \lambda U^{n+1}.$$

It follows that  $(1 - \lambda k)U^{n+1} = U^n$ , 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  $\lambda k$ -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

$$\begin{bmatrix} U^{n+1} \\ V^{n+1} \end{bmatrix} = \begin{bmatrix} 1 & -k \\ 9k & 1 \end{bmatrix}^{-1} \begin{bmatrix} U^n \\ V^n \end{bmatrix}$$

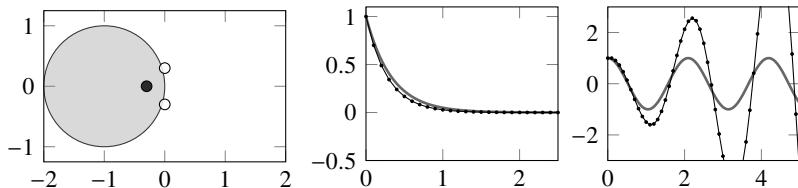
is a damped sine wave that dissipates within a few oscillations when  $k$  is large.

---

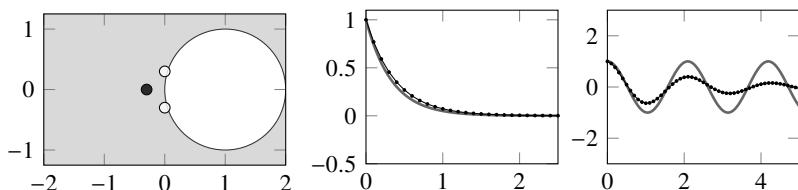
Region of absolute stability $\bullet \quad u' = -3u$  $\circ \quad u'' = -9u$ 

---

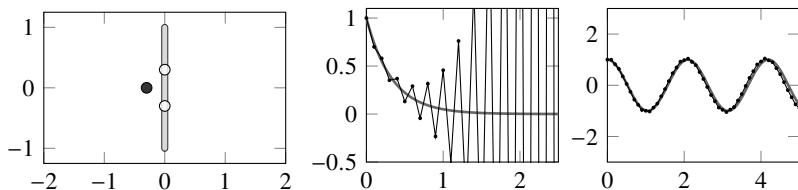
## forward Euler



## backward Euler



## leapfrog



## trapezoidal

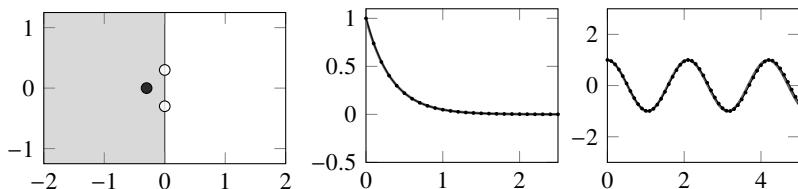


Figure 12.4: Regions of absolute stability and solutions to  $u' = -3u$  and  $u'' = -9u$  for step size  $k = 0.1$ . The eigenvalues for these ODEs are  $-3$  and  $\pm 3i$ , respectively. The value  $\bullet \lambda k = -0.3$  falls outside of the region of absolute stability for the leapfrog method and the value  $\circ \lambda k = \pm 0.3i$  falls outside of the region of absolute stability for the forward Euler method resulting in an instability.

**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. 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  $|r| = 1$  along the entire imaginary axis, and the region of absolute stability includes the entire left half  $\lambda k$ -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 discussing specific numerical methods in-depth, let's 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 nonspecific placeholder. It could stand-in for some spatial dimensions if we have a partial differential equation. Or, it could be nothing at all if we have an ordinary differential equation. In this 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 of 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  $c_\tau$  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  $\alpha$ , 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,

$$\|e_k(t_n, \cdot)\| \leq c_s \left( \|e_k(0, \cdot)\| + k \sum_{j=1}^n \|\tau_k(t_{j-1}, \cdot)\| \right).$$

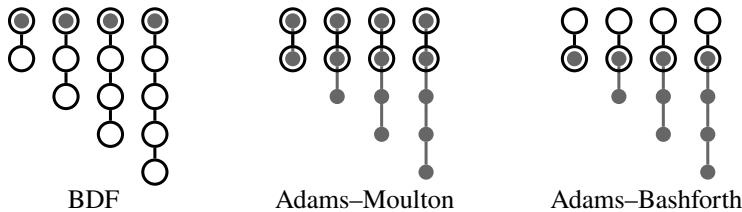
We have  $\|\tau_k(t_{j-1}, \cdot)\| \rightarrow 0$  as  $k \rightarrow 0$  from the consistency condition. And if the method is  $p$ th order consistent, then  $\|\tau_k(t_{j-1}, \cdot)\| \leq c_\tau k^p$ . So,

$$\|e_k(t_n, \cdot)\| \leq c_s (\|e_k(0, \cdot)\| + c_\tau T k^p).$$

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 generate first-order approximations of the solution using piecewise linear functions. We can get more accurate approximations of the derivative by using higher-order piecewise polynomials—piecewise quadratic, piecewise cubic, etc. 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 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

$$\frac{\frac{3}{2}u(t_{n+1}) - 2u(t_n) + \frac{1}{2}u(t_{n-1})}{k} = u'(t_{n+1}) + O(k^2).$$

Using 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  $\lambda k$  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  $\lambda k$  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); plot!(aspect_ratio=:equal)
```

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( $\alpha$ )-stable*. A method is *A( $\alpha$ )-stable* if the region of stability contains the sector in the negative  $\lambda k$ -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.

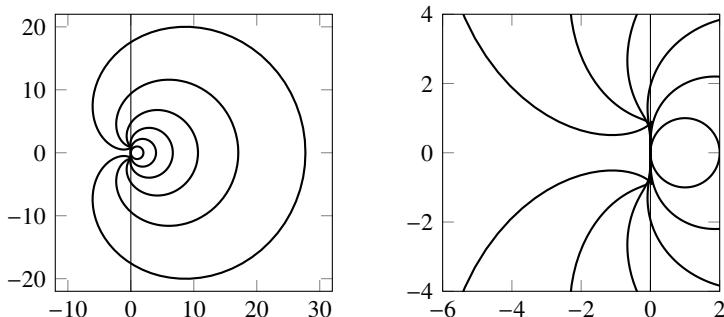


Figure 12.5: Left: internal contours of the regions of absolute stability for BDF1–BDF6. The regions of absolute stability are the areas outside these contours. Right: a close-up of the same contours.

### ► 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

$$\begin{aligned} u(t_{n-j}) &= u(t_n) - jku'(t_n) + \frac{1}{2}(jk)^2 u''(t_n) + \dots \\ f(u(t_{n-j})) &= u'(t_n) - jku''(t_n) + \frac{1}{2}(jk)^2 u'''(t_n) + \dots . \end{aligned}$$

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:

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & s \\ 1 & 2^2 & 3^2 & \cdots & s^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2^s & 3^s & \cdots & s^s \end{bmatrix} \begin{bmatrix} 1 \\ a_0 \\ a_1 \\ \vdots \\ a_{s-1} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 1 \\ 2 & 4 & 6 & \cdots & 2s \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s & s2^{s-1} & s3^{s-1} & \cdots & s^{s-1} \end{bmatrix} \begin{bmatrix} b_{-1} \\ b_0 \\ b_1 \\ \vdots \\ b_{s-1} \end{bmatrix} = 0.$$

We can solve this system with a few lines of code in Julia. The following function returns the multistep coefficients for stencils given by  $m$  and  $n$ :

```
function multistepcoefficients(m,n)
    s = length(m) + length(n) - 1
    A = (m.+1).^(0:s-1) .|> Rational
    B = (0:s-1).*((n.+1).^[0;s-2])
    c = -[A[:,2:end] B]\ones(Int64,s)
    [1;c[1:length(m)-1]], c[length(m):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  $0$  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  $m = [1 \ 2]$  and  $n = [1 \ 2 \ 3]$ , 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
```

```
m = [0 1]; n = [0 1 2]
a, b = zeros(maximum(m)+1), zeros(maximum(n)+1)
a[m.+1], b[n.+1] = multistepcoefficients(m,n)
plotstability(a,b)
```

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

$$\begin{bmatrix} \mathbf{a}^T \\ \mathbf{b}^T \end{bmatrix} = \begin{bmatrix} 1 & a_1 & a_2 & \cdots & a_{s/2} & 0 & \cdots & 0 \\ b_0 & b_1 & b_2 & \cdots & a_{s/2} & 0 & \cdots & 0 \end{bmatrix}.$$

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*.<sup>1</sup>

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

---

<sup>1</sup>In 1963, Dahlquist established his second barrier stating that no explicit scheme can be A-stable.

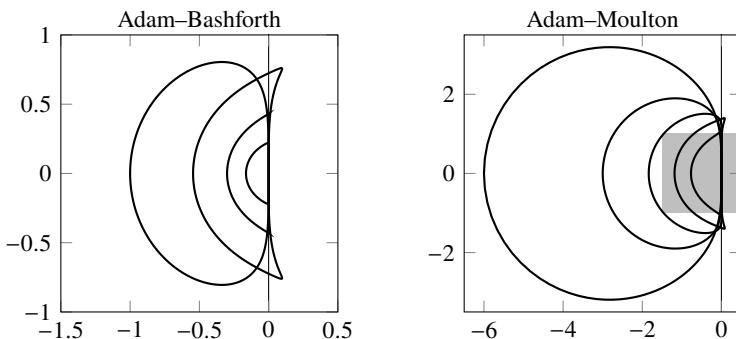


Figure 12.6: External contours of the regions of absolute stability for the explicit Adams–Bashforth methods AB2–AB5 (left) and implicit Adams–Moulton methods AM3–AM7 (right). The plot region of the Adams–Bashforth is shown with the gray rectangle overlaying the plot region of the Adams–Moulton.

intermediate time somewhere 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,

$$U^{n+1} = U^n + k \sum_{j=-1}^{s-1} b_j f(U^{n-j}).$$

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 requires 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 approach 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 as predictor–corrector methods, using the Adams–Bashforth method as a predictor and one or more functional iterations of the Adams–Moulton method as a corrector—an approach called an Adams–Bashforth–Moulton (ABM) method:

$$\text{predictor: } \tilde{U}^{n+1} = U^n + k \sum_{j=0}^{s-1} b_j f(U^{n-j}) \quad (12.7\text{a})$$

$$\text{corrector: } U^{n+1} = U^n + k b_0^* f(\tilde{U}^{n+1}) + k \sum_{j=0}^{s-1} b_j^* f(U^{n-j}). \quad (12.7\text{b})$$

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) $^\infty$  is simply an implementation of the implicit method.

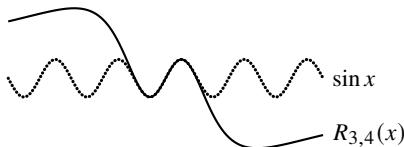
### ► Padé approximation

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

$$R_{3,4}(x) = \frac{x - \frac{31}{294}x^3}{1 + \frac{3}{49}x^2 + \frac{11}{5880}x^4}$$



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 an approximant of the logarithm.

We can compute the Padé approximant of  $\log r$  using a symbolic mathematics language like Wolfram.<sup>2</sup> For example,  $R_{1,3}(x)$  is constructed using

---

<sup>2</sup>WolframCloud (<https://www.wolframcloud.com>) provides free, online access to Mathematica.

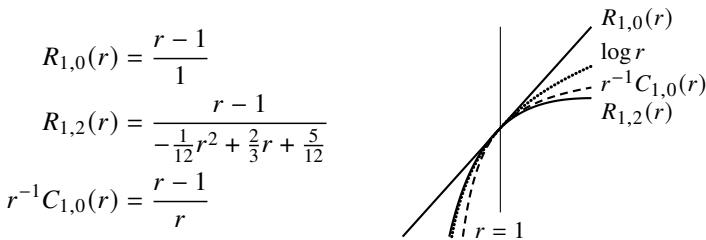


Figure 12.7: Padé approximant representations of the backward Euler, third-order Adams–Moulton, and forward Euler methods.

```
m = 1; n = 3;
Simplify[PadeApproximant[Log[r], {r, 1, {m, n}}]]
```

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 347.

Implicit methods correspond to the Padé approximations of  $\log r$ . An order- $p$  BDF method 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 an order- $p$  Adams–Bashforth method corresponds to  $r^{-1}C_{1,p-1}(r)$ . The figure above shows the Padé approximants corresponding 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 evaluate the integral numerically. 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 to 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

$$\int_a^b f(x) dx \approx (b - a)f\left(\frac{b + a}{2}\right).$$

We'll need to approximate  $f(t_{n+1/2}, U^{n+1/2})$  this time. 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 expressions yields 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

$$U^{n+1} = U^n + k \sum_{i=1}^s b_i K_i \quad \text{with} \quad K_i = f\left(t_n + c_i k, U^n + k \sum_{j=1}^s a_{ij} K_j\right). \quad (12.8)$$

Computing the coefficients  $a_{ij}$  is not a trivial exercise. We need to take  $c_i = \sum_{j=1}^s a_{ij}$  and  $\sum_{i=1}^s b_i = 1$  for consistency. These coefficients are often conveniently given as a *Butcher tableau*

$$\begin{array}{c|cc} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array}$$

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, the method is a diagonally implicit Runge–Kutta Method (DIRK).

The second-order Heun method (RK2) is given by

$$\begin{array}{lll} K_1 = f(t_n, U^n) & & 0 \\ K_2 = f\left(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kK_1\right) & & \frac{1}{2} \\ \hline U^{n+1} = U^n + kK_2 & & 0 \quad 1 \end{array}$$

The Butcher tableau of the trapezoidal rule is

$$\begin{array}{l}
 K_1 = f(t_n, U^n) \\
 K_2 = f(t_n + k, U^n + kK_1) \\
 U^{n+1} = U^n + k\left(\frac{1}{2}K_2 + \frac{1}{2}K_1\right)
 \end{array}
 \quad \left| \begin{array}{c|cc}
 0 & 0 & 0 \\
 1 & 0 & 1 \\
 \hline
 \frac{1}{2} & \frac{1}{2} & \frac{1}{2}
 \end{array} \right.$$

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.

$$\begin{array}{l}
 K_1 = f(t_n, U^n) \\
 K_2 = f(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kK_1) \\
 K_3 = f(t_n + \frac{1}{2}k, U^n + \frac{1}{2}kK_2) \\
 K_4 = f(t_n + k, U^n + kK_3) \\
 U^{n+1} = U^n + \frac{1}{6}k(K_1 + 2K_2 + 2K_3 + K_4).
 \end{array}
 \quad \left| \begin{array}{c|cccc}
 0 & 0 & 0 & 0 & 0 \\
 \frac{1}{2} & \frac{1}{2} & & & \\
 \frac{1}{2} & 0 & \frac{1}{2} & & \\
 1 & 0 & 0 & 1 & \\
 \hline
 \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} &
 \end{array} \right.$$

**Example.** Plot the regions of stability for the Runge–Kutta methods given by the following tableaus:

$$\left| \begin{array}{c|cc}
 0 & 0 & 0 \\
 \hline
 1 & 1 & 1
 \end{array} \right| \quad \left| \begin{array}{c|cc}
 0 & 0 & 0 \\
 \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\
 \hline
 1 & -1 & 2
 \end{array} \right| \quad \left| \begin{array}{c|ccc}
 0 & 0 & 0 & 0 & 0 \\
 \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & & \\
 \frac{1}{2} & 0 & \frac{1}{2} & & \\
 1 & 0 & 0 & 1 & \\
 \hline
 \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} &
 \end{array} \right|$$

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  $\lambda k$  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

$$\mathbf{K} = \lambda(U^n \mathbf{E} + k\mathbf{A}\mathbf{K}) \quad (12.9)$$

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

$$r = 1 + \lambda k \mathbf{b}^T (\mathbf{I} - \lambda k \mathbf{A})^{-1} \mathbf{E}. \quad (12.10)$$

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.  $\blacktriangleleft$

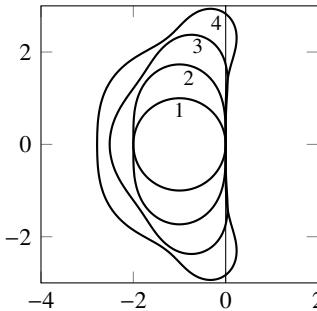


Figure 12.8: External contours of the regions of absolute stability for the explicit Runge–Kutta methods RK1–RK4. The regions of stability get progressively larger with increasing order.

**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.

$$\begin{array}{l} K_1 = f(U^n) \\ K_2 = \frac{1}{4}f(U^n) + \frac{1}{4}f(U^{n+1/2}) \\ K_3 = \frac{1}{3}f(U^n) + \frac{1}{3}f(U^{n+1/2}) + \frac{1}{3}f(U^{n+1}) \\ U^{n+1} = U^n + k\left(\frac{1}{3}K_1 + \frac{1}{3}K_2 + \frac{1}{3}K_3\right) \end{array} \quad \left| \begin{array}{c} 0 \\ \frac{1}{2} \\ 1 \\ \hline \frac{1}{3} \end{array} \right| \quad \left| \begin{array}{ccc} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{array} \right|$$

Instead of breaking the time step into two equal subintervals, we could also take the intermediate point at some fraction  $\alpha$  of the time step. In this case, we take the trapezoidal method over a subinterval  $\alpha k$  and the BDF2 method over the two subintervals  $\alpha k$  and  $(1 - \alpha)k$  with  $0 < \alpha < 1$ . 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 on 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 that 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	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$
		$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$
		$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$
				0
				$\frac{1}{8}$

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  $2k$ .

The notation  $RKm(n)$  is occasionally 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 use  $(m, n)$  or  $n(m)$  instead. 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.

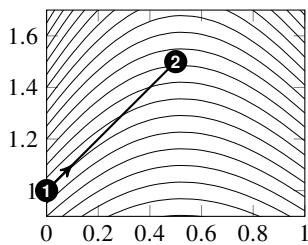
### ► Implicit Runge–Kutta methods

Another approach to Runge–Kutta methods is through collocation. Collocation is a method of problem-solving that uses 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

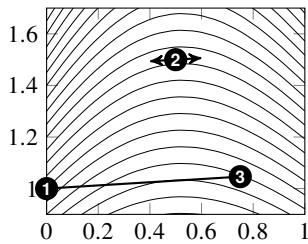
$$u'(t_0 + c_i k) = f(t_0 + c_i k, u(t_0 + c_i k)) \quad (12.11)$$

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{3}{4}$	0	$\frac{3}{4}$		
1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$	
	$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{8}$

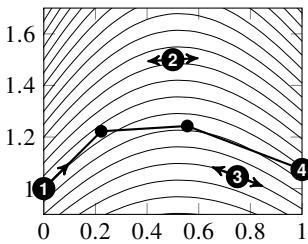
The Bogacki–Shampine scheme given by the Butcher tableau on the left is implemented in Julia using BS3. It can be implemented as an adaptive step size routine.



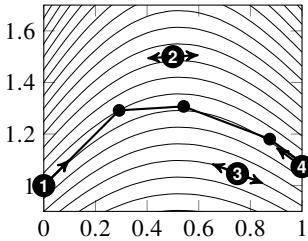
We start with the second row of the Butcher tableau. Using the slope  $f(0, u(0))$  at ① we find the approximate solution at  $t = \frac{1}{2}$  by using a simple forward Euler approximation to arrive at ②.



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:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \quad \begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \quad \begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}.$$

**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 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

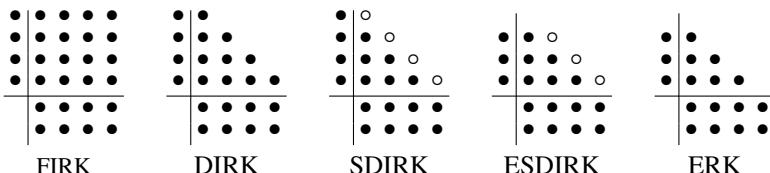
$$K_i = f\left(U^n + k \sum_{j=1}^i a_{ij} K_j\right).$$

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 nonzero values.



Stability, accuracy, and computational complexity tend to decrease 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 often be solved analytically. This section briefly discusses nonlinear differential equations, starting with the stability condition. For linear differential equations, we used the eigenvalue  $\lambda$  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

$$u' = f'(u^*)u + (f(u^*) - f'(u^*)u^*).$$

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. We may opt for an iterative technique like SOR for large, sparse systems. 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

$$U^{n(j+1)} = k b_0 f(U^{n(j)}) + U^{n-1} + k \sum_{i=1}^s b_i f(U^{n-i}),$$

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  $\phi(u)$ . We can compute the Jacobian matrix analytically or numerically using a finite difference approximation. We don't necessarily need to compute it 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.

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. To handle such a situation, one might use an explicit method to get a sufficiently close guess and then an implicit method to get even closer for each time step. These approaches are called predictor-corrector methods and often combine Adams–Bashforth and Adams–Moulton methods.

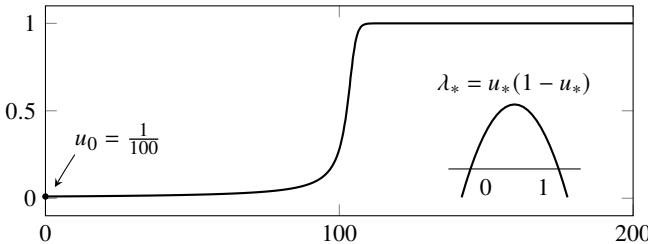
## 12.7 Stiff equations

A differential equation is said to be *stiff* if the dynamics of the solution react on at least two vastly 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 combustion model, which he credits to Larry Shampine.<sup>3</sup> Imagine striking a match. The ball of flame expands quickly until the amount of oxygen available through the surface balances the amount 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  $u(0) = u_0$ . By taking an initial condition of  $u_0 = \frac{1}{100}$ , we have the following dynamics:

---

<sup>3</sup>Shampine 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,  $\lambda_*$  is close to zero. As  $u_*$  increases,  $\lambda_*$  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 many 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 a 1952 article. They motivated their discussion by describing free radicals that are rapidly created and destroyed compared to an overall chemical reaction. The dynamics of the equations occur faster than the desired resolution of the solution. 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.<sup>4</sup>

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 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  $\lambda$  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 a 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.

---

<sup>4</sup>Söderlind, 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.

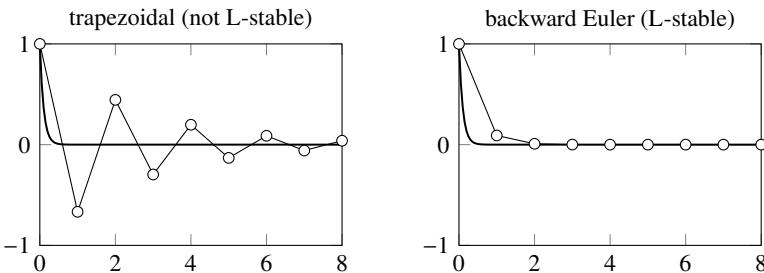


Figure 12.10: The numerical solutions to  $u' = -10u$  using the second-order trapezoidal method and first-order backward Euler method with step size  $k = 1$ .

Consider the initial value problem  $u' = -\lambda u$  for a positive factor  $\lambda$ . We can write the trapezoidal method  $U^{n+1} - U^n = -\frac{1}{2}\lambda k (U^{n+1} + U^n)$  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  $\lambda k$  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.

We can write the backward Euler method  $U^n - U^{n-1} = -\lambda k U^n$  as

$$U^n = \left( \frac{1}{1 + \lambda k} \right)^n U^0 = (r(\lambda k))^n U^0.$$

Now, when  $\lambda k$  is large, the multiplying factor  $r(\lambda k) \approx 0$ , and the numerical solution decays quickly.

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

Partial differential equations often have stiff, linear terms coupled with nonstiff, 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	$i \frac{\partial \psi}{\partial t} = -\frac{1}{2} \Delta \psi +  \psi ^2 \psi$

We can treat the linear terms implicitly to handle stiffness and the nonlinear terms explicitly to simplify implementation—an approach 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  $u' = \mathbf{N}u + \mathbf{L}u$ , where  $\mathbf{N}u = \cos^2 u$  and  $\mathbf{L}u = -10u$ . The solution  $u$  is being forced by  $\mathbf{N}u$  and  $\mathbf{L}u$  *concurrently*. We can't solve this problem analytically, but we can approximate the solution over a short time interval by considering  $\mathbf{N}u$  and  $\mathbf{L}u$  as acting *successively*. We can do this in two ways:

1. First solve  $u' = \mathbf{N}u$ . Then use the solution as initial conditions of  $u' = \mathbf{L}u$ .
2. First solve  $u' = \mathbf{L}u$ . Then use the solution as initial conditions of  $u' = \mathbf{N}u$ .

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

$$u(k) = \tan^{-1}(k + \tan(e^{-10k} u_0)).$$

The natural question is, “how close are these approximations to the exact solution?” Both of these solutions are plotted along curves ❶ and ❷ in Figure 12.11 on the following page.

Take the problem  $u' = \mathbf{A}u + \mathbf{B}u$ , where  $\mathbf{A}$  and  $\mathbf{B}$  are arbitrary operators. Consider a simple splitting where we alternate by solving  $u' = \mathbf{A}u$  and then solving  $u' = \mathbf{B}u$  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.

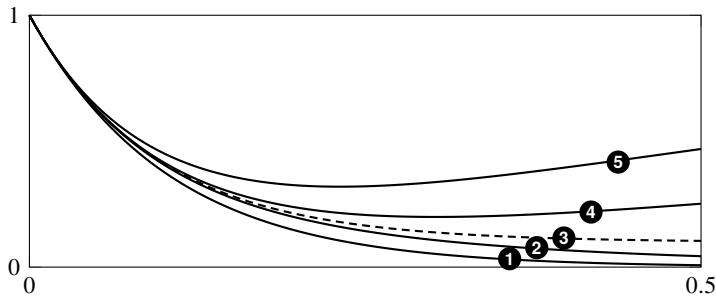


Figure 12.11: Solution to (12.12) using general splitting in ❶ and ❸, Strang splitting in ❷ and ❹, and exactly in ❻. Each operator is inverted without numerical error over a time step  $k = 0.5$ .

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  $u' = \mathbf{A} u$  for a half time step. Because

$$e^{\frac{1}{2}k\mathbf{A}} e^{k\mathbf{B}} e^{\frac{1}{2}k\mathbf{A}} - e^{k(\mathbf{A} + \mathbf{B})} = O(k^3),$$

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  $u' = \mathbf{A} u$  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 approach to a stiff problem is an implicit-explicit (IMEX) method. An IMEX method uses the approximation of  $u'(t)$  for the implicit, linear term and the explicit, nonlinear term.

A typical second-order IMEX method combines the second-order Adams–Bashforth and the second-order Adams–Moulton methods (the trapezoidal method). Both of these methods approximate the time derivative to second order at  $t_{n+1/2}$ . For  $u' = \mathbf{L} u + \mathbf{N} u$ , we have

$$\frac{U^{n+1} - U^n}{k} = \mathbf{L} U^{n+1/2} + \mathbf{N} U^{n+1/2} \approx \frac{1}{2} \mathbf{L} U^{n+1} + \frac{1}{2} \mathbf{L} U^n + \frac{3}{2} \mathbf{N} U^n - \frac{1}{2} \mathbf{N} U^{n-1}. \quad \blacktriangleleft$$

Alternatively, we can build a second-order IMEX scheme using the BDF2 method for the linear part  $u' = \mathbf{L} u$ :

$$\frac{\frac{3}{2}U^{n+1} - 2U^n + \frac{1}{2}U^{n-1}}{k} = \mathbf{L} U^{n+1}.$$

A BDF method approximates the derivative at  $t_{n+1}$ . To avoid a splitting error, we should also evaluate the nonlinear term  $\mathbf{N} U$  at  $t_{n+1}$  by extrapolating 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. To simplify the discussion, we'll take unit mass, length, and gravitational acceleration. 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<sup>5</sup>

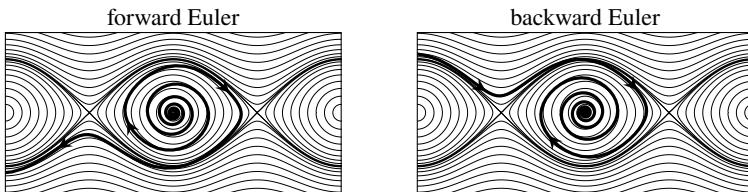
$$\frac{dq}{dt} = p \quad \text{and} \quad \frac{dp}{dt} = -\sin q.$$

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

---

<sup>5</sup>This problem can be solved analytically in terms of elliptic functions. See 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 code at the bottom of this page. In both cases, the pendulum starts in the same position. When the pendulum has sufficient momentum  $|p_0| > 2$ , it has enough energy 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 a tiny swing, it 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. Energy is not conserved in either method, resulting in unphysical solutions. In problems such as planetary motion and molecular dynamics, a scheme that conserves energy over a long period is essential to getting a physically correct solution. Smaller timesteps and higher-order methods can help, but energy may still not be conserved over time. 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  $\mathbf{r} \equiv (p, q)$ :

$$\frac{d\mathbf{r}}{dt} = D_H \mathbf{r} \equiv \mathbf{J} \nabla H(\mathbf{r}) \quad \text{where} \quad \mathbf{J} = \begin{bmatrix} 0 & -I \\ I & 0 \end{bmatrix}.$$

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 implicit forward Euler method and the 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

$$P^{n+1} = P^n - k \nabla_q H(P^n, Q^{n+1}) \quad (12.14a)$$

$$Q^{n+1} = Q^n + k \nabla_p H(P^n, Q^{n+1}). \quad (12.14b)$$

We'll 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  $\alpha k$  followed by a Verlet with timestep  $(1 - 2\alpha)k$  followed by a Verlet with timestep  $\alpha k$  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 entirely 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 two hundred—far too many to cover in any depth here.<sup>6</sup> Several of these methods correspond to the standard solvers in Matlab and Python—see the table in Figure 12.12. This section summarizes these standard solvers and their suitability for different problem types. Given Matlab's influence on Python's SciPy and Julia, it makes some sense to start the discussion with it.

---

<sup>6</sup>For a complete list of Julia's ODE solvers, see [https://diffeq.sciml.ai/latest/solvers/ode\\_solve/](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 principal 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>effic.</i>
Rosenbrock (order 4)	Rodas4	—	—	●	●
Radau IIA 5(3)	RadauIIA5	—	Radau	●	●
BDF (DAE/implicit)	DFBDF	ode15i	IDA	●	○
Rosenbrock (order 2)	Rosenbrock23	ode23s	—	●	○
TR-BDF2 SDIRK	TRBDF2	ode23tb <sup>†</sup>	—	●	○
BDF	FBDF	ode15s	BDF, LSODA, CVODE	○	●
Additive IMEX RK	KenCarp4	—	—	○	●
Trapezoidal ESDIRK	Trapezoid	ode23t <sup>†</sup>	—	●	○
Verner 9(8)	Vern9	ode89 <sup>†</sup>	—	○	●
Verner 8(7)	Vern8	ode78 <sup>†</sup>	—	○	●
Dormand–Prince 8(5,3)	DP8	—	DOP853	○	●
Tsitouras 5(4)	Tsit5	—	—	○	●
Dormand–Prince 4(5)	DP5	ode45	RK45	○	●
ABM	VCABM	ode113 <sup>‡</sup>	LSODA, CVODE	○	○
Bogacki–Shampine 2(3)	BS3	ode23	RK23	○	○

○ low   ● medium   ● high

Figure 12.12: Equivalent routines in Julia, Matlab, and Python. The table includes only a fraction of all methods available in Julia. Matlab and Python routines are available in Julia using wrappers. <sup>†</sup>Only MATLAB. <sup>‡</sup>Octave uses lsode instead.

## ► Matlab

For a complete 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: the suite should have a uniform interface so the different specialized and robust solvers could be called *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.

Matlab has four explicit Runge–Kutta methods designed for nonstiff problems. The low-order ode23 routine is the third-order Bogacki–Shampine Runge–Kutta method presented on page 356. The medium-order ode45 routine is the popular 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 sometimes recommended as the go-to solver for nonstiff problems. The high-order ode78 and ode89 routines—Verner’s “most

efficient” 7(6) and 8(9) Runge–Kutta methods—often outperform other methods, particularly on problems with smooth solutions. Another routine designed for nonstiff problems, `ode113`, is a variable-step, variable-order Adams–Bashforth–Moulton PECE. It computes a solution up to order 12 and an error estimate up to order 13 to control the variable step size. The `ode15s` routine uses a variation on Klopfenstein–Shampine numerical differentiation formulas. These formulas are modifications of BDF methods with variable-step, variable-order to order 5 and have good stability properties, especially at higher orders. The `ode15i` routine is a similar variable-step, variable-order BDF method designed for fully implicit problems  $f(t, u, u') = 0$ . All 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 routine will compute it using a finite difference approximation. The `ode23s` routine uses a second-order modified Rosenbrock formula designed especially for stiff problems with sharp changes (quasi-discontinuities) in the solutions. The `ode23t` routine implements the trapezoidal Runge–Kutta method. Because 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. The method doesn’t have an embedded error estimate—instead, 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 355 taking  $\alpha = 2 - \sqrt{2}$ .

Octave’s ODE solvers have some notable differences from the Matlab solvers. In addition to five Matlab-compatible solvers `ode23`, `ode23s`, `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 nonstiff problems, using functional iteration to evaluate the implicit terms.

## ► Python

Python’s `scipy.integrate` library has several ODE solvers. The `RK45` (default), `RK23`, and `BDF` routines are equivalent to Matlab’s `ode45`, `ode23`, and `ode15s` routines. The `DOP853` routine implements the Dormand–Prince 8(5,3) Runge–Kutta. `LSODA` 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` routine is an order-5 Radau IIA (fully-implicit Runge–Kutta) method. 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 using a finite difference approximation. These routines can also be called from `solve_ivp`, a generic interface that uses adaptive step size over a

prescribed interval of integration, through the `method` option. These methods can also be called for one time step, allowing them to be integrated into time-splitting routines.

The scikits.odes package provides a few more routines. Two of these routines, `CVODE` and `IDA`, come from LLNL’s Sundials<sup>7</sup> library. The Sundials routine `CVODE`<sup>8</sup> includes Adams–Moulton formulas (with orders up to 12) for nonstiff problems and BDFs (with orders up to 5)—similar to `LSODA`. The Sundials `IDA` routine solves differential-algebraic equation systems in the form  $f(t, u, u') = 0$ .

## ► Julia

Julia’s flagship ODE package is `DifferentialEquations.jl`, developed by Chris Rackauckas and Qing Nie. `DifferentialEquations.jl` includes routines equivalent to those in Matlab and Python and many other methods that may be more efficient than the standard ones. 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 routines, check out the documentation at <https://diffeq.sciml.ai>.

Julia also has a wrapper for Python’s `scipy.integrate.solve_ivp` module (`SciPyDiffEq.jl`), a wrapper for the Sundials library (`Sundials.jl`), and a wrapper for licensed MATLAB installations (`MATLABDiffEq.jl`), among others. The `Sundials.jl` package has all six Sundials methods, including `ARKode`. Additive Runge–Kutta (`ARK`) methods are IMEX methods that combine an `ERK` for nonstiff terms and a `DIRK` for stiff terms. The routines found in `ARKode`, a class of `ARK` methods developed by Christopher Kennedy and Mark Carpenter, are also available through a class of `DifferentialEquations.jl` routines, such as `KenCarp4`.

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 symplectic or almost symplectic method, 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):

---

<sup>7</sup>A portmanteau of SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers

<sup>8</sup>The acronym `CVODE` refers to a C-language implementation of the Fortran `VODE`, which uses variable-coefficient methods instead of the fixed-step-interpolate methods in `LSODE`.

```

1. Load the module      using DifferentialEquations, Plots
2. Set up the parameters pendulum(du,p,t) = [u[2]; -sin(u[1])]
u₀= [8π/9,0]; tspan = [0,8π]
3. Define the problem   problem = ODEProblem(pendulum, u₀, tspan)
4. Choose the method    method = Trapezoid()
5. Solve the problem    solution = solve(problem,method)
6. Present the solution 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  $f(u, t) \equiv L u$  as a linear operator using the `DiffEqArrayOperator` function from the `DiffEqOperators.jl` library.

• DifferentialEquations.jl is a suite of utilities and hundreds of ODE solvers.

## ► Efficiency

No one solver works best for all problems in all scenarios. One routine may be relatively efficient for one class of problems; another one may be better suited for a different class. As we've seen throughout this chapter, a routine's accuracy and computing time are a combination of several competing and compounding factors. Benchmarking different solvers on various standard problems is one approach to measuring performance. Figure 12.12 summarizes the relative efficiencies of routines determined in the following benchmarking example.

**Example.** Let's examine the efficiencies of different numerical routines for nonstiff and stiff problems. The Lotka–Volterra equation is a predator-prey population model

$$x' = \alpha x - \beta xy, \quad y' = -\gamma y + \delta xy.$$

The variable  $x$  is the population of the prey (e.g., rabbits) and the variable  $y$  is the population of the predators (e.g., lynxes). We'll take the parameters  $\{\alpha, \beta, \gamma, \delta\} = \{1.5, 1, 3, 1\}$ . Notice how the populations of prey — and predators — fluctuate symbiotically over time:



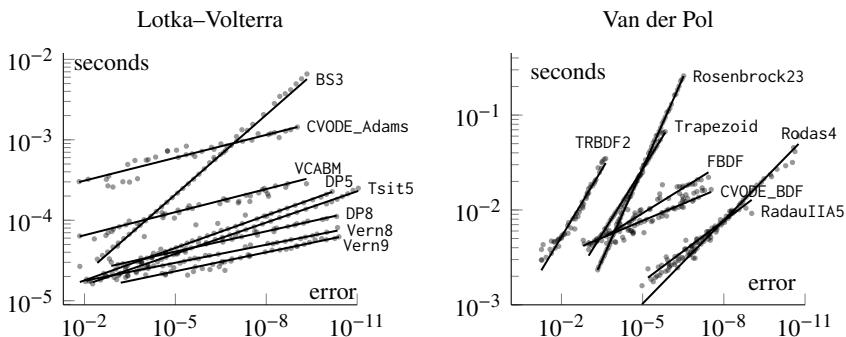


Figure 12.13: Computing time versus precision for the Lotka–Volterra equation (nonstiff) and the Van der Pol equation with  $\mu = 1000$  (stiff).

The Van der Pol equation models a nonlinear relaxation oscillator

$$x'' - \mu(1 - x^2)x' + x = 0,$$

where the nonnegative parameter  $\mu$  controls damping. When  $\mu$  is zero, the Van der Pol equation is a simple linear oscillator. When  $\mu$  is large, the Van der Pol equation exhibits slow decay punctuated by rapid phase shifts.<sup>9</sup> The following plot shows the Van der Pol oscillations when  $\mu = 100$  with  $x(0) = 2$ :



Let's examine the relative efficiencies of some standard routines by comparing the compute times and solution accuracies across a range of tolerances. We'll use Julia's `DiffEqDevTools.jl` package, which has utilities for benchmarking solvers.<sup>10</sup> The efficiency of several nonstiff solvers and several stiff solvers are depicted using a log-log plot in Figure 12.13. Outliers in the data have been removed using the `LinRegOutliers.jl` package, and regression lines are fit through the points. More efficient methods are in the lower-right quadrant, and less efficient methods are in the upper-left quadrant. For example, on the Lotka–Volterra problem, Verner's “most efficient” method is, in fact, the most

<sup>9</sup>In his original paper, Balthazar van der Pol examined the oscillating current driven by a triode, which required that  $\mu$  be smaller than one. Van der Pol later studied behavior for large values of  $\mu$  and coined the term “relaxation oscillation.”

<sup>10</sup>The code for this example is available in the Jupyter notebook that accompanies this book. Chris Rackauckas' `SciMLBenchmarks.jl` project also presents summaries and plots of several benchmarks using the `DiffEqDevTools.jl` package.

efficient—about 14 times faster than the Adams–Moulton method (CVODE\_Adams). The high-order Adams–Moulton method is also relatively inefficient compared to the third-order Bogacki–Shampine method at low tolerances, but it outperforms the Bogacki–Shampine method at high tolerances. 

## 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	$\frac{2}{3}$	$\frac{1}{6}$

12.5. S676.1.5 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 that the error using Strang splitting is  $O(k^2)$ .

12.6. The multistage trapezoidal and BDF2 introduced in the example on page 355 are both implicit methods. Implementing each stage typically requires computing a Jacobian matrix for Newton's 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 method's stability 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 350. 

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 pendulums 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

$$x''(t) + \gamma x'(t) + \alpha x + \beta x^3 = \delta \cos \omega t$$

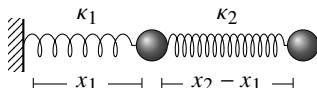
models the motion of a nonlinear damped driven oscillator. Think of a weight attached to a spring attached to a piston. The spring becomes stiffer or softer as it is stretched by adding a cubic term to the linear Hooke's law. The parameter  $\gamma$  is a damping coefficient,  $\alpha$  is the stiffness 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.



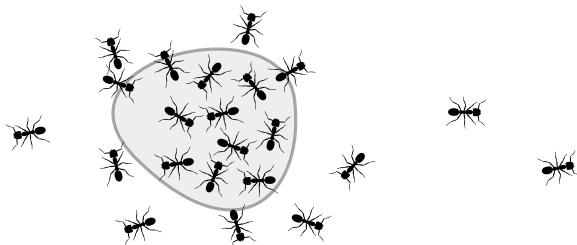
$$\begin{aligned} x_1'' &= -\kappa_1 x_1 + \kappa_2(x_2 - x_1) \\ x_2'' &= -\kappa_2(x_2 - x_1) \end{aligned}$$

Suppose that the 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. This behavior can be modeled on a large scale 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 others 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 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



$$J = -\alpha(x) \frac{\partial u}{\partial x}.$$

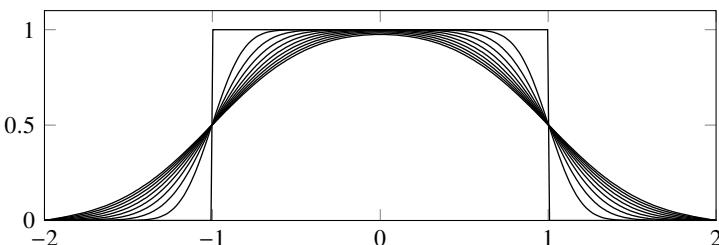
And, in general,

$$\frac{\partial u}{\partial t} = \nabla \cdot (\alpha(\mathbf{x}) \nabla u).$$

Consider an insulated heat-conducting bar. The 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 the method of 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  $[-2, 2]$ . The following figure and the QR link at the bottom of this page show snapshots of the solution taken at equal time intervals from  $t \in [0, 0.1]$ :



solution to the heat equation

Notice how initially the solution rapidly evolves 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, but 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 a uniform mesh  $x_j = jh$ , where  $h$  is the grid spacing. Let  $U_j(t)$  be a finite difference approximation to  $u(t, x_j)$  in space. We can use a central difference

---

<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.

approximation

$$\mathbf{D}^2 U_j = \mathbf{D} \left( \frac{U_{j+1/2} - U_{j-1/2}}{h} \right) = \frac{U_{j+1} - 2U_j + U_{j-1}}{h^2}$$

to get a second-order approximation of the second derivative. 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 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. The previous chapter discussed the forward Euler, the backward Euler, the leapfrog, and the trapezoidal methods to solve ODEs. Let's apply these methods to the linear diffusion equation. The leapfrog method applied to the diffusion equation is called the Richardson method, and the trapezoidal method applied to the diffusion equation is called the Crank–Nicolson method. British mathematicians John Crank and Phyllis Nicolson developed their method in 1948 as a replacement for the unstable method proposed by Lewis Fry Richardson in 1910. The following stencils correspond to the stencils on page 333.

	Forward Euler	$O(k + h^2)$
		$\frac{U_j^{n+1} - U_j^n}{k} = \alpha \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}$

(13.3)

	Backward Euler	$O(k + h^2)$
		$\frac{U_j^{n+1} - U_j^n}{k} = \alpha \frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{h^2}$

(13.4)

	Richardson ( <i>unstable!</i> )	$O(k^2 + h^2)$
		$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = \alpha \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}$

(13.5)



Crank–Nicolson       $O(k^2 + h^2)$

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{1}{2}\alpha \frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{h^2} + \frac{1}{2}\alpha \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \quad (13.6)$$

## ► 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.<sup>2</sup>

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

$$\frac{\partial}{\partial t} U_0 = \alpha \frac{2U_1}{h^2} + 2\alpha \frac{u_L}{h^2}. \quad (13.7)$$

We can similarly eliminate the right ghost point.

**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  $(1, -2, 1)$ . We need to invert a tridiagonal matrix at each step by using a

<sup>2</sup>“Science is a differential equation. Religion is a boundary condition.” —Alan Turing

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.

```
 $\Delta x = .01; \Delta t = .01; L = 2; \nu = \Delta t / \Delta x^2; u_l = 0; u_r = 0;$ 
 $x = -L : \Delta x : L; m = \text{length}(x)$ 
 $u = (\text{abs.}(x). < 1)$ 
 $u[1] += 2\nu * u_l; u[m] += 2\nu * u_r$ 
 $D = \text{Tridiagonal}(\text{ones}(m-1), -2\text{ones}(m), \text{ones}(m-1))$ 
 $D[1, 2] = 0; D[m, m-1] = 0$ 
 $A = I - \nu * D$ 
 $\text{for } i \text{ in } 1:20$ 
 $u = A \backslash u$ 
 $\text{end}$ 
```

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  $D = \text{Matrix}(D)$ , 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 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 using 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

$$(2 + 2\nu)U_0^{n+1} - 2\nu U_1^{n+1} = (2 - 2\nu)U_0^n + 2\nu U_1^n$$

$$-\nu U_{j-1}^{n+1} + (2 + 2\nu)U_j^{n+1} - \nu U_{j+1}^{n+1} = \nu U_{j-1}^n + (2 - 2\nu)U_j^n + \nu U_{j+1}^n$$

$$2\nu U_{m-1}^{n+1} - (2 + 2\nu)U_m^{n+1} = 2\nu U_{m-1}^n + (2 - 2\nu)U_m^n$$

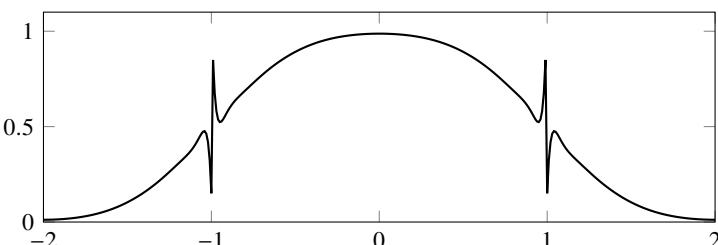
Now, we need to solve the system  $(\mathbf{I} - \nu\mathbf{D})\mathbf{U}^{n+1} = (\mathbf{I} - \nu\mathbf{D})\mathbf{U}^n$  where  $\mathbf{D}$  is a modified tridiagonal matrix  $(1, -2, 1)$ . The following Julia code implements the Crank–Nicolson method with reflecting boundary conditions over the 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 = float.(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 code 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 364 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, 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 the boundary 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*  $u(t, 0) = u(t, 1)$ , we can no longer make the system tridiagonal. We instead have 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 to 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

The previous chapter 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, periodic boundary conditions will not significantly change the analysis because numerical stability is a local behavior.

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 wavenumber  $\xi$  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 nonpositive real numbers.

The regions of absolute 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 absolute stability. The eigenvalues  $\lambda_\xi$  can be as large as  $|\lambda_\xi| = 4\alpha/h^2$ . We can determine the stability conditions for the different numerical schemes using this bound and the associated regions of absolute stability.

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 absolute stability for all  $\xi$ . Because the magnitude of  $\lambda_\xi$  can be as large as

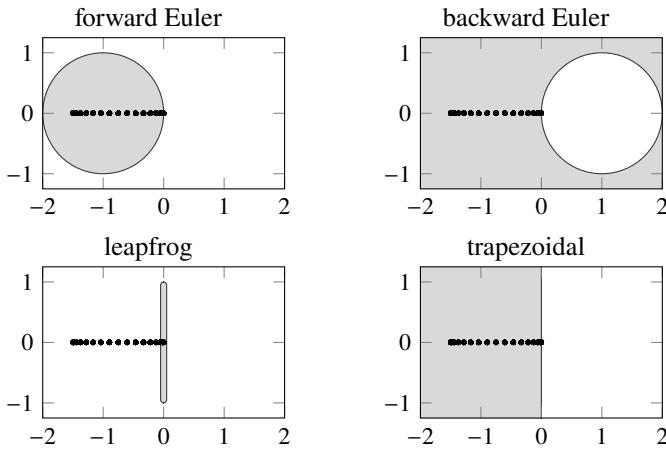


Figure 13.1: Regions of absolute stability (shaded) in the  $\lambda k$ -plane for the forward Euler, backward Euler, leapfrog, and trapezoidal methods along with eigenvalues  $\lambda_\xi$  of the central difference approximation with  $\alpha k/h^2 = \frac{3}{2}$ .

$4\alpha/h^2$ , it follows that  $\lambda_\xi k$  stays within the region of absolute stability as long as  $k \leq h^2/2\alpha$ . 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  $h$ , 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 tenfold, we need to increase the number of time steps a hundredfold. Such a restriction is impractical.

There is no constraint on the time step  $k$  for the backward Euler method when the eigenvalues are on the negative real axis. The backward Euler method is *unconditionally stable* for (13.8). 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. We'll still need to keep  $k = O(h^2)$  to achieve second-order accuracy overall. We'd be better off pairing (13.8) with a scheme that is  $O(k^2)$  in time.

In the above analysis, we used the property that the set  $\{e^{i\xi j h}\}$  forms a linearly independent basis, and therefore, we only need to compare the Fourier coefficients. We can abbreviate the stability analysis 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

Richardson (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 Richardson method is unconditionally unstable! That the Richardson method is unconditionally unstable should be quite clear 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 slightly modify to the Richardson 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  $A^n e^{i\xi j h}$  in (13.10) and dividing by  $A^{n-1} e^{i\xi j h}$  gives us

$$\frac{A^2 - 1}{2k} = \alpha \frac{A e^{i\xi h} - (A^2 + 1) + A e^{-i\xi h}}{h^2} = \alpha \frac{2A \cos \xi h - (A^2 + 1)}{h^2}.$$

Let  $\nu = \alpha k / h^2$ , then

$$(1 + 2\nu)A^2 - (4\nu \cos \xi h)A - (1 - 2\nu) = 0.$$

The roots of 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>3</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>3</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

$$\frac{U_{ij}^{n+1} - U_{ij}^n}{k} = \frac{1}{2} (\delta_x^2 U_{ij}^{n+1} + \delta_x^2 U_{ij}^n + \delta_y^2 U_{ij}^{n+1} + \delta_y^2 U_{ij}^n). \quad (13.13)$$

For von Neumann analysis in two dimensions, we substitute  $\hat{U}_{ij}^n = A^n e^{i(\xi h + j \eta h)}$  and determine the CFL condition such that  $|A| \leq 1$ . The Crank–Nicolson method is second-order in time and space and unconditionally stable.

We no longer have a simple tridiagonal system when moving to two or three dimensions. 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. And we will need to invert a  $10^6 \times 10^6$  matrix in three dimensions. A more efficient approach is to use operator splitting. Let's examine two ways of splitting the right-hand operator of (13.13)

$$\frac{1}{2} (\overset{\textcircled{1}}{\delta_x^2 U_{ij}^{n+1}} + \overset{\textcircled{2}}{\delta_x^2 U_{ij}^n} + \overset{\textcircled{3}}{\delta_y^2 U_{ij}^{n+1}} + \overset{\textcircled{4}}{\delta_y^2 U_{ij}^n})$$

that will ensure that the method remains implicit. The fractional step method splits  $\textcircled{1} + \textcircled{2}$  from  $\textcircled{3} + \textcircled{4}$

$$\frac{U_{ij}^* - U_{ij}^n}{k} = \frac{1}{2} (\delta_x^2 U_{ij}^* + \delta_x^2 U_{ij}^n) \quad \text{and} \quad \frac{U_{ij}^{n+1} - U_{ij}^*}{k} = \frac{1}{2} (\delta_y^2 U_{ij}^* + \delta_y^2 U_{ij}^{n+1}),$$

and the alternate direction implicit (ADI) method splits  $\textcircled{1} + \textcircled{3}$  from  $\textcircled{2} + \textcircled{4}$

$$\frac{U_{ij}^* - U_{ij}^n}{k} = \frac{1}{2} (\delta_x^2 U_{ij}^* + \delta_y^2 U_{ij}^n) \quad \text{and} \quad \frac{U_{ij}^{n+1} - U_{ij}^*}{k} = \frac{1}{2} (\delta_x^2 U_{ij}^* + \delta_y^2 U_{ij}^{n+1}).$$

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

$$\frac{U^{n+i/p} - U^{n+(i-1)/p}}{k} = D_i \frac{U^{n+i/p} + U^{n+(i-1)/p}}{2} \quad \text{for } i = 1, 2, \dots, p.$$

The Crank–Nicolson method is unconditionally stable at each stage, so 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 by Donald Peaceman and Henry Rachford in 1955, while working at Humble Oil and Refining Company, in a short-lived effort to simulate petroleum reserves. The ADI method applied to the heat equation 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  $y \mapsto \eta$ )

$$\hat{U}^{n+1/2} = \hat{U}^n + \frac{1}{2}\nu \left( -4 \sin^2 \frac{1}{2}\xi h \right) \hat{U}^{n+1/2} + \frac{1}{2}\nu \left( -4 \sin^2 \frac{1}{2}\eta h \right) \hat{U}^n$$

$$\hat{U}^{n+1} = \hat{U}^{n+1/2} + \frac{1}{2}\nu \left( -4 \sin^2 \frac{1}{2}\xi h \right) \hat{U}^{n+1/2} + \frac{1}{2}\nu \left( -4 \sin^2 \frac{1}{2}\eta h \right) \hat{U}^{n+1}$$

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,

$$\hat{U}^{n+1} = \underbrace{\frac{(1 - 2\nu \sin^2 \frac{1}{2}\eta h)(1 - 2\nu \sin^2 \frac{1}{2}\eta h)}{(1 + 2\nu \sin^2 \frac{1}{2}\eta h)(1 + 2\nu \sin^2 \frac{1}{2}\eta h)}}_{\lambda} \hat{U}^n.$$

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 has the same order as the Crank–Nicolson scheme  $O(h^2 + k^2)$ . The fractional step method is easier to implement than the ADI method, but the ADI method is generally 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 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. Fortunately, 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 Netwon'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
Verner 7(6)	Vern7	○	10513	2.7
Rosenbrock 3(4)	Rodas4	●	430	12.0
Trapezoidal ESDIRK	Trapezoid	○	948	39.9
Rosenbrock 2(3)	Rosenbrock23	●	554	41.0

○ 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 as a gif

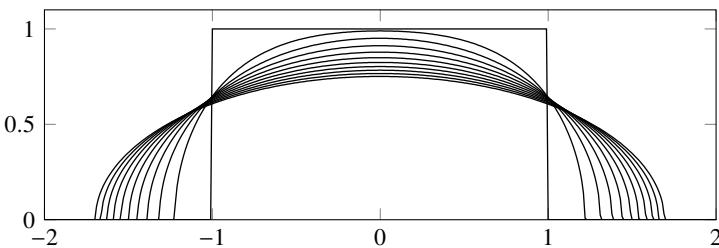
```
anim = @animate for t in LinRange(0,2,200)
    plot(x,solution(t),legend=:none,fill=(0,0.4,:red),ylims=(0,1))
end
gif(anim, "porous.gif", fps = 15)
```

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` afterward:

```
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 forward and backward 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 show 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 380.

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 nonstiff methods are shown in Figure 13.2 on the previous page. 

## ▷ Stability

We cannot use linear von Neumann stability analysis on variable-coefficient and nonlinear problems. Instead, we will use the *energy method* to check 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 nonincreasing. 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,

$$\frac{1}{2} \frac{\partial}{\partial t} \int_0^1 u^2 \, dx = - \int_0^1 \alpha(u) \left( \frac{\partial}{\partial x} u \right)^2 \, dx.$$

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 use the same approach on the numerical scheme (13.17) with the discrete  $\ell^2$ -norm. Now,

$$\sum_{j=1}^{m-1} U_j \frac{\partial}{\partial t} U_j = \sum_{j=1}^{m-1} \frac{1}{h^2} \left[ \alpha_{j+1/2}(U_{j+1} - U_j) U_j - \alpha_{j-1/2}(U_j - U_{j-1}) U_j \right],$$



solution to the porous  
medium equation

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 inequality tells us that the  $\ell^2$ -norm of the solution is nonincreasing in time.

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 nonnegative 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. S676.2.0 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  $u_t = u_{xx}$ .

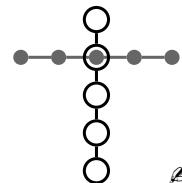
13.3. Suppose we want to solve the heat equation, but instead of using the values  $\{U_{j-1}, U_j, U_{j+1}\}$  to approximate  $u_{xx}$  at  $x_j$ , we decide to use the values  $\{U_j, U_{j+1}, U_{j+2}\}$ . For example, the stencil using the backward Euler method would be



Our approach is terrible for several 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. The Applied Mathematics Panel, an organization established by Vannevar Bush to conduct mathematical research for the U.S. military during World War II, proposed the stencil on the right to solve the heat equation. John Crank and Phyllis Nicolson later dismissed the method as being “very cumbersome.” Discuss the stability restrictions of the method.



13.5. 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  $k = O(h)$ .



13.6. 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  $h \ll k$ .

### 13.7. The Schrödinger equation

$$i\epsilon \frac{\partial \psi}{\partial t} = -\frac{\epsilon^2}{2} \frac{\partial^2 \psi}{\partial x^2} + V(x)\psi$$

models the quantum behavior of a particle in a potential field  $V(x)$ . The parameter  $\epsilon$  is the scaled Plank constant, which gives the relative length and time scales. The probability of finding a particle at a position  $x$  is  $\rho(t, x) = |\psi(t, x)|^2$  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.

- 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, Richardson, Crank–Nicolson, and Runge–Kutta.
- 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>4</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  $\varepsilon = 0.3$ . Choose a wide domain to ensure that interactions with the boundaries are negligible. Compute the slope of the log-log plot of the error with the analytic solution at  $t = 2\pi$  to confirm that the method is  $O(k^2 + h^2)$ . 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  $h \ll k$ .

- What happens when the mesh spacing  $h$  or the time step  $k$  is about equal to or larger than  $\varepsilon$ ? 

13.8. 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. 

---

<sup>4</sup>Coherent states refer to Gaussian wave packet solutions of the harmonic oscillator. Schrödinger derived them in 1926 in response to criticism that the wave function  $\psi(t, x)$  did not display classical motion and satisfy the correspondence principle.

13.9. Suppose 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 closely space points near the origin and spread them out away from the origin.

- Derive a central difference approximation to the Laplacian operator for arbitrarily-spaced gridpoints. Discuss the error of the approximation.
- Solve the heat equation by approximating open boundary conditions using a nonuniform grid. Use an inverse sigmoid function, such as  $x = \tanh^{-1} \xi$ , to generate the grid. 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  $u(x, t) = (1 + 4st)^{-1/2} e^{-sx^2/(1+4st)}$ . 

### 13.10. 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 the 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 approaching 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 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 developed considerably in the last fifty years. This chapter is an introduction—see Randall LeVeque’s textbook *Finite Volume Methods for Hyperbolic Problems* for 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:

$$\frac{d}{dt}u(t, x(t)) = \frac{\partial u}{\partial t} + \frac{dx}{dt}\frac{\partial u}{\partial x} \quad \text{and} \quad \frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0.$$

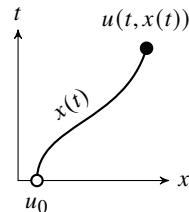
By taking  $dx/dt = c$ , the total derivative

$$\frac{d}{dt}u(t, x(t)) = 0,$$

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 the characteristic curve back 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 develop numerical schemes to solve the simple linear problem  $u_t + cu_x = 0$ . The schemes designed for this linear problem will be prototypic methods that we later extend to solve nonlinear hyperbolic problems.

### ► Upwind method

The most straightforward 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*.

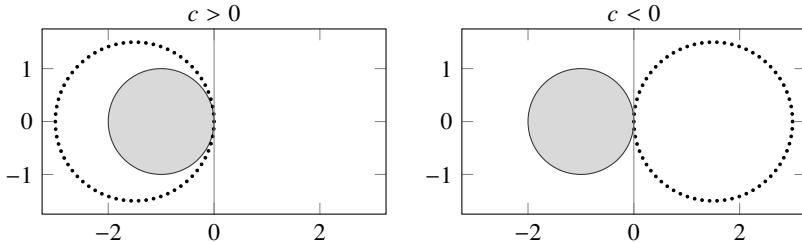
A natural first question about such a scheme might be, “what is its stability condition?” We can answer this question using the von Neumann analysis developed in the previous chapter. Consider the semidiscrete system of equations

$$\frac{\partial}{\partial t} u(t, x_j) = c \frac{u(t, x_{j-1}) - u(t, x_j)}{h}.$$

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 a numerical method is by overlaying the eigenvalues of the right-hand-side operator 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 get a sense of why we need two upwind schemes (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 a characteristic of the advection equation (14.1). In a finite 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, that same point in space can only influence a bounded region of space over a finite time. This region is called the *domain of influence*. The domain of dependence looks into the past, and the domain of influence looks into the future.

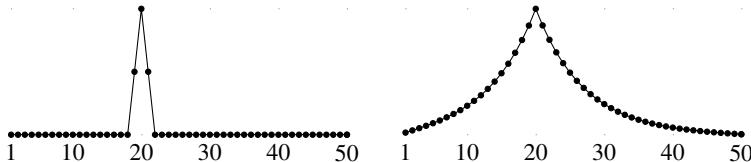
Consider the solution to the advection equation:  $u(t_n, x_j) = u_0(x_j - ct_n)$ . Take a uniform step size  $k$  in time and a uniform grid spacing  $h$ . The domain of dependence of the true solution is  $[x_j, x_j + cnk]$ . The domain of dependence of the numerical solution is  $[x_j, x_j + nh]$ . By keeping  $ck/h < 1$ , we ensure that the domain of dependence of the numerical solution contains the domain of dependence of the true 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 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

<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?

gridpoint is propagated to every other grid point in one time step. The normalized plots below show the values of a column taken from the matrices  $\mathbf{I} + k/h^2 \mathbf{D}$  with  $h = \frac{1}{2}k^2$  and  $(\mathbf{I} - k/h^2 \mathbf{D})^{-1}$  with  $h = k$ :

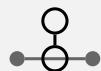


### ► Lax–Friedrichs method

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 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> )	$O(k + h^2)$
$\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_{j+1}^n + U_{j-1}^n}{2h} = 0$	(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 the imaginary axis, like the leapfrog scheme. Another way is to somehow push the eigenspectrum into the left half-plane, where we can then shrink them to fit in 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 stabilize the unstable central difference scheme (14.5). Using the Taylor series expansion about  $U_j^n = u(t_n, x_j)$ , we have

$$\begin{aligned} U_j^{n+1} &= u + ku_t + \frac{1}{2}k^2u_{tt} + O(k^3) \\ U_{j+1}^n &= u + hu_x + \frac{1}{2}h^2u_{xx} + \frac{1}{6}h^3u_{xxx} + O(k^4) \\ U_{j-1}^n &= u - hu_x + \frac{1}{2}h^2u_{xx} - \frac{1}{6}h^3u_{xxx} + O(k^4). \end{aligned}$$

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 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.

	<b>Lax–Wendroff Method</b>	$O(k^2 + h^2)$
		$\frac{U_j^{n+1} - U_j^n}{k} + c \frac{U_{j+1}^n - U_{j-1}^n}{2h} = \frac{c^2}{2} k \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}$ <span style="float: right;">(14.8)</span>

Note that by rearranging the terms of the Lax–Friedrichs scheme, we get a form that 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

$$-4 \frac{\alpha}{h^2} \sin^2 \frac{\xi h}{2} - i \frac{c}{h} \sin \xi h.$$

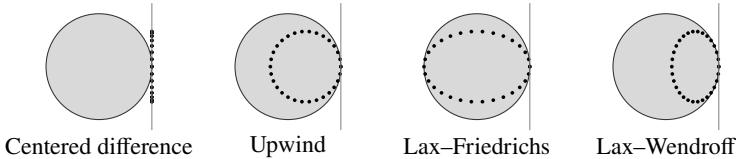
The eigenvalues lie on an ellipse in the negative half-plane bounded by  $-4\alpha/h^2$  along the real axis and  $\pm c/h$  along the imaginary axis. For a forward Euler method, this ellipse must be scaled by  $k$  to lie entirely within the unit circle centered at  $-1$ . For the Lax–Friedrichs method,  $\alpha = h^2/2k$  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 entirely inside the unit circle, and when  $k > h/|c|$ , the  $k$ -scaled ellipse is entirely 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 the region of absolute stability, so the 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

We can better understand the behavior of a numerical solution by examining the truncation error of the numerical scheme. Let's start by looking at the upwind scheme. Using Taylor series expansion, we find that the upwind method is a numerical approximation of

$$u_t + cu_x = \frac{1}{2}ch \left(1 - \frac{ck}{h}\right) u_{xx} + O(h^2). \quad (14.10)$$

From this expression, 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 matching the original problem (14.11), it does a better job matching the parabolic equation (14.12). Notice what happens if we take tiny 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, 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.

So, what's the distinction between dissipation and dispersion? Take a Fourier component

$$u(t, x) = e^{i(\omega t - \xi x)}, \quad (14.14)$$

where the value  $\omega$  is some angular frequency and the value  $\xi$  is the wave number of the Fourier component. Let's use the Fourier component as an *ansatz* to the advection equation  $u_t + cu_x = 0$ . Then

$$i\omega e^{i(\omega t - \xi x)} - ic\xi e^{i(\omega t - \xi x)} = 0,$$

from which we have  $\omega = c\xi$ . We can plug this relation back into the ansatz to get a solution

$$u(t, x) = e^{i(\omega t - \xi x)} = e^{i\xi(ct - x)}.$$

Each Fourier component moves at a velocity  $c = \omega/\xi$ , the ratio of the angular frequency and the wavenumber. We call this ratio the *phase velocity*.

Now, let's use the same ansatz (14.14) in the modified upwind equation (14.12)  $u_t + cu_x = \alpha u_{xx}$ . We get

$$i\omega e^{i(\omega t - \xi x)} - ic\xi e^{i(\omega t - \xi x)} = -\alpha\xi^2 e^{i(\omega t - \xi x)},$$

from which we have  $i\omega - ic\xi = -\alpha\xi^2$ . From this equation, we can determine a *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 components with high wavenumbers  $\xi$ .

The 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

$$u(t, x) = e^{i(c\xi + \beta\xi^3)t} e^{-i\xi x} = \underbrace{e^{i\xi(ct-x)}}_{\text{advection}} \cdot \underbrace{e^{i\beta\xi^3 t}}_{\text{dispersion}}.$$

Numerical dispersion causes oscillations in the solution.

Fourier components don't act alone—every function is a linear combination of many (infinitely many) Fourier components of varying wavenumbers. For a function to move as a group of Fourier components, the phase  $\omega t - \xi x$  must be independent of wavenumber, i.e.,

$$\frac{d(\omega t - \xi x)}{d\xi} = 0 \quad \text{from which} \quad x = \frac{d\omega}{d\xi} \cdot t$$

The *group velocity* is the derivative of the angular frequency with respect to the wave number,  $v_{\text{group}} = d\omega/d\xi$ . See the QR code at the bottom of the page.

A solution is dispersionless when the phase and group velocity are the same  $\omega/\xi = d\omega/d\xi = c$ . Otherwise, the solution is *dispersive*. For example, the phase velocity of the solution to the modified Lax–Wendroff equation is

$$v_{\text{phase}} = \frac{\omega}{\xi} = c + \beta\xi^2,$$

and group velocity is

$$v_{\text{group}} = \frac{d\omega}{d\xi} = c + 3\beta\xi^2.$$

Let's summarize. Every function is a combination of Fourier components of varying wavenumbers. These components travel at a different speed when the solution is dispersive. High wavenumbers are associated with large derivatives, e.g., functions with discontinuities. A dispersive scheme creates oscillations around a discontinuity, and a dissipative scheme smooths out the discontinuity.

Figure 14.1 on the facing page shows the 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 when  $k$  is greater than  $h$ .) When  $k$  is much smaller than  $h$ , the solutions exhibit significant diffusion in the upwind and Lax–Friedrichs methods and dispersion in the Lax–Wendroff method.

## 14.4 Linear hyperbolic systems

Let's turn our attention to the one-dimensional linear wave equation

$$u_{tt} - c^2 u_{xx} = 0. \tag{14.15}$$



group and  
phase velocities

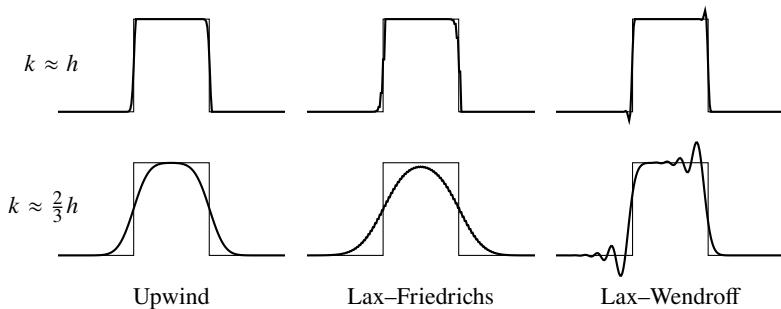


Figure 14.1: Solution to the advection equation for an initial rectangle function. Solutions move from left to right. See the QR code at the bottom of the page.

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 in real space as  $\mathbf{T}^{-1}\mathbf{A}\mathbf{T} = \Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ , where  $\mathbf{T}$  is a matrix of eigenvectors  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ . That is,  $\mathbf{A}\mathbf{v}_i = \lambda_i\mathbf{v}_i$ . The eigenvalues  $\lambda_i$  are the *characteristic speeds*. The characteristic speeds for the wave equation (14.15) are  $\lambda_{\pm} = \pm c$ .

By diagonalizing the matrix  $\mathbf{A}$ , we completely decouple the system into 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  $\mathbf{T}^{-1}\mathbf{u}_t + \Lambda\mathbf{T}^{-1}\mathbf{u}_x = \mathbf{0}$ . By letting  $\mathbf{v} = \mathbf{T}^{-1}\mathbf{u}$ , we have the equivalent system

$$\mathbf{v}_t + \Lambda\mathbf{v}_x = \mathbf{0},$$

in which each variable is now uncoupled as

$$\frac{\partial v_i}{\partial t} + \lambda_i \frac{\partial v_i}{\partial x} = 0 \tag{14.16}$$



for  $i = 1, 2, \dots, n$ . We get the solution  $v_i(t, x) = v_i(0, x - \lambda_i t)$  by using the method of characteristics. The initial conditions are given by  $\mathbf{v}(0) = \mathbf{T}\mathbf{u}(0)$ . Finally, setting  $\mathbf{u}(t) = \mathbf{T}\mathbf{v}(t)$ , we've solved the problem.

Now, let's solve the problem numerically. We can discretize the uncoupled system (14.16) using the upwind scheme

$$\frac{(V_i)_j^{n+1} - (V_i)_j^n}{k} + \frac{\lambda_i + |\lambda_i|}{2} \frac{(V_i)_j^n - (V_i)_{j-1}^n}{h} + \frac{\lambda_i - |\lambda_i|}{2} \frac{(V_i)_{j+1}^n - (V_i)_j^n}{h} = 0.$$

If we let  $\Lambda^+$  denote the diagonal matrix whose elements are  $\frac{1}{2}(\lambda_i + |\lambda_i|)$  and  $\Lambda^-$  denote the diagonal matrix whose elements are  $\frac{1}{2}(\lambda_i - |\lambda_i|)$ , 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

$$\frac{\partial}{\partial t} u + \frac{\partial}{\partial x} f(u) = 0 \quad (14.17)$$

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 (14.17) 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 expression 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 when differentiability falls apart, we'll examine the inviscid Burgers equation, a toy model for fluid and gas dynamics.

### ► Burgers' equation

Burgers' equation  $u_t + uu_x = \varepsilon u_{xx}$  is a simplification of the 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.

Given the right initial conditions, it seems plausible that a fast-moving characteristic might eventually catch up with and overtake a slow-moving one. If a characteristic curve with a higher slope is to the left of one with a lower slope, then the two curves will intersect at some point. See Figure 14.2 on page 415 and

---

<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’).”

the QR code at the bottom of this page. The solution at this point would then be multivalued because it 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 the solution overturns by computing when the derivative first becomes infinite.

$$\frac{du}{dx} = \frac{du}{dx_0} \frac{dx_0}{dx} = u'_0(x_0) \left( \frac{dx}{dx_0} \right)^{-1} = \frac{u'_0(x)}{u'_0(x)t + 1} \quad (14.18)$$

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  $-(u'_0(x_0))^{-1}$ . 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, something is wrong with either our formulation or our reasoning. We relied on the fact that  $u(t, x)$  is smooth to apply the method of characteristics. But as soon as the derivative of  $u(t, x)$  blows up, the method is invalidated. In the next section, we will look at how to solve the ill-posed problem.<sup>3</sup>

## ► Weak solutions

Burgers' equation using the method of characteristics is ill-posed when the solution is multivalued. We need to choose one 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 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  $\varphi$ . There will be no boundary terms because  $\varphi$  vanishes at the boundaries. 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. This chapter uses weak solutions to determine an appropriate analytical solution. The next chapter will apply them 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

---

<sup>3</sup>An alternative 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  $\varepsilon \rightarrow 0$ .



multivalued and  
weak solutions to  
Burgers' equation

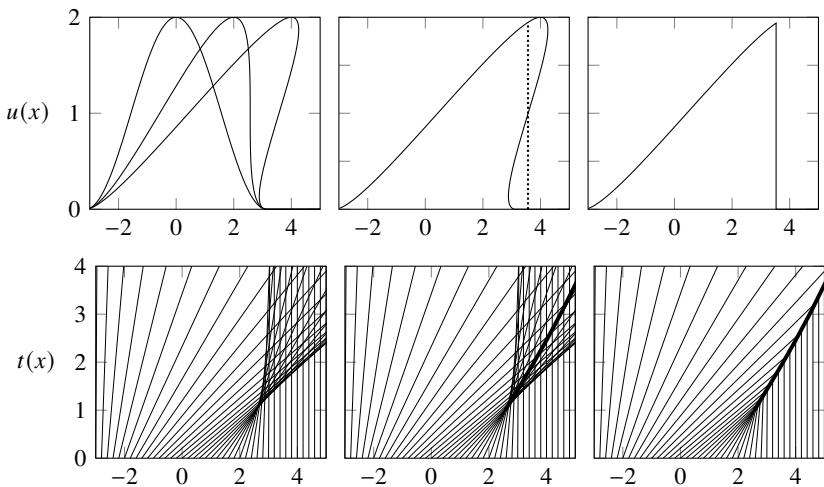


Figure 14.2: Top left: solutions  $u$  to Burgers' equation at  $t = 0$  (initial shock formation),  $t_c = 1$  (initial shock formation), and  $t = 2$  (unphysical multivalued solution). Top right: weak solution at  $t = 2$ . Bottom: characteristic curves  $x(t)$ .

is bounded. Notably,  $\varphi$  is zero at the boundaries, where we need it to be zero. We denote this by  $\varphi = C_0^1(\Omega)$ . 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

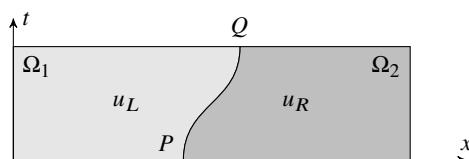
$$\iint_{\Omega} (\varphi u_t + \varphi f(u)_x) \, dx \, dt = 0.$$

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 initial data given by two constant states separated by a discontinuity. For the advection equation, it is

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{with} \quad u(0, x) = \begin{cases} u_L & x < 0 \\ u_R & x > 0 \end{cases}. \quad (14.19)$$

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 symmetry 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  $\zeta = x/t = \hat{x}/\hat{t}$ . Taking  $u(t, x) = u(\zeta)$  in Burgers' equation,

$$0 = u_t + \left(\frac{1}{2}u^2\right)_x = \zeta_t u' + \zeta_x uu' = \left(-\frac{x}{t^2}\right)u' + \left(\frac{1}{t}\right)uu'.$$

Multiplying by  $t$  and replacing  $x/t$  by  $\zeta$ , we have

$$-\zeta u' + uu' = u'(u - \zeta) = 0.$$

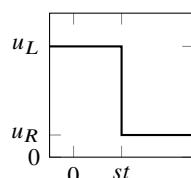
Either  $u' = 0$ , in which case  $u(x)$  is constant; or  $u = \zeta$ , in which case  $u(t, x) = x/t$ . The solution may also be a piecewise combination of these solutions. Let's consider the two cases: when  $u_L > u_R$  and when  $u_L < u_R$ .

1. If  $u_L > u_R$ , then the Rankine–Hugoniot condition says

$$s = \frac{\frac{1}{2}u_R^2 - \frac{1}{2}u_L^2}{u_R - u_L} = \frac{1}{2}(u_R + u_L),$$

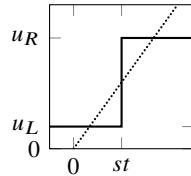
and the exact solution to the Riemann problem is

$$u(t, x) = \begin{cases} u_L, & \text{if } x/t < s \\ u_R, & \text{if } x/t > s \end{cases}$$



2. If  $u_L < u_R$ , then one exact solution is

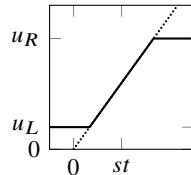
$$u(t, x) = \begin{cases} u_L, & \text{if } x/t < s \\ u_R, & \text{if } x/t > s \end{cases}$$



where  $s = (u_R + u_L)/2$  is given by the Rankine–Hugoniot condition.

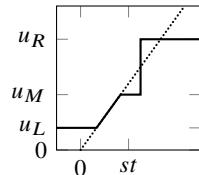
Another solution is

$$u(t, x) = \begin{cases} u_L, & \text{if } x/t < u_L \\ x/t, & \text{if } u_L < x/t < u_R \\ u_R, & \text{if } x/t > u_R \end{cases}$$



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*

$$f'(u_L) > s \equiv \frac{f(u_R) - f(u_L)}{u_R - u_L} > f'(u_R).$$

The Lax entropy condition says that the characteristics with speeds  $f'(u_R)$  and  $f'(u_L)$  must intersect the shock with speed  $s$  from both sides. For Burgers' equation  $f(u) = \frac{1}{2}u^2$ , so  $f'(u) = u$ .

- 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.
- 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. A density  $\rho = 0$  says that the road is completely empty, and a density  $\rho = 1$  says 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 of 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.

In this case, the simplest model for  $u(\rho)$  is  $u(\rho) = u_{\max}(1 - \rho)$ . Notice that the flux  $f(\rho) = u(\rho)\rho = u_{\max}(\rho - \rho^2)$  is zero when the roads are empty (there are no cars to move) and when the roads are 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). 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 Burgers' equation. Consider two simple examples: cars approaching a red stoplight and cars leaving after the stoplight 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

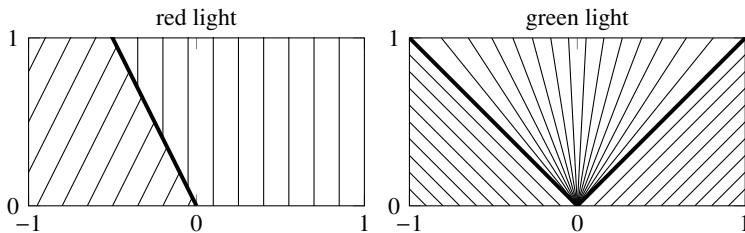
$$\rho(x, 0) = \begin{cases} \rho_L, & x < 0 \\ \rho_R = 1, & x \geq 0. \end{cases}$$

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 backward as the cars queue up.

**Green light.** We imagine several cars in a 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 speed  $u_{\max}$ , and a rarefaction wave moves backward with speed  $f'(\rho) = u_{\max}(1 - 2\rho)$  with  $\rho = 1$ , i.e., speed  $f'(1) = -u_{\max}$ . See Figure 14.3 on the following page.  $\blacktriangleleft$

Figure 14.3: Characteristics for traffic flow with  $\rho_L = \frac{1}{2}$ .

## 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

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial}{\partial x} \mathbf{f}(\mathbf{u}) = 0, \quad (14.20)$$

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 (14.20) is called *hyperbolic*. In this case, there is an invertible map  $\mathbf{T}(\mathbf{u})$  such that  $\mathbf{T}^{-1} \mathbf{f}' \mathbf{T} = \mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$ . The eigenvalues  $\lambda_i$  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

$$h_t + (hu)_x = 0 \quad (14.21a)$$

$$(hu)_t + (hu^2 + \frac{1}{2}gh^2)_x = 0, \quad (14.21b)$$

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 (14.21) becomes

$$h_t + m_x = 0$$

$$m_t + \frac{2m}{h}m_x - \frac{m^2}{h^2}h_x + gh h_x = 0,$$

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  $\mathbf{f}'(\mathbf{u})$  are given by the zeros of

$$\begin{vmatrix} -\lambda & 1 \\ gh - u^2 & 2u - \lambda \end{vmatrix} = \lambda^2 - 2u\lambda + u^2 - gh.$$

So  $\lambda_{\pm} = u \pm \sqrt{gh}$ . 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 is a function of the depth of the water  $h$  along with the velocity of the current  $u$ . For example, a tsunami occurs when a disturbance creates an ocean wave with a wavelength of a hundred or more kilometers 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 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 water’s surface. 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. 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,

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{f}'(\mathbf{u}) \frac{\partial \mathbf{u}}{\partial x} = \mathbf{0} \quad (14.23)$$

where  $\mathbf{f}'$  is the Jacobian matrix of  $\mathbf{f}$  with real eigenvalues  $\{\lambda_1, \lambda_2, \dots, \lambda_n\}$ . Let  $\nabla_{\mathbf{u}} w_i$ —for some  $w(\mathbf{u})$ —be the *left eigenvector* of  $\mathbf{f}'(\mathbf{u})$  corresponding to the eigenvalues  $\lambda_i$ . Then left multiplying (14.23) 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

$$\nabla_{\mathbf{u}} w_i \frac{\partial \mathbf{u}}{\partial t} + \lambda_i \nabla_{\mathbf{u}} w_i \frac{\partial \mathbf{u}}{\partial x} = \mathbf{0}.$$

And by the chain rule, we have

$$\frac{\partial w_i}{\partial t} + \lambda_i \frac{\partial w_i}{\partial x} = \mathbf{0}, \quad (14.24)$$

which is simply the scalar advection equation. So  $w_i$  is a Riemann invariant because it is constant along the characteristic curve given by  $dx/dt = \lambda_i$ . Notice that (14.24) is the diagonalization of the original system.

**Example.** Let's find 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  $\lambda_{\pm} = u \pm \sqrt{gh}$ . 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 (14.20) 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, we may want to prescribe inflow boundary conditions to model incoming water waves in the ocean. Or we may want to model airflow over an airfoil by injecting a flow upwind of the airfoil. We want the gas to escape the domain downwind of the airfoil 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.

A straightforward fix is to use a larger domain and stop computation before the reflected information pollutes the solution—effectively putting the boundaries outside 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.

We'll examine a characteristics-based approach to implementing open boundary conditions over the next several paragraphs. 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  $\mathbf{u}(t, x) = \mathbf{u}(x/t) = \mathbf{u}(\zeta)$ . Then

$$\left(-\frac{x}{t^2}\right)\mathbf{u}' + \frac{1}{t}\mathbf{f}'(\mathbf{u})\mathbf{u}' = \mathbf{0},$$

from which we have

$$(\mathbf{f}'(\mathbf{u}) - \zeta \mathbf{I})\mathbf{u}' = \mathbf{0}.$$

So either  $\mathbf{u}' = \mathbf{0}$  in which case  $\mathbf{u}$  is constant in  $\zeta$  or  $\det(\mathbf{f}'(\mathbf{u}) - \zeta \mathbf{I}) = \mathbf{0}$  in which case  $\zeta$  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  $\det(\mathbf{f}'(\mathbf{u}) - \zeta \mathbf{I}) = \mathbf{0}$  is a *contact discontinuity* that travels with characteristic speed  $\zeta$ . 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 move 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

$$\frac{\partial}{\partial t} U_j + \frac{F_{j+1/2} - F_{j-1/2}}{h} = 0, \quad (14.25)$$

where  $F_{j+1/2}$  is the numerical flux. In this case, by summing (14.25) 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: 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 407 that the second term adds artificial viscosity to an otherwise unstable central difference scheme. However, the diffusion coefficient  $\alpha = h^2/2k$  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 the 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

$$\begin{aligned} & \int_{x_{j-1/2}}^{x_{j+1/2}} u(t_{n+1}, x) \, dx - \int_{x_{j-1/2}}^{x_{j+1/2}} u(t_n, x) \, dx \\ & + \int_{t_n}^{t_{n+1}} f(u(t, x_{j+1/2})) \, dt - \int_{t_n}^{t_{n+1}} f(u(t, x_{j-1/2})) \, dt = 0. \end{aligned} \quad (14.26)$$

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

$$\frac{U_j^{n+1} - U_j^n}{k} + \frac{F_{j+1/2}^{n+1/2} - F_{j-1/2}^{n+1/2}}{h} = 0. \quad (14.27)$$

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  $k \leq h|\lambda_{\max}|$ , where  $\lambda_{\max}$  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.

We need to use piecewise linear approximations to extend the method to second order (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 (14.27), 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,

$$F_j^{n+1/2} = \frac{1}{k} \int_{t_n}^{t_{n+1}} f(u(t, x_j)) dt \approx f(u(t_{n+1/2}, x_j)),$$

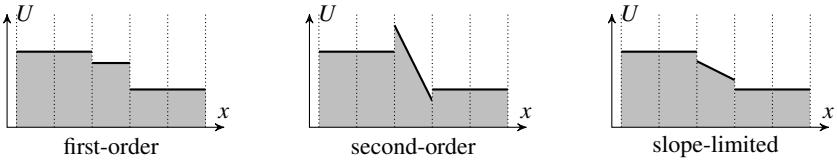


Figure 14.4: A second-order method may introduce oscillations near a discontinuity. A slope limiter is used to minimize oscillations.

with

$$u(t_{n+1/2}, x_j) \approx u(t_n, x_j) + \frac{k}{2} \frac{\partial}{\partial t} u(t_n, x_j) = u(t_n, x_j) - \frac{k}{2} \frac{\partial}{\partial x} f(u(t_n, x_j)),$$

where we use  $u_t + f(u)_x = 0$  for the equality above. This approximation gives us the staggered, two-step predictor-corrector method

$$\begin{aligned} U_j^{n+1/2} &= U_j^n - \frac{k}{2} \frac{\partial}{\partial x} f(U_j^n) \\ U_{j+1/2}^{n+1} &= \frac{1}{2}(U_j^n + U_{j+1}^n) + \frac{h}{8} \left( \frac{\partial}{\partial x} U_j^n - \frac{\partial}{\partial x} U_{j+1}^n \right) - \frac{k}{h} \left( f(U_{j+1}^{n+1/2}) - f(U_j^{n+1/2}) \right). \end{aligned}$$

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 a 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 it 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  $U$ .

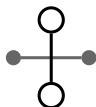
• Julia has several packages for modeling and solving hyperbolic problems. Trixi.jl is a numerical simulation framework for hyperbolic conservation laws. Kinetic.jl is a toolbox for the study of computational fluid dynamics and scientific machine learning. Oceananigans.jl is a fluid flow solver for modeling ocean dynamics. ViscousFlow.jl models viscous incompressible flows around bodies of various shapes.

## 14.7 Exercises

14.1. Show that the error of the Lax–Friedrichs scheme is  $O(k + h^2/k)$ . 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. exercise:32 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 (leapfrog in time and central difference in space):



Determine the order and stability conditions for a consistent scheme. Discuss whether the scheme is dispersive or dissipative. How does dispersion or dissipation affect the scheme's stability? 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 and stability conditions of the scheme. Is the scheme dispersive or dissipative?

14.5. Derive the conservative form of the Lax–Wendroff scheme (page 426).

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 gas

$$E = \frac{1}{2}\rho u^2 + \frac{p}{\gamma - 1}.$$

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. 

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. Confirm the solutions match by plotting them together at several time steps. 

14.8. The dam-break problem. Consider the shallow water equations

$$h_t + (hu)_x = 0 \tag{14.29a}$$

$$(hu)_t + (hu^2 + \frac{1}{2}gh^2)_x = 0, \tag{14.29b}$$

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 scheme, you will need a slope limiter.) Use 100 and 200 points for space discretization and choose time steps according to the CFL condition. Plot the numerical result at  $t = 0.25$  against the “exact” solution obtained using a fine mesh in space and step size in time. 

## Chapter 15

---

# Elliptic Equations



This chapter introduces the finite element method (FEM) and its applications in solving 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 law of conservation of energy says that  $q'(x) = f(x)$ . From these equations, we have the steady-state problem  $-u''(x) = f(x)$ . We solve the problem by formulating it in three different ways—as a *boundary value problem* (B), as a *minimization problem* (M), or as a *variational problem* (V) (also called the weak formulation).

(B) *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 relatively 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  $\langle u', v' \rangle = \langle f, v \rangle$ . We reverse the steps to show that (V) implies (D) given that  $u'$  is differentiable. 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  $\varepsilon > 0$ . We define the *variational derivative* of  $F$  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 out 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 essential idea of the finite element method is to find a solution to the variational problem (V) or the minimization problem (M) 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:

(M) *Rayleigh–Ritz*: Find  $u_h \in V_h$  such that  $F(u_h) \leq F(v_h)$  for all  $v_h \in V_h$ .

(V) *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

$$\varphi_j(x) = \begin{cases} 1 + t_j(x), & t_j(x) \in (-1, 0] \\ 1 - t_j(x), & t_j(x) \in (0, 1] \\ 0, & \text{otherwise} \end{cases} \quad \begin{array}{c} \text{---} \\ | \\ \bullet \\ | \\ \text{---} \end{array} \quad \begin{matrix} x_{j-1} & x_j & x_{j+1} \end{matrix} \quad (15.1)$$

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 expression is the same as

$$\sum_{j=1}^m v_j \left\langle \sum_{i=1}^m \xi_i \varphi'_i(x), \varphi'_j(x) \right\rangle = \sum_{j=1}^m v_j \langle f, \varphi_j(x) \rangle$$

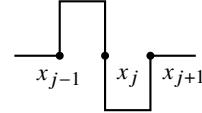
For this equality to hold for all  $v_j$ , we must have

$$\left\langle \sum_{i=1}^m \xi_i \varphi'_i(x), \varphi'_j(x) \right\rangle = \langle f, \varphi_j(x) \rangle$$

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*. By using regular, piecewise-linear basis functions, the system  $\mathbf{A}\xi = \mathbf{b}$  is

$$\frac{1}{h^2} \begin{bmatrix} 2 & -1 & & \cdot & & \\ -1 & 2 & & & & \\ & \ddots & \ddots & & & \\ & & \ddots & \ddots & -1 & \\ & & & & -1 & 2 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_m \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}.$$

## 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  $\Omega \in \mathbb{R}^2$ . We'll need a few more concepts before writing the Poisson equation in variational form. 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, we define  $H_0^1(\Omega)$  to be the set of all Sobolev functions that vanish on the boundary of  $\Omega$ . The Sobolev norm is an extension of the  $L^2$ -norm that includes a measure of the smoothness or regularity of a function.

**Theorem 50** (Green's first identity).

$$\iint_{\Omega} v \Delta u d\mathbf{x} = \int_{\partial\Omega} v \frac{\partial u}{\partial n} ds - \iint_{\Omega} \nabla v \cdot \nabla u d\mathbf{x}$$

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 a two-dimensional domain into triangles and higher-dimensional domains into simplices such as tetrahedrons. 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 the

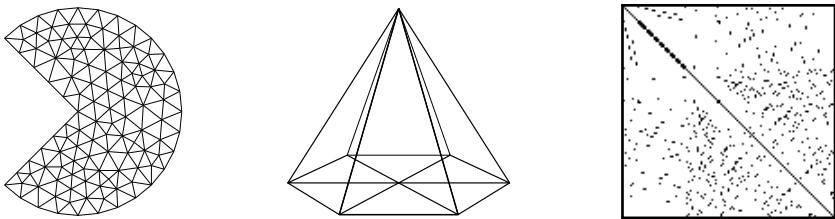


Figure 15.1: Triangulation of a domain  $\Omega$  (left) using a simple pyramidal basis functions (middle) and the resultant sparse stiffness matrix (right).

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  $\mathbf{A}\xi = \mathbf{b}$ , 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 nonzero 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 a more straightforward 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 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 the 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

We can also use the finite element method to solve 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  $\langle u_t, v \rangle + \langle u_x, v_x \rangle = \langle f, v \rangle$  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

$$\frac{\xi^{n+1} - \xi^n}{\Delta t} + \frac{1}{2}\mathbf{A}^{-1}\mathbf{B}\xi^{n+1} + \frac{1}{2}\mathbf{A}^{-1}\mathbf{B}\xi^n = \mathbf{A}^{-1}\mathbf{c}.$$

We can confirm numerical stability by showing 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. Because FEniCS is an external program, you'll need to download and install it in addition to the wrapper.<sup>2</sup> For a finite-element library in Julia without external dependencies, try the relatively new Gridap.jl library.

 Julia's FEM packages include Ferrite.jl, Gridap.jl, and FEniCS.jl.

---

<sup>1</sup>The letters “FE” stand for “finite element,” the letters “CS” stand for “computational software,” and the letters “ni” are simply used to string the others together into a homophone of *phoenix*, the mascot of University of Chicago, where the project originated.

<sup>2</sup><https://fenicsproject.org/>

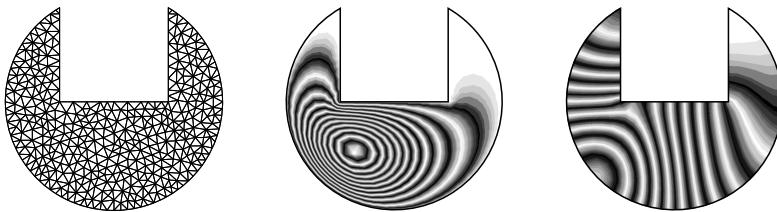


Figure 15.2: FEM mesh (left), solution of the Poisson equation with a heat source centered at  $(-0.5, -0.5)$  (middle), and solution of the heat equation with initial conditions centered at  $(-0.5, -0.5)$  (right). Each contour depicts a 10 percent change in value.

**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) and later visualized using ParaView:<sup>3</sup>

```
File("poisson_solution.pvd",w)
```

The figure on the preceding page shows the solution. 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 condition  $u(0, x, y) = f(x, y) = e^{-(4x+2)^2 - (4y+2)^2}$ . 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 condition, 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 them by setting `bc = []`.

```
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.wt, [])
```

---

<sup>3</sup>ParaView is an open-source application for visualizing scientific data sets such as VTK.

```
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 figure on page 443 shows the solution. 

## 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

$$u'''' = f \quad \text{with} \quad u(0) = u'(0) = u(1) = u'(1) = 0$$

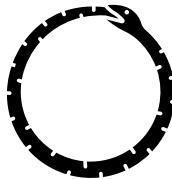
for the deflection  $u(x)$  of a uniform beam under the load  $f(x)$ . Formulate a finite element method and find the corresponding linear system of equations in the case of a uniform partition. Determine the solution for  $f(x) = 384$ . You may wish to use the basis elements  $\phi_j(t) = H_{00}(|t|)$  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 can improve the accuracy of finite difference methods by using wider stencils that provide better approximations to the derivatives. With each additional grid point added to the stencil, we 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—something sometimes called exponential or spectral accuracy. If the solution is at most  $p$ -times differentiable, convergence is at most  $O(h^{p+1})$ . This chapter examines an essential 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 nonperiodic solutions. Some tricks may also allow us 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 regarding the fast Fourier transform and 10.3 regarding function approximation. This section

re-examines the Fourier transform by focusing on trigonometric 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 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  $c_\xi$  so 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 expression, we have the discrete Fourier transform and the inverse discrete 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 definitions with the (continuous) Fourier transform and the inverse (continuous) 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

$$u_p(x) = \sum_{\xi=-\infty}^{\infty} u(x + 2\pi\xi)$$

if the series converges. If  $u_p(x)$  exists, then it is periodic:  $u_p(x) = u_p(x + 2\pi\xi)$ . 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  $\hat{u}(\xi)$ . We are left with

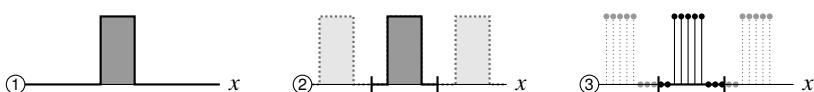
$$\sum_{\xi=-\infty}^{\infty} u(x + 2\pi\xi) = \sum_{\xi=-\infty}^{\infty} \hat{u}(\xi) e^{i\xi x},$$

known as the *Poisson summation formula*. If  $u(x)$  vanishes outside of  $[0, 2\pi]$ ,

$$u(x) = \sum_{\xi=-\infty}^{\infty} \hat{u}(\xi) e^{i\xi x}.$$

By truncating the Fourier series (16.2) from  $-m$  to  $m - 1$  (i.e., restricting the frequency domain), we have the discrete Fourier transform (16.1).

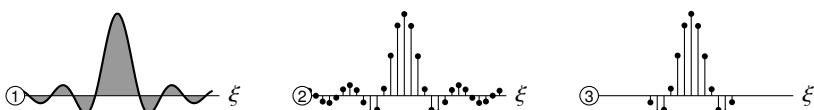
Let's summarize the Fourier transform, the Fourier series, and the discrete Fourier transform over the spatial domain  $x$  and the frequency domain  $\xi$ .



① *Fourier transform*: the spatial domain is the real line  $(-\infty, \infty)$

② *Fourier series*: the spatial domain is periodic

③ *discrete Fourier transform*: the spatial domain is periodic and discrete



① *Fourier transform*: the frequency domain is the real line  $(-\infty, \infty)$

② *Fourier series*: the frequency domain is discrete over the real line  $(-\infty, \infty)$

③ *discrete Fourier transform*: the frequency domain is 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,

$$\frac{d}{dx}u(x) = \sum_{\xi=-m}^{m-1} c_\xi \frac{d}{dx} e^{i\xi x} = \sum_{\xi=-m}^{m-1} (i\xi c_\xi) e^{i\xi x}.$$

The discrete Fourier transform of the derivative has the memorable form

$$F\left[\frac{d}{dx}u\right] = i\xi F[u].$$

From this equation, we can see the origin of the term “spectral method.” When we expand a solution  $u(t, x)$  as 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.

When the function  $u(x)$  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  $i(2\pi/\ell)\xi\hat{u}$ . 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

$$\hat{u}(t, \xi) = e^{-\xi^2 t} \hat{u}(0, \xi).$$

We can write the formal solution to the heat equation from this expression as

$$u(t, x) = F^{-1}[e^{-\xi^2 t} F u(0, x)]. \quad (16.4)$$

We'll use a fast Fourier transform (FFT) to compute the discrete transforms in practice. Let's examine the numerical implementation of the discrete solution in Julia. We need to exercise some care 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  $m$  nodes, we'll take

$$\xi = 0, 1, 2, \dots, \frac{1}{2}(m-1), -\frac{1}{2}m, -\frac{1}{2}m+1, \dots, -2, -1.$$

where we round half toward zero. 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 trials to select from among several FFT algorithms. When first called, the library may try several different algorithms for the same problem and, after that, 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 = (fftwshift(-(m-1)÷2:m÷2)*(2π/ℓ)).^2
u = (t,u₀) -> real(ifft(exp.(-ξ²*t).*fft(u₀)))
```

If a problem repeatedly uses Fourier transforms 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 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 over all 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 back into 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  $\hat{B}_0(\xi) = F[B_0(x)]$ :

$$\int_{-\infty}^{\infty} B_0(x) e^{-i\xi x} dx = \int_{-\frac{1}{2}h}^{\frac{1}{2}h} e^{-i\xi x} dx = \frac{e^{i\xi/2h} - e^{-i\xi/2h}}{i\xi} = \frac{\sin \xi/2h}{\xi/2} = \frac{1}{h} \operatorname{sinc} \frac{\xi}{2h}.$$

The rectangular function and the sinc function<sup>1</sup> are plotted below.



The rectangular function is not differentiable at  $x = \pm \frac{1}{2}h$ ; 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 the rectangular function  $B_0(x+a)$  is

$$\int_{-\frac{1}{2}h}^{\frac{1}{2}h} e^{-i\xi(x+a)} dx = e^{ia\xi} \frac{\sin \xi/2h}{\xi/2} = \frac{1}{h} e^{ia\xi} \operatorname{sinc} \frac{\xi}{2h}.$$

So the Fourier transform a piecewise constant function is a linear combination of sinc functions.

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 240. 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

$$\hat{B}_1(\xi) = (i\xi)^{-1} \left( e^{i\xi/2h} - e^{-i\xi/2h} \right) \frac{1}{h} \operatorname{sinc} \frac{\xi}{2h} = \frac{1}{h^2} \operatorname{sinc}^2 \frac{\xi}{2}.$$

---

<sup>1</sup>British mathematician Phillip Woodward introduced the sinc function (pronounced “sink”) in his 1952 paper “Information theory and inverse probability in telecommunication,” saying that the function  $(\sin \pi x)/\pi x$  “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 and the sinc-squared function are plotted below.



The triangular function is not differentiable at  $x \in \{-1, 0, 1\}$ , 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. It follows that the Fourier transform of piecewise linear function is a linear combination of sinc-squared functions.

Continuing, we can construct a degree- $p$  piecewise polynomial using B-splines of order  $p$ . The Fourier transform of a B-spline with  $h$ -separated knots is given by

$$\hat{B}_p(x) = \frac{1}{h^{p+1}} \operatorname{sinc}^{p+1} \frac{\xi}{2h}.$$

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 253 to compute the  $\ell^2$ -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

$$|c_{\xi}| = O(h^{-(p+1)} \operatorname{sinc}^{p+1}(\xi/2h)) = O(|\xi|^{-(p+1)})$$

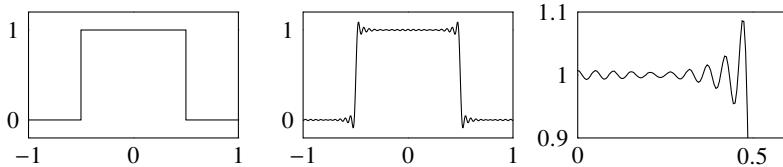
and

$$\|\tau_m\|^2 = \sum_{\xi > |m|} |c_{\xi}|^2 = \sum_{\xi > |m|} O(|\xi|^{-2p-2}) = O(m^{-2p-1}).$$

Therefore, the truncation error  $\|\tau_m\| = O(m^{-p-1/2})$ .

For a rectangular function or any function with a discontinuity,  $p=0$  and  $|\tau_n| = O(n^{-1/2})$ . We should expect slow convergence, as we see in the next

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 Fourier polynomial approximation results in a *Gibbs phenomenon*, oscillations in the neighborhood of a discontinuity.



## 16.2 Nonlinear stiff equations

As seen in previous chapters, some partial differential equations, such as the Navier–Stokes equation, combine a stiff linear term with a nonstiff nonlinear term. Techniques for solving such equations include time splitting, integrating factors, and implicit-explicit (IMEX) methods. Time splitting allows us to avoid the stability issues caused by the stiff linear term. Still, we are typically unable to get beyond second-order-in-time accuracy using them. Integrating factors can bring us high accuracy, but we may not entirely avoid the stability issues caused by the stiffness. IMEX methods can generate moderately accurate solutions with some reduced stability.

### ► Time splitting

Time splitting methods are familiar from chapters 12 and 13. Consider the equation  $u_t = L u + N u$ , 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 365 that after one time step, the solution  $u(\Delta t, x) = w(\Delta t, x) + O(\Delta t^2)$ . The method has a second-order splitting error with each time step, resulting in 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 subsequently becomes 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.** An 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 the 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  $-i\varepsilon^{-1} V(x) u$ . 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, if the problem is stiff, we must use an implicit L-stable method to avoid stability

issues altogether. High-order implicit schemes can be challenging 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

$$\hat{u}_t - \hat{L}\hat{u} = F[N u].$$

Multiply both sides of the equation by  $e^{-t\hat{L}}$  and simplify to get

$$(e^{-t\hat{L}}\hat{u})_t = e^{-t\hat{L}}F[N u]. \quad (16.5)$$

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  $\hat{w}(0, \xi) = e^{0\cdot\hat{L}} F[u(0, x)] = F[u(0, x)]$ .

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

$$\begin{aligned} \text{set} \quad & \hat{w}(t_n, \xi) = F u(t_n, x) \\ \text{solve} \quad & \frac{\partial}{\partial t} \hat{w}(t, \xi) = e^{-\Delta t \hat{L}} F \left[ N \left( F^{-1} \left[ e^{\Delta t \hat{L}}\hat{w}(t, \xi) \right] \right) \right] \\ \text{set} \quad & u(t_{n+1}, x) = F^{-1} \left( e^{\Delta t \hat{L}}\hat{w}(t_{n+1}, \xi) \right). \end{aligned}$$

It isn't necessary to switch back to the variable  $u(t, x)$  with each iteration. Instead, we can solve the problem in the variable  $\hat{w}(t, \xi)$  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 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 these methods 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. 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 injects instability into the system at large scales. The fourth-order term brings stability back into the system at small scales. The nonlinear convection term transfers stability between scales resulting in the formation of fronts. Eventually, the KSE exhibits chaotic behavior.

The KSE is quite stiff and 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
```

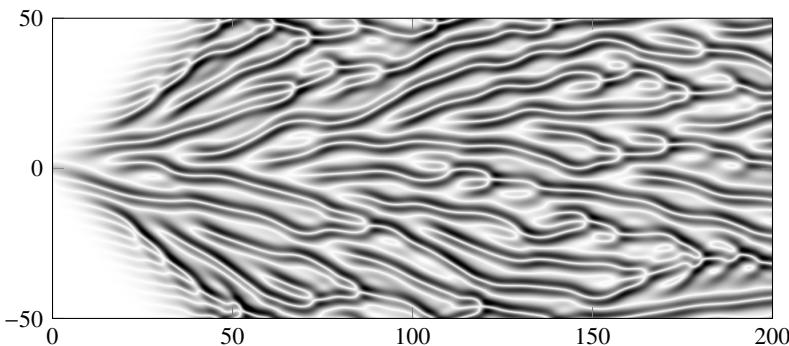


Figure 16.1: Solution to the Kuramoto–Sivashinsky equation for a Gaussian bump. Space is along the vertical axis and time is along the horizontal axis.

```
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.

```
using ImageShow,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 code at the bottom of this page. ▶

### 16.3 Incompressible Navier–Stokes equation

We can modify a one-dimensional Fourier spectral method to handle two- and three-dimensional problems by replacing the vector  $\xi$  with the tuple  $(\xi_1, \xi_2)$  and  $(\xi_1, \xi_2, \xi_3)$ . To see this in practice, we'll solve the two-dimensional Navier–Stokes equation following an approach from Deuflhard and Hohmann [2003].



KSE animation

Consider the incompressible Navier–Stokes equation

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \varepsilon \Delta \mathbf{u} \quad (16.8)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (16.9)$$

which models viscous fluid flow. The vector  $\mathbf{u} = (u(t, x, y), v(t, x, y))$  is the velocity,  $p(t, x, y)$  is the pressure, and  $\varepsilon$  is the inverse of the Reynolds number. Explicitly, the two-dimensional Navier–Stokes equation says

$$\begin{aligned} u_t + uu_x + vu_y &= -p_x + \varepsilon(u_{xx} + u_{yy}) \\ v_t + uv_x + vv_y &= -p_y + \varepsilon(v_{xx} + v_{yy}) \\ u_x + u_y &= 0. \end{aligned}$$

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

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = \underbrace{-\frac{1}{2}(\nabla p^{n+1} + \nabla p^n)}_{\text{Crank–Nicolson}} - \underbrace{\left(\frac{3}{2}\mathbf{H}^n - \frac{1}{2}\mathbf{H}^{n-1}\right)}_{\text{Adams–Bashforth}} + \underbrace{\frac{1}{2}\varepsilon(\Delta \mathbf{u}^{n+1} + \Delta \mathbf{u}^n)}_{\text{Crank–Nicolson}},$$

where convective acceleration  $\mathbf{H}^n$  is  $\mathbf{u}^n \cdot \nabla \mathbf{u}^n$ .

There are a couple of problems with this setup. 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

$$\mathbf{u}^* = \mathbf{M}_-^{-1} \left( -\frac{1}{2}\nabla p^n - \frac{3}{2}\mathbf{H}^n + \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+ \mathbf{u}^n \right). \quad (16.11)$$

We need to determine  $\nabla p^{n+1}$  in such a way as to 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 time step. Substituting  $-\frac{1}{2}\nabla p^n$  back into (16.11) yields

$$\begin{aligned} \mathbf{u}^* &= \mathbf{M}_-^{-1}(\mathbf{M}_-(\mathbf{u}^n - \mathbf{u}^{*-1}) - \frac{3}{2}\mathbf{H}^n + \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+\mathbf{u}^n) \\ &= \mathbf{u}^n - \mathbf{u}^{*-1} + \mathbf{M}_-^{-1}(-\frac{3}{2}\mathbf{H}^n + \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+\mathbf{u}^n). \end{aligned}$$

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

$$\begin{aligned} \mathbf{u}^* &= \mathbf{u}^n - \mathbf{u}^{*-1} + \mathbf{M}_-^{-1}(-\frac{3}{2}\mathbf{H}^n + \frac{1}{2}\mathbf{H}^{n-1} + \mathbf{M}_+\mathbf{u}^n) \\ \mathbf{u}^{n+1} &= \mathbf{u}^* - \nabla\Delta^{-1}\nabla\cdot\mathbf{u}^*. \end{aligned}$$

To initialize the multistep method, we'll take

$$\mathbf{u}^{-1} = \mathbf{u}^{*-1} = \mathbf{u}^0.$$

In the Fourier domain, we replace

$$\nabla \rightarrow (i\xi_1, i\xi_2) \quad \text{and} \quad \Delta = \nabla \cdot \nabla \rightarrow -(\xi_1^2 \oplus \xi_2^2) = -|\xi|^2,$$

where the sum  $\xi_1^2 \oplus \xi_2^2$  is broadcasted as  $\xi_1^2 \mathbf{1}^\top + \mathbf{1}(\xi_2^2)^\top$ . From (16.13) we get (using the standard hat notation  $\hat{\square}$  for the Fourier transform  $\mathbf{F} \square$ ),

$$\hat{\mathbf{u}}^* = \hat{\mathbf{u}}^n - \hat{\mathbf{u}}^{*-1} + \hat{\mathbf{M}}_+^{-1} \left( -\frac{3}{2}\hat{\mathbf{H}}^n + \frac{1}{2}\hat{\mathbf{H}}^{n-1} + \hat{\mathbf{M}}_- \hat{\mathbf{u}}^n \right) \quad (16.14a)$$

$$\hat{\mathbf{u}}^{n+1} = \hat{\mathbf{u}}^* - \frac{(\xi \cdot \hat{\mathbf{u}}^*)}{|\xi|^2} \xi \quad (16.14b)$$

---

<sup>2</sup>Do you recognize the projection operators  $\mathbf{P}_A = A(A^\top A)^{-1}A^\top$  and  $\mathbf{I} - \mathbf{P}_A$ ?

where

$$\hat{M}_- = \frac{1}{At} - \frac{1}{2}\varepsilon|\xi|^2 \quad \text{and} \quad \hat{M}_+ = \frac{1}{At} + \frac{1}{2}\varepsilon|\xi|^2.$$

We must be cautious when dividing by  $|\xi|^2$  in (16.14b) because one of the components is zero. In this case, we modify the term by replacing the zeros with ones. This fix is itself fixed when we subsequently multiply by  $\xi$ .

We haven't discussed the implementation of the nonlinear term  $\hat{\mathbf{H}}$  yet. If  $\mathbf{u}$  is given by  $(u, v)$ , the  $x$ -component of  $\mathbf{H}$  is

$$H(u, v) = uu_x + vu_y,$$

with a similar form for a  $y$ -component. The Fourier transform of a product of two functions  $uv$  is the convolution  $\hat{u} * \hat{v}$ —an inefficient operator to implement directly. Therefore, we'll evaluate  $\hat{\mathbf{H}}$  in the spatial domain rather than the Fourier domain. Hence, we take

$$\hat{H}(\hat{u}, \hat{v}) = F(F^{-1}\hat{u}F^{-1}(i\xi_1\hat{u}) + F^{-1}\hat{v}F^{-1}(i\xi_2\hat{u})).$$

To visualize the solution, we use a tracer. Imagine dropping ink (or smoke) into a moving fluid and following the ink's motion. Numerically, we can simulate a tracer by solving the advection equation

$$\frac{\partial Q}{\partial t} + \mathbf{u} \cdot \nabla Q = 0 \tag{16.15}$$

using the Lax–Wendroff method, where  $\mathbf{u}$  is the solution to the Navier–Stokes equation at each time step.

Consider a stratified fluid moving in the  $x$ -direction separated from a stationary fluid by a narrow interface. Suppose we further modulate the speed of the fluid in the flow direction. Think of wind blowing over water. The pressure decreases in regions where the moving fluid moves faster, pulling on the stationary fluid and causing it to bulge. This situation leads to Kelvin–Helmholtz instability, producing surface waves in water and turbulence in otherwise laminar fluids. Kelvin–Helmholtz instability can be 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 in a  $2 \times 2$  square, we will take zero vertical velocity  $v$  and

$$u = \frac{1}{4}(2 + \sin 2\pi x)Q(x, y) \text{ where } Q(x, y) = \frac{1}{2} + \frac{1}{2}\tanh(10 - 20|y - 1|).$$

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.

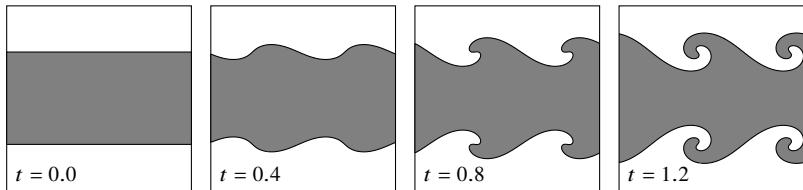


Figure 16.2: The emergence of Kelvin–Helmholtz instability in the Navier–Stokes equation. Also, see the QR code at the bottom of this page.

```

 $\Delta^\circ(Q, \text{step}=1) = Q - \text{circshift}(Q, (\text{step}, 0))$ 
flux(Q, c) = c.* $\Delta^\circ(Q)$  - 0.5c.*(1 - c).*( $\Delta^\circ(Q, 1)$ + $\Delta^\circ(Q, -1)$ )
H = (u, v, i $\xi_1$ , i $\xi_2$ ) -> F*((F\u).* (F\ (i $\xi_1$ .*u)) + (F\v).* (F\ (i $\xi_2$ .*u)))

```

Next, we initialize the variables.

```

using FFTW
 $\ell = 2$ ; n = 128;  $\epsilon = 0.001$ ;  $\Delta t = 0.001$ ;  $\Delta x = \ell/n$ 
x = LinRange( $\Delta x, \ell, n$ )'; y = x'
Q = (@. 0.5(1 + tanh(10 - 20abs(1 - 2y/ $\ell$ ))).*ones(1, n))
un = Q .* (1 .+ 0.5sin.( $\ell*\pi*x$ ))
vn = zeros(n, n)
F = plan_fft(un)
un, vn = F*un, F*vn
uo, vo = un, vn
i $\xi_1$  = im*fftfreq(n, n)'*(2 $\pi/\ell$ )
i $\xi_2$  = transpose(i $\xi_1$ )
 $\xi^2$  = i $\xi_1$ .^2 .+ i $\xi_2$ .^2
H1n, H2n = H(un, vn, i $\xi_1$ , i $\xi_2$ ), H(vn, un, i $\xi_2$ , i $\xi_1$ )
M+ = (1/ $\Delta t$  .+ ( $\epsilon/2$ )* $\xi^2$ )
M- = (1/ $\Delta t$  .- ( $\epsilon/2$ )* $\xi^2$ );

```

Finally, we loop in time.

```

for i = 1:1200
    Q -= flux(Q, ( $\Delta t/\Delta x$ ).*real(F\vn)) + flux(Q', ( $\Delta t/\Delta x$ ).*real(F\un))'
    H1n-1, H2n-1 = H1n, H2n
    H1n, H2n = H(un, vn, i $\xi_1$ , i $\xi_2$ ), H(vn, un, i $\xi_2$ , i $\xi_1$ )
    uo = un - uo + (-1.5H1n + 0.5H1n-1 + M+.*un)./M-
    vo = vn - vo + (-1.5H2n + 0.5H2n-1 + M+.*vn)./M-
     $\phi$  = (i $\xi_1$ .*uo + i $\xi_2$ .*vo)./( $\xi^2$  .+ ( $\xi^2$  .~ 0))
    un, vn = uo - i $\xi_1$ .* $\phi$ , vo - i $\xi_2$ .* $\phi$ 
end
contour(x', y, Q, fill=true, levels=1, legend=:none)

```



Kelvin–Helmholtz  
instability

## 16.4 Exercises

16.1. Solve the Schrödinger equation discussed on page 455 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 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. This exercise aims to explore the behavior of 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 classes 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. Sudden phase separation can set in if a homogeneous molten alloy of two metals is rapidly cooled.<sup>3</sup> 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  $t = 150$ .

We can enforce the mean-zero restriction 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.




---

<sup>3</sup>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.

# 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

$$\begin{aligned}\mathbf{Ax} &= a\mathbf{Av}_1 + b\mathbf{Av}_2 = a\lambda_1\mathbf{v}_1 + b\lambda_2\mathbf{v}_2, \\ \mathbf{A}^2\mathbf{x} &= a\mathbf{A}^2\mathbf{v}_1 + b\mathbf{A}^2\mathbf{v}_2 = a\lambda_1^2\mathbf{v}_1 + b\lambda_2^2\mathbf{v}_2,\end{aligned}$$

and so on...

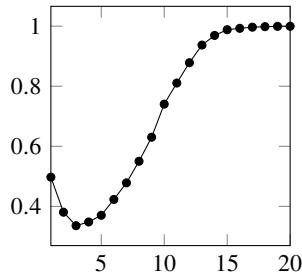
So the Krylov subspace is spanned by linear combinations of the eigenvectors  $\mathbf{v}_1$  and  $\mathbf{v}_2$ , and the dimension is two.

- (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 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. We'll use a subscript to indicate a matrix's size.

- (a) Instead of attacking  $\mathbf{D}_n$  head on, let's use  $\mathbf{M}_n = \mathbf{D}_n + 2\mathbf{I}$ , a matrix with ones along the upper and lower diagonals. If  $(\lambda, \mathbf{x})$  is an eigenpair of  $\mathbf{M}_n$ , then

$$\mathbf{D}_n \mathbf{x} = (\mathbf{M}_n - 2\mathbf{I})\mathbf{x} = (\lambda - 2)\mathbf{x}.$$

So  $(\lambda - 2, \mathbf{x})$  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 Laplace (or cofactor) expansion. We have

$$\begin{aligned} p_n(\lambda) &= \det(\mathbf{M}_n - \lambda\mathbf{I}) \\ &= -\lambda \det(\mathbf{M}_{n-1} - \lambda\mathbf{I}) - \det(\mathbf{M}_{n-2} - \lambda\mathbf{I}) \\ &= -\lambda p_{n-1}(\lambda) - p_{n-2}(\lambda). \end{aligned}$$

We can confirm the values of  $p_2(\lambda)$  and  $p_1(\lambda)$  by expanding  $p_3(\lambda)$ .

$$p_3(\lambda) = \begin{vmatrix} -\lambda & 1 & 0 \\ 1 & -\lambda & 1 \\ 0 & 1 & -\lambda \end{vmatrix} = -\lambda \begin{vmatrix} -\lambda & 1 \\ 1 & -\lambda \end{vmatrix} - 1 \begin{vmatrix} 1 & 1 \\ 0 & -\lambda \end{vmatrix},$$

from which we have  $p_2(\lambda) = \lambda^2 - 1$  and  $p_1(\lambda) = -\lambda$ . With mathematical foresight (or *deus ex machina*?), we may recognize the expression  $p_n(\lambda)$  as the three-term recurrence relation for the Chebyshev polynomial of the second kind  $U_n(-2\lambda)$ . The roots of  $U_n(x)$  are

$$x_i = \cos \frac{i\pi}{n+1} \quad \text{for } i = 1, 2, \dots, n.$$

Therefore, the eigenvalues of  $\mathbf{D}_n$  are given by

$$\lambda_i - 2 = -2 - 2 \cos \frac{i\pi}{n+1} \quad \text{for } i = 1, 2, \dots, n.$$

- (b) The spectral radius of  $\mathbf{D}_n + 2\mathbf{I}$  is

$$\rho(\mathbf{D}_n + 2\mathbf{I}) = \max_{j=1,\dots,n} \left| -2 \cos \frac{i\pi}{n+1} \right| = 2 \cos \frac{\pi}{n+1}.$$

- (c) The 2-norm condition number for a symmetric matrix is the ratio of the largest eigenvalue to the smallest eigenvalue.

$$\kappa_2(\mathbf{D}_n) = \frac{1 + \cos \frac{\pi}{n+1}}{1 - \cos \frac{\pi}{n+1}} \approx \frac{2 - \frac{1}{2} \left( \frac{\pi}{n+1} \right)^2}{\frac{1}{2} \left( \frac{\pi}{n+1} \right)^2} \approx \frac{4}{\pi^2} (n+1)^2$$

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))
```

We can compute the eigenvalues and eigenvectors using `eigen(D)`. Alternatively, we can use the pipe operator `D |> eigen`. We confirm the solution of parts (b) and (c) using

```
( 2cos(pi/(n+1)) , maximum(eigen(D+2I).values) ) |> display
( 4(n+1)^2/pi^2, cond(D) ) |> display
```

### 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),
- $$\|\mathbf{QA}\|_F^2 = \text{tr}((\mathbf{QA})^\top \mathbf{QA}) = \text{tr}(\mathbf{A}^\top \mathbf{Q}^\top \mathbf{QA}) = \text{tr}(\mathbf{A}^\top \mathbf{A}) = \|\mathbf{A}\|_F^2.$$
- (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 equals the product of the determinants. We can use this identity and LU factorization to efficiently determine the compute of a matrix. The LU factorization of a matrix  $\mathbf{A}$  using partial pivoting 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 equals  $+1$ , if the permutation matrix was built from identity matrix using an even number of row exchanges (even parity). It equals  $-1$ , if it was built using an odd number (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  $\mathbf{P}$ . We still 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 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 the 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  $\mathbf{A}$  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(!iszzero, 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 49. The figure on the next page shows the sparsity plots and graphs before and after Cuthill–McKee resorting.

```
p = rcuthillmckee(A)
gplot(Graph(A[p,p]),layout=circular_layout)
```

2.7. Loading the CSV file as a dataframe will allow us to inspect and interpret the data.

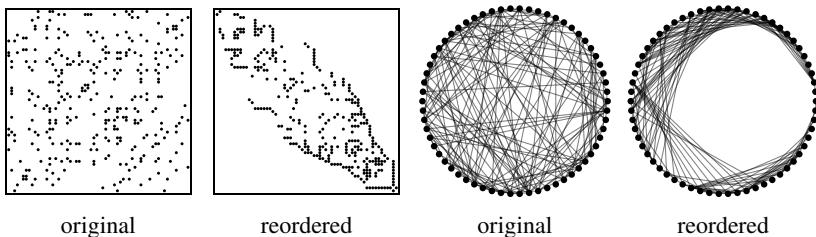


Figure A.1: Sparsity plots and graph drawings of the dolphins of Doubtful Sound.

```
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 are 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. We’ll explicitly call it using `JuMP.value` to avoid confusion or conflicts.

```
using JuMP, GLPK
model = Model(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 run out of new nodes to visit, indicating 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 49 to construct 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  $\mathbf{B}$  indicates which actors (columns) appeared in which movies (rows). The matrix  $\mathbf{B}\mathbf{B}^T$  gives us the movie-movie adjacency matrix, and the matrix  $\mathbf{B}^T\mathbf{B}$  gives us the actor-actor adjacency matrix. We can write the adjacency matrix of a bipartite graph as a block matrix

$$\mathbf{A} = \begin{bmatrix} \mathbf{0} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{bmatrix}.$$

Using **A** will tell us both the actors and their connecting movies.

```
(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.

**2.9.** An  $m \times n$  matrix with density  $d$  has  $d m n$  nonzero elements. Storing a matrix using CSC format requires at most  $m + 1$  memory addresses for the column indexes,  $d m n$  addresses to identify the rows of each nonzero element, and  $d m n$  addresses for each of the values. A 64-bit processor storing double-precision floating-point numbers will need  $2 d m n + m + 1$  eight-byte blocks to store a CSC matrix and  $m n$  blocks for full matrix in memory. The CSC format saves space as long as the density is less than around 0.5. We can check this result by explicitly computing the bytes of memory used in a sparse matrix:

```
d = 0.5; A = sprand(800,600,d)
Base.summarysize(A)/Base.summarysize(Matrix(A))
```

A complex floating-point number is 16-bytes long. We'd need  $3 d m n + m + 1$  blocks for a CSC matrix versus  $2 m n$  blocks for a full matrix. The break-even point for the density is around 0.66. We can check this result by with  $A .+= 0im$ .

**3.4.** Take **X** to grayscale image (with pixel intensity between 0 and 1), **A** and **B** to be Gaussian blurring matrices with standard deviations of 20 pixels, and **N** to be a matrix of random values from the uniform distribution over the interval  $(0, 0.01)$ .

```
X = load(download(bucket*"laura.png")) .|> Gray
m,n = size(X)
blur = x -> exp(-(x/20).^2/2)
A = [blur(i-j) for i=1:m, j=1:m]; A ./= sum(A,dims=2)
```

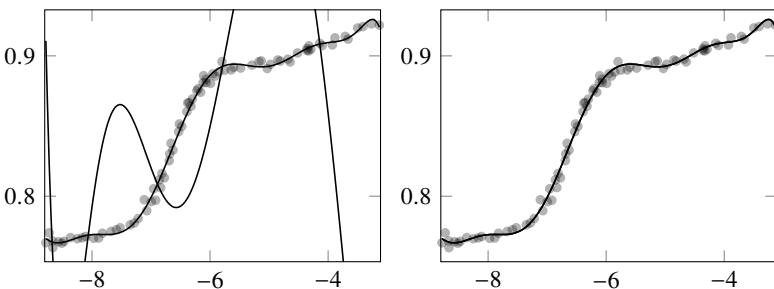


Figure A.2: Solutions to Exercise 3.5. The left plot shows solutions using the Vandermonde matrix built using the original  $x$ -data and the right plot shows solutions using the Vandermonde matrix built using the mean-centered  $x$ -data.

```
B = [blur(i-j) for i=1:n, j=1:n]; B ./= sum(B,dims=1)
N = 0.01*rand(m,n)
Y = A*X*B + N;
```

We'll compare three deblurring methods: inverse, Tikhonov regulation, and the pseudoinverse. We can find a good value for regularization parameter  $\alpha = 0.05$  with some trial-and-error.

```
alpha = 0.05
X1 = A\Y/B
X2 = (A'*A+alpha^2*I)\A'*Y*B'/(B*B'+alpha^2*I)
X3 = pinv(A,alpha)*Y*pinv(B,alpha)
Gray.([X Y X1 X2 X3])
```

3.5. 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 equation 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 preceding page.

Mean-centering the  $x$ -data reduces the 2-condition number of the new Vandermonde matrix to  $1.3 \times 10^5$ . Further scaling the  $x$ -data between  $-1$  and  $1$  reduces the 2-condition number 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.

3.7. 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 period is one year. Second, we can write  $\sin(x + y)$  as  $\sin x \cos y + \sin y \cos x$ . In 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:

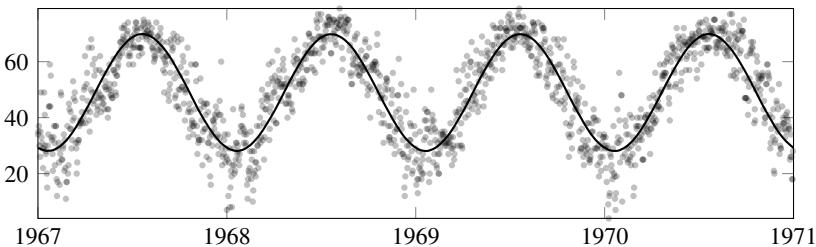


Figure A.3: Daily temperatures of Washington, DC between 1967 and 1971

```
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, width=3)
```

The solution is plotted in the figure above.

3.7. 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 combination 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 method's accuracy. 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 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=1:10]
r = hcat(r...)
prediction = [argmin(r[i,:]) for i=1:10000] .- 1
[sum(prediction[label_test== i]== j) for j=0:9, i=0:9]
```

3.9. 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  $\cos \theta_i = \mathbf{q}_j^T \mathbf{q}_i / \| \mathbf{q}_j \| \| \mathbf{q}_i \|$ . Let's start by importing the data and building the adjacency matrices using the functions `get_names` and `get_adjacency_matrix` defined on page 473

```
actors = get_names("actors")
genres = get_names("genres")
```

```
A = get_adjacency_matrix("movie-genre")
A = (A.*Diagonal(1 ./ sum(A,dims=1)[:,1]))
B = get_adjacency_matrix("actor-movie");
```

We'll use the Arpack.jl package to compute the SVD of 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:

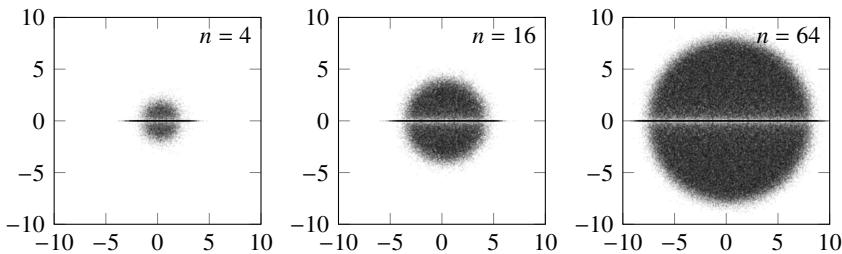
```
index = x -> findfirst(actors.==x)[1]
q = Q[:,index("Steve Martin")][:]
z = Q'*q
r = sortperm(-z)
actors[r[1:10]]
```

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, but 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 a serious film for comic relief. Actors change over their careers and the data set is limited to the years 1972–2012. Classification of movies into genres is simplistic and sometimes arbitrary. Perhaps using more keywords would add more fidelity. The information was crowdsourced through Wikipedia editors and contains errors. How does the dimension of the latent space change the model? By choosing a large-dimensional latent space, we keep much of 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, choosing a small-dimensional space effectively removes the underrepresented genres, and we are left with the main genres. Such a model is underfit.

*3.10.* 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

Figure A.5: Distribution of eigenvalues of normal random  $n \times n$  real matrices.

$(x_i, y_i, t_i) \leftarrow (x_i, y_i, t_i) - (x_1, y_1, t_1)$ . 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 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]
xols = 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:

```

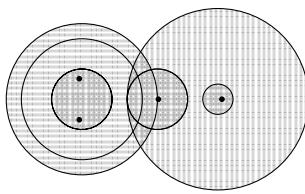
E = collect(Iterators.flatten([eigvals(randn(n,n)) for i∈1:2000]))
scatter(E, mc=:black, ma=0.05, legend=nothing, aspect_ratio=:equal)

```

The figure above shows the distribution for  $n = 4, 16$ , and  $64$ . The distribution 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  $\blacksquare\blacksquare$ , and the eigenvalues  $\bullet$  lie in their intersections.

4.4. The following Julia code finds an eigenpair using Rayleigh iteration starting with a random initial vector. We can compute all four eigenpairs by running the code repeatedly for different initial guesses. The method converges rapidly, and  $\mathbf{A} - \rho \mathbf{I}$  becomes poorly 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 473.

```

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. Samuel L. Jackson, M. Emmet Walsh, Gene Hackman, Bruce Willis, Christopher Walken, Robert De Niro, Steve Buscemi, Dan Hedaya, Whoopi Goldberg, and Frank Welker are the ten most central actors. 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, and convert it to an array:

```
using 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

$$\varepsilon_k = \frac{\sqrt{\sum_{i=k+1}^n \sigma_i^2}}{\sqrt{\sum_{i=1}^n \sigma_i^2}} = 1 - \frac{\sqrt{\sum_{i=1}^k \sigma_i^2}}{\|\mathbf{A}\|_F},$$

which can be implemented as

```
epsilon = 1 .- sqrt.(cumsum(S.^2))/norm(A)
plot(epsilon,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  $\text{Gray} \cdot (U \cdot \text{Diagonal}(S) \cdot V')$ .



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 Jacobi method's spectral radius was derived in exercise 1.5(b). We will confirm the Gauss–Seidel method's spectral radius using Julia.

```
n = 20; D = SymTridiagonal(-2ones(n),ones(n-1))
P = -diagm(0 => diag(D,0), -1 => diag(D,-1))
N = diagm(1 => diag(D,1))
eigvals(N,P)[end], cos(pi/(n+1))^2
```

5.2. 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  $|\sigma| < 1$ .

5.4. Let's start by setting up the problem and defining the exact solution  $u_e$ . We can define `kron` as an infix operator using the Unicode character  $\otimes$  within Julia to improve readability.

```
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 = spdiags([-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][:]
u_e = [φ(x)*φ(y)*φ(z)/2 for x∈x, y∈x, z∈x][:]
```

In practice, we could use routines from `IterativeSolvers.jl`, but because we want to evaluate the error at intermediate steps, we'll write our own. 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 - u_e, 1))
    end
    return(ε)
end
```

We also 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 - u_e, 1))
    end
    return(ε)
end
```

Finally, we run each method and plot the errors.

```
ε = zeros(400,4)
```

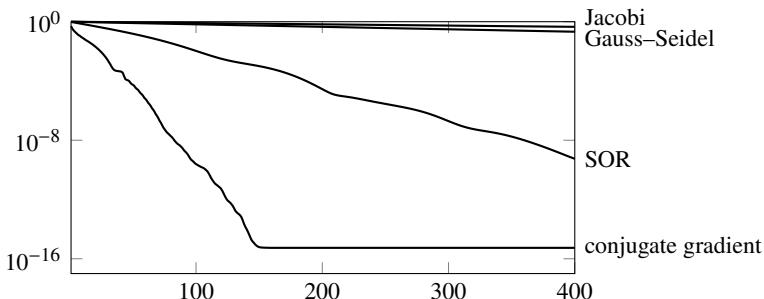


Figure A.6: Relative error as a function of the number of iterations.

```
@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 figure above shows the errors. Notice that the conjugate 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 iterations using a Jacobi method and 10 iterations using the conjugate gradient method. Convergence of the Jacobi method is extremely 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.

5.5. The  $(1, 1)$ -entry of  $\mathbf{A}^{-1}$  is the first element of the solution  $\mathbf{x}$  to  $\mathbf{Ax} = \boldsymbol{\xi}$  for  $\boldsymbol{\xi} = (1, 0, 0, \dots, 0)$ . 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 in d]
A = sparse(P) + spdiags(n, n, (d .=> B)...)
b = zeros(n); b[1] = 1
cg(A, b; Pl=P)[1]
```

The computed solution 0.7250783... agrees with 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

$$\begin{aligned} \omega_m^{k(m+j)} &= \omega_m^{km} \omega_m^{kj} = \omega_m^{kj}, & \text{and} & \omega_n^{m+j} &= \omega_n^m \omega_n^j = \omega_3 \omega_n^j, \\ \omega_m^{k(2m+j)} &= \omega_m^{2km} \omega_m^{kj} = \omega_m^{kj}, & \omega_n^{2m+j} &= \omega_n^{2m} \omega_n^j = \omega_3^2 \omega_n^j, \end{aligned}$$

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

$$\mathbf{F}_n \mathbf{c} = (\mathbf{F}_3 \otimes \mathbf{I}_{n/3}) \boldsymbol{\Omega}_n (\mathbf{I}_3 \otimes \mathbf{F}_{n/3}) \mathbf{P} \mathbf{c}$$

where  $\boldsymbol{\Omega} = \text{diag}(1, \omega_n, \omega_n^2, \dots, \omega_n^{n-1})$ .

```
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 two series, one with even indices running forward  $\{0, 2, 4, \dots\}$  and the other with odd indices running backward  $\{\dots, 5, 3, 1\}$ . 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  $\text{DCT}(f_k) = \operatorname{Re}(e^{-i\xi\pi/2n} \cdot \text{DFT}(f_{P(k)}))$ , 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 backward.

Let's define the functions `dctII` and `idctII`. Note that the FFTW.jl functions `fft` and `ifft` are multi-dimensional unless we specify a dimension.

```
function dctII(f)
    n = size(f,1)
    ω = exp.(-0.5im*π*(0:n-1)/n)
    return real(ω.*fft(f[[1:2:n; n-mod(n,2):-2:2],:],1))
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(ifft(f./ω,1))
    return f
end
```

We can verify our code by computing the DCT directly:

```
y1 = [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:

```
y2 = dct(f)/sqrt(2/n) ; y2[1] *=sqrt(2)
```

The two-dimensional DCT or IDCT applies the one-dimensional DCT or IDCT in each direction:

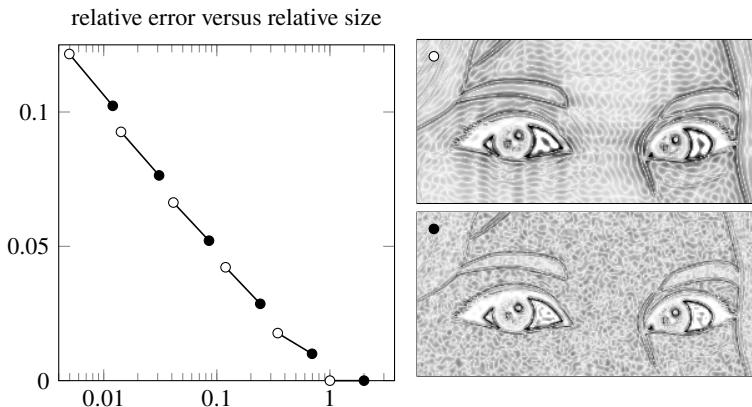


Figure A.7: The relative Frobenius error of a DCT-compressed image as a function of relative storage requirements, by  $\circ$  clipping high-frequency Fourier components and by  $\bullet$  zeroing out low-magnitude Fourier components.

```
dct2(f) = dctII(dctII(f')')
idct2(f) = idctII(idctII(f')')
```

6.4. The following function modifies the one on 163, which removes low-magnitude Fourier components, to instead crop high-frequency Fourier components and keep a full subarray. The compressed image  $A_0$  is recovered by padding the cropped components with zeros.

```
function dctcompress2(A,d)
    B = dct(Float64.(A))
    m,n = size(A)
    m0,n0 = sqrt(d).* (m,n) .|> floor .|> Int
    B0 = B[1:m0,1:n0]
    A0 = idct([B0 zeros(m0,n-n0); zeros(m-m0,n)])
    A0 = A0 .|> clamp01! .|> Gray
    sizeof(B0)/sizeof(B), sqrt(1-(norm(B0)/norm(B))^2), A0
end
```

The two approaches— $\circ$  cropping high frequency components and  $\bullet$  removing low magnitude components—are compared in Figure A.7 above. The lollipops show the relative compressed size and relative error for the same target density  $d$ . Surgically zeroing out low-magnitude components results in lower error than cropping high-frequency components, but it also requires additional overhead

for sparse matrix format. As a result, both approaches have similar effective error-compression ratios.

Let's look at the compression artifacts. Figure A.7 shows a close-up of the error at 95 percent compression. See the QR codes at the bottom of this page and page 163. Cropping high-frequency components results in the Gibbs phenomenon that appears as local, parallel ripples around the edges in the image that diminish away from the edges. On the other hand, by zeroing out low-magnitude components, instead of the Gibbs phenomenon, we see high-frequency graininess throughout the entire image.

## 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

$$f(x + ih) = f(x) + ihf'(x) - \frac{1}{2}h^2 f''(x) - \frac{1}{2}ih^3 f'''(\xi)$$

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.8 on the following page.

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

Because binary  $10 - 1 = 1$ , the double-precision representation of  $7/3 - 4/3 - 1$  equals

which is machine epsilon.



an image with  
increasing levels of  
DCT compressed

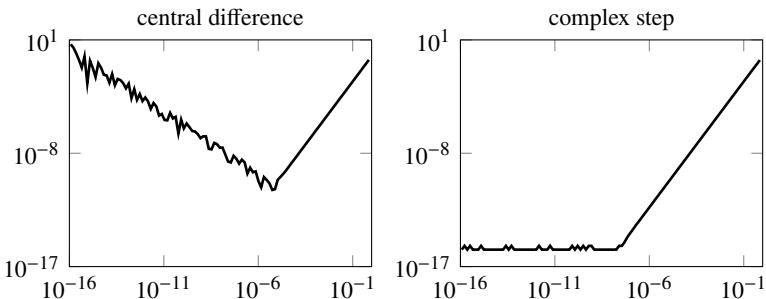


Figure A.8: The total error as a function of the stepsize  $h$  in the central difference and complex step approximations of  $f'(x)$ .

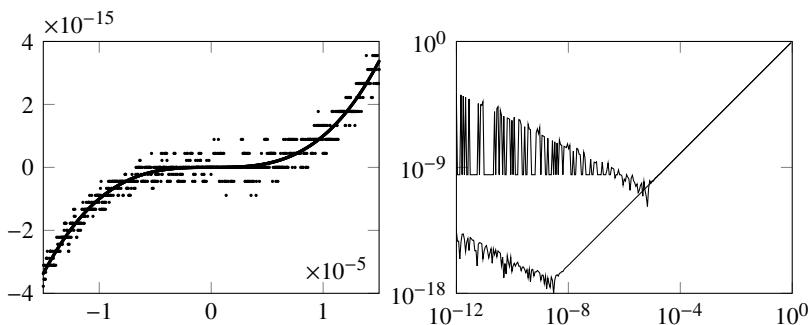


Figure A.9: Left:  $(x - 1)^3$  (solid line) and  $x^3 - 3x^2 + 3x - 1$  (dots). The  $x$ -axis is offset from  $x = 1$ . Right: The total error in the central difference approximation of  $f'(x)$  as a function of stepsize  $h$ .

7.3. See Figure A.9 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  $(x - 1)^3$

8.3. 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

$$e^{(k+1)} \approx \frac{f^{(n)}(x^*)}{n(n+1)} (e^{(k)})^2.$$

8.4. 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

1.00000000000000000000000000000000000000000000000000000000000000...  
 1.40000000000000000000000000000000000000000000000000000000000000...  
**1.41421319796954314720812182741116751269035532994923...**  
**1.41421356237309504879564008075425994635423824014524...**  
**1.41421356237309504880168872420969807856967187537694...**

The correct number of decimals—1, 2, 7, 19, and 62, respectively—triples with each iteration.

8.6. When the error is sufficiently small,  $|e^{(k+1)}| \approx r|e^{(k)}|^p$  for some positive  $r$ . We can 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  $e^{(k)} = x^{(k)} - x^*$  as

$$e^{(k+1)} = e^{(k)} - \frac{f(x^* + e^{(k)})(e^{(k)} - e^{(k-1)})}{f(x^* + e^{(k)}) - f(x^* + e^{(k-1)})}.$$

The Taylor series expansion of  $f(x^* + e^{(k)})$  is

$$f'(x^*)e^{(k)} + \frac{1}{2}f''(x^*)(e^{(k)})^2 + o(e^{(k)}) = f'(x^*)(1 + Me^{(k)})e^{(k)} + o(e^{(k)}),$$

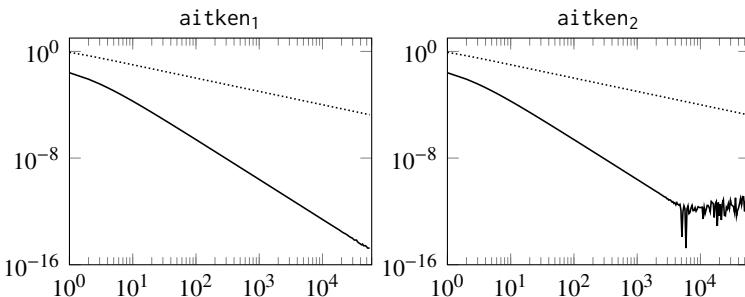


Figure A.10: Error in the  $n$ th partial sum of Leibniz's formula (dotted) and Aitken's extrapolation—(8.7) on the left and (8.6) on the right—of that sum (solid).

where  $M = f''(x^*)/2f'(x^*)$ . It follows that

$$\begin{aligned} e^{(k+1)} &= e^{(k)} - \frac{e^{(k)} f'(x^*) (1 + M e^{(k)}) (e^{(k)} - e^{(k-1)})}{f'(x^*) (e^{(k)} - e^{(k-1)}) (1 + M (e^{(k)} + e^{(k-1)}))} + o((e^{(k)})^2) \\ &= e^{(k)} - \frac{e^{(k)} (1 + M e^{(k)})}{1 + M (e^{(k)} + e^{(k-1)})} + o((e^{(k)})^2) \\ &= \frac{M e^{(k)} e^{(k-1)}}{(1 + M (e^{(k)} + e^{(k-1)}))} + o((e^{(k)})^2) \\ &\approx M e^{(k)} e^{(k-1)}. \end{aligned}$$

Substituting  $|e^{(k+1)}| \approx r |e^{(k)}|^p$  and  $|e^{(k)}| \approx r |e^{(k-1)}|^p$  into the expression for error yields  $r |e^{(k)}|^p \approx |M/r| |e^{(k)}| |e^{(k)}|^{1/p}$ , from which  $r^2 = |M|$  and  $p = 1+1/p$ . Therefore  $p = (\sqrt{5} + 1)/2 \approx 1.618$ .

8.7. Formulations (8.7) and (8.6) for Aitken's extrapolation are

$\text{aitken1}(x_1, x_2, x_3) = x_3 - (x_3 - x_2)^2 / (x_3 - 2x_2 + x_1)$ 
 $\text{aitken2}(x_1, x_2, x_3) = (x_1 * x_3 - x_2^2) / (x_3 - 2x_2 + x_1)$

Let's approximate  $\pi$  using the Leibniz formula for values  $n = 1, 2, \dots, 60000$ .

```

n = 60000
p = cumsum([(-1)^i*4/(2i+1) for i=0:n])
p1 = aitken1.(p[1:n-2],p[2:n-1],p[3:n])
p2 = aitken2.(p[1:n-2],p[2:n-1],p[3:n])
plot(abs.(π.-p)); plot!(abs.(π.-p2)); plot!(abs.(π.-p1))
plot!(xaxis=:log,yaxis=:log)

```

The left plot of Figure A.10 shows the errors in Leibniz's formula and Aitken's correction. The log-log slopes are 1.0 and 3.0. The errors after the  $n$  term are approximately  $1/n$  and  $1/4n^3$ , respectively. The methods are both linearly convergent, but using Aitken's acceleration is significantly faster. Instead of a trillion terms to compute the first 13 digits of  $\pi$ , we need nine thousand terms. It's still an exceptionally slow way of approximating  $\pi$ . The right plot of Figure A.10 shows the numerically less-stable Aitken's method.

We can express the Aitken's extrapolation to Leibniz's formula explicitly as

$$\pi_n = \sum_{i=1}^n \frac{(-1)^{i+1} 4}{2i-1} + r_n \quad \text{where} \quad r_n = (-1)^n \frac{2n-3}{(n-1)(2n-1)}.$$

The 14th-century Indian mathematician Madhava of Sangamagrama formulated a better correction

$$r_n = (-1)^n \frac{4n^2 + 4}{4n^3 + 5n}.$$

The error using Madhava's correction goes as  $3/4n^7$ . With it, we can compute the first 13 digits of  $\pi$  in sixty terms.

Modern methods have quadratic or higher rates of convergence. The Gauss-Legendre algorithm uses iteration with arithmetic and harmonic means to get quadratic convergence—the first 16 digits of  $\pi$  need three terms, and the correct digits double with each additional term. Jonathan and Peter Borwein's algorithm for  $1/\pi$  converges quartically, nonically, and even hexadecimally. Each iteration multiplies the correct number of digits by four, nine, and sixteen.

8.8. Consider Newton's method where  $f(x) = x - \phi(x) = 0$ :

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - \phi(x^{(k)})}{1 - \phi'(x^{(k)})}.$$

Let's further approximate the slope  $\phi'(x^{(k)})$  as

$$\phi'(x^{(k)}) \approx \frac{\phi(x^{(k+1)}) - \phi(x^{(k)})}{x^{(k+1)} - x^{(k)}} = \frac{\phi(\phi(x^{(k)})) - \phi(x^{(k)})}{\phi(x^{(k)}) - x^{(k)}}.$$

Substituting this expression for  $\phi'(x^{(k)})$  yields

$$x^{(k+1)} = x^{(k)} - \frac{(\phi(x^{(k)}) - x^{(k)})^2}{\phi(\phi(x^{(k)})) - 2\phi(x^{(k)}) + x^{(k)}}. \quad (\text{A.1})$$

We can further rearrange the terms of this expression to match the form of (8.8)

$$x^{(k+1)} = \phi(\phi(x^{(k)})) - \frac{(\phi(\phi(x^{(k)})) - \phi(x^{(k)}))^2}{\phi(\phi(x^{(k)})) - 2\phi(x^{(k)}) + x^{(k)}}.$$

Note that we can also express Steffenson's method (A.1) in the form

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{q(x^{(k)})} \quad \text{with} \quad q(x) = \frac{f(x + f(x)) - f(x)}{f(x)}$$

where  $q(x)$  is an approximation for the slope  $f'(x)$ .

We'll use Wolfram Cloud to determine the convergence rate.<sup>1</sup>

```
a = Series[ f[x]^2, {x, 0, 3}] /. f[0]->0;
b = Series[ f[x+f[x]] - f[x], {x, 0, 3}] /. f[0]->0 /. f[0]->0;
Simplify[x - a/b] /. x->ε
```

We find the error of Steffenson's method is

$$e^{(k+1)} = \frac{(1 + f'(x^*))f''(x^*)}{2f'(x^*)}(e^{(k)})^2 + O((e^{(k)})^3).$$

Recall that Newton's method has a similar error

$$e^{(k+1)} = \frac{f''(x^*)}{2f'(x^*)}(e^{(k)})^2 + O((e^{(k)})^3).$$

**8.10.** 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  $x^{(k+1)} = x^{(k)} - f(x^{(k)})/f'(x^{(k)})$ .

**8.12.** 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:

---

<sup>1</sup>Wolfram Cloud (<https://www.wolframcloud.com>) has a free basic plan that provides complete access to Mathematica with a few limitations.

```
using DifferentialEquations
function homotopy(f,dx,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,dx,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...));
[4*x*(x^2+y^2-1) 4y*(x^2+y^2+1);
 6*x*(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]))
```

**8.13.** 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
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 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

$$x(x(x(\cdots(x + a_{n-1}) + \cdots + a_2) + a_1) + a_0).$$

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 latter. This approach 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)

```

8.14. 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, if  $(1 - \alpha m + \beta)^2 < 4\beta$ , the discriminant is negative and the eigenvalues are complex. From this,

$$|\lambda_{\pm}|^2 = \frac{1}{4}(1 - \alpha m + \beta)^2 - \frac{1}{4}(1 - \alpha m + \beta)^2 + \beta = \beta.$$

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

$$\det(\mathbf{V}) = \prod_{j=0}^n \prod_{\substack{i \neq j \\ i \neq k}} (x_i - x_j). \quad (\text{A.2})$$

The relation clearly holds for  $n = 1$ . Assume that (A.2) 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

$$\det(\mathbf{V}[x_k]) = \prod_{\substack{j=0 \\ j \neq k}}^n \prod_{\substack{i \neq j \\ i \neq k}} (x_i - x_j) \quad (\text{A.3})$$

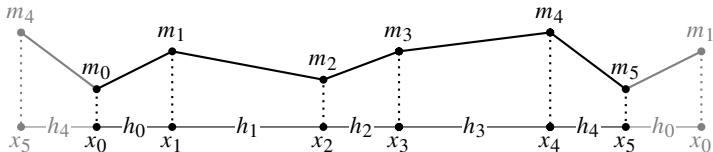
By cofactor expansion  $\det(\mathbf{V})$  equals

$$x_0^n \det(\mathbf{V}[x_0]) + \dots + (-1)^k x_k^n \det(\mathbf{V}[x_k]) + \dots + (-1)^n x_n^n \det(\mathbf{V}[x_n])$$

By (A.3),

$$\det(\mathbf{V}) = \sum_{k=0}^n (-1)^k x_k^n \prod_{\substack{j=0 \\ j \neq k}}^n \prod_{\substack{i \neq j \\ i \neq k}} (x_i - x_j).$$

**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:



The following code modifies `spline_natural` on page 238 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])
```

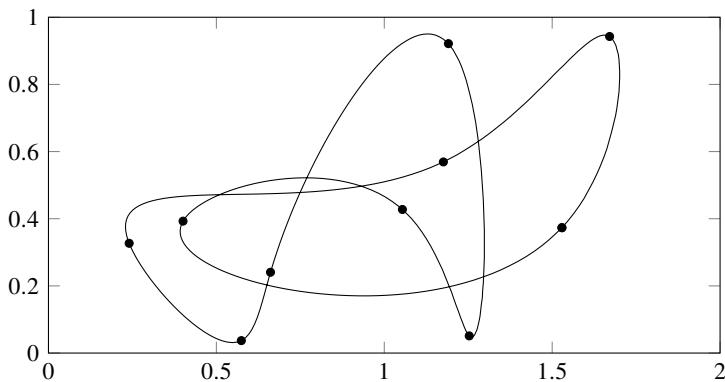


Figure A.11: A closed parametric spline passing through 10 knots.

```

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  $n \times n$  points through a set of  $n$  random points using the function `evaluate_spline` defined on page 239. One solution is shown in Figure A.11 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)

```

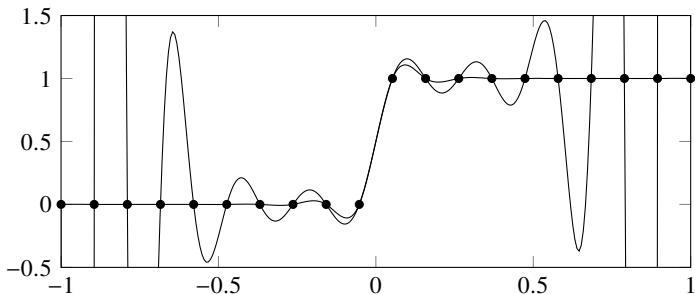
9.3. First, let's set up the domain  $x$  and the function  $y$ , which we are interpolating.

```

n = 20; N = 200
x = LinRange(-1,1,n)
y = float(x .> 0);

```

Next, define the radial basis functions and the polynomials bases.

Figure A.12: Interpolation using polynomial and  $|x|^3$  radial basis function .

```

 $\phi_1(x, a) = \text{abs}(x - a)^3$ 
 $\phi_2(x, a) = \exp(-20(x - a)^2)$ 
 $\phi_3(x, a) = x^a$ 

```

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 = :o, legend = :none)
plot!(X,[Y1,Y2,Y3], ylim=[-0.5,1.5])

```

As seen in Figure A.12 above, the polynomial interpolant suffers from the Runge phenomenon with a maximum error of about 500. On the other hand, the radial basis function gives us a good approximation. The radial basis function  $\phi(x) = |x|^3$  generates 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

We have 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  $\sum_{j=-1}^{n+1} c_j v_j(x)$ . We start by defining a general collocation solver for  $L u(x) = f(x)$ . This solver takes boundary conditions  $bc$  and an array of equally-spaced collocation points  $x$ . It 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]]
    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  $A[end, end-2]$ . Next, we define a function that will interpolate between collocation points:

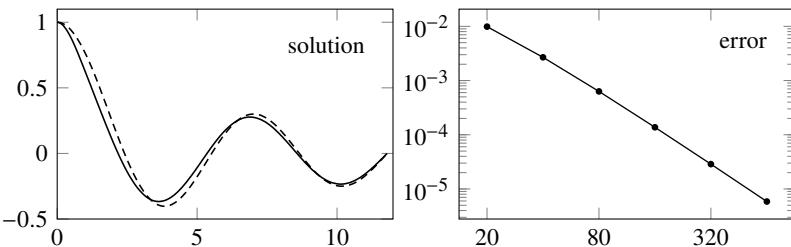


Figure A.13: Numerical solution (solid) and exact solution (dashed) to Bessel's equation for 15 collocation points (left). The log-log plot of the error as a function of number of points (right).

```
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,:))
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.13 above using 15 collocation points. Even with only 15 points, the numerical solution is fairly accurate. Finally, let's measure the convergence rate by increasing the 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 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.

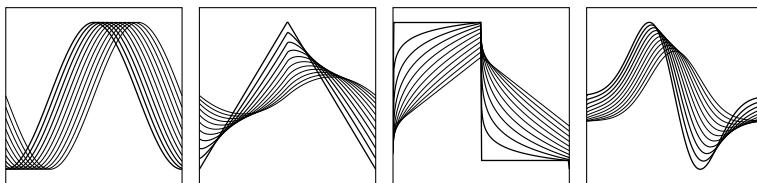
```

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 follows.

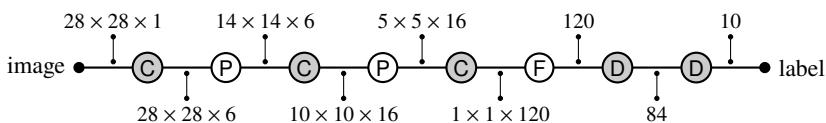
```
using FFTW
n = 256; ℓ = 2
x = (0:n-1)/n*ℓ .- ℓ/2
ξ = [0:(n/2-1); (-n/2):-1]*(2π/ℓ)
f₁ = exp.(-16*x.^2)
f₂ = sin.(π*x)
f₃ = x.* (1 .- abs.(x))
deriv(f,p) = real(ifft((im*ξ).^p.*fft(f)))
```

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 code 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 below 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



fractional derivatives  
of the Gaussian  
function

maps. Each of these feature maps is 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  $5 \times 5$  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 of 120 filters, producing a  $1 \times 1 \times 120$  feature map that 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 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 = MLDatasets.MNIST.traindata(Float32)
image_test, label_test = MLDatasets.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

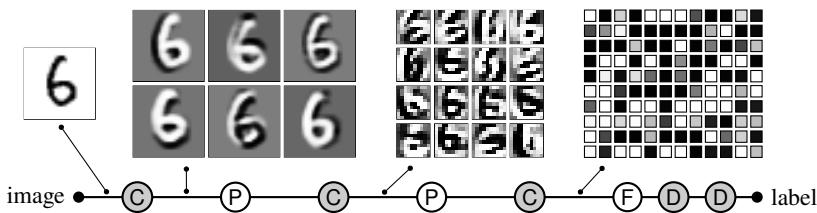


Figure A.14: An input image and the intermediate transformations of that image following the three convolutional matrices of LeNet-5.

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 five 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 the 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 a 2 percent error rate—better than the 5 percent error rate obtained using principal component analysis in exercise 3.7. The figure above shows the intermediate layer inputs. The first convolutional layer appears to detect edges in the image. The second convolutional layer then 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  $x$ ,  $x + h$ ,  $x + 2h$  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  $C * d.^n / \text{factorial}(n)$ :

```
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

$$h^{-1} \left( \left| -\frac{11}{6} \right| + |3| + \left| -\frac{3}{2} \right| + \left| \frac{1}{3} \right| \right) \text{eps} \approx 7h^{-1} \text{eps},$$

where  $\text{eps}$  is machine epsilon. The truncation error decreases and round-off error increases error as  $h$  gets smaller. Following the discussion on page 187, 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 term. 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 301 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...` 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)` 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

$$f'(a) = \frac{f(a + i\varepsilon) - f(a - i\varepsilon)}{2i\varepsilon} + O(\varepsilon^2) = \frac{\text{Im } f(a + i\varepsilon)}{\varepsilon} + O(\varepsilon^2),$$

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

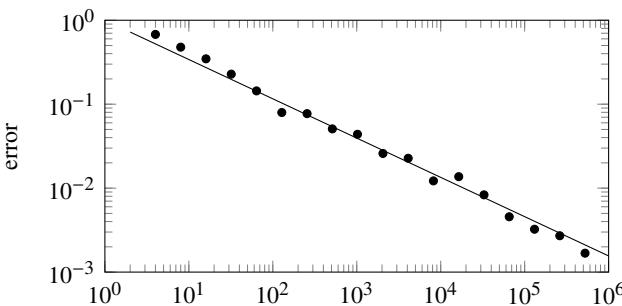
```
mc_π(n) = sum(sum(rand(2,n).^2,dims=1).<1>)/n*4
```

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 log-log slope of the error is  $-0.468$ , confirming the expected  $O(N^{-1/2})$  half-order of convergence. The Monte Carlo method would need roughly  $10^{25}$  samples for 13 digits of accuracy—taking about 800 years on my laptop. 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,dim=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.** The  $\theta$ -scheme corresponds to the forward Euler scheme, the trapezoidal scheme, and the backward Euler scheme when  $\theta = 1$ ,  $\theta = \frac{1}{2}$ , and  $\theta = 0$ , respectively. So the regions of absolute stability of the  $\theta$ -scheme will correspond with those three schemes when  $\theta$  is any of those three values. Let's find the region of absolute stability for a general  $\theta$ . 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 determine the values  $\lambda k$  for which  $|r| \leq 1$ . To do this, we'll find the boundary of the region  $\lambda k = x + iy$  where  $|r| = 1$ .

$$\begin{aligned} 1 = |r|^2 = r\bar{r} &= \frac{1 + \theta(x + iy)}{1 - (1 - \theta)(x + iy)} \frac{1 + \theta(x - iy)}{1 - (1 - \theta)(x - iy)} \\ &= \frac{1 + 2\theta x + \theta^2(x^2 + y^2)}{1 - 2(1 - \theta)x + (1 - \theta)^2(x^2 + y^2)}, \end{aligned}$$

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 yields

$$\left(x - \frac{1}{1-2\theta}\right)^2 + y^2 = \frac{1}{(1-2\theta)^2}.$$

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'll pair a BDF3 scheme with an appropriate explicit scheme. The BDF3 method approximates  $u'$  at time  $t_{n+1}$ , so we'll need to evaluate the explicit scheme at the same time  $t_{n+1}$  to avoid a splitting error. To do this, we need a third-order approximation of  $g(U^{n+1})$ .

The coefficients of the BDF3 scheme can be computed using the function `multistepcoefficients` from page 347 with an input `m = [0 1 2 3]` and `n = [0]`. Alternately, we can compute them by solving the Taylor polynomial system

$$u'(t - ik) = u - iku' + \frac{1}{2}i^2 k^2 u'' - \frac{1}{6}i^3 k^3 u''' + O(k^4), \quad (\text{A.4})$$

keeping the  $u'$  term while eliminating the  $u$ ,  $u''$ , and  $u'''$  terms.

```
i = 0:3; c = ((-i)'.^i ./ factorial.(i))\[0;1;0;0]
```

The coefficients are `[11//6 -3//1 3//2 -1//3]`, and the BDF3 scheme is correspondingly

$$\frac{11}{6}U^{n+1} - 3U^n + \frac{3}{2}U^{n-1} - \frac{1}{3}U^{n-2} = kf(U^{n+1}).$$

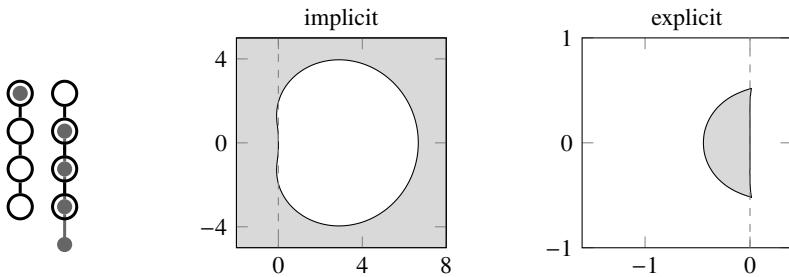


Figure A.15: The stencil and regions of absolute stability (in gray) for the implicit and explicit parts of the third-order IMEX method.

To approximate  $g(U^{n+1})$ , we again solve (A.4). This time, we keep the  $u$  term and eliminate the  $u'$ ,  $u''$  and  $u'''$  terms.

```
i = 0:3; c = ((-(i.+1)).^i./factorial.(i))\ [1;0;0;0]
```

The coefficients are  $[4 \ -6 \ 4 \ -1]$ , and the approximation of  $g(U^{n+1})$  is

$$\tilde{g}(U^n, U^{n-1}, U^{n-2}, U^{n-3}) = g \left( 4U^n - 6U^{n-1} + 4U^{n-2} - U^{n-3} \right).$$

We combine the implicit and explicit methods to get

$$\frac{\frac{11}{6}U^{n+1} - 3U^n + \frac{3}{2}U^{n-1} - \frac{1}{3}U^{n-2}}{k} = f(U^{n+1}) + \tilde{g}(U^n, U^{n-1}, U^{n-2}, U^{n-3}).$$

The region of absolute stability for the BDF3 is known from Figure 12.5. To determine the region of stability for the explicit scheme, we take  $g(u) = \lambda u$  and let  $r = U^{n+1}/U^n$ :

$$\frac{11}{6}r^4 - 3r^3 + \frac{3}{2}r^2 - \frac{1}{3}r = \lambda k \left( 4r^3 - 6r^2 + 4r - 1 \right).$$

We express  $\lambda k(r)$  as a rational function and find the boundary of the domain  $|r| \leq 1$  by taking  $r = e^{i\theta}$ . Figure A.15 above shows the regions of stability.

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

$$\tilde{U}^{n+1} = U^n + \alpha z \tag{A.5a}$$

$$U^{n+1} = U^n + (\tilde{U}^{n+1} + \beta)z, \tag{A.5b}$$

where  $z = \lambda k b_{-1}^*$ . Substituting (A.5a) into (A.5b) yields

$$U^{n+1} = U^n + (U^{n+1} + \beta)z + \alpha z^2.$$

For an additional corrector iteration, we substitute this new expression for  $\tilde{U}^{n+1}$  in (A.5b), giving us

$$U^{n+1} = U^n + (U^n + \beta)(z + z^2) + \alpha z^3.$$

Continuing for additional corrector iterations produces

$$U^{n+1} = U^n + (z + z^2 + \cdots + z^s)(U^n + \beta) + z^{s+1}\alpha.$$

To find the boundary of the region of absolute stability, we let  $r = U^n/U^{n+1}$  and look for the solutions to the following equation when to  $|r| = 1$

$$1 = r + (z + z^2 + \cdots + z^s)(r + \beta(r)) + \alpha(r)z^{s+1}, \quad (\text{A.6})$$

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.6) by finding the roots of the polynomial. We'll use the function `multistepcoefficients` from page 347 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 = multistepcoefficients([0 1],hcat(1:n-1...))
    _,b = multistepcoefficients([0 1],hcat(0:n-1...))
    α(r) = a · r.^(1:n-1)/b[1]
    β(r) = b[2:end] · r.^(1:n-1)/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
```

We plot the first-order ABM PE(CE) $^m$  using

```
z = vcat([PECE(2,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.16 on the facing 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

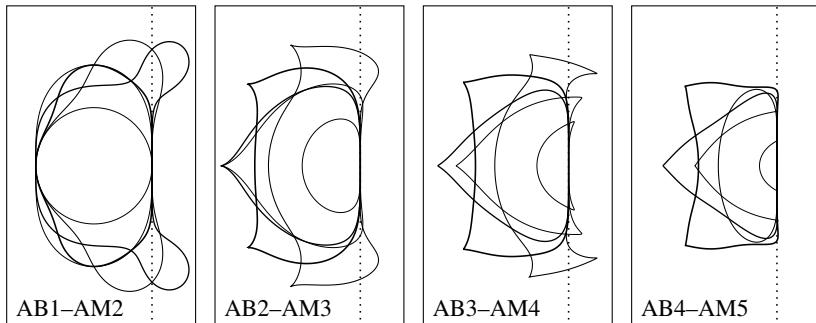


Figure A.16: Regions of stability for Adams–Bashforth–Moulton PE(CE) $^m$  methods for  $m = 0$  (thin curve) to  $m = 4$  (thick curve). The domain is  $[-2.5, .75] \times [-2.75, 2.75]$ .

$$P_m(x) = Q_n(x)T(x) + O(x^{m+n+1}):$$

$$\sum_{i=0}^m p_i x^i = \left(1 + \sum_{i=1}^n q_i x^i\right) \left(\sum_{i=0}^{\infty} a_i x^i\right) + O(x^{m+n+1}). \quad (\text{A.7})$$

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.7).

```
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  $r - 1$  for  $x$  in the Padé approximant and regroup by powers of  $r$ . This amounts to multiplying the two coefficient vectors by upper inverse Pascal matrices.

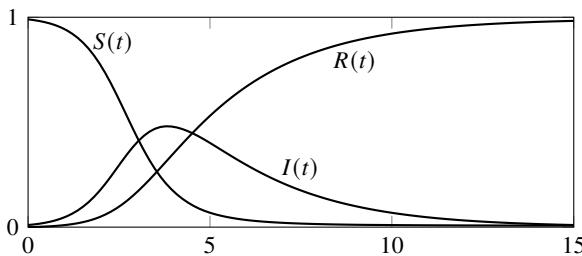


Figure A.17: Solution to the SIR model in problem 12.13.

```
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. While we can write the code that solves this problem succinctly, Julia also lets us write code that clarifies the mathematics. Note that `SIR!` is an in-place function.

```
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.17.

**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:

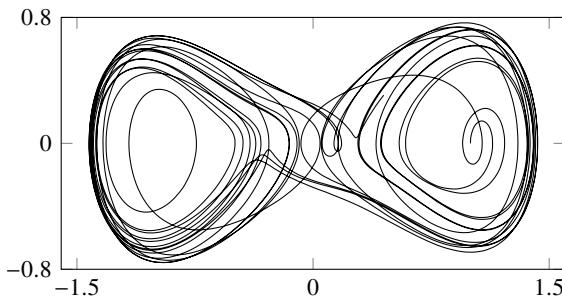


Figure A.18: The phase map of the Duffing equation in problem 12.14.

```
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.18 above.

**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
```

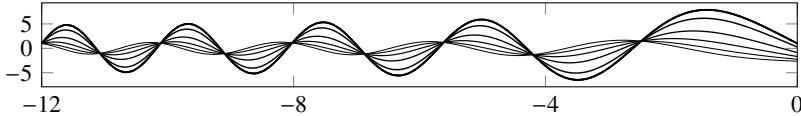
Now, we can solve the boundary value problem using the shooting method with boundary condition `bc` and an initial guess `guess`. Because `find_zero` takes only two arguments, we'll use an anonymous function `shoot_airy` to let us format and pass our additional parameters to `solveIVP`.

```
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 the initial condition  $v \approx 16.149$ . Let's plot the solution:

```
sol = solve(ODEProblem(airy,[bc[1],v],domain))
plot(sol)
```

We can see the shooting method in action by plotting each intermediate solution of `solveIVP`. In the following figure, the lines grow thicker with each iteration until we reach the solution—the widest 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

$$\begin{aligned} U_j^n : \quad u(t, x) &= u \\ U_j^{n+1} : \quad u(t+k, x) &= u + ku_t + \frac{1}{2}k^2u_{tt} + O(k^3) \\ U_{j+1}^n : \quad u(t, x+k) &= u + hu_x + \frac{1}{2}h^2u_{xx} + \frac{1}{6}h^3u_{xxx} + \frac{1}{24}h^4u_{xxxx} + O(h^5) \\ U_{j-1}^n : \quad u(t, x-k) &= u - hu_x + \frac{1}{2}h^2u_{xx} - \frac{1}{6}h^3u_{xxx} + \frac{1}{24}h^4u_{xxxx} - O(k^5). \end{aligned}$$

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 tells us that the Crank–Nicolson is  $O(k + h^2)$  to  $u_t = u_{xx}$ .

### 13.3. By making the substitution

$$U_{j+2}^n = \hat{u}(t_n, \xi) e^{ijh\xi}$$

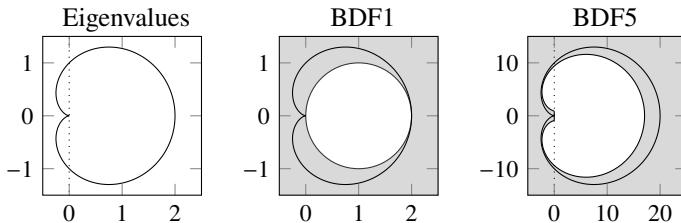
in the expression

$$\frac{U_{j+2}^n - 2U_{j+1}^n + U_{j-1}^n}{h^2},$$

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) method, we would need to take  $k > h^2/2$ . And for the BDF5 method, we would need to take  $k > 5h^2$ . Every other method we've examined is unconditionally unstable.

**13.4.** We'll use the function `multistepcoefficients`, defined on page 347.

```
m, n = [0 1 2 3 4], [1]
a, b = zeros(maximum(m)), zeros(maximum(n))
a[m], b[n] = multistepcoefficients(m,n)
```

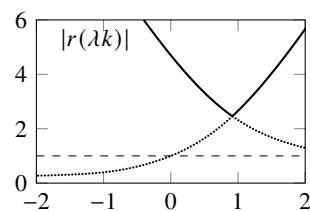
The stencil has the following approximation for the time derivative

$$\frac{1}{4}U^{n+1} + \frac{5}{6}U^n - \frac{3}{2}U^{n-1} + \frac{1}{2}U^{n-2} - \frac{1}{12}U^{n-3}. \quad (\text{A.8})$$

We can examine the stability using the multiplier  $r(\lambda k)$  along the negative real axis similar to Figure 12.2 on page 337.

```
lambda_k = r -> (a * r.^-(1:length(a))) ./ (b * r.^-(1:length(b)))
r = LinRange(0.2, 6, 100)
plot([lambda_k.(r) lambda_k.(-r)], [r r], xlim=[-2, 2])
```

The plot on the right shows  $|r(\lambda k)|$ . One branch of the curve closely matches the exponential function (as we should expect). The other branch increases as  $\lambda k$  decreases. The minimum of the maximum of  $|r(\lambda k)|$  is around 2.46—it never gets below 1. So the method is unconditionally unstable.



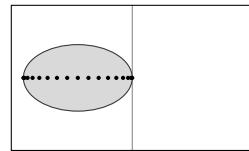
**13.5.** Start with the Dufort–Frankel scheme as (13.11) and move the second term on the right over to the left-hand side:

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} + a \frac{U_j^{n+1} - 2U_j^n + U_j^{n-1}}{h^2} = a \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

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



$$\lambda_\xi = i \frac{1}{k} \sin \theta - \frac{2a}{h^2} (\cos \xi h - 1),$$

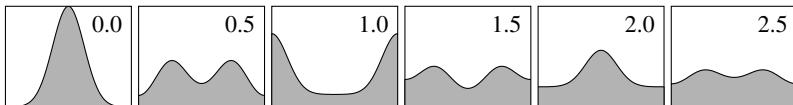
which is an ellipse in the negative half-plane that extends from 0 to  $-4a/h^2$ , exactly the size that we need to fit the eigenspectrum.

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 the consistency of the Dufort–Frankel, we'll simply set  $U^1$  equal to  $U^0$ .

```

 $\Delta x = 0.01; \Delta t = 0.01$ 
 $\ell = 1; x = -\ell:\Delta x:\ell; m = \text{length}(x)$ 
 $U^n = \exp(-8*x.^2); U^{n-1} = U^n$ 
 $\nu = \Delta t/\Delta x^2; \alpha = 0.5 + \nu; \gamma = 0.5 - \nu$ 
 $B = \nu * \text{Tridiagonal}(\text{ones}(m-1), \text{zeros}(m), \text{ones}(m-1))$ 
 $B[1,2] *= 2; B[end, end-1] *= 2$ 
 $@gif for i = 1:300$ 
 $global U^n, U^{n-1} = (B*U^n + \gamma * U^{n-1})/\alpha, U^n$ 
 $plot(x, U^n, ylim=(0,1), label=:none, fill=(0, 0.3, :red))$ 
end

```



The snapshots above and the QR code at the bottom of the page show the solution to the heat equation using the Dufort–Frankel scheme. While ultimately dissipative, the solution is mainly oscillatory, behaving more like a viscous fluid than a heat-conducting bar.

**13.7.** Let's determine the CFL condition. We can write the constant-potential Schrödinger equation 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

$$\frac{\partial \hat{u}}{\partial t} = -i \frac{2\varepsilon}{h^2} \sin^2 \left( \frac{\xi h}{2} \right) \hat{u} - \varepsilon^{-1} V \hat{u}. \quad (\text{A.9})$$

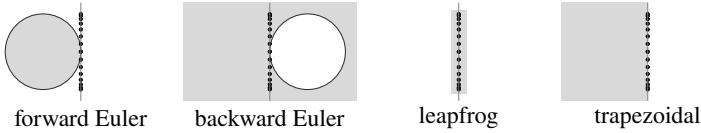


the Dufort–Frankel  
solution to the  
heat equation

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  $|\lambda_\xi| \leq 2\varepsilon/h^2 + \varepsilon^{-1}V$ . We'll need to keep  $h$  much smaller than  $\varepsilon$  to model the dynamics of the Schrödinger equation accurately. Consequently,  $2\varepsilon/h^2$  will dominate  $\varepsilon^{-1}V$ . So let's only consider the contribution of  $2\varepsilon/h^2$  on stability. Note where the regions of absolute stability intersect the imaginary axis for different methods in the following figures:



We see that the backward Euler and trapezoidal methods are unconditionally stable and that the forward Euler method is unconditionally unstable. The leapfrog method is stable when  $|k\lambda_\xi| < 1$ , meaning its CLF condition is  $k < h^2/2\varepsilon$ . Figure 12.8 shows that the Runge–Kutta method (RK4) is stable when  $|k\lambda_\xi| < 2.5$ , so its CFL condition is  $k < 1.25h^2/\varepsilon$ . Figures 12.5 and 12.6 show that the BDF2 method 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 the 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,ε)
    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; ex=[]; et=[]
N = floor.(Int,exp10.(LinRange(2,3.7,6)))
x = LinRange(-4,4,m)
ψm = -exp.(-(x.-1).^2/2ε)/(π*ε)^(1/4);
for n ∈ N
    x = LinRange(-4,4,n)
    ψn = -exp.(-(x.-1).^2/2ε)/(π*ε)^(1/4)
    append!(et,norm(ψm - schroedinger(m,n,ε))/m)
    append!(ex,norm(ψn - schroedinger(n,m,ε))/n)
end
plot(2π./N,et,shape=:circle, xaxis=:log, yaxis=:log)
plot!(8 ./N,ex,shape=:circle)

```

The error is plotted in Figure A.19 on the facing page. We can compute the slopes of the lines using `polyfit(log(n), log(error_x), 1)`. The slope for  $k$  is 2.0, and for  $h$ , it is 2.5.

Finally, let's examine the effect  $\varepsilon$  and  $h$  have on the solution  $\rho(t,x) = |\psi(t,x)|^2$ . We find the solution at  $t = \pi$  using  $10^4$  time steps. We compute the error for several 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 time step  $k$  is sufficiently small, and its contribution to the error is negligible. The middle plot of Figure A.19 shows the error as a function of  $h$  for each of the four values of  $\varepsilon$ , decreasing from right to left. The log-log curves are linear and parallel 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. Consequently, the wave packet gradually slows until the numerical solution no longer coincides with the true solution. If we plot the error against  $h/\varepsilon$ , 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  $O((k/\varepsilon)^2 + (h/\varepsilon)^2)$ .

*13.8.* 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

$$\frac{\partial u}{\partial t} = 2 \frac{\partial^2 u}{\partial r^2}.$$

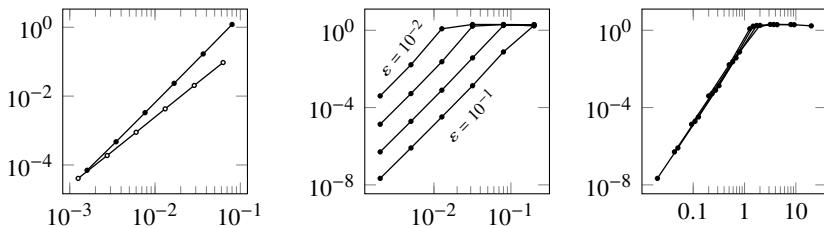


Figure A.19: Left: the error in solution  $\rho(t, x)$  as a function of step size  $h$  • or time step  $k$  ◊. Middle: the error versus step size  $h$  for various  $\epsilon$ . Right: the error versus step size as a multiple of  $\epsilon$ .

Let's discretize space to get  $\frac{\partial}{\partial t} U_j = D U_j$  where

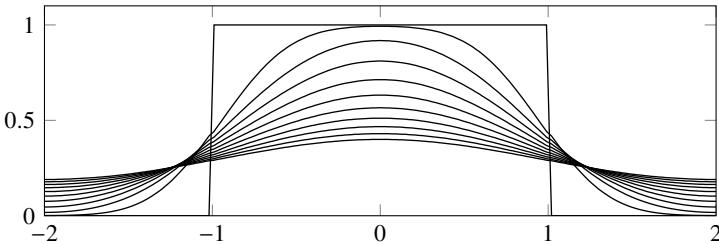
$$\begin{aligned} D U_0 &= 2 \frac{U_1 - 2U_0 + U_{-1}}{h^2}, \text{ and} \\ D U_j &= \frac{U_{j+1} - 2U_j + U_{j-1}}{h^2} + \frac{1}{r_j} \frac{U_{j+1} - U_{j-1}}{2h} \quad \text{for } j = 1, 2, \dots, m. \end{aligned}$$

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  $U_j^{n+1} = (I - \frac{1}{2}k D)^{-1}(I + \frac{1}{2}k D)U_j^n$ :

```
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
```

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 code below show the solution with an initial distribution given by a step function. The snapshots are at equal intervals from  $t \in [0, \frac{1}{2}]$  of the solution reflected about  $x = 0$ . Compare this solution to the one-dimensional heat equation on page 380.

**13.9.** 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  $\frac{1}{3}(h_2 - h_1)f'''(x_2) + O(h_1^2 + h_2^2)$ . If  $h_2 = h_1$ , 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 to `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]
    d₀ = @. -2/[(Δx2[end]).^2; Δx1*Δx2; Δx1[1].^2]
    d_+ = @. 2/[(Δx2[end]).^2; Δx2*(Δx1+Δx2) ]
    Tridiagonal(d_-,d₀,d_+)
end
```



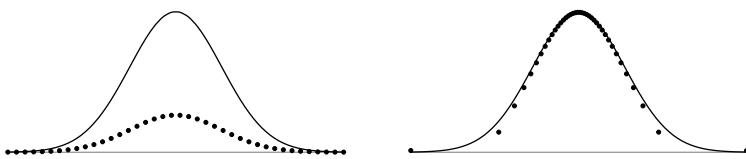
the heat equation in  
polar coordinates

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 define an initial condition and compute the solutions.

```
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,shape=:circle,label="equal")
x = loginspace(20,m,0.999)
u₂ = heat_equation(x,t,m,ϕ(x,0,10))
plot!(x,u₂,shape=:circle,label="logit")
```



The plots above compare the solution using equally-spaced nodes (left) and the solution using the inverse-sigmoid-spaced nodes (right) with the exact solution.

**13.10.** We'll solve the Allen–Cahn equation using Strang splitting. The differential equation  $u'(t) = \varepsilon^{-1}u(1 - u^2)$  has the analytical solution  $u(t) = (u_0^2 - (u_0^2 - 1)e^{-t/\varepsilon})^{-1/2}u_0$ . 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)

```

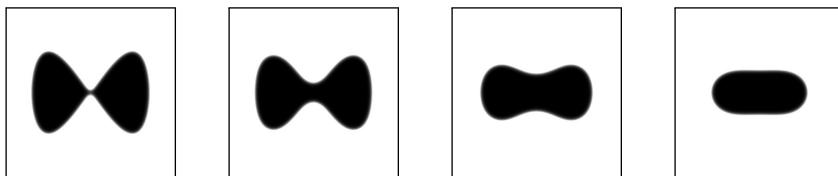
To watch the solution's evolution, we add the following three commands (the first before the loop, the second inside the loop, and the third after the loop) to the code above.

```

anim = Animation()
(i%10)==1 && (plot(Gray.(u),border=:none); frame(anim))
gif(anim, "allencahn.gif", fps = 30)

```

The figures below show the solutions to the Allen–Cahn equation at times  $t = 0.05, 0.5, 2.0$ , and  $4.0$ . Also, see the QR code 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.



the Allen–Cahn  
equation

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

$$u_t + cu_x + \frac{1}{2}ku_{tt} - \frac{1}{2}\frac{h^2}{k}u_{xx} + \{ \text{higher order terms} \} = 0,$$

from which we see that the truncation error is  $O(k + h^2/k)$ .

We examine the leading order truncation

$$u_t + cu_x + \frac{1}{2}ku_{tt} - \frac{1}{2}\frac{h^2}{k}u_{xx} = 0 \quad (\text{A.10})$$

to find the dispersion relation. We will eliminate the  $u_{tt}$  term by first differentiating (A.10) with respect to  $t$  and  $x$  and then substituting the expression for  $u_{tt}$  back into the original equation, discarding the higher-order terms. We have

$$\begin{aligned} u_{tt} &= -cu_{xt} + O(k + h^2/k) \\ u_{tx} &= -cu_{xx} + O(k + h^2/k) \end{aligned}$$

from which

$$u_{tt} = c^2u_{xx} + O(k + h^2/k).$$

After eliminating higher-order terms, equation (A.10) 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

$$\left[ i\omega + c(-i\xi) - \frac{1}{2} \left( c^2 k - \frac{h^2}{k^2} \right) \xi^2 \right] e^{i(\omega t - \xi x)} = 0.$$

So

$$i\omega + c(-i\xi) - \frac{1}{2} \left( c^2 k - \frac{h^2}{k^2} \right) \xi^2 = 0,$$

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

$$\begin{aligned} u(t+k, x) &= u + ku_t + \frac{1}{2}k^2 u_{tt} + \frac{1}{6}k^3 u_{ttt} + O(k^4) \\ u(t-k, x) &= u - ku_t + \frac{1}{2}k^2 u_{tt} - \frac{1}{6}k^3 u_{ttt} + O(k^4) \\ u(t, x+h) &= u + hu_x + \frac{1}{2}h^2 u_{xx} + \frac{1}{6}h^3 u_{xxx} + O(h^4) \\ u(t, x-h) &= u - hu_x + \frac{1}{2}h^2 u_{xx} - \frac{1}{6}h^3 u_{xxx} + O(h^4) \end{aligned}$$

from which

$$u_t + \frac{1}{2}k^2 u_{ttt} + cu_x + \frac{1}{2}h^2 u_{ttt} = O(k^2 + h^2).$$

So, the method is  $O(k^2 + h^2)$ .

The scheme is leapfrog 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

$$\frac{\partial}{\partial t} \hat{u}(t, \xi) = -c \frac{e^{i\xi h} - e^{-i\xi h}}{h} \hat{u}(t, \xi) = -i \frac{c}{h} \sin(\xi h) \hat{u}(t, \xi).$$

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

$$\rho_t + (\rho u)_x = 0 \quad (\text{A.11a})$$

$$(\rho u)_t + (\rho u^2 + p)_x = 0 \quad (\text{A.11b})$$

$$E_t + ((E + p)u)_x = 0 \quad (\text{A.11c})$$

in semilinear form using  $\rho$ ,  $u$  and  $p$  as independent variables. First, rewrite (A.11a) and (A.11b) as

$$\rho_t + u\rho_x + \rho u_x = 0 \quad (\text{A.12})$$

$$u\rho_t + \rho u_t + u^2 \rho_x + 2\rho u u_x + p_x = 0 \quad (\text{A.13})$$

We can eliminate the  $u\rho_t$  term in (A.13) by first multiplying (A.12) by  $u$  to get

$$u\rho_t + u^2 \rho_x + \rho u u_x = 0$$

and then subtracting this equation from (A.13) to get

$$\rho u_t + \rho u u_x + p_x = 0.$$

$$u_t + uu_x + \frac{1}{\rho} p_x = 0 \quad (\text{A.14})$$

To derive an equation for  $p$ , we will use the equation of state

$$E = \frac{1}{2}\rho u^2 + \frac{p}{\gamma - 1} \quad (\text{A.15})$$

to eliminate  $E$  from (A.11c). Differentiating (A.15) with respect to  $t$  yields

$$E_t = \frac{1}{2}u^2 \rho_t + \rho u u_t + \frac{1}{\gamma - 1} p_t.$$

Substituting (A.12) and (A.14) in this expression for  $\rho_t$  and  $u_t$  produces

$$E_t = -\frac{1}{2}u^3\rho_x - \frac{3}{2}\rho u^2u_x - up_x + \frac{1}{\gamma-1}p_t. \quad (\text{A.16})$$

By expanding  $((E+p)u)_x$  with the equation of state, we have

$$\begin{aligned} ((E+p)u)_x &= \left( \frac{1}{2}u^3\rho + \frac{\gamma}{\gamma-1}pu \right)_x \\ &= \frac{1}{2}u^3\rho_x + \frac{3}{2}\rho u^2u_x + \frac{\gamma}{\gamma-1}pu_x + \frac{\gamma}{\gamma-1}up_x \end{aligned} \quad (\text{A.17})$$

By combining (A.16) and (A.17), 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

$$p_t + \gamma pu_x + up_x = 0. \quad (\text{A.18})$$

We now can express (A.12), (A.14) and (A.18) 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 are  $\{u, u+c, u-c\}$  where  $c$  is the sound speed.

**14.7.** The solution starts with a shock beginning at  $x = 1$  and 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 leading edge position, 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  $x = 2$ . So  $c = 0$ . Altogether,

$$\begin{aligned} \text{when } t < 2 \quad u(x,t) &= \begin{cases} x/t, & 0 < x < t \\ 1, & t < x < 1 + \frac{1}{2}t \\ 0, & \text{otherwise} \end{cases} \\ \text{when } t \geq 2 \quad u(x,t) &= \begin{cases} x/t, & 0 < x < \sqrt{2t} \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

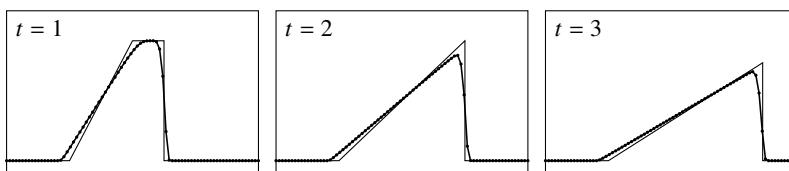
because the area is conserved. We can write the exact solution as the function:

```
U = (x,t) -> @. t<2 ?
    (0<x<t)*(x/t) + (t<x<1+t/2) : (x/t)*(0<x<sqrt(2t))
```

We can implement the local Lax–Friedrichs method 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])
    global 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 figure below and the QR code at the bottom of the page show the analytical and numerical solutions.



solutions to Burgers' equation



### 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

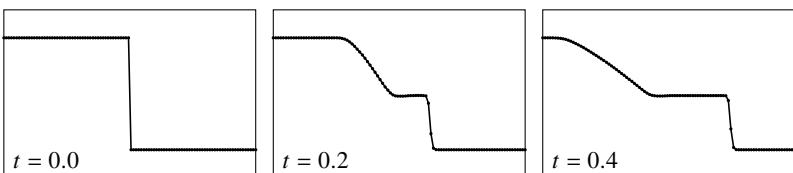
$$\phi(\theta) = \frac{|\theta| + \theta}{1 + |\theta|}.$$

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,:])
δx(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 - δ(∂x(U))/8 - Δt/Δx*δ(F(U°))
    U° = U-0.5*Δt/Δx*∂x(F(U))
    Uj .= (Uj+Uj+1)/2 - δ(∂x(U))/8 - Δt/Δx*δ(F(U°))
end
```

The figure and QR code below show the numerical solution for  $h(t, x)$ .



Notice the shock wave moving to the right and the rarefaction wave moving to the left from the initial discontinuity.



the dambreak problem

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

$$\sum_{j=0}^{m+1} v_j \sum_{i=0}^{m+1} \xi_i a(\varphi_j, \varphi_i) = \sum_{i=0}^{m+1} v_i L(\varphi_i).$$

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 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)
```

Figure A.20 shows the finite element solution and the exact solution. The finite element solution has little error even with only ten nodes.

15.2. Let  $V$  be the space of twice differentiable functions that take the same boundary conditions as  $u(x)$ . Multiply the equation  $u'''' = f$  by  $v \in V$  and integrate by parts twice to get a symmetric bilinear form  $\int_0^1 u''v'' \, dx = \int_0^1 fv \, dx$ . (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  $v_h \in V_h$

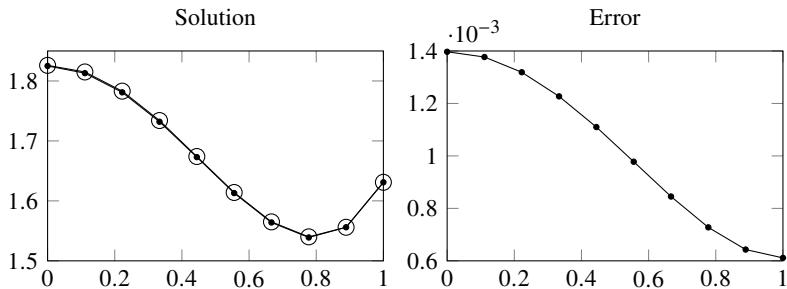


Figure A.20: Solutions to the Neumann problem 15.1. The marker  $\bullet$  is used for the finite element solution and  $\circ$  for the analytical solution.

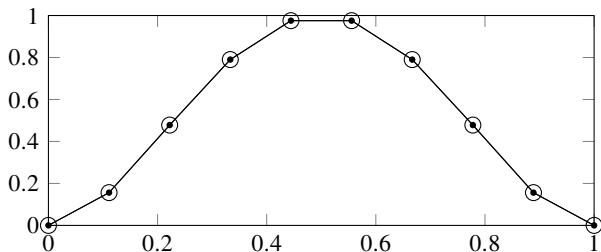


Figure A.21: Solutions to the beam problem. The marker  $\bullet$  is used for the finite element solution and  $\circ$  for the analytical solution.

The finite elements are

$$u_h(x) = \sum_{i=0}^{m+1} \xi_i \phi_i(x) + \eta_i \psi_i(x) \text{ and } v_h(x) = \sum_{j=0}^{m+1} \alpha_j \phi_j(x) + \beta_j \psi_j(x),$$

where the basis elements  $\phi_j(x)$  prescribe the value of  $u(x)$  at the nodes, and the basis elements  $\psi_j(x)$  prescribe the value of the derivative of  $u(x)$  at the nodes. The coefficients  $\xi_j$  and  $\eta_j$  are unknown, and the coefficients  $\alpha_j$  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

$$a\left(\sum_{i=1}^m \xi_i \phi_i + \eta_i \psi_i, \phi_j\right) = L(\phi_j) \text{ and } a\left(\sum_{i=1}^m \xi_i \phi_i + \eta_i \psi_i, \psi_j\right) = L(\psi_j)$$

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 can write the system of  $2m$  equations as the block matrix

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^\top & \mathbf{C} \end{bmatrix} \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{\eta} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \end{bmatrix},$$

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

```
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)
```

Figure A.21 shows the solution using 8 interior nodes. Because the problem has a constant load, the finite element solution matches the analytical solution exactly.

16.2. We can solve Burgers' equation as  $u_t = -\frac{1}{2} F^{-1} [i\xi F(u^2)]$  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());
```

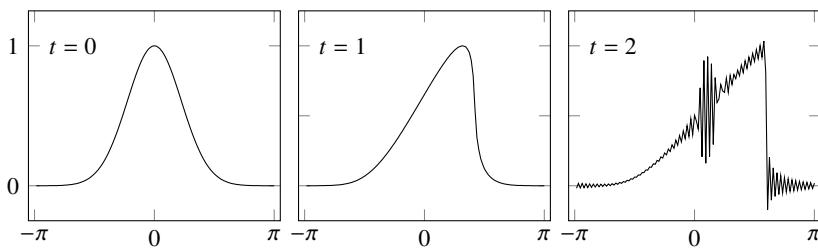


Figure A.22: Solutions to Burgers' equation in exercise 16.2 at  $t = 0, 1$ , and  $2$ . Gibbs oscillations overwhelm the solution after a discontinuity develops.

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
```

Figure A.22 and the QR code on the bottom of this page show the solution. 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. As a result, truncation error is half-order in space and time once a discontinuity develops. This is bad—and it gets worse. Gibbs oscillations develop around the discontinuity. These oscillations will spread and grow because Burgers' equation is dispersive. Ultimately, the oscillations overwhelm the solution.

**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) )
```



Fourier spectral  
solution to Burgers'  
equation

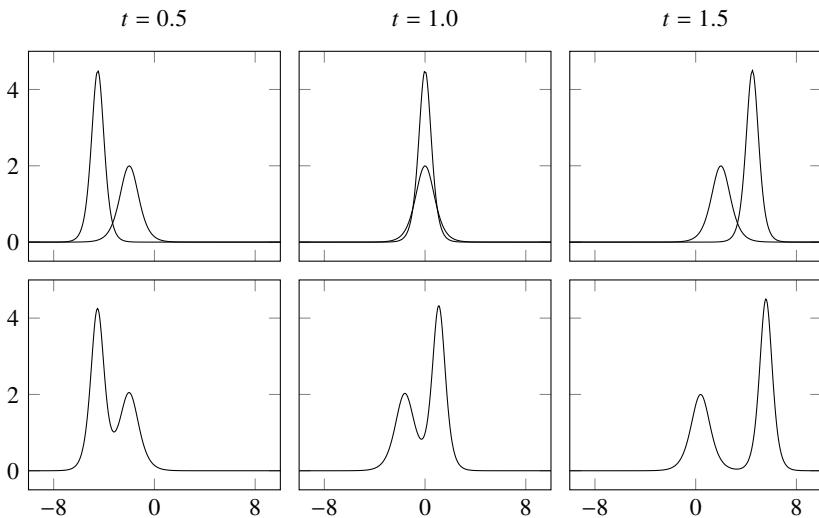


Figure A.23: Solutions of the KdV equation in exercise 16.3. The top row shows independent solitons. The bottom row shows the interacting solitons.

Then, we solve the problem.

```
w = solve(ODEProblem(f,F*u₀,(0,T)),DP5())
u = t -> real(F\w(t).*G(t))
```

Finally, we animate the solution.

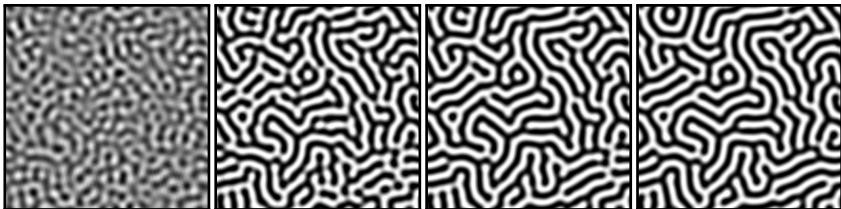
```
@gif for t=0:.01:2
    plot(x,u(t),ylim=(0,5),legend=:none)
end
```

Figure A.23 shows the soliton solution to the KdV equation. 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 different initial conditions. The bottom row shows the two-soliton solution for  $u(x, 0) = \phi(x; -4, 4) + \phi(x; -9, 9)$ . After colliding, the tall, fast soliton in the bottom row is about 1 unit in front of the corresponding soliton in the top row. The small, 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 the collision as it had before the collision—its position is merely shifted.

**16.4.** Let's use Strang splitting on  $u' = N u + L u$ . There are two reasonable ways to define  $N$  and  $L$ . The first with  $N u = \varepsilon u - u^3$  and  $L u = -(\Delta + 1)^2 u$ , and the second with  $N u = -u^3$  and  $L u = \varepsilon u - (\Delta + 1)^2 u$ . In the first case, the

soliton interaction in  
the KdV equation



Figure A.24: Solution to the Swift–Hohenberg equation at  $t = 3, 10, 25$ , and  $100$ .

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 equilibria at  $u = \pm\sqrt{\varepsilon}$  and 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  $u' = N u$  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

$$E \hat{u}_0 = e^{-(\hat{\Delta}+1)^2\Delta t} \hat{u}_0.$$

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.24 and the QR code on 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))
```



the Swift–Hohenberg  
equation

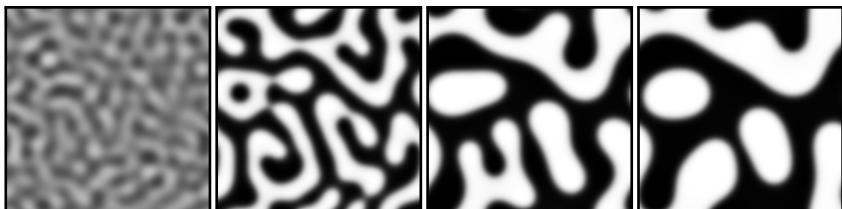


Figure A.25: Solution to the Cahn–Hilliard equation at  $t = 0.1, 1, 4$ , and  $8$ .

```
for i=1:600
    U = f(F\(\mathbf{E} \cdot (\mathbf{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)))
```

Making a call to ffmpeg produces a movie.

```
run(`ffmpeg -i "temp%03d.png" -pix_fmt yuv420p swifthohenberg.mp4`)
```

## 16.5. We'll use a fourth-order ETDRK scheme.

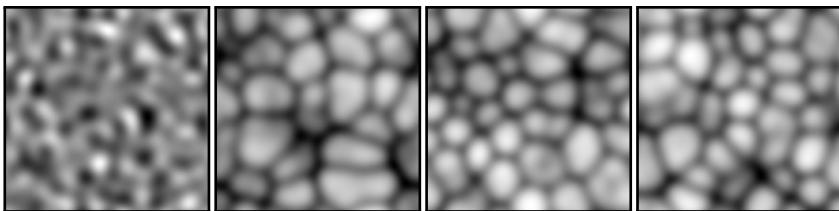
```
using FFTW, DifferentialEquations, DiffEqOperators
ℓ = 8; T = 8.0; m = 256; ε² = 0.01; α = 1
ξ = [0:(m÷2);(-m÷2+1):-1]*2π/ℓ
Δ = -(ξ.^2 .+ ξ'.^2); Δ² = Δ.^2
u₀ = randn(m,m)
F = plan_fft(u₀)
L = DiffEqArrayOperator(Diagonal((-ε²*Δ²+α*Δ)[:]))
N = (u,_,_) -> (v = F\reshape(u,m,m); Δ.*(F*(v.^3-(1+α)*v))[:])
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
```

Figure A.25 and the QR code on the current page show the solution to the Cahn–Hilliard equation. Observe how the dynamics slow down over time.



Figure A.26: The Kuramoto–Sivashinsky equation at  $t = 1, 50, 100$ , and  $150$ .

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; u₀ = randn(n,n)
x = LinRange(-ℓ/2, ℓ/2, n+1)[2:end]
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 evolution of the solution using the `Interact.jl` library. See Figure A.26 and the QR code on this page.

```
using Interact, ColorSchemes, Images
@manipulate for t in slider(0:0.5:T; value=0, label="time")
    get(colorschemes[:magma], -u(t), :extrema)
end
```



the two-dimensional  
KSE

## 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 (a 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 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 TeX, 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 the 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 an 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 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 :: sgessv
    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 sgessv(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

---

<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, computer memory and printing limitations made uppercase a necessity. In the 1990s, “FORTRAN,” on which MATLAB was based and styled, adopted the naming “Fortran,” and “MATLAB” stayed “MATLAB.”

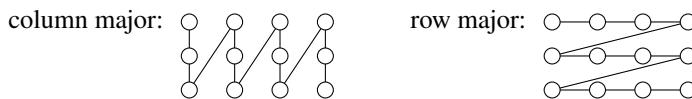
a free, open-source clone of MATLAB. While the syntax of Octave is nearly identical to MATLAB, there are a few differences. In general, Octave allows a little more freedom in syntax than MATLAB, while MATLAB tends to have more and newer features and functions. For example, Octave permits C-style increment 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 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 in 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 forward 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 faster, cached compiled version. Julia has arbitrary precision arithmetic (arbitrary-precision integers and

---

<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 some 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 a minute or so the first run and around a second after that. Julia continues to improve and compile-time latency tends to decrease with each version.

floating-point numbers using GNU Multiple Precision Arithmetic Library (GMP) and the GNU MPFR Library) with BigInt and BigFloat.

## B.2 Python

This section provides a Python supplement to the Julia commentary and code in the body of this book. The code is written and tested using Python version 3.9.7. 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 sps
```

We'll also define the following variables:

```
bucket = "https://raw.githubusercontent.com/nmfsc/data/master/"
π = np.pi
```

Finally, we'll define a few helper functions for downloading and displaying images:

```
import PIL, requests, io
def rgb2gray(rgb): return np.dot(rgb[...,:3], [0.2989,0.5870,0.1140])
def getimage(url):
    response = requests.get(url)
    return np.asarray(PIL.Image.open(io.BytesIO(response.content)))
def showimage(img):
    display(PIL.Image.fromarray(np.int8(img.clip(0,255)), mode='L'))
```

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 code 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 code at the bottom of this page.

Define the array `x = np.arange(n)`. There are several different ways to create a 5 column vector using NumPy:

Python notebook  
on Binder



Python notebook  
on Google Colab



```
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()
```

- 23 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

```

The following Python code implements the simplex method. We start by defining 42 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.argwhere(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)

```

The `scipy.optimize` function `linprog` solves the linear programming problem 45 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.

The `scipy.sparse.linalg` function `spsolve` uses the `scikit-umfpack` wrapper of 46 UMFPACK to solve sparse linear systems.

- 46 We'll construct a sparse, random matrix, use the `nnz` method to get the number of nonzeros, and the `matplotlib.pyplot` function `spy` to draw the sparsity plot.

```
A = sps.rand(60,80,density=0.2)
print(A.nnz), plt.spy(A,markersize=1)
```

- 49 The NetworkX package provides tools for constructing, analyzing, and visualizing graphs.
- 49 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.

```
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=sps.lil_matrix((m,1)); z[i] = 1; return z.tocsr()
    N = np.arange(n); B = np.arange(n,n+m)
    A = sps.hstack([sps.csr_matrix(A),sps.identity(m)],format="csr")
    ABinv = sps.identity(m).tocsr()
    b = sps.csr_matrix(b)
    c = sps.vstack([sps.csr_matrix(c),sps.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)

```

For a  $2000 \times 1999$  matrix `la.lstsq(A1,b)` takes 3.4 seconds and `la.pinv(A1)@b` 57 takes 6.3 seconds.

Python does not have a dedicated function for the Givens rotations matrix. Instead, 64 use the `scipy.linalg QR` decomposition `Q, _ = qr([[x],[y]])`.

The `scipy.linalg` function `qr` implements LAPACK routines to compute the QR 65 factorization of a matrix using Householder reflection.

The `numpy.linalg` and `scipy.linalg` function `lstsq` solves an overdetermined 66 system using the LAPACK routines `gelsd` (using singular value decomposition) `gelsy` (using QR factorization), or `gelss` (singular value decomposition).

The Zipf's law coefficients `c` for the canon of Sherlock Holmes computed using 67 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]

```

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]]

```

The `numpy.linalg` or `scipy.linalg` command `U,S,V=svd(A)` returns the SVD de- 71 composition of a matrix `A` and `la.svd(A,0)` returns the “economy” version of the SVD. The `scipy.sparse.linalg` command `svds(A,k)` returns the first `k` singular values and associated singular vectors.

The NumPy function `la.pinv` computes the Moore–Penrose pseudoinverse by 73 computing the SVD using a default tolerance of `1e-15*norm(A)`.

The following code implements total least squares:

```

def tls(A,B):
    n = A.shape[1]
    _,_,VT = la.svd(np.c_[A,B])

```

```
    return -la.solve(V^T[:,n:],V^T[:,,:n]).T
```

77 A = np.array([[2,4],[1,-1],[3,1],[4,-8]]);  
b = np.array([[1,1,4,1]]).T  
x\_ols = la.lstsq(A,b)[0]  
x\_tls = tls(A,b)

79 We'll use the helper functions `getimage` and `showimage` defined on page 549.

```
A = rgb2gray(getimage(bucket+'red-fox.jpg'))  

U,s,V^T = 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  $\sum_{i=k+1}^n \sigma^2$ .

```
A_k = U[:, :k] @ np.diag(s[:k]) @ V^T[:, :]  

la.norm(A-A_k,'fro') - la.norm(s[k:])
```

Finally, we'll show the compressed image and plot the error curve.

```
showimage(np.c_[A,A_k])  

r = np.sum(A.shape)/np.prod(A.shape)*range(1,min(A.shape)+1)  

error = 1 - np.sqrt(np.cumsum(s**2))/la.norm(s)  

plt.semilogx(r,error,'.-'); plt.show()
```

85 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)
```

- 87 The NumPy function `vander` generates a Vandermonde matrix with rows given by  $[x_i^p \quad x_i^{p-1} \quad \cdots \quad x_i \quad 1]$  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. Instead, we can compute it 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)
```

The code to compute the PageRank of the graph in Figure 4.3 is

97

```
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
```

The `scipy.linalg` function `hessenberg` computes the unitarily similar upper Hessenberg form of a matrix.

Both `numpy.linalg` and `scipy.linalg` have the function `eig` that computes the eigenvalues and eigenvectors of a matrix.

The `scipy.sparse.linalg` command `eigs` computes several eigenvalues of a sparse matrix using the implicitly restarted Arnoldi process.

The `scipy.sparse.linalg` function `cg` implements the conjugate gradient method— preconditioned conjugate gradient method if a preconditioner is also provided.

The `scipy.sparse.linalg` function `gmres` implements the generalized minimum residual method and `minres` implements the minimum residual method.

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

149

```
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(fft2(np.conj(y)))/len(y)
```

- 150 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.
- 153 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 `fft2` and `ifft2` 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 ifft2(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 discrete sine and discrete cosine transforms. The PyFFTW package provides a Python wrapper to the FFTW routines. According to the SciPy documentation: “users for whom the speed of FFT routines is critical should consider installing PyFFTW.”
- 159 The `scipy.fft` functions `fftshift` and `ifftshift` shift the zero frequency component to the center of the array.
- 160 The following code solves the Poisson equation using a naive fast Poisson solver and then compares the solution with the exact solution.

```
from scipy.fft import dstn, idstn
```

```

n = 50; x = np.arange(1,n+1)/(n+1); Δx = 1/(n+1)
v = 2 - 2*np.cos(x*π)
λ = np.array([[[ (v[i]+v[j]+v[k])/Δx**2 \
    for i in range(n)] for j in range(n)] for k in range(n)])
f = np.array([[[ (x-x**2)*(y-y**2) + \
    (x-x**2)*(z-z**2)+(y-y**2)*(z-z**2) \
    for x in x] for y in x] for z in x])
u = idstn(dstn(f,type=1)/λ,type=1)
U = np.array([[[ (x-x**2)*(y-y**2)*(z-z**2)/2 \
    for x in x] for y in x] for z in x])
la.norm(u - U)

```

The `scipy.fft` functions `dct` and `dst` return the type 1–4 DCT and DST, respectively. 163  
The following function returns a sparse matrix of Fourier coefficients along with  
the reconstructed compressed image:

```

from scipy.fft import dctn, idctn
def dctcompress(A,d):
    B = dctn(A)
    idx = int(d*np.prod(A.size))
    tol = np.sort(abs(B.flatten()))[-idx]
    B[abs(B)<tol] = 0
    return sps.csr_matrix(B), idctn(B)

```

We'll test the function on an image and compare it before and after using `getimage`  
and `showimage` defined on page 549.

```

A = rgb2gray(getimage(bucket+"red-fox.jpg"))
_, A0 = dctcompress(A,0.001)
showimage(np.c_[A,A0])

```

The function `type` returns the data type of a variable. 180

The following function returns a double-precision floating-point representation 181  
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)

```

The NumPy function `finfo` returns machine limits for floating-point types. For 181  
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.

- 182 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))
```

- 184  $a = 77617$ ;  $b = 33096$   

$$333.75 \cdot b^{**6} + a^{**2} \cdot (11 \cdot a^{**2} \cdot b^{**2} - b^{**6} - 121 \cdot b^{**4} - 2) + 5.5 \cdot b^{**8} + a / (2 \cdot b)$$

- 185 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.
- 186 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()
```

- 187 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.
- 191 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
```

- 197 The `scipy.optimize` function `fsolve(f,x0)` uses Powell's hybrid method to find the zero of the input function.
- 204 The following function takes the parameters `bb` for the lower-left and upper-right corners of the bounding box, `xn` for the number of horizontal pixels, and `n` for the maximum number of iterations. 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 escape(n,z,c):
    M = np.zeros_like(c,dtype=int)
    for k in range(n):
```

```

mask = np.abs(z)<2
M[mask] += 1
z[mask] = z[mask]**2 + c[mask]
return M

```

```

def mandelbrot(bb, xn=800, n=200, z=0):
    yn = int(np.round(xn*(bb[3]-bb[1])/(bb[2]-bb[0])))
    z = z*np.ones((yn,xn),dtype=complex)
    c = np.linspace(bb[0],bb[2],xn).reshape(1,-1) +
        1j*np.linspace(bb[3],bb[1],yn).reshape(-1,1)
    return escape(n,z,c)

```

The following commands produce image (c) of Figure 8.5 on page 204:

```

import matplotlib.image as mpimg
M = mandelbrot([-0.1710,1.0228,-0.1494,1.0443])
mpimg.imsave('mandelbrot.png', -M, cmap='magma')

```

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.

The NumPy function `roots` returns the roots of a polynomial by finding the eigenvalues of the companion matrix.

The following code finds the roots using homotopy continuation:

217

```

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]

```

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)
```

- 227 The NumPy function `vander` constructs a Vandermonde matrix. Or you can build one yourself with `x**np.arange(n)`.
- 238 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 a 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.argmin(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)
```

- 239 The `scipy.interpolate` function `spline` returns a cubic (or any other order) spline. The `splprep` and `splev` commands break up the steps of finding the coefficients for an interpolating spline and evaluating the values of that spline. Perhaps the easiest approach is the function `CubicSpline` that returns a cubic spline interpolant class. The `Spline` classes in `scipy.interpolate` are wrappers for the Dierckx Fortran library.

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  $c$ .

The `scipy.interpolate` function `pchip` returns a cubic Hermite spline.

242

We can build a Bernstein matrix using the following function which takes a column vector such as  $t = np.linspace(0,1,50).[:,None]$ :

```
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)
```

```
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)
```

255

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');
```

- 263 The ChebPy package, a Python implementation of the Matlab Chebfun package, includes methods for manipulating functions in Chebyshev space.
- 269 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] = 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()
```

- 271
- ```
psi = np.zeros_like(phi); n = len(c)-1; L = len(phi)//(2*n)
for k in range(n):
    psi[k*L:(k+n)*L] += (-1)**k*c[n-k]*phi[::2]
```
- 273 The scipy.signals library includes several utilities for wavelet transforms. The PyWavelets (pywt) package provides a comprehensive suite of utilities.
- 274 We'll use the PyWavelets package. The helper functions getimage and showimage are defined on page 549. The following code gives the two-dimensional Haar DWT of a grayscale image:

```
import pywt
def adjustlevels(x): return 1-np.clip(np.sqrt(np.abs(x)),0,1)
A = rgb2gray(getimage(bucket+"laura_square.png"))/255
A = pywt.wavedec2(A,'haar')
c, slices = pywt.coeffs_to_array(A)
showimage(adjustlevels(c)*255)
```

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')
showimage(B*255)
```

We first need a numerically approximation for the Jacobian.

283

```
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 implement 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)
if c is not None: plt.plot(x,y,'.',X,f(X,c))
```

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)
```

We'll first define the logistic function and generate synthetic data. Then, we apply Newton's method.

286

```
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 do logistic regression:

```
import statsmodels.api as sm
results = sm.Logit(y, X).fit()
w = results.params
```

The command `results.summary()` gives a summary of the results.

- 289 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
```

After determining the parameters `W1` and `W2`, we can plot the results.

```
plt.scatter(x[1,:],y,color='#ff000050')
x̂ = np.linspace(-1.2,1.2,200); x̂ = np.c_[np.ones_like(x̂),x̂].T
ŷ = W2 @ φ(W1@x̂)
plt.plot(x̂[1,:],ŷ[0],color='#000000')
```

Alternatively, we can swap out the activation and loss functions.

```
def ϕ(x): return 1/(1+np.exp(-x))
def dϕ(x): return ϕ(x)*(1-ϕ(x))
```

```
L = la.norm(ŷ-y)
dLdy = 2*(ŷ-y)/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 and 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;
θ = tf.linspace(0.0,π,N)
x = tf.math.cos(θ); y = tf.math.sin(θ) + 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)
ŷ = model.predict(x)
plt.plot(x,ŷ,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)
```

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  $C @ d^{**n} / np.math.factorial(n)$ .

- 297 The fractions package allows 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)`.
- 298 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)
```

- 301 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)
```

We can define the dual numbers as

302

```

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)

```

Python has several packages for automatic differentiation, including dedicated 303 libraries like JAX and machine learning frameworks like TensorFlow and PyTorch

We can use the following trapezoidal quadrature to make a Romberg method 306 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

```

The following code examines the convergence rate of the composite trapezoidal 307 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={:.1f}'.format(k+1,s[k]) for k in range(7)]
lines = plt.loglog(n,error)
plt.legend(lines, info); plt.show()

```

```
310 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] )
```

```
311 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] )
```

316 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 the Jacobi matrix, followed by one Newton step to refine the calculation.

```
nodes, weights = np.polynomial.legendre.leggauss(n)
np.dot(f(nodes), weights)
```

325 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.

```
r = np.exp(2j*np.pi*np.linspace(0,1,100)) 345
z = (3/2*r**2 - 2*r + 0.5)/r**2
plt.plot(z.real,z.imag); plt.axis('equal'); plt.show()
```

The following function determines the coefficients for stencil given by  $m$  and  $n$ : 347

```
def multistepcoefficients(m,n):
    s = len(m) + len(n) - 1
    A = (np.array(m)+1)**np.c_[range(s)]
    B = [[i*((j+1)**max(0,i-1)) for j in n] for i in range(s)]
    c = la.solve(-np.c_[A[:,1:],B],np.ones((s,1))).flatten()
    return ( np.r_[1,c[:len(m)-1]], c[len(m)-1:] )
```

We'll use the arrays returned by `multistepcoefficients` to plot the region of stability.

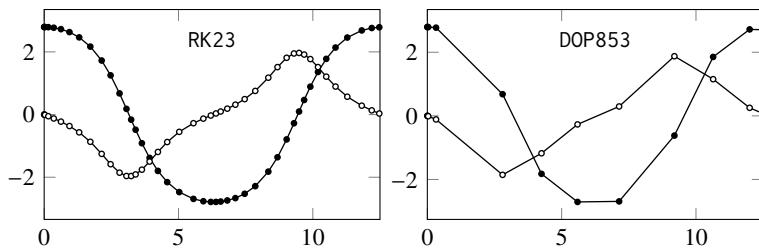
```
def plotstability(a,b):
    r = np.exp(1j*np.linspace(0,2*pi,200))
    z = [np.dot(a,r**np.arange(len(a))) / \
          np.dot(b, r**np.arange(len(b))) for r in r]
    plt.plot(np.real(z),np.imag(z)); plt.axis('equal')
```

The Python recipe for solving a differential equation is quite similar to Julia's 373 recipe, except 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=[' $\theta$ ', ' $\omega$ ']); plt.show()
```

- 384 We can use `numpy.linalg.solve_banded` as a tridiagonal solver.

```
 $\Delta x = .01; \Delta t = .01; L = 2; c = \Delta t / \Delta x^{**2}; u_L = 0; u_R = 0;$ 
x = np.arange(-L,L,Delta x); 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.

- 384 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 dividing in Fourier

space). The `scipy.sparse.linalg` function `spsolve` can be used to solve general sparse systems.

The following code solves the heat equation using the Crank–Nicolson method. 385 While we can use `numpy.dot` and `numpy.linalg.solve_banded` for tridiagonal multiplication and inverse, let's use Python's sparse matrix routines instead.

```
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 = sps.diags(diagonals, [-1,0,1], format='csr')
D[0,1] *= 2; D[-1,-2] *= 2
A = 2*sps.identity(n) + ν*D
B = 2*sps.identity(n) - ν*D
for i in range(20):
    u = spsolve(B,A@u)
plt.plot(x,u); plt.show()
```

We'll use the LSODA solver, which automatically switches between Adams– 394 Moulton and BDF routines for stiff nonlinear problems. Using the `ipywidgets` package, we can plot the results in a Jupyter notebook as an interactive animation.

```
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))
```

PyClaw is a Python-based interface to Clawpack (Conservation Laws package), 429 a collection of finite volume methods for hyperbolic systems of conservation laws. The collection was initially 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.

- 443 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
```

- 450 The numpy.fft function `fftfreq` returns the DFT sample frequencies.

```
iξ = 1j*np.fft.fftfreq(n,1/n)*(2*π/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
ξ² = (fftfreq(n,1/n)*(2*π/L))**2
def u(t,u₀): return np.real(ifft(np.exp(-ξ²*t)*fft(u₀)))
```

- 461 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  $\hat{\mathbf{H}}$  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,iξx,iξy): return fft2(ifft2(u)*ifft2(iξx*u) + \
    ifft2(v)*ifft2(iξy*u))
```

The variable initialization is designed on equal  $x$  and  $y$  dimensions, but we can easily modify the code for arbitrary domain size.

```
ℓ, n, Δt, ε = 2, 128, 0.001, 0.001; Δx = ℓ/n
x = np.linspace(Δx, ℓ, n)[None,:,:]; y = x.T
q = 0.5*(1+np.tanh(10*(1-np.abs(ℓ/2 - y)/(ℓ/4))))
Q = np.tile(q, (1,n))
u = Q*(1+0.5*np.sin(ℓ*π*x))
```

```
v = np.zeros_like(u)
u,v = fft2(u),fft2(v)
us,vs = u,v
i\xi x = (1j*fftfreq(n)*n*(2*pi/l))[None,:]
i\xi y = i\xi x.T
\xi 2 = i\xi x**2+i\xi y**2
Hx, Hy = H(u,v,i\xi x,i\xi y), H(v,u,i\xi y,i\xi x)
M1 = 1/Dt + (epsilon/2)*\xi 2
M2 = 1/Dt - (epsilon/2)*\xi 2
```

Finally, we iterate on (16.14) and the Lax–Wendroff solver for (16.15).

```
for i in range(1200):
    Q -= flux(Q, (Dt/Dx)*np.real(ifft2(v))) + \
        flux(Q.T, (Dt/Dx)*np.real(ifft2(u)).T).T
    Hxo, Hyo = Hx, Hy
    Hx, Hy = H(u,v,i\xi x,i\xi y), H(v,u,i\xi y,i\xi 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\xi x*phi, vs - i\xi y*phi
plt.imshow(Q,'seismic'); plt.show()
```

```
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()
```

468

An easy way to input  $\mathbf{D}_n$  in Python is with the `diag` function

469

```
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`.

```
470 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))
```

- 471 The following function implements a naïve reverse Cuthill–McKee algorithm for symmetric matrices:

```
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 = sps.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 = sps.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()
```

- 471 We'll reuse the code on page 552.

```
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()
```

The following code solves the Stigler diet problem using the naïve simplex 471 algorithm developed on page 42.

```
import pandas as pd
diet = pd.read_csv(bucket+'diet.csv')
A = diet.values[1:,3:1].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)]
```

Let's first define a couple of helper functions to import the data.

473

```
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 sps.csr_matrix((np.ones_like(i[:,0]), (i[:,0]-1,i[:,1]-1)))
```

We use the `scipy.sparse` command `bmat` to form a 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 = sps.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 = sps.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.

- 474 Python formats indices and index pointers in CSR matrices using four-byte integers instead of eight-byte integers as Julia and Matlab do. Storing double-precision floating-point numbers will need  $3dmn + m + 1$  four-byte blocks to store a CSR array and  $2mn$  blocks for full array in memory—a break-even point of about 0.66. Similarly, double-precision floating-point numbers will need  $5dmn + m + 1$  four-byte blocks for the sparse matrix versus  $4mn$  blocks for the full matrix—a break-even point of around 0.8.

```

d = 2/3; A = sps.rand(60,80,format='csr',density=d)
 nbytes = A.data.nbytes + A.indptr.nbytes + A.indices.nbytes
 nbytes, 4*(3*d*np.prod(A.shape) + A.shape[0] + 1)

```

- 474 `X = rgb2gray(getimage(bucket+"laura.png"))`  
`m,n = X.shape`  
`def blur(x): return np.exp(-(x/20)**2/2)`  
`A = [[blur(i-j) for i in range(m)] for j in range(m)]`  
`A /= np.sum(A, axis=1)`  
`B = [[blur(i-j) for i in range(n)] for j in range(n)]`  
`B /= np.sum(B, axis=0)`  
`N = 0.01*np.random.rand(m,n)`  
`Y = A@X@B + N`

```

α = 0.05
X1 = la.lstsq(A,Y)[0]
X2 = la.solve(A.T@A+α**2*np.eye(m),A.T@Y).T
X2 = la.solve(B@B.T+α**2*np.eye(n),B@X2).T
X3 = la.pinv(A,α) @ Y @ la.pinv(B,α)
showimage(np.c_[X,Y,X1,X2,X3])

```

The Python code for exercise 3.5 follows:

475

```
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 evaluates 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])

```

- 476 We'll use pandas to download CSV file. Pandas is built on top of NumPy, so converting from a pandas data frame 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*pi*t),\
    np.cos(2*pi*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()

```

- 477 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
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 = sps.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."

```

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 method's accuracy.

```

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 data frame.

```

import pandas as pd
pd.DataFrame(confusion)

```

We use singular value decomposition to find a low-dimensional subspace relating 478 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 575.

```

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]]

```

```

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]

```

```

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()

```

```
481 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)
```

- 481 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)
```

- 482 We'll use the helper functions `get_names` and `get_adjacency_matrix` from page 575.

```
actors = get_names("actors")
B = get_adjacency_matrix("actor-movie")
r,c = (B.T@B).nonzero()
M = sps.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]]
```

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: 483

```
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.

```
A = rgb2gray(getimage(bucket+'red-fox.jpg'))
U, S, VT = randomizedsvd(A,10);
img = np.c_[A, np.minimum(np.maximum((U*S)@VT,0),255)]
showimage(img)
```

```
from scipy.sparse.linalg import svds
m = 5000
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]
```

The following functions solve the Poisson equation: 485

```
n = 50; xi = np.arange(1,n+1)/(n+1); dx = 1/(n+1)
I = sps.identity(n)
D = sps.diags([-1, 2, -1], [0, 1, 0], shape=(n, n))
A = ( sps.kron(sps.kron(D,I),I) + sps.kron(I,sps.kron(D,I)) +
    sps.kron(I,sps.kron(I,D)) )/dx**2
f = np.array([(x-dx**2)*(y-dx**2) + (x-dx**2)*(z-dx**2)+(y-dx**2)*(z-dx**2)
    for x in xi for y in xi for z in xi])
ue = np.array([(x-dx**2)*(y-dx**2)*(z-dx**2)/2
    for x in xi for y in xi for z in xi])
```

```
from scipy.sparse.linalg import spsolve
def stationary(A,b,ω=0,n=400):
    ε = []; u = np.zeros_like(b)
    P = sps.diags(A.diagonal(),0) + ω*sps.tril(A,-1)
    for i in range(n):
        u += spsolve(P,b-A@u,'NATURAL')
        ε.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.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 to complete 400 iterations. In contrast, the conjugate gradient method takes around 1.2 seconds to complete the same number of iterations—and 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.

- 486 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).

```

from scipy.sparse.linalg import cg
n = 20000
d = 2 ** np.arange(15); d = np.r_[ -d, d]
P = sps.diags(np.loadtxt(bucket+"primes.csv"))
B = [np.ones(n - abs(d)) for d in d]
A = P + sps.diags(B, d)
b = np.zeros(n); b[0] = 1
cg(A, b, M=P)[0][0]

```

Python code for the radix-3 FFT in exercise 6.1 is

487

```

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**(2*k) * fftx3(c[2::3])))
        F = np.exp(-2j*π/3)*np.array([[0,0,0],[0,1,2],[0,2,4]])
        return (F @ u).flatten()
    else:
        k = np.arange(n)[:,None]
        F = ω**(k*k.T);
        return F @ c

```

Python code for exercise 6.2 is as follows

488

```

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 to ensure that the zero-padded array has an 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  $pq = \text{convolve}(p, q)$ . Note that Python uses arbitrary-precision integers, so we can simply multiply the numbers directly. Python uses the recursive Karatsuba algorithm to multiply integers, which is significantly faster than grade school multiplication but still slower than the Schönhage–Strassen algorithm for larger numbers.

- 489 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]
    ω = np.exp(-0.5j * π * np.arange(n) / n).reshape(-1, 1)
    i = [*range(0, n, 2), *range(n - 1 - n % 2, 0, -2)]
    return np.real(ω * 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)
```

- 490 The following function returns a sparse matrix of Fourier coefficients along with the reconstructed compressed image:

```
from scipy.fft import dctn, idctn
def dctcompress2(A, d):
    n = A.shape
    n0 = tuple(int(np.sqrt(d) * i) for i in A.shape)
    B = dctn(A)[:n0[0], :n0[1]]
    return B, idctn(B, s=n)
```

```
def aitken(x1,x2,x3):
    return x3-(x3-x2)**2/(x3-2*x2+x1), (x1*x3-x2**2)/(x3-2*x2+x1)
n = 50000
p = np.cumsum([(-1)**i*i**4/(2*i+1) for i in range(n)])
p1,p2 = aitken(p[:n-2],p[1:n-1],p[2:n])
plt.loglog(abs(pi-p)); plt.loglog(abs(pi-p2)); plt.loglog(abs(pi-p1))
```

494

We'll find the solution to the system of equations in exercise 8.12. 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
```

We'll write a function that computes point addition and point doubling.

497

```
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)
nmP = dbl_add(n,mP)
```

- 500 The following function modifies `spline_natural` on page 560 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])
    alpha = h[:-1]
    beta = h + np.r_[h[-1],h[:-1]]
    C = np.diag(beta)+np.diag(alpha,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  $nx*n$  points through a set of  $n$  random points using the function `evaluate_spline` defined on page 560:

```
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()
```

- 501 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((-5,1.5)); plt.show()
```

We define a function to solve linear boundary value problems.

503

```
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))

```

- 506 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*np.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*xi)**derivative*fft(f(x))))
    plt.plot(x,u,color='k'); plt.show()
interactive(anim, derivative = (0,2,0.01))

```

- 507 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 a two percent error rate. With more epochs, it has about a one percent error rate.

```
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]]
```

509

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 565.

```
509 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]
```

- 510 We'll extend the Dual class on page 566 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(delta) > epsilon:
        fx = f(Dual(x))
        delta = value(fx)/deriv(fx)
        x -= delta
    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)))
delta = deriv(value(fx))/deriv(deriv(fx))
```

```
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)
```

The following function computes the nodes and weights for Gauss–Legendre quadrature by using Newton’s method to find the roots of  $P_n(x)$ : 511

```
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) )
```

```
ξ,w = np.polynomial.hermite.hermgauss(40)
def u0(x): return np.sin(x)
def u(t,x):
    return [np.dot(w,u0(x-2*np.sqrt(t)*ξ)/np.sqrt(π)) for x in x]
x = np.linspace(-12,12,100)
plt.plot(x,u(1,x)); plt.show()
```

Let’s define a general function 512

```
def mc_π(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 by looping over several values of  $n$ .

```
m = 20; error = []; N = 2**np.arange(20)
error = [sum(abs(π - mc_π(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()
```

- 514 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()
```

- 514  $i = \text{np.arange}(4)[\cdot, \text{None}]$   
 $\text{def factorial}(k): \text{return np.cumprod(np.r_[1,range(1,k)])}$   
 $c1 = \text{la.solve}(((\text{-}i)^{\star\star} i.\text{T}/\text{factorial}(4)).\text{T}, \text{np.array}([0,1,0,0]))$   
 $c2 = \text{la.solve}(((\text{-}(i+1))^{\star\star} i.\text{T}/\text{factorial}(4)).\text{T}, \text{np.array}([1,0,0,0]))$

- 516 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 multistepcoefficients defined on page 569.

```
def PECE(n,m):
    _,a = multistepcoefficients([0,1],range(1,n))
    _,b = multistepcoefficients([0,1],range(0,n))
    def c(r): return np.r_[r-1,\n        np.full(m, r + np.dot(b[1:],r**np.arange(1,n))/b[0]),\n        (a @ r**np.arange(1,n))/b[0]]
    return [np.roots(np.flip(c(r)))/b[0] \
        for r in np.exp(1j*np.linspace(0,2*\pi,200))]
```

```
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()
```

- 516 The `scipy.interpolate` function `pade(a,m,n)` computes a Padé approximant from Taylor polynomial coefficients. For 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 the 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
```

The solution to the SIR problem is

518

```
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.

The solution to the Duffing equation is

518

```
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()
```

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.

519

```
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.

- 522 The following code implements the Dufort–Frankel method to solve the heat equation. (As mentioned elsewhere, this approach is not recommended.)

```
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
B = c*sps.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))
```

- 523 The following code solves the Schrödinger equation:

```
from scipy.sparse.linalg import spsolve
def ψθ(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 = 2π/m; V = x**2/2
    ψ = ψθ(x,ε)
    D = 0.5j*ε*sps.diags([1, -2, 1], [-1, 0, 1], shape=(n, n))/dx**2 \
        - 1j/ε*sps.diags(V,0)
    D[0,1] *= 2; D[-1,-2] *= 2
    A = sps.eye(n) + (Δt/2)*D
    B = sps.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 = -ψθ(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()

```

We'll solve a radially symmetric heat equation. Although we divide by zero at  $r=0$  when constructing the Laplacian operator, we subsequently overwrite the resulting `inf` term when we apply the boundary condition.

```

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 = sps.diags([1, -2, 1], [-1, 0, 1], shape=(m,m))/Δr**2 \
    + sps.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 = sps.eye(m) - 0.5*Δt*D
B = sps.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()

```

We'll first define a function as a logit analog to `np.linspace`.

```

def loginspace(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.

```

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 sps.diags(2/d,[-1,0,1],shape=(n, n)).T

```

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 = sps.eye(len(x)) - 0.5*Δt*D
    B = sps.eye(len(x)) + 0.5*Δt*D
    for i in range(n):
        u = spsolve(A,B@u)
    return u

```

Finally, we compare the uniformly-spaced and logit-spaced solutions.

```

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 = loginspace(20,m,.999)
laplacian(x).toarray()
u = heat_equation(x,t,φ(x,t,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,t,10))
plt.plot(x,u,'.-',x,φ(x,t,10),'k'); plt.show()

```

527 Here's a solution to the Allen–Cahn equation using Strang splitting:

```

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 = sps.diags([1, -2, 1], [-1, 0, 1], shape=(m,m)).tocsr() / Δx**2
D[0,1] *= 2; D[-1,-2] *= 2;
A = sps.eye(m) + 0.5*Δt*D
B = sps.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@spsolve(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()
```

The following code solves Burgers' equation.

533

```
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]))
```

Let's first define a few functions.

534

```
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);
```

```
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,'.-');
```

535

```

537 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],'.-');

```

```

537 from scipy.integrate import solve_ivp
from scipy.fft import fftshift, fft, ifft
m = 128; x = np.linspace(-pi,pi,m,endpoint=False)
xi = 1j*fftshift(np.arange(-m/2,m/2))
def f(t,u): return -np.real(ifft(xi*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()

```

- 538 The following code solves the KdV equation using integrating factors. We first set the parameters.

```

from scipy.fft import fftshift, fft, ifft
def phi(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)
ixi = 1j*fftshift(np.arange(-m/2,m/2))*(2*pi/L)

```

Next, we define the integrating factor  $G$  and the right-hand side function  $f$ .

```

def G(t): return np.exp(-ixi**3*t)
def f(t,w): return -(3*ixi*ifft(ifft(G(t)*w)**2))/G(t)

```

Then we solve the problem using RK45.

```

from scipy.integrate import solve_ivp
u = phi(x,-4,4) + phi(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 displays 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 requires the ffmpeg video codecs).

```
from scipy.fft import fftshift, fft2, ifft2
epsilon = 1; m = 256; l = 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/l)
D2 = -xi[:,None]**2 - xi[None,:]**2
E = np.exp(-(D2+1)**2*Delta_t)
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()
```

540

### B.3 Matlab

This section provides a Matlab supplement to the Julia commentary and code elsewhere in the book. This book uses *Matlab* to describe the scientific programming language and syntax common to both MATLAB,<sup>4</sup> the proprietary computer software sold by MathWorks, and GNU Octave, the open-source computer software developed by John Eaton and others. The book explicitly uses *MATLAB* or *Octave* when referring to a particular implementation of the language.

The code is written and tested using Octave version 6.4.0. Page references to the Julia commentary are listed in the left margins. For brevity, the variable `bucket` is assumed throughout. The function `rgb2gray` is a standard function in MATLAB it is available in Octave in the image toolbox, or we can write it ourselves.

```
bucket = "https://raw.githubusercontent.com/nmfsc/data/master/";
rgb2gray = @(x) 0.2989*x(:,:,1) + 0.5870*x(:,:,2) + 0.1140*x(:,:,3);
```

The Matlab code in this section is available as a Jupyter notebook at

---

<sup>4</sup>MATLAB® is a registered trademark of The MathWorks, Inc., Python® is a registered trademark of the Python Software Foundation, Julia® is a registered trademark of Julia Computing, Inc., and GNU® is a registered trademark of Free Software Foundation.

<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. 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  $A = \text{rand}(3, 4)$ , then  $A - \text{mean}(A)$  will subtract the row vector  $\text{mean}(A)$  from each row of  $A$ . Similarly,  $A - \text{mean}(A, 2)$  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);
```



Octave notebook  
on Binder

```

end
for i=n:-1:1
    b(i) = ( b(i) - A(i,i+1:n)*b(i+1:n) )/A(i,i);
end
end

```

The following Matlab code implements the simplex method. We start by defining 42 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

```

The MATLAB function `linprog` solves the LP problem. The built-in Octave 45 function `glpk` (GNU Linear Programming Kit) can solve LP problems using either the revised simplex method or the interior point method.

Matlab's backslash and forward slash operators use UMFPACK to solve sparse 46 linear systems.

- 46 We'll construct a sparse, random matrix, get the number of zeros, and draw the sparsity plot.

```
A = sprand(60,80,0.2); nnz(A), spy(A, '.')
```

- 49 The function `gplot` plots the vertices and edges of an adjacency matrix. MATLAB, but not Octave, has additional functions for constructing and analyzing graphs.
- 49 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;
```

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. 52

The following code implements the revised simplex method. 54

```
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
```

For a  $2000 \times 1999$  matrix  $A \setminus b$  takes 3.8 seconds and  $\text{pinv}(A) \cdot b$  takes almost 56 57 seconds.

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 65 of a matrix using Householder reflection.

Matlab's `mldivide (\)` solves an overdetermined system using a QR solver. 66

- 67 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
```

- 70 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
```

- 64 The command  $[U,S,V]=\text{svd}(A)$  returns the SVD of a matrix  $A$  and  $\text{svd}(A,0)$  returns the “economy” version of the SVD. The command  $\text{svds}(A,k)$  returns the first  $k$  singular values and associated singular vectors.
- 73 The function  $\text{pinv}$  computes the Moore–Penrose pseudoinverse by computing the SVD using a default tolerance of  $\max(\text{size}(A)) * \text{norm}(A) * \text{eps}$ , 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  $A = [2 \ 4; 1 \ -1; 3 \ 1; 4 \ -8]; b = [1; 1; 4; 1];$   
 $x_{ols} = A\backslash b$   
 $x_{tls} = \text{tls}(A,b)$

- 79 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. Take  $k$  to be some nominal value like 20.

```
A = rgb2gray(imread(['bucket "laura.png']));  
[U, S, V] = svd(A,0);  
sigma = diag(S);
```

We can confirm that the error  $\|\mathbf{A} - \mathbf{A}_k\|_F^2$  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([A,Ak])
```

The 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 of a floating-point array between 0 and 1. 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,'.-');
```

The following function is a naïve implementation of NMF: 85

```
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
```

The function `vander` generates a Vandermonde matrix for input  $(x_0, x_1, \dots, x_n)$  87 with rows given by  $[x_i^p \quad x_i^{p-1} \quad \dots \quad x_i \quad 1]$ .

The function `roots` finds the roots of a polynomial  $p(x)$  by computing `eig` for 89 the companion matrix of  $p(x)$ .

The Matlab function `condeig` returns the eigenvalue condition number. 90

The following code computes the PageRank of the graph in Figure 4.3. 97

```
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 = 1:9
    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.
- 141 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  $\mathbf{A} \otimes \mathbf{B}$ .
- 149 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

```

The Matlab function `toeplitz` constructs a Toeplitz matrix. 150

A circulant matrix can be built in Matlab from a vector  $v$  using 150

```
toeplitz(v,circshift(flipud(v),1))
```

The convolution of  $u$  and  $v$  is  $\text{ifft}(\text{fft}(u) \cdot \text{fft}(v))$ . 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(fft2(x1).*fft2(x2));
y = y(1:n);
end
```

```
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
```

The Matlab function `fft` implements the Fastest Fourier Transform in the 158 West (FFTW) library. The inverse FFT can be computed using `ifft` and multidimensional FFTs can be computed using `fftn` and `ifftn`.

The functions `fftshift` and `ifftshift` shift the zero frequency component to the 159 center of the array.

Matlab does not have a three-dimensional DST function, so we'll need to build 160 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))));;
end

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  $n = 50$ .

- 163 The MATLAB function `dct` returns the type 1–4 DCT. The equivalent function in Octave’s `signal` package returns a DCT-2. The following function returns a sparse matrix of Fourier coefficients along with the reconstructed compressed image:

```

pkg load signal
function [B,A] = dctcompress(A,d)
B = dct2(A);
idx = floor(d*prod(size(A)));
tol = sort(abs(B(:)), 'descend')(idx);
B(abs(B)<tol) = 0;
A = idct2(B); B = sparse(B);
end

```

We’ll test the function on an image and compare it before and after.

```

A = rgb2gray(imread(['laura.png']));
[~,A0] = dctcompress(A,0.01);
imshow([A,A0])

```

- 180 The function `class` returns the data type of a variable.
- 181 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
```

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`.

```
function y = Q_sqrt(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
```

```
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)
```

The command `realmax` returns the largest floating-point number. The command  
`realmin` returns the smallest normalized floating-point number.

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])
```

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.

```
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
```

- 197 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.

```
204 function M = escape(n,z,c)
    M = zeros(size(c));
    for k = 0:n
        mask = abs(z)<2;
        M(mask) = M(mask) + 1;
        z(mask) = z(mask).^2 + c(mask);
    end
end
```

```
function M = mandelbrot(bb,xn,n,z)
    yn = round(xn*(bb(4)-bb(2))/(bb(3)-(bb(1)))); % calculate number of iterations
    z = z*ones(yn,xn); % initialize z
    c = linspace(bb(1),bb(3),xn) + 1i*linspace(bb(4),bb(2),yn)'; % complex plane
    M = escape(n,z,c);
end
```

```
M = mandelbrot([-0.1710,1.0228,-0.1494,1.0443],800,200,0);
imwrite(1-M/max(M(:)), 'mandelbrot.png');
```

- 208 The function `polyval(x,p)` uses Horner's method to evaluate a polynomial with coefficients  $p = [a_0, a_1, \dots, a_n]$  at  $x$ .
- 211 The function `roots` returns the roots of a polynomial by finding the eigenvalues of the companion matrix.

```
217 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.

- 221 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)
```

The function `vander` generates a Vandermonde matrix with rows given by  $[x_i^n \quad x_i^{n-1} \quad \dots \quad 1]$ . Add `fliplr` to reverse them  $[1 \quad \dots \quad x_{i-1}^{n-1} \quad x_i^n]$ .

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')\gammaamma;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 = sum(x<=x');` broadcasts the element-wise operator  $\leq$  across a row and a column array.

- 239 The function `spline` returns a cubic spline with the not-a-knot condition.
- 241 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.
- 242 The Matlab function `pchip` returns a cubic Hermite spline.
- 244 We can build a Bernstein matrix in Matlab from a column vector `t` with the following function.

```
B = @(n,t)[1 cumprod((n:-1:1)./(1:n))].*t.^ (0:n).* (1-t).^(n:-1:0);
```

- 255
- ```
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
```

- 261 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,'.-')
```

The Chebfun package (<https://www.chebfun.org>) includes methods for manipulating functions in Chebyshev space. 263

```
function [x,phi] = scaling(c,z,n) 269
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)
```

```
psi = zeros(size(phi)); n = length(c)-1; L = length(phi)/(2*n) 271
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)
```

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. 273

274 We'll use the LTFAT package.

```
pkg load ltfat
adjustlevels = @(x) 1 - min(max((sqrt(abs(x))),0),1);
A = rgb2gray(double(imread(['laura_square.png'])))/255;
c = fwt2(A,"db2",9);
imshow(adjustlevels(c))
```

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);
imshow([A,max(min(B,1),0)]);
```

283 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
```

We can then implement the Levenberg–Marquardt method.

```
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
```

We can solve the problem using 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)
```

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
```

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
```

After determining 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;
```

- 297 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))`:

- 297 The command `rats` returns the rational approximation of number as a string by using continued fraction expansion. For example `rats(0.75)` returns  $3/4$ .
- 298 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
```

- 301 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  $y'(0) = 2$ .

We can define the dual numbers as

302

```
x1 = dual(2,[1 0]);
```

```
x2 = dual(pi,[0 1]);
y1 = x1*x2 + sin(x2);
y2 = x1*x2 - sin(x2);
```

306    **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**

307    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**

310    **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
```

316 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)
```

```
r = exp(2i*pi*(0:0.01:1));
plot((1.5*r.^2 - 2*r + 0.5)./r.^2); axis equal;
```

345

The following function returns the coefficients for a stencil given by  $m$  and  $n$ : 347

```

function [a,b] = multistepcoefficients(m,n)
    s = length(m) + length(n) - 1;
    A = (m+1).^(0:s-1)';
    B = ((0:s-1).* (n+1).^ [0,0:s-2])';
    c = -[A(:,2:end) B]\ones(s,1);
    a = [1;c(1:length(m)-1)];
    b = [c(length(m):end)];
end

```

We can express floating-point numbers  $c$  as fractions using the `rats( $c$ )`.

```

function plotstability(a,b)
    r = exp(1i*linspace(.01,2*pi-0.01,400));
    z = (r.^((1:length(a))*a) ./ (r.^((1:length(b))*b));
    plot(z); axis equal
end

```

The Matlab recipe for solving a differential equation is similar to the Julia recipe. 373 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 is almost identical.

- |                                                                                                      |                                                                                                                                                                                                       |
|------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. Define the problem<br>2. Set up the parameters<br>3. Solve the problem<br>4. Present the solution | <pre>pendulum = @(t,u)[u(2);-sin(u(1))];<br/> u0 = [8*pi/9,0]; tspan = [0,8*pi];<br/> [t,u] = ode23(pendulum,tspan,u0);<br/> plot(t,u(:,1),'.-',t,u(:,2),'.-')<br/> legend('\\theta','\\omega')</pre> |
|------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

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`. In MATLAB, we can alternatively request a structure `sol` as the output of the ODE solver. We can subsequently evaluate `sol` 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. Octave does not have the function `deval`.

- 384 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 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.

- 384 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.

- 385 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);
```

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([-1 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];
```

We can express the discrete indices of  $\xi$  as

450

```
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)))
```

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) fft2(ifft2(u).*ifft2(ikx.*u) ...
+ ifft2(v).*ifft2(iky.*u));
```

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  $H_x$ , computed using the function  $H$ , is the  $x$ -component of  $\mathbf{H}^n$  and  $H_{x0}$  is  $x$ -component of  $\mathbf{H}^{n-1}$ . Similarly,  $H_y$  and  $H_{y0}$  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;
```

```
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,'.-')
```

468

An easy way to input  $\mathbf{D}_n$  in Matlab is with the `diag` function:

469

```
D = diag(ones([n-1 1]),1) ...
    - 2*diag(ones([n 1]),0) + diag(ones([n-1 1]),-1);
eig(D)
```

The following function implements a naïve determinant:

470

```
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
```

The following function implements a naïve reverse Cuthill–McKee algorithm for symmetric matrices:

471

```
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
    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
```

- 471 We'll reuse the code on page 602.

```
p = rcuthillmckee(A);
gplot(A(p,p),circular_layout(A),'.-')
axis equal; axis off;
```

- 471 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

```
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)}
```

- 473 We'll reuse the function `get_adjacency_matrix` from page 602. 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 block sparse 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)){:}
```

We can use the whos function to determine the bytes of a matrix.

474

```
d = 0.5; A = sprand(60,80,d);
s = whos('A'); nbytes = s.bytes;
nbytes, 8*(2*d*prod(size(A)) + size(A,2) + 1)
```

```
X = rgb2gray(double(imread(['bucket "laura.png'])))/255;
[m,n] = size(X);
blur = @(x) exp(-(x/20).^2/2);
A = blur([1:m] - [1:m]'); A = A./sum(A,2);
B = blur([1:n] - [1:n]'); B = B./sum(B,1);
N = 0.01*rand(m,n);
Y = A*X*B + N;
```

474

```
a = 0.05;
X1 = A\Y/B;
```

```
X2 = (A'*A+a^2*eye(m))\A'*Y*B'/(B*B'+a^2*eye(n));
X3 = pinv(A,a)*Y*pinv(B,a);
imshow(max(min([X Y X1 X2 X3],1),0))
```

475 The Matlab code for exercise 3.5 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 evaluates a polynomial given those coefficients. The command `qr(V,0)` 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])
```

476

```

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;
tempsmodel = @(t) [sin(2*pi*t) cos(2*pi*t) ones(size(t))];
c = tempsmodel(day)\temp;
plot(day,temp,'.',day,tempsmodel(day)*c,'k');

```

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 is a low-dimensional column space of the space of digits. 477

```

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 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.

- 478 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 624.

```
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
```

- 478
- ```
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'
```

- 480
- ```
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
```

- 481
- ```
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
```

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
    eigenvalues = diag(H);
end
```

```
n = 20; S = randn(n);
D = diag(1:n); A = S*D*inv(S);
implicitqr(A)
```

We'll reuse `get_names` and `get_adjacency_matrix` from page 624.

482

```
actors = get_names(bucket,"actors");
B = get_adjacency_matrix(bucket,"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)){:}
```

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:

483

```
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']))));
[U,S,V] = randomizedsvd(A,10);
imshow([A U*S*V'])
```

484

```
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)
```

485 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(-f); toc` takes around 20 seconds, considerably shorter than either Julia or Python.

```

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)

```

486

The Matlab code for the radix-3 FFT in exercise 6.1 is

487

```

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

```

- 488 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

```

- 489
- ```

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

```

Two-dimensional DCT and IDCT operators apply the DCT or IDCT once in each dimension.

```

dct2 = @(f) dct(dct(f)');
idct2 = @(f) idct(idct(f'));

```

The following function returns a sparse matrix of Fourier coefficients along with the reconstructed compressed image:

```
pkg load signal
function [B,A] = dctcompress2(A,d)
    n = size(A) ; n0 = floor(n*sqrt(d));
    B = dct2(A)(1:n0(1),1:n0(2));
    A = idct2(B,n);
end
```

We'll test the function on an image and compare it before and after.

```
aitken1 = @(x1,x2,x3) x3 - (x3-x2).^2./(x3 - 2*x2 + x1); 494
aitken2 = @(x1,x2,x3) (x1.*x3 - x2.^2)./(x3 - 2*x2 + x1);
n = 20000;
p = cumsum((-1).^[0:n]*4./(2*[0:n]+1));
p1 = aitken1(p(1:n-2),p(2:n-1),p(3:n));
p2 = aitken2(p(1:n-2),p(2:n-1),p(3:n));
loglog(abs(pi-p)); hold on; loglog(abs(pi-p2)); loglog(abs(pi-p1));
```

We'll find the solution to the system of equations in exercise 8.12. 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,df,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,df,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
end
```

- 500 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
```

The following script picks  $n$  random points and interpolates a parametric spline with periodic boundary conditions using  $nx*n$  points. To evaluate the spline we use `evaluate_spline` discussed on page 611.

```
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);
```

- 501 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]);
```

- 503 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).*(1-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))

```

```
506 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
```

```
509 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)`.

```
509 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
```

510 We'll extend the dual class on page 617 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;
```

```
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
```

511

```
f = @(x) exp(x)/(cos(x).^3 + sin(x).^3)
cauchyderivative(f, 0, 6)
```

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

```

- 512 We'll modify the implementation of the Golub–Welsch algorithm from page 618 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))

```

- 512 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 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,'.');

```

- 514 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 = linspace(-4,4); y = x'; r = zeros(N,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]);

```

```

i = (0:3)';
a = ((-(i+1)').^i./factorial(i))\[1;0;0;0];
b = ((-i').^i./factorial(i))\[0;1;0;0];

```

514

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 multistepcoefficients defined on page 619.

```

function z = PECE(n,m)
[_,a] = multistepcoefficients([0,1],1:n-1);
[_,b] = multistepcoefficients([0,1],0:n-1);
for i = 1:200
    r = exp(2i*pi*(i/200));
    c(1) = r - 1;
    c(2:m+1) = r + r.^(1:n-1)*b(2:end)'/b(1);
    c(m+2) = r.^1:n-1*a/b(1);
    z(i,:) = roots(fliplr(c))'/b(1);
end
end

for i= 1:4
    plot(PECE(2,i),'.k'); hold on; axis equal
end

```

517

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 the Taylor polynomial approximation of  $\log(x + 1)$ .

```
m = 3; n = 2;
a = [0 ((-1).^(0:m+n)./(1:m+n+1))]';
[p,q] = pade(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
```

- 518 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 ode45 with linspace(0,15,100) to evaluate the solution at additional points for a smoother plot.

- 518 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));
```

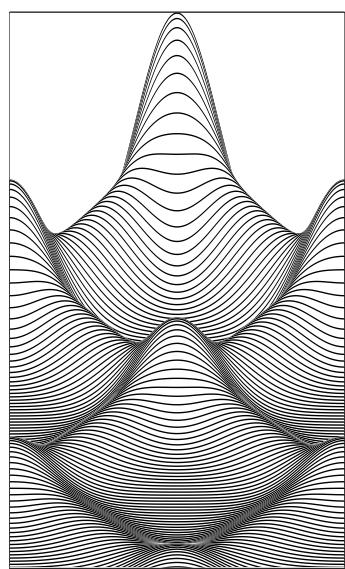
- 519 The following code solves the initial value problem and computes the error at the second boundary point:

```
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)
```

Displaying time-varying dynamics can be challenging. A simple approach in Matlab is adding a plot command 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 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, each new one shifted 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',[])
```

The following Matlab code solves the Schrödinger equation:

523

```
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
```

```
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');
```

- 525 Let's solve a radially symmetric heat equation. Although we divide by zero at  $r = 0$  when constructing the Laplacian operator, we subsequently overwrite the resulting  $\inf$  term 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);
```

- 526 We'll first define a function as a logit analogue to `linspace`.

```
loginspace = @(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:n
        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 = loginspace(20,m,.999);
u = heat_equation(x,t,phi(x,0,10));
plot(x,u,'.-',x,phi(x,t,10),'k')

```

Here's a solution to the Allen–Cahn equation using Strang splitting:

527

```

L = 16; m = 400; dx = L/m;
T = 4; n = 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.

- 533 The following code solves 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]);
```

- 534 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));
```

```
m=10; 535
x=linspace(0,1,m)';
h=x(2)-x(1);
A=diag(repmat(-1/h-h/6,[m-1 1]),-1)+diag(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);
plot(x,s,'o-',x,u,'.-');
```

```
m = 8; 537
x = linspace(0,1,m+2); h = x(2)-x(1);
D = @(a,b,c) (diag(a*ones(m-1,1),-1) + ...
    diag(b*ones(m,1),0) + diag(c*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 = [ones([m 1])*h*384;zeros([m 1])];
u = M\b;
plot(x,16*(x.^4 - 2*x.^3 + x.^2),'o-',x,[0;u(1:m);0],'.-');
```

```
m = 128; x = (1:m)'/m*(2*pi)-pi; 537
k = 1i*[0:m/2 -m/2+1:-1]';
f = @(t,u) -(ifft(k.*fft(0.5*u.^2)));
[t,u] = ode45(f,[0 1.5],exp(-x.^2));
plot(x,u(end,:))
```

The following Matlab code solves the KdV equation using integrating factors. 538  
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`. My carelessness and use of the conjugate transpose (`'`) instead of a regular transpose (`.`) provided me with what seemed like hours of debugging amusement.

- 540 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.
- Alexander Craig Aitken. *Gallipoli to the Somme: Recollections of a New Zealand infantryman*. Oxford University Press, 1963.
- 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. Démo istration 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.
- Jonathan M. Borwein and Peter B. Borwein. *Pi and the AGM: A study in the analytic number theory and computational complexity*. Wiley–Interscience, 1987.
- 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.
- Claude Brezinski. *History of continued fractions and Padé approximants*, volume 12. Springer Science & Business Media, 2012.
- 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.
- Augustin-Louis Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Comptes Rendu de à l'Académie des Sciences*, 25:536–538, 1847.
- 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.

- John Crank and Phyllis Nicolson. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 43, pages 50–67. Cambridge University Press, 1947.
- 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.
- Susan T. Dumais, George W. Furnas, Thomas K. Landauer, Scott Deerwester, and Richard Harshman. Using latent semantic analysis to improve access to textual information. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 281–285, 1988.
- 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 tr 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.
- Uriel Frisch and Jérémie Bec. Burgulence. In *New trends in turbulence*, pages 341–383. Springer, 2001.
- 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. *Nautilus 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 Nørsett, 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.
- William H. Kruskal and Stephen M. Stigler. Normative terminology: “Normal” in statistics and elsewhere. *Statistics and Public Policy*, pages 85–111, 1997.
- 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 Mandelbrot. How long is the coast of Britain? Statistical self-similarity and fractional dimension. *science*, 156(3775):636–638, 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.
- Leonid V. Moroz, Cezary J. Walczyk, Andriy Hrynyshyn, Vijay Holimath, and Jan L. Cieśliński. Fast calculation of inverse square root with the use of magic constant—analytical approach. *Applied Mathematics and Computation*, 316:245–255, 2018.
- Philip M. Morse. Trends in operations research. *Journal of the Operations Research Society of America*, 1(4):159–165, 1953.
- K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations*. Cambridge University Press, Cambridge, second edition, 2005.
- Jean-Michel Muller. Elementary functions and approximate computing. *Proceedings of the IEEE*, 108(12):2136–2149, 2020.

- 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://thewinnower.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.
- Alan M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.
- 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.
- James Hamilton Verner. Explicit Runge–Kutta methods with estimates of the local truncation error. *SIAM Journal on Numerical Analysis*, 15(4):772–790, 1978.
- 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. Zwieten, 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, 345  
A-conjugate, 135  
A-energy inner product, 13  
A-stable, 339  
Abel, Niels, 207  
Abel–Ruffini theorem, 89, 104, 191, 207  
ABM method, 350  
absolutely stable, 337  
acceptance-rejection sampling, 323  
activation function, 276  
Adams–Bashforth method, 349, 516  
Adams–Moulton method, 349, 516  
adaptive methods, 307, 318, 319, 362,  
    363, 569, 602, 621  
adjacency matrix, 47  
adjoint, 6, 303  
affine transformation, 24, 276, 317  
Agnesi, Maria, 235  
Airy equation, 262, 378, 593  
algorithms  
    Arnoldi method, 112  
    Babylonian method, 194  
    bisection method, 191  
    Bluestein factorization, 154–155  
    Broyden’s method, 214  
    Cholesky decomposition, 37  
    conjugate gradient method, 136  
    Cooley–Tukey radix-2 FFT, 149

Cuthill–McKee algorithm, 51–52, 471,  
    552, 574, 603, 623  
de Boor’s algorithm, 240  
de Casteljau’s algorithm, 244  
Dekker–Brent method, 196  
fast inverse square root, 182  
fast Toeplitz multiplication, 153  
full orthogonalization method, 139  
Gauss–Seidel method, 121  
Gaussian elimination, 33  
general minimized residual, 140  
Golub–Kahan–Reinisch algorithm,  
    115  
Golub–Welsch algorithm, 315, 638  
gradient descent, 132, 219  
Gram–Schmidt method, 62, 254  
implicit QR method, 108  
multigrid method, 128  
Newton’s method, 193, 212, 493, 496,  
    497, 585, 590, 591, 594, 633, 637  
Newton–Horner method, 209  
nonnegative matrix factorization, 85  
Rader factorization, 155–158  
randomized SVD algorithm, 116  
Rayleigh quotient iteration, 101  
revised simplex method, 54  
Richardson extrapolation, 298  
simplex method, 42  
successive over-relaxation, 126  
Allen–Cahn equation, 400, 455, 527

almost L-stable, 364  
 ansatz, 409  
 Arnoldi method, 110–114  
     implicitly restarted, 113  
 Arnoldi process, 138  
 artificial intelligence, 276  
 asymptotic convergence, 198  
     rate, 124  
 autodiff, *see* automatic differentiation  
 automatic differentiation, 291, 299–303  
 automatic programming, 545

**B**

B-splines, 240–241, 266, 278, 453, 502, 561  
 Bézier curves, 243–246  
 Bézier, Pierre, 245  
 Babbage, Charles, xx, 229, 310  
 Babylonian method, 194  
 Backus, John, 545  
 backward differentiation formula, 344  
 backward Euler method, 333–341, 351, 364, 369, 383, 388, 404, 441, 523  
 Bacon number, 56, 575, 624  
 Bacon, Kevin, 56  
 Banach fixed-point theorem, 93, 198, 360  
 Banach space, 253  
 banded matrix, 8, 50  
 basic feasible solution, 42  
 basic variables, 41  
 basin of attraction, 213  
 BDF, *see* backward differentiation formula  
 benchmarking, 374  
 Bernoulli number, xx, 310  
 Bernoulli trial, 245, 285  
 Bernoulli’s theorem, 245  
 Bernstein matrix, 244, 561, 612  
 Bernstein polynomial, 244, 245, 505  
 Bernstein, Serge, 245  
 Bessel’s equation, 247, 503  
 best approximation, 249  
 bifurcation diagram, 202  
 big O notation, 171  
 bilinear form, 435  
 biomimicry, 276

bisection method, 191  
 Bitcoin, 223  
 Bluestein factorization, 154–155  
 Bogacki–Shampine method, 356, 357, 371  
 Bonnet’s formula, 326  
 Boole’s rule, 305  
 Born, Max, xxiv  
 Borwein’s algorithm, 495  
 boundary conditions, 383–386  
     open, 400, 423  
 boundary value problem  
     Chebyshev polynomials, 262  
     collocation, 247  
     finite element method, 431  
     shooting method, 378, 519  
 Box, George, 324, 479  
 Box–Muller transform, 324  
 breadth-first search, 51  
 Broyden’s method, 214  
 Broyden–Fletcher–Goldfarb–Shanno algorithm, 219  
 Buffon’s needle, 320  
 Burgers’ equation, 413–414  
 Burgers, Jan, 413  
 Butcher tableau, 353  
 butterfly matrix, 148

**C**

C programming language, 547  
 Céa’s lemma, 440  
*Caenorhabditis elegans*, 55  
 Cahn–Hilliard equation, 464, 541  
 Cantor set, 204  
 Cardano’s formula, 207  
 Cardano, Girolamo, 207  
 cardioid, 202, 520  
 catastrophic cancellation, 184  
 Cauchy differentiation formula, 326  
 Cauchy problem, 414  
 Cauchy sequence, 253  
 Cauchy, Augustin–Louis, 130  
 Cauchy–Schwarz inequality, 14, 439, 442  
 centrality, 96  
 CFL condition, 388, 403  
 CFL number, 403

- chaotic map, 202  
 characteristic curves, 402  
 characteristic decomposition, 412  
 characteristic functions, 241  
 characteristic polynomial, 89, 210, 377,  
     468  
 characteristic speed, 402, 411  
 Chebyshev approximation, 249  
 Chebyshev differentiation matrix, 260  
 Chebyshev equioscillation theorem, 234  
 Chebyshev node, 234  
 Chebyshev polynomial, 233, 259–263,  
     311, 468  
 Chebyshev spectral method, 259  
 Cholesky decomposition, 35–37  
 circulant matrix, 11, 150–153  
 citardaum formula, 175  
 Clawpack, 571  
 Clenshaw–Curtis quadrature, 310–311,  
     319, 568, 618  
 coercive, 439  
 cofactor expansion, 210  
 colleague matrix, 263  
 collocation, 247, 356, 502  
 column space, 6, 24, 41, 58, 59, 71, 72,  
     76, 80, 116  
 companion matrix, 210, 263  
 compatible norm, 15  
 complete metric space, 253  
 complete pivoting, 35  
 complex-step derivative, 189, 212, 283  
 composite Simpson’s rule, 307  
 composite trapezoidal rule, 306  
 compressed sparse column, 46  
 comrade matrix, 263  
 condition number, 21, 173–175, 178  
     eigenvalue, 90  
 conditioning, *see* condition number,  
     ill-conditioned, well-conditioned  
 confusion matrix, 478, 578, 627  
 congruent, 19, 36  
 conjugate gradient method, 134–137  
 conjugate transpose, 6  
 connectivity, 47  
 conservation law, 412  
 conservative, 425  
 consistent, 177, 336, 342  
 contact discontinuity, 424  
 continuous, 439  
 contraction mapping, 93, 123, 197, 360  
 convergence, 178, 342  
     global, 192  
     local, 192  
     order of, 192  
     spectral, *see* spectral accuracy  
 convergence factor, 124  
 convolution, 11, 151, 152, 291, 461, 556,  
     607  
 convolutional neural network, 293, 506,  
     588  
 Cooley–Tukey radix-2 FFT, 149  
 coset, 145  
 cosine similarity, 81  
 Courant–Fischer theorem, 99  
 Courant–Friedrichs–Lewy, 388  
 covector, 19  
 Crank, John, 382  
 Crank–Nicolson method, 441, 525, 527  
 cubature, 304  
 cubic splines, 235–239  
 cumulative length spline, 239  
 curse of dimensionality, 289, 321  
 Cuthill–McKee algorithm, 51–52, 471,  
     552, 574, 603, 623  
 Cybenko, George, 278
- D**
- da Vinci, Leonardo, 276  
 Dahlquist barrier, 348  
 dam break problem, 430  
 dam-break problem, 430  
 Dantzig, George, 37  
 de Boor’s algorithm, 240  
 de Casteljau’s algorithm, 244  
 decision variables, 38  
 decomposition  
     incomplete LU, 122  
 deep learning, 181, 290, 565  
 defective, 13  
 deflation  
     of a matrix, 106  
     of a polynomial, 208  
 degree

- of a polynomial, 207, 225, 232, 312
  - of a vertex, 47
  - of freedom, 108, 236, 280
  - of separation, 55
  - degree matrix, 47
  - Dekker–Brent method, 196
  - Deming regression, 76
  - density, 46
  - determinant, 24, 55, 470
  - de Broglie, Louis, 162
  - de Casteljau, Paul, 245
  - DFT, *see* discrete Fourier transform
  - Diaconis, Persi, 147
  - diagonally dominant, 123
  - diet problem, *see* Stigler diet problem
  - Diffie–Hellman key exchange, 205
  - Dijkstra's algorithm, 51
  - Dirac notation, 19
  - direct problem, 173
  - Dirichlet boundary condition, 383
  - discrete Fourier transform, 144
  - discrete wavelet transform, 272–276
  - dispersion relation, 409
  - divided differences, 229
  - dolphins of Doubtful Sound, 49
  - domain of dependence/influence, 404
  - dominant eigenvalue, 93
  - Dormand–Prince method, 356, 362, 371, 395
  - double-and-add, 207, 498
  - dual number, 300
  - dual problem, 44
  - dual space, 19
  - Duffing equation, 378, 518, 593, 640
  - Dufort–Frankel method, 389–390, 521
  - DWT, *see* discrete wavelet transform
- E**
- Eaton, John, 546
  - Eckart–Young–Mirsky theorem, 74
  - eigenimage, 79
  - eigenpictures, 79
  - eigenspace, 7
  - eigenvalue, 6, 24
    - condition number, 90
    - estimation, 91
    - methods for computing, 93–114
  - eigenvector, 6, 24
    - left eigenvector, 90
  - eigenvector centrality, 96
  - elementary row operations, 30
  - elliptic curve, 206
  - energy method, 396
  - energy norm, 14
  - ENIAC, xiv, 321, 543
  - entropy, 418
  - epoch, 289
  - equivalent matrices, 8, 12
  - equivalent norms, 14, 15, 397
  - Erdős, Paul, 56
  - Euclidean inner product, 13
  - Euclidean norm, 14, 24
  - Euler equations, 430
  - Euler method, *see* backward Euler method, forward Euler method
  - Euler, Leonhard, 46
  - Euler–Maclaurin formula, 309
  - explicit Euler method, *see* forward Euler method
  - exponential convergence, 258
- F**
- fast Fourier transform, 143–163, 259, 450
  - fast inverse square root, 182, 189
  - fast Toeplitz multiplication, 153
  - father wavelet, *see* scaling function
  - Favard's theorem, 254, 314
  - Feigenbaum constant, 202
  - FEniCS, 442–445, 572
  - FFT, *see* fast Fourier transform
  - FFTW, 158, 160, 451, 556, 607
  - Fickian diffusion, 380
  - Filippelli problem, 86
  - finite volume method, 426
  - fixed point, 93, 201, 496
  - fixed-point method, 197–200, 360
  - fixed-point number, 185
  - fjord, 49, 299
  - floating-point number, 180–182
  - flux, 412
  - Fontana, Niccolò, 207
  - force-directed graph drawing, 48

Fortran, 184, 211, 239, 319, 372, 545–546, 560, 571  
 forward Euler method, 176, 333–341, 351, 369, 387, 403, 404, 407, 523  
 Fourier polynomial, 256–258, 448  
 Fourier series, 256  
 Fourier transform, 257  
 Fourier, Joseph, xix, 161  
 Fox, Laura, 68, 80, 162, 164, 274, 275, 490  
 fractal, 299  
 Francis, John, 101  
 Frejér quadrature, *see* Clenshaw–Curtis quadrature  
 Frobenius norm, 17, 26, 74, 84, 162, 214, 470, 550  
 Fry, Hannah, 290  
 full orthogonalization method, 139  
 fundamental subspaces, 6, 72  
 fundamental theorem of Gaussian quadrature, 312  
 linear algebra, 6  
 linear programming, 40  
 numerical analysis, 178, 341

**G**

Galerkin formulation, 433  
 Galois field, 3  
 Galois, Évariste, 207  
 Gauss, Carl Friedrich, xix, 31, 146, 312  
 Gauss–Chebyshev, 312, 313  
 Gauss–Christoffel quadrature, *see* Gaussian quadrature  
 Gauss–Hermite, 312, 313, 316, 327, 638  
 Gauss–Jacobi, 312  
 Gauss–Kronrod, 317–319  
 Gauss–Laguerre, 312, 313, 317  
 Gauss–Legendre, 312, 313, 316–319, 326, 327, 511, 591  
 Gauss–Legendre algorithm, 495  
 Gauss–Lobatto, 319  
 Gauss–Newton method, 283  
 Gauss–Radau, 319  
 Gauss–Seidel method, 121  
 Gaussian distribution, 58, 161, 277, 282, 451  
 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, 311–319  
 Gaussian–Legendre, 637  
 Gear’s method, *see* backward differentiation formula  
 general minimized residual, 140  
 generalized linear model, 285  
 generator  
     pseudo-random number, 325  
     subgroup, 144  
     subspace, 4  
 Gershgorin circle theorem, 91  
 ghost point, 382  
 Gibbs phenomenon, 162, 454  
 Girko’s circular law, 480  
 Givens rotation, 116  
 global truncation error, 346  
 golden ratio, 218  
 golden section search method, 218  
 Goldstine, Herman, 31  
 Golub–Kahan–Reinsch algorithm, 115  
 Golub–Welsch algorithm, 315, 638  
 Google matrix, 97  
 gradient descent, 130–134, 137, 219–221, 289, 508, 565  
 Gram matrix, 70, 251  
 Gram–Schmidt method, 61–62, 116, 254  
 graph Laplacian, 47  
 graphs, 46–50  
 Green’s first identity, 435  
 Green’s theorem, 416  
 group, 144, 206  
 group velocity, 410

**H**

Haar, Alfréd, 264  
 Hadamard product, 5, 70, 152  
 Hadamard, Jacques, 21, 173  
 Halley’s method, 222  
 Halmos, Paul, xiv, xxiii  
 Hamiltonian, 19, 369  
 heat equation, 380–381  
     backward Euler solution, 383  
     Crank–Nicolson solution, 385

- Dufort–Frankel solution, 521  
 Fourier solution, 450  
 fundamental solution, 397  
 polar coordinates, 399  
 heavy ball method, 220  
 Hein, Piet, 6  
 Hermite interpolation, 230  
 Hermitian matrix, 8  
 Hessian matrix, 219  
 Heun’s method, 352  
 Hilbert approximation, 249  
 Hilbert basis, 253  
 Hilbert matrix, 23, 72, 252  
 Hilbert space, 253  
 homotopy, 215  
 Hopf–Cole transformation, 414  
 Horner form, 208, 498  
 Horner’s method, 208  
 Householder matrix, 64, 104  
 Householder method, 222  
 Householder reflection, 64, 116  
 hundred-dollar problem, 118, 142, 172  
 hyperbolic equation, 401–408  
 hyperbolic system, 410–413, 420–426  
 hyperoperation, 170  
 hyperspectral images, 84
- I**  
 identification problem, 173  
 IDFT, *see* discrete Fourier transform  
 IEEE 754 standard, 180–182  
 ill-conditioned, 22, 174  
 ill-posed, 21, 173  
 image compression  
     discrete cosine transform, 161  
     discrete wavelet transform, 273  
     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, 15, 22, 25, 123, 254  
 infinitesimal number, 300  
 infinity norm, 14, 16, 58, 92, 189, 233,  
     249, 292, 397  
 inner product, 13, 250  
 inner product space, 250  
 integral curve, 334  
 integration, *see* Clenshaw–Curtis quadrature, Gaussian quadrature, Monte Carlo integration, Newton–Cotes quadrature  
 intrinsic dimension, 280  
 invariant, 7  
 inverse problem, 173  
 iteratively reweighted least squares, 286
- J**  
 Jacobi method, 121  
     weighted, 125  
 Jacobi operator, 315  
 Jenkins–Traub method, 211  
 JPEG 2000, 78, 273  
 Julia set, 204, 213  
 Jupyter, 544, 549, 599
- K**  
 Königsberg bridge puzzle, 47  
 Kahan, William, 182  
 Kantorovich, Leonid, 37  
 Karinthy, Frigyes, 55  
 Karpinski, Stefan, 548  
 Karush–Kuhn–Tucker conditions, 70, 84  
 Katz centrality, 96  
 Kelvin–Helmholtz instability, 461  
 kernel, 6  
 Kirchhoff matrix, 47  
 Kirchhoff, Gustav, 46  
 KKT conditions, *see* Karush–Kuhn–Tucker conditions  
 knots, 235  
 Knuth, Donald, xiv, 170, 544  
 Kortweg–deVries equation, 463, 538  
 Kronecker product, 148  
 Kronrod, Aleksandr, 318  
 Krylov matrix, 111, 150  
 Krylov methods, 137–141  
 Krylov subspace, 25, 111, 138  
 Kublanovskaya, Vera, 101  
 Kuramoto–Sivashinsky equation, 457,  
     464, 542

**L**

L-stable, 364  
 Lagrange multiplier, 70, 214  
 Lambert W function, 361  
 Lanczos method, 114, 139  
 Landau notation, 171  
 Landau, Edmund, 95  
 LAPACK, 33, 546, 553, 603, 606  
 Laplace equation, 431  
 Laplace expansion, 210  
 Laplacian matrix, 47  
 Laplacian operator, 51, 120, 124, 159, 400  
 Larry, Shampine, 361  
 latent semantic analysis, 81  
 law of large numbers, 245, 322  
 law of the unconscious statistician, 321  
 Lax entropy condition, 418  
 Lax equivalence theorem, 178, 342  
 Lax, Peter, 3  
 Lax–Friedrichs method, 405–411, 425, 529  
 Lax–Milgram lemma, 439  
 Lax–Wendroff method, 406–411, 426, 461  
 Lax–Wendroff theorem, 425  
 leapfrog method, 333–341, 523  
 learning rate, 219, 284  
 least squares  
     constrained, 69  
     multiobjective, 69  
     regularized, 67  
     sparse, 70  
 LeCun, Yann, 293  
 left eigenvector, 90, 93, 422  
 left null space, 6, 71, 72  
 Legendre polynomial, 4, 252, 255, 318, 358  
 Leland, Wilkinson, 547  
 LeNet, 293, 506  
 Levenberg–Marquardt method, 219, 283, 284  
 LeVeque, Randall, 571  
 likelihood function, 285  
 Lindemann–Weierstrass theorem, 304  
 linear functional, 434

linear independence, 4  
 linear map, 5  
 linear programming, 37–45  
     equational form, 41  
     standard form, 39  
 linear regression, 58, 66, 281, 476, 578, 627  
 link function, 285  
 Lipschitz continuous, 173, 332  
 literate programming, 544  
 little O notation, 171  
 load vector, 434

local Lax–Friedrichs method, 426, 533  
 local truncation error, 346  
 logistic equation, 176  
 logistic map, 176, 201  
 logit function, 526  
 Lorenz equation, 377  
 Lotka–Volterra equation, 374  
 Lovelace, Ada, xx, 229, 310  
 LU decomposition, 31–35

**M**

Müller’s method, 196  
 machine epsilon, 181, 186, 212, 557, 604, 609  
 Madhava of Sangamagrama, 495  
 magic number, 183, 189  
 Mandelbrot set, 204, 558  
 Mandelbrot, Benoit, xix, 299  
 manifold hypothesis, 280  
 Markov chain, 93  
 Markov matrix, 93  
 Mathematica, *see* Wolfram Language  
 Matplotlib, 547  
 matrices  
     adjacency, 47  
     banded, 8, 50  
     Bernstein, 244, 561, 612  
     Chebyshev differentiation, 260  
     circulant, 11, 150–153  
     colleague, 263  
     companion, 210, 263  
     comrade, 263  
     degree, 47  
     Gram, 70, 251  
     Hermitian, 8

- Hilbert, 23, 72, 252  
 Householder, 64, 104  
 Krylov, 111, 150  
 Laplacian, 47  
 normal, 8  
 orthogonal, 8, 24  
 Pascal, 517  
 permutation, 8, 24  
 positive definite, 8, 19, 35  
 primitive, 93  
 projection, 8, 24, 460  
 sparse, 45–52  
 Toeplitz, 150–153  
 tridiagonal, 8, 29, 121, 159, 241, 315,  
     384, 385, 394, 404  
 unitary, 8  
 upper Hessenberg, 8, 104, 108, 113,  
     139, 555, 606  
 upper triangular, 8, 24  
 Vandermonde, 86, 87, 226, 228, 246,  
     296  
 zero-one, 25, 47
- matrix decomposition, *see* Cholesky, LU,  
     nonnegative matrix, QR, Schur, singular value, spectral decompositions  
 matrix norm, 15  
 maximum likelihood, 58  
 maximum principle, 381  
 May, Robert, 201  
 McCulloch, Warren, 276  
 mean, 291  
 mean value theorem, 231  
 median, 291  
 Mersenne twister, 325  
 Merson method, 376  
 method of characteristics, 402  
 method of lines, 331, 381  
 Metropolis algorithm, 324  
 Metropolis, Nicholas, 321  
 midrange, 291  
 Millennium Prize, 381  
 million-dollar problem, 381  
 min-max theorem, 99  
 mindblowingly awesome, *see* Fox, Laura  
 minimization problem, 431  
 mixed boundary condition, 386  
 MNIST, 87, 294, 477, 507, 578  
 mode, 291  
 Moler, Cleve, 27, 182, 361, 546  
 Monte Carlo integration, 320–327  
 Monty Python, 547  
 Moore–Penrose pseudoinverse, *see*  
     pseudoinverse  
 mother wavelet, *see* wavelet function  
 motion by mean curvature, 528  
 multigrid method, 128  
 multilateration, 88  
 multiresolution, 265
- N**
- Nakamoto, Satoshi, 223  
 NaN, *see* not a number  
 Navier–Stokes equation, 381, 413, 458–  
     462  
 Nelder–Mead method, 218  
 neural network, 276–281, 287–291, 506–  
     508  
 Newton’s method, 193, 212, 219, 493,  
     496, 497, 585, 590, 591, 594, 633,  
     637  
 Newton, Isaac, 222  
 Newton–Cotes quadrature, 303–311  
 Newton–Horner method, 209  
 Newton–Raphson method, *see* Newton’s  
     method  
 Nicolson, Phyllis, 382  
 node, 47  
 nonbasic variables, 41  
 nonnegative matrix factorization, 83–85  
 nonstandard analysis, 300  
 norm, *see* energy norm, Euclidean norm,  
     Frobenius norm, induced norm, infinity  
     norm, matrix norm, p-norm,  
     spectral norm, vector norm  
 normal equation, 58–60, 251, 281  
 normal matrix, 8  
 not a number, 185, 558, 609  
 null space, 6, 24, 67, 71, 72, 92  
 nullity, 6  
 numerical diffusion/dispersion, 408–410  
 numerical viscosity, 408  
 numerically stable, 177

**O**

O, o, 171  
 objective function, 38  
 Octave, 546  
 Octave, Levenspiel, 546  
 Oliphant, Travis, 547  
 operator splitting, *see* splitting methods  
 optimal vector, 131  
 orbit, 198  
 order of convergence, 192  
 orthogonal, 13, 250  
 orthogonal complement, 250  
 orthogonal matrix, 8, 24  
 orthogonal polynomial, 253–255, 311–314, 318  
 orthogonal regression, 76  
 orthonormal, 253  
 overflow, 185  
 Arianne 5 rocket, 185

**P**

p-norm, 14, 21, 550  
 Pólya, George, 202  
 Padé approximation, 350  
 PageRank algorithm, 96  
 Parseval’s theorem, 253, 453  
 partial pivoting, 35  
 partition of unity, 240, 269, 433  
 Pascal matrix, 517  
 path, 47  
 pathological function, 169  
 PCA, *see* principal component analysis  
 perfect shuffle, 147, 273  
 permutation matrix, 8, 24  
 Perron–Frobenius theorem, 93, 97  
 phase velocity, 409  
 Pitts, Walter, 276  
 pivoting, 34–35  
 Poincaré inequality, 439  
 Poisson equation, 120, 126, 141, 159, 434, 436, 440, 443  
 Poisson summation formula, 449  
 polynomial, *see* Bernstein, characteristic, Chebyshev, Fourier, Lagrange, Legendre, orthogonal, Stieltjes, Taylor polynomials

polynomial interpolation error theorem, 232  
 positive definite, 131, 464  
 positive definite matrix, 8, 19, 35  
 positive matrix, 93  
 power method, 93–101  
 preconditioner, 122  
 predictor-corrector method, 349  
 prestige, 96  
 primal problem, 44  
 primitive matrix, 93  
 principal component analysis, 74–76, 87, 478  
 probability, 20, 93, 97, 245, 320, 321  
 probability vector, 97  
 projection matrix, 8, 24, 460  
 pseudo-inner product, 256  
 pseudoinverse, 57, 69, 72–74, 86, 282, 283, 553, 604  
 pseudospectral, 450  
 public-key encryption, 205  
 Python, 547

**Q**

Q\_rsqrt algorithm, *see* fast inverse square root  
 QR code, xxiv, 79, 82, 83, 100, 105, 113, 130, 163, 170, 172, 202, 244, 276, 289, 369, 380, 385, 396, 410, 411, 414, 458, 462, 491, 506, 522, 526, 528, 533, 534, 538–542, 549, 600  
 QR decomposition, 60–65  
 QR method, 101–109  
   implicit QR, 107–109  
   shifted QR, 105  
 quadratic approximation, 196  
 quadratic form, 18  
 quadratic map, 201  
 quadrature, 303, *see also* Clenshaw–Curtis quadrature, Gaussian quadrature, Monte Carlo integration, Newton–Cotes quadrature

**R**

R programming language, 547  
 Rackaukas, Chris, 375  
 Rader factorization, 155–158

- radial basis function, 247, 277  
 randomized SVD, 483  
 randomized SVD algorithm, 116  
 rank of a matrix, 6  
 Rankine–Hugoniot condition, 416  
 rate of convergence, 192  
 Rayleigh quotient, 99  
 Rayleigh quotient iteration, 99–101  
 Rayleigh quotient shifting, 105  
 Rayleigh–Bénard convection, 463  
 Rayleigh–Ritz formulation, 433  
 Rayleigh–Ritz method, 110  
 RBF, *see* radial basis function  
 recursion, 233, 269, 280, 298, 326  
 reduced row echelon form, 30  
 region of absolute stability, 338, 339  
 regression  
     Deming, 76  
     linear, 58, 281, 476, 578, 627  
     orthogonal, 76  
     total least squares, 76  
 regula falsi, 221  
 Reiss combustion model, 361  
 relaxation, 126  
 ReLU, 277  
 residual, 58  
 reverse Cuthill–McKee algorithm, *see*  
     Cuthill–McKee algorithm  
 revised simplex method, 54  
 Richardson effect, 299  
 Richardson extrapolation, 298, 307, 326  
 Richardson method, 389  
 Richardson, Lewis Fry, 299, 382  
 ridge regression, 68  
 Riemann invariant, 402, 421  
 Riemann problem, 417  
 Riemann, Bernhard, 169  
 Ritchie, Dennis, 547  
 Ritz estimate, 113  
 Ritz pair, 110  
 Rolle’s theorem, 232  
 Romberg’s method, 307, 319  
 root of unity, 144, 257  
 root-finding methods  
     bisection method, 191  
     Broyden’s method, 214  
     Companion matrix method, 210  
     fixed-point method, 197  
     Halley’s method, 222  
     Homotopy continuation, 215  
     Horner’s method, 208  
     Householder method, 222  
     Newton’s method, 193  
     secant method, 195  
 Rosenbrock function, 220  
 Rosenbrock method, 359  
 rough number, 153  
 round-off error, 186–187  
     Patriot missile, 185  
     Vancouver Stock Exchange, 187  
 row space, 6, 71, 72  
 Ruffini’s rule, 209  
 Rump, Siegfried, 184  
 Runge’s phenomenon, 233, 311, 312  
 Runge, Carl, 235  
 Runge–Kutta method, 351–360  
 Runge–Kutta methods  
     Bogacki–Shampine, 356, 357, 371  
     Dormand–Prince, 371, 537  
     ETDRK, 456, 541, 542  
     Heun, 352  
     Radau, 359  
     RK4, 354  
     Rosenbrock, 359  
     trapezoidal, 353  
     Verner’s, 372
- S**
- S programming language, 547  
 scaling function, 265  
 Schönhage–Strassen algorithm, 165  
 Schrödinger equation, 19, 398, 455, 463,  
     523  
 Schur decomposition, 9–10, 102, 109  
 Schur decomposition theorem, 9  
 secant method, 195  
 self-adjoint, 8  
 self-similar solution, 417  
 self-similarity, 170, 204, 265, 272, 280  
 semi-norm, 256  
 Shanks’ transformation, 222  
 Sherlock Holmes, 66  
 Sherman–Morrison formula, 54, 215

- shift-and-invert, 98  
shock, 414  
  speed, 416  
shooting method, 378, 519  
sigmoid function, 277, 278, 386, 527  
similar matrices, 7  
similarity transform, 7  
simplex, 218  
simplex method, 40–45  
  revised, 52–55  
Simpson’s rule, 305, 307, 319, 353  
sinc function, 452  
singular value, 12, 17, 24, 71, 72  
singular value decomposition, 12–13, 23,  
  71–83, 115–116  
singular vectors, 7  
smooth number, 153  
Sobolev space, 258, 435  
soliton, 463  
Sonneborn–Berger system, 95  
sparse matrix, 45–52, 119, 437  
sparse SVD, 116, 479, 483  
sparsity, 46  
spectral accuracy, 309, 447, 538  
spectral decomposition, 10–12, 90, 99  
spectral layout drawing, 48  
spectral norm, 16, 74  
spectral radius, 7  
spectral theorem, 10  
spectrum, 7  
splines, 235–242  
  cardinal B-spline, 241  
  computation, 237  
  cubic Hermite spline, 242  
  cumulative length, 239  
  parametric, 239  
splitting methods, 364–368  
  ADI, 392  
  fractional step, 391  
  IMEX, 367, 457  
  integrating factors, 368, 455  
  Strang splitting, 366  
stable, 21, 332, 337, 342  
Stan programming language, 321  
Steffensen’s method, 222  
stencil, 334  
Stieltjes polynomial, 318  
stiff equation, 361–364, 394, 454–458  
stiffness matrix, 434  
stiffness ratio, 363  
Stigler diet problem, 38, 55, 471, 575,  
  624  
Stigler, George, 38  
stochastic matrix, 93  
Strang splitting, 366, 527  
Strang, Gilbert, 3, 6, 143  
strictly hyperbolic, 411  
strong duality theorem, 45  
strong solution, 415  
submultiplicative, 15  
subspace, 4  
successive over-relaxation, 126  
Sundials, 373  
SVD, *see* singular value decomposition  
Swift–Hohenberg equation, 463  
swish function, 277  
symmetric, 439  
symplectic Euler method, 370  
symplectic methods, 368–370
- T**
- Taylor series expansion, 526  
Taylor’s theorem, 441  
ten-digit algorithm, xv  
ten-digit problem, 328  
test function, 414, 433  
tetration, 170  
theorems  
  Abel–Ruffini theorem, 89, 104, 191,  
  207  
  Banach fixed-point theorem, 93, 198,  
  360  
  Bernoulli’s theorem, 245  
  Céa’s lemma, 440  
  Chebyshev equioscillation theorem,  
  234  
  Courant–Fischer theorem, 99  
  Eckart–Young–Mirsky theorem, 74  
  Favard’s theorem, 254, 314  
  fundamental theorem of  
  Gaussian quadrature, 312  
  linear programming, 40  
  numerical analysis, 178

Gershgorin circle theorem, 91  
 Golub–Welsch algorithm, 315  
 Green’s first identity, 435  
 Green’s theorem, 416  
 implicit Q theorem, 108  
 Lax equivalence theorem, 178, 342  
 Lax–Wendroff theorem, 425  
 Lindemann–Weierstrass theorem, 304  
 mean value theorem, 231  
 min–max theorem, 99  
 Parseval’s theorem, 253, 453  
 Perron–Frobenius theorem, 93, 97  
 Poincaré inequality, 439  
 polynomial interpolation error theorem, 232  
 Rolle’s theorem, 232  
 Schur decomposition theorem, 9  
 spectral theorem, 10  
 strong duality theorem, 45  
 Taylor’s theorem, 441  
 universal approximation theorem, 277  
 Weierstrass approximation theorem, 245  
 theta method, 376  
 Thomae’s function, 170  
 three-term recurrence relation, 254, 293, 314, 468  
 Tikhonov regularization, 68, 283  
 time to first plot, 548  
 time-marching, 382  
 Toeplitz matrix, 150–153  
 total least squares regression, 76  
 total potential energy, 432  
 traffic equation, 418  
 transpose, 5  
 trapezoidal method, 333–341, 363, 523  
 trapezoidal rule, 305  
 Trefethen, Nick, xv, 118, 142, 172, 260, 263, 328  
 trial function, 433  
 tridiagonal matrix, 8, 29, 121, 159, 241, 315, 384, 385, 394, 404  
 truncation error, 187, 342, 408  
 Turing machine, 276

**U**

Ulam, Stanislaw, 321

ultimate shift, 109  
 UMFPACK, 46, 551, 601  
 unconditionally stable, 388  
 underflow, 185  
 unit round-off, *see* machine epsilon  
 unitarily similar, 7  
 unitary matrix, 8  
 universal approximation theorem, 277  
 upper Hessenberg matrix, 8, 104, 108, 113, 139, 555, 606  
 upper triangular matrix, 8, 24  
 upwind method, 402, 403, 408, 411

**V**

V-elliptic, 439  
 Van der Pol equation, 375  
 Vandermonde matrix, 86, 87, 143, 226, 228, 246, 296, 475, 500, 509, 554, 605  
 van Gogh, Vincent, 461  
 van Loan, Charles, 27  
 van Rossum, Guido, 547  
 variational problem, 431  
 vector norm, 13, 24  
 vector space, 3  
 Verlet method, 370  
 versine function, 235  
 vertex, 47  
 von Neumann analysis, 386–391, 403, 406  
 von Neumann, John, 31, 37, 324, 325

**W**

w-orthogonal, 250  
 wavelet function, 271  
 wavelets  
     B-spline, 266  
     Cohen–Daubechies–Feauveau, 273  
     Coiflet, 276  
     Daubechies, 269  
     Haar, 264, 274  
     LeGall–Tabatabai, 273  
 weak solution, 415  
 Weierstrass approximation theorem, 245  
 Weierstrass function, 169  
 Weierstrass, Karl, 234  
 well-conditioned, 122, 138, 174

well-posed, 21–23, 175–179, 331  
Wickham, Hadley, 547  
Wiener weight, 73  
Wikipedia, xiv, 56, 479, 545  
Wilkinson shift, 106  
Wilkinson, James, 45  
wisdom, 158  
witch of Agnesi, 235  
Wolfram Language, 211, 350, 496, 505,  
    544

**X**

Xoshiro, 325

**Z**

zero-based indexing, 548  
zero-one matrix, 25, 47  
zero-stability, 341  
zeros of polynomials  
    companion matrix method, 210  
    Newton–Horner method, 209  
Zipf's law, 66



# Julia Index

## Functions

append, 471	gmres, 141
argmax, 42	gplot, 49
argmin, 42	Gray, 483
bitstring, 181	hessenberg, 105, 481
cg, 137	hilbert, 23
Circulant, 150	idct, 159
cond, 23	ifft, 158, 506
conv, 152	ifftshift, 159
curve_fit, 285	interp, 502
dct, 159	irfft, 158
det, 468	isinf, 186
Diagonal, 483, 485	isnan, 186
diff, 500	kron, 148, 485
eigen, 109	log1p, 187
eigs, 114	LowerTriangular, 485
eigvals, 109	Matrix, 384, 483, 500
eigvecs, 109	minres, 141
eps, 181	nlsolve, 197
evalpoly, 208	nnz, 46
expm1, 187	norm, 17
fft, 158, 506	ones, 485
fftshift, 159	opnorm, 17
findall, 471	optimize, 221
findfirst, 54	parse, 488
floatmax, 185	permutedims, 477
floatmin, 185	pinv, 73
fzero, 197	popfirst, 471
gausslegendre, 316	qr, 65, 483
givens, 64, 481	rand, 483
	randn, 501

- randstring, 488
- Rational, 297
- rationalize, 297
- readdlm, 476
- reshape, 477
- rfft, 158
- Roots, 504
- roots, 89, 211
- rref, 30
- scaleminmax, 477
- solve, 496
- sortperm, 471
- sparse, 54, 485
- spdiagram, 485
- SpecialFunctions, 504
- SplitODEProblem, 457
- spy, 46
- spzeros, 54
- summarysize, 474
- svd, 71, 483
- svds, 71, 477, 484
- SymTridiagonal, 500
- Toeplitz, 150
- transpose, 6
- Tridiagonal, 384
- typeof, 180
- Vandermonde, 87, 228
- Vector, 54
  
- Macros**
- @animate, 395, 506
- @., 5
- @btime, 384
- @constraint, 472
- @elapsed, 30
- @gif, 522
- @less, 29
- @manipulate, 395
- @objective, 472
- @time, 30, 384, 483
- @variable, 472
  
- Packages**
- AbstractFFTs, 451
- LinRegOutliers, 375
- ApproxFun, 263
- Arpack, 114, 477, 479, 483, 484
- BenchmarkTools, 183, 384
- COSMO, 45
- CSV, 471, 476
- DataFrames, 471, 476
- Dates, 476
- DelimitedFiles, 49, 471, 476
- Dierckx, 239
- DiffEqDevTools, 375
- DiffEqOperators, 374, 457
- DifferentialEquations, 370, 373, 457, 496
- Distributions, 325
- DSP, 152
- FastGaussQuadrature, 316, 512
- FastPolynomialRoots, 211
- FEniCS, 442
- Ferrite, 442
- FFTW, 158, 159, 451, 488, 506
- FinEtools, 52
- Flux, 290, 507
- ForwardDiff, 303
- GLPK, 45, 472
- GraphPlot, 49, 471
- Graphs, 49, 471
- Gridap, 442
- HomotopyContinuation, 218
- Images, 477, 483
- Interact, 395, 506, 541
- Interpolations, 241
- IterativeSolvers, 137, 141
- JuliaFEM, 52
- JuMP, 45, 472
- Kinetic, 429
- LinearAlgebra, xxiv, 17, 481
- LsqFit, 285
- MLDatasets, 87, 477, 507
- NLsolve, 197
- NMF, 85
- Oceananigans, 429
- OffsetArrays, 269
- Optim, 221
- Plots, xxiv, 548
- Polynomials, 89, 211, 517
- PolynomialsRoots, 211

- ProgressMeter, 508
- Random, 488
- ReverseDiff, 303
- Roots, 197, 378
- RowEchelon, 30
- SciPyDiffEq, 373
- SparseArrays, 49, 54, 471, 485
- SpecialMatrices, 23, 87, 228
- Sundials, 373, 394
- ToeplitzMatrices, 150
- Trixi, 429
- ViscousFlow, 429
- Wavelets, 273
- Zygote, 303



# Python Index

## Functions

bspline, 561  
Bspline, 561  
cg, 555  
circulant, 556  
comb, 561  
cond, 550  
conj, 550  
convolve, 556  
CubicSpline, 560  
curve\_fit, 563  
det, 557  
dot, 551, 571  
dst, 557  
eig, 555  
eigs, 555  
eps, 557  
expm1, 558  
fft, 556  
fftfreq, 572  
fftshift, 556  
finfo, 557  
fsolve, 558  
hessenberg, 555  
hilbert, 550  
ifft, 556  
ifftshift, 556  
isinf, 558  
isnan, 558  
kron, 555  
leggauss, 568  
linear\_solver, 551  
linprog, 551  
log1p, 558  
lstsq, 553  
max, 558  
min, 558  
minimize, 560  
minres, 555  
norm, 550  
outer, 551  
pchip, 561  
pinv, 553  
polyval, 559  
pywraplp, 551  
qr, 553  
quad\_vec, 319  
reshape, 550, 600  
reverse\_cuthill\_mckee, 552  
romb, 319  
romberg, 319  
roots, 554, 559  
rref, 550  
simpson, 319  
solve, 550  
solve\_banded, 571  
solve\_circulant, 571  
solve\_ivp, 373, 559, 569, 570  
solveh\_banded, 560, 571  
splev, 560

- spline, 560
- splprep, 560
- spsolve, 551, 571
- svd, 553
- timeit, 550
- toeplitz, 556
- type, 557
- vander, 554, 560
- vstack, 550, 600
- ipywidgets, 571
- JAX, 566
- Matplotlib, 549
- Mshr, 572
- NetworkX, 552
- networkx, 552
- NumPy, 549
- OR-Tools, 551
- Pandas, 552
- PIL, 549
- PyWavelets (pywt), 562
- requests, 549
- scikits, 373
- SciPy, 549
- sys, 558
- TensorFlow, 565
- timeit, 570

## Packages

- Autograd, 566
- ChebPy, 562
- FEniCS, 572
- FiPy, 571
- fractions, 566

# Matlab Index

## Functions

class, 608

condeig, 605

dct, 608

eig, 605, 606

eigs, 606

eps, 609

expm1, 609

fft, 607

fftshift, 607

fminbnd, 611

fminsearch, 611

fminunc, 611

fzero, 610

givens, 603

glpk, 601

gplot, 602

hess, 606

hilb, 550, 600

ifft, 607

ifftshift, 607

integral, 319

isinf, 609

isnan, 609

kron, 606

leasqr, 615

linprog, 601

log1p, 609

lsode, 372

lsqcurvefit, 615

minres, 606

mkpp, 612

mldivide (\), 601, 603

mlrividie (/), 601

num2hex, 608

ode113, ode15i, ode15s, ode23, ode23s,  
ode23t, ode23tb, ode45, ode78,  
ode89, 371

pcg, 606

pchip, 612

pinv, 603, 604

polyval, 610

ppval, 612

qr, 603

quad, quadcc, quadl, quadgk, quadv, 319

rats, 616

realmax, realmin, 609

rgb2gray, 604

roots, 605, 610

rref, 600

spline, 612

svd, 604

svds, 604

symrcm, 603

toeplitz, 607

trapz, 319

vander, 605, 611

## Packages

Chebfun, 613

image, 604  
LTFAT, 613

optim, 611, 615  
signal, 608



