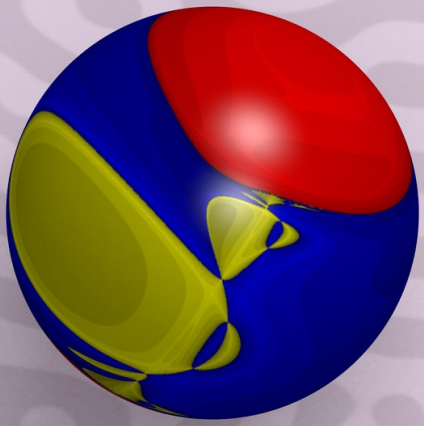
NUMERICAL METHODS FOR SCIENTIFIC COMPUTING



KYLE A. NOVAK

Numerical Methods for Scientific Computing

Numerical Methods for Scientific Computing

The Definitive Manual for Math Geeks

KYLE A. NOVAK

Copyright ©, 2017, by Kyle A. Novak

Version 1.4.1L

ISBN 978–1–365–65991–1

Printed in the United States of America

Cover illustration: Basins of attraction for the Rayleigh iteration for a three-dimensional matrix and the solution to the Swift–Hohenberg equation with random initial conditions.

Disclaimer: This book is a work in progress and will be progressively updated. Great efforts have been made to correct typographical and formatting errors. Nonetheless, several errors undoubtedly still lurk within these pages.

Dedicated to Laura

$$(x^2 + y^2 - 1)^3 - x^2y^3 = 0$$
$$(x^2 + y^2)^2 - 2(x^2 - y^2) = 0$$

Contents

Preface **v**

Linear Algebra **1**

1 Review: Theory and Definitions **3**
1.1 Vector spaces 3
1.2 Matrices 5
1.3 Eigenspaces 8
1.4 Measuring vectors and matrices 13
1.5 Well-posedness 20
1.6 Geometric interpretation of linear algebra 23

2 Direct Methods for Linear Systems **29**
2.1 Gaussian elimination 29
2.2 Cramer's rule is not your friend 36
2.3 Cholesky decomposition 36
2.4 Sparse matrices 39

3 Least Squares Problem and SVD **45**
3.1 Overdetermined systems 45
3.2 Normal equation 46
3.3 QR decomposition 48
3.4 Least squares problem (again) 54
3.5 Underdetermined (and inconsistent) systems 56
3.6 Singular value decomposition 57

4 Eigenvalue Problems **69**

4.1 Eigenvalue sensitivity 69
4.2 Eigenvalue estimation 71
4.3 The power method 73
4.4 The QR method 76
4.5 Speeding up the QR method 80
4.6 Implicit QR 84
4.7 Getting the eigenvectors 86
4.8 Arnoldi method 88
4.9 Singular value decomposition 93

5 Iterative Methods for Linear Systems 99
5.1 Jacobi and Gauss–Seidel methods 101
5.2 Successive over-relaxation 105
5.3 Multigrid 107
5.4 Solving the minimization problem 110
5.5 Krylov methods 117

6 Fast Fourier Transform 125
6.1 Discrete Fourier transform 125
6.2 Fast Fourier transform 128
6.3 Toeplitz and circulant matrices 132
6.4 Bluestein and Rader factorization 136
6.5 The FFT in MATLAB 139
6.6 Applications 140

Analysis 147

7 Preliminaries 149
7.1 Well-behaved functions 149
7.2 Well-posed problems 151
7.3 Well-posed methods 156
7.4 Floating-point arithmetic 160
7.5 Computational error 163

8 Solutions to Nonlinear Equations 169
8.1 Bisection method 169
8.2 Newton's method 171
8.3 Fixed-point iterations 175
8.4 Stopping criteria 178
8.5 Dynamical systems 179
8.6 Roots of polynomials 182
8.7 Systems of nonlinear equations 186

8.8 Homotopy continuation 191

9 Interpolation **197**
9.1 Review of linear algebra terms 197
9.2 Polynomial interpolation 198
9.3 How good is polynomial interpolation? 203
9.4 Hermite interpolation 207
9.5 Splines 208

10 Approximating Functions **221**
10.1 Least-squares approximation 221
10.2 Orthonormal systems 225
10.3 Complete orthonormal systems 226
10.4 Fourier polynomials 229
10.5 Wavelets 232
10.6 Data Fitting 236

11 Differentiation and Integration **241**
11.1 Differentiation 241
11.2 Numerical integration 245
11.3 Gaussian quadrature 248
11.4 Monte Carlo integration 252

Partial Differential Equations **258**

12 Ordinary Differential Equations **261**
12.1 Brief theory of ODEs 261
12.2 Numerical methods and stability 264
12.3 Multistep methods 271
12.4 Multistage: Runge–Kutta method 278
12.5 Nonlinear ODEs 283
12.6 Stiff equations 284
12.7 Splitting methods 286
12.8 Symplectic integrators 291
12.9 Theory: consistency, stability, convergence 293

13 Parabolic Equations **299**
13.1 Method of lines 301
13.2 Stability: von Neumann analysis 305
13.3 Higher-dimensional methods 310
13.4 Nonlinear diffusion equation 313

14 Hyperbolic Equations **321**

14.1 Linear hyperbolic equations 322
14.2 Methods for linear hyperbolic equations 323
14.3 Numerical diffusion and numerical dispersion 329
14.4 Linear hyperbolic systems 331
14.5 Nonlinear hyperbolic equations 333
14.6 Hyperbolic systems of conservation laws 341
14.7 Methods for nonlinear hyperbolic systems 347

15 Finite Element Method 355
15.1 A one-dimensional example 355
15.2 A two-dimensional example 359
15.3 Stability and convergence 364
15.4 Time-dependent problems 366

16 Fourier Spectral Methods 371
16.1 Discrete Fourier transform 371
16.2 Nonlinear stiff PDES 377
16.3 Incompressible Navier–Stokes equation 380

Exercise Solutions 389

Linear Algebra 389

Analysis 399

Partial Differential Equations 415

References 439

Index 445

Preface

This book is a compilation of lecture notes that I developed and used for a sequence of three numerical methods courses taught in the Department of Mathematics and Statistics at the Air Force Institute of Technology. The three courses Numerical Linear Algebra, Numerical Analysis, and Numerical Methods for Partial Differential Equations were taken primarily by mathematics, physics, and engineering masters and PhD students. Each course was designed to be independent of the other two courses, so there was some repetition, essential to provide preliminary background and guide special topics. The goal of the courses was to present some of the foundational principles and essential tools of scientific computing in an informal manner, provide practical application of the principles being taught, and generate interest in the topic. When exacting mathematical rigor hindered developing mathematical intuition, I would typically favor intuition. There are many other great texts in numerical methods to fill in the details. These notes were themselves influenced by lectures from the a two sequence numerical analysis course taught by my doctoral advisor Shi Jin at the University of Wisconsin–Madison.

The purpose of my lecture notes was to first guide lectures and discussion and secondly 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 to explain the principles in an intuitive manner. Simple picture proofs and explanations were preferred over more rigorous but abtruse analytic derivations. Student who wanted greater rigor could find it in other referenced numerical mathematics textbooks.

This book is divided into three parts, each itself is a self-contained course.

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 operator $\mathbf{A}$ and vector $\mathbf{b}$, find vector $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$.

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

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

where is the $f(x, y)$ is a 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 largely left up to a black box. Part One of this book breaks open that black box.

Numerically, one can solve problem $\mathbf{Ax} = \mathbf{b}$ either directly or iteratively. The basic direct solver is Gaussian elimination. We can get more efficient methods such as banded solvers, block solvers and Cholesky decomposition by exploiting properties of the matrix such as low bandwidth and symmetry. Chapter 2 looks at direct methods. Large, sparse matrices often arise in the numerical methods for partial differential equations. Such systems can be effectively solved 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, and SOR, and Krylov methods such as the conjugate gradient method. It also examines multigrid methods.

In some cases, the 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.

The second fundamental problem of linear algebra can be stated as

2. Find the scalar λ and the vector $\mathbf{x}$ given $\mathbf{Ax} = \lambda\mathbf{x}$.

The eigenvalue problem arises in the study of stability, steady-state behavior, and ordinary differential equations. In general, it is impossible to solve the eigenvalue problem directly and we must instead use iterative methods. Chapter 4 takes a close look at how this is done.

Finally, Chapter 6 looks briefly at the fast Fourier transform which has revolutionized computational mathematics.

Numerical methods for analysis

Numerical analysis develops tools and methods of problem solving in such a manner that they can be implemented *efficiently* and *accurately* using a computer. Because of this, numerical analysis can be studied from a rather theoretical framework, heavy in functional analysis. But numerical analysis can also be studied by looking at its motivation in solving physical problems, an approach often called scientific computing. And, 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 a finite dimension, 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. It also discusses the nonlinear least-squares problem. Chapter 10 looks at methods of approximating whole functions using basis functions such as Chebyshev polynomials, Fourier polynomials, and radial basis functions. Finally, Chapter 11 provides a brief overview of numerical differentiation and integration.

Numerical methods for partial differential equations

Part Three examines three classifications of partial differential equations (PDEs): parabolic, hyperbolic and elliptic.

Parabolic equations include equations such as 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 describe 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 describe 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. To handle these equations mathematically, an alternative weak solution is needed. 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 Laplace's equation and Poisson's equation. Unlike parabolic and hyperbolic equations which are typically time-dependent, elliptic equations are usually 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 which describes fluid flow is predominantly diffusive on small length and time scales but it is predominately hyperbolic on large scales. The melting of ice occurs only at 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 some consideration.

Part Three examines three main numerical tools for solving PDEs: 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 as a means of solving problems that finite difference methods are not well adapted to such as handling complicated boundaries. Hence, finite element methods are attractive solutions to engineering problems. Finite element methods often require much more numerical and mathematical machinery than finite difference methods.

Fourier spectral methods were also developed in the 1960s with the development of the fast Fourier transform (FFT). The FFT allows efficient employment of Fourier spectral solutions. They are important in fluid modeling and analysis where boundary effects are not important and can be assumed 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 said "the only way to learn mathematics is to do mathematics." Doing mathematics involves visualizing complex data structures, thinking logically and creatively, gaining conceptual understanding and insight. Doing mathematics is about problem solving and pattern recognition and forging the connections between seemingly disparate fields. It is recognizing that the same class 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 have the familiarization, technique and confidence in order to start doing mathematics. Each chapter concludes with a set of problems. Solutions to problems marked with a ✍ are provided in Part Four.

Not surprisingly, programming plays a key role in scientific computing. This book focuses on MATLAB and the equivalent open source Octave language because of the relatively straightforward syntax when dealing with mathematical expressions. For example, to solve the the linear system $\mathbf{Ax} = \mathbf{b}$ in MATLAB we simply write `A\b`, and to find an eigenvalue we just as simply write `eig(A)`. Moreover, MATLAB uses efficient numerical algorithms. One of the goals in this book will be to understand the mathematical what and why of MATLAB and determine what can go wrong.

Of course, there are other numerical software packages such as Python (SciPy and NumPy), R, Julia, Scilab,C++, and Fortran. With little modification to the code, all of the algorithms developed in this book can

be implemented using other packages. MATLAB, Octave, Python, and R all shares similar libraries such as LAPACK. Resources such as the wiki-based website rosettacode.org compare various languages to implement different algorithms. Additionally, the thesaurus of mathematical languages located online at mathesaurus.sourceforge.net provides a quick reference for converting between MATLAB, Python, and R.

Kyle A. Novak
Washington, D.C.
December 2016

About the author

Kyle Novak has over twenty years of research and analysis experience in modeling, applied mathematics, and scientific computing and visualization including teaching mathematics at the undergraduate and graduate levels. As an Air Force mathematician, he provided decision analysis to senior military leaders in diverse topics such as cryptanalysis, autonomous navigation systems, multiscale networks, and superconducting materials.

Numerical Methods for Linear Algebra

CHAPTER 1

Review: Theory and Definitions

We'll start with a review of concepts from elementary linear algebra and introduce a few important definitions as needed. The review is rather cursory—see other texts such as Lax [2007], Strang [2005c], Watkins [2002] for the missing details.

1.1 Vector spaces

A *vector space* is a nonempty set V which is closed under linear combinations (addition and scalar multiplication over a field $\mathbb{F}$) of its elements called *vectors*. Typically, the field is the real numbers $\mathbb{R}$ or the complex numbers $\mathbb{C}$. Examples include

- $V = \mathbb{R}^n$ (or $V = \mathbb{C}^n$);
- $V = \mathbb{P}_n$: the set of polynomials $p_n = \sum_{k=0}^{n} a_k x^k$ where $a \in \mathbb{R}$ (or $a \in \mathbb{C}$);
- the space of piecewise linear functions on $[a, b]$, which is important for the finite-element method;
- cubic splines for a given set of nodes;
- $V = \mathrm{GF}(p^n)$, Galois field with prime characteristic p with n terms. For example, $\mathrm{GF}(2^6)$ gives byte xor arithmetic,

$$\{011011\} \text{ xor } \{001101\} = \{010110\}.$$

A nonempty set $W \subset V$ is a *subspace* if and only if W is a vector space over $\mathbb{F}$ ($\mathbb{R}$ or $\mathbb{C}$). Consider a system of n vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\} \in V$.

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

$$W = \text{span}\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}.$$

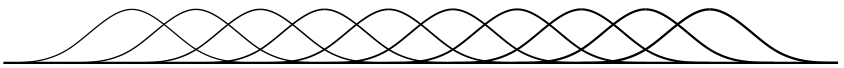
The system $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$ is called the *spanning set* or the *generators* of W.

The system of vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$ of a vector space V is called *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 $a_1, a_2, \ldots, a_n$ are identically zero: $a_1 = a_2 = \cdots = a_n = 0$. Otherwise, we say that the system is *linearly dependent*.

A *basis* of V is a system of linear independent generators of V, i.e., linear independent vectors that span V. For example, the monomials $\{1, x, x^2, \cdots, x^k\}$ are a basis for the space of kth-order polynomials, and B-splines are a basis for the space of splines:



The number of elements in a basis of a vector space V is called the *dimension* of V. A vector space is called infinite dimensional, if for any n, there always exists linearly independent vectors of V. If $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$ 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 given in matrix representation as the vector

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

where the basis is implicitly understood. Hence, any n-dimensional vector space is isomorphic to $\mathbb{R}^n$ (or $\mathbb{C}^n$). The standard basis of $\mathbb{R}^n$ is $\{\boldsymbol{\xi}_1, \boldsymbol{\xi}_2, \cdots, \boldsymbol{\xi}_n\}$ where

$$\boldsymbol{\xi}_i = (\underbrace{0, \ldots, 0}_{i-1}, 1, 0, \ldots, 0).$$

Example. It should be emphasized that the choice of a basis is not unique, although 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 are 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 is 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 $\mathbb{R}^3$. For example, the vector $1 + x^2$ is isomorphic to $(1, 0, 1)$ given the basis $\{1, x, x^2\}$. The same vector $1 + x^2$ is isomorphic to $(\frac{4}{3}, 0, \frac{2}{3})$ given the Legendre basis $\{1, x, \frac{1}{2}(3x^2 - 1)\}$. ◀

1.2 Matrices

A *linear map* from $\mathbb{C}^n$ into $\mathbb{C}^m$ is a function $f : \mathbb{C}^n \to \mathbb{C}^m$ 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 $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$. For any linear map f, there exists a unique matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$ such that $f(\mathbf{y}) = \mathbf{A}\mathbf{x}$ for all $\mathbf{x} \in \mathbb{C}^n$.

Every n-dimensional vector space is isomorphic (equivalent) to $\mathbb{R}^n$. Furthermore, every linear map from an n-dimensional vector space into an m-dimensional vector space can be represented by an $m \times n$ matrix.

Example. Consider the derivative operator defined over the space of quadratic polynomials ($\frac{\mathrm{d}}{\mathrm{d}x} : \mathbb{P}_2 \mapsto \mathbb{P}_2$). It is easy to check 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 $\frac{\mathrm{d}}{\mathrm{d}x}(a + bx + cx^2) = b + 2cx$. It follows from

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

for quadratic polynomials using the monomial basis. ◀

The *transpose* $\mathbf{B} = \mathbf{A}^T$ of a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ is the $n \times m$ matrix obtained by exchanging the rows of $\mathbf{A}$ with columns of $\mathbf{A}$. That is, $b_{ij} = a_{ji}$. Note that

$$(\mathbf{A}^T)^T = \mathbf{A}, \qquad (\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T,$$
$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T, \qquad (\mathbf{A}^{-1})^T = (\mathbf{A}^T)^{-1}.$$

The *conjugate transpose* (or *adjoint*) $\mathbf{B} = \mathbf{A}^H$ 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 $\mathbf{A}$. That is, $b_{ij} = \overline{a}_{ji}$.

Ⓜ In MATLAB the conjugate transpose of `A` is produced by `A'`.

Ⓜ The MATLAB data structure for an array is by columns. (Fortran is also by columns, but C is by rows.) Using `A(:,j)` in MATLAB is about three times faster than using `A(i,:)` for large arrays.

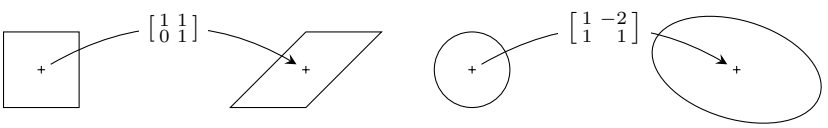
Consider a matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$. The *column space* (or range) of $\mathbf{A}$ is the subspace of $\mathbb{C}^n$ generated by the columns of $\mathbf{A}$. The dimension of the column space $\mathbf{A}$ equals the dimension of the rows space of $\mathbf{A}$ and is called the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

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

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

Example. Consider the derivative operator over the space of quadratic polynomials (using the monomial basis $\{1, x, x^2\}$ again). The derivative operator maps constants to zero, so the nullspace 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 $\text{span}\{(0, 0, 1)\}$. ◀

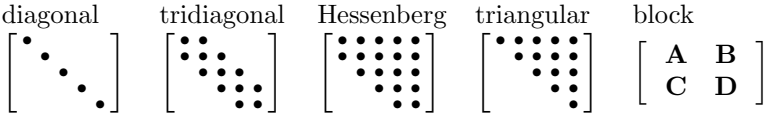
Several important types of matrices are listed on the facing page. By considering the vector space $\mathbb{R}^2$, we have a nice geometric interpretation of linear maps. A circle (centered at the origin) is mapped to an ellipse (also centered at the origin). In higher dimensions, a sphere is mapped to an ellipsoid. Similarly, a square is mapped to a parallelogram and a cube is mapped to a parallelipid.



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 such that $\mathbf{x}^H\mathbf{A}\mathbf{x} > 0$ for any non-zero vector $\mathbf{x}$.
- *Orthogonal*: $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$.
- *Unitary*: $\mathbf{Q}^H\mathbf{Q} = \mathbf{I}$. 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 $\mathbf{P}^T = \mathbf{P}$. A projection matrix maps a vector into a subspace but does not change an vectors already in that subspace. We use a subscript to denote the column space of $\mathbf{P}$. An orthogonal projection matrix $\mathbf{P}_A = \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ maps any vector orthogonal to the column space of $\mathbf{A}$ to the zero vector. That is, if $\mathbf{A}^T\mathbf{B} = \mathbf{0}$ then $\mathbf{P}\mathbf{B} = \mathbf{0}$.
- *Diagonal*: $a_{ij} = 0$ for $i \neq j$. We denote it: $\mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn})$.
- *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 us called the *bandwidth*.
- *Upper Hessenberg*: $a_{ij} = 0$ for $i > j + 1$.

diagonal tridiagonal upper Hessenberg upper triangular block



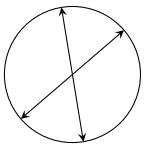
1.3 Eigenspaces

The number $\lambda \in \mathbb{C}$ is called an *eigenvalue* of a square matrix $\mathbf{A}$ if there exists a nonzero vector $\mathbf{x}$ called the *eigenvector* such that $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$. A subspace S is called *invariant* with respect to a square matrix $\mathbf{A}$ if $\mathbf{A}S \subset S$. Eigenvectors are directionally invariant. The set of eigenvalues of $\mathbf{A}$ is the *spectrum* of $\mathbf{A}$ and is denoted by $\lambda(\mathbf{A})$. The maximum of the absolute values of the eigenvalues is the *spectral radius* of $\mathbf{A}$:

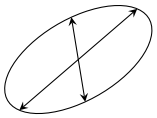
$$\rho(\mathbf{A}) = \max_{\lambda \in \lambda(\mathbf{A})} |\lambda|.$$

The set of eigenvectors (including the zero vector) associated with an eigenvector λ is called the *eigenspace*. The dimension of the eigenspace is called the *geometric multiplicity* of λ. The multiplicity of the eigenvalue in the characteristic polynomial is called the *algebraic multiplicity*.

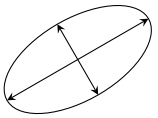
Eigenvectors have a simple geometric interpretation. Let's take $\mathbf{R}^2$. Start with two unit eigenvectors for $\mathbf{A}$. They fit in a unit circle. The matrix $\mathbf{A}$ will stretch the eigenvectors by their respective eigenvalues. Our unit circle becomes an ellipse. Singular vectors lie along the semi-major and semi-minor axes. If the eigenvectors where perpendicular (as they are for a normal matrix), then the eigenvectors would be singular vectors and the eigenvalues would be singular values.



unit eigenvectors



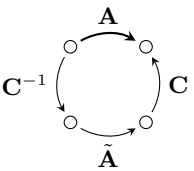
stretched by eigenvalues



singular vectors

Similar matrices

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



If $\mathbf{C}$ is a unitary matrix, we say that $\tilde{\mathbf{A}}$ and $\mathbf{A}$ are *unitarily similar*. Similarity transformations can simplify a problem by transforming a matrix into a diagonal or triangular matrix. It is often faster to take two

detours and hook up with a superhighway than it is to take the direct route.

Diagonalization

A matrix that has a complete set of eigenvectors is said to be *semisimple*. Otherwise, it is *defective*. For semisimple matrices, we can rewrite the eigenvalue problem $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ as

$$\mathbf{A}\mathbf{S} = \mathbf{S}\mathbf{\Lambda}$$

where $\mathbf{S}$ is a matrix of the eigenvectors and $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues. This formulation leads to the *diagonalization* of $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1},$$

which says that $\mathbf{A}$ is similar to the diagonal matrix $\mathbf{\Lambda}$ in its eigenvector basis. The system is uncoupled in this eigenvector basis and much easier to manipulate.

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

$$\begin{aligned} f(\mathbf{A}) &= \sum_{k=0}^{\infty} c_k \mathbf{A}^k \\ &= \sum_{k=0}^{\infty} c_k \left(\mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}\right)^k = \sum_{k=0}^{\infty} c_k \mathbf{S}\mathbf{\Lambda}^k\mathbf{S}^{-1} = \mathbf{S}\left(\sum_{k=0}^{\infty} c_k \mathbf{\Lambda}^k\right)\mathbf{S}^{-1} \\ &= \mathbf{S}\,\mathrm{diag}(f(\lambda_1), f(\lambda_2), \ldots, f(\lambda_n))\mathbf{S}^{-1}. \end{aligned}$$

Similarly, we can evaluate the solution to the system of differential equations $\mathbf{y}' = \mathbf{A}\mathbf{y}$ as $\mathbf{y}(t) = \exp(\mathbf{A}t)\mathbf{y}(0)$ by solving the simple problem $\tilde{\mathbf{y}}' = \mathbf{\Lambda}\tilde{\mathbf{y}}$. ◀

Schur decomposition

Theorem 1 (Schur Decomposition Theorem)**.** *Every square matrix is unitarily similar to an upper-triangular matrix. In other words, for any* $\mathbf{A} \in \mathbb{C}^{n\times n}$*, 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. When $n = 1$, the hypothesis is clearly true: $\mathbf{A} = \lambda$ is already an upper triangular 1×1 matrix with $\mathbf{U} = 1$. Now, assume that the hypothesis holds for $n - 1$ and show that it also holds for n. Let λ be an eigenvalue of $\mathbf{A}$ and $\mathbf{v}$ be the associated eigenvector with $\|\mathbf{v}\|_2 = 1$. Let $\mathbf{U}_1$ be a unitary matrix whose first column is $\mathbf{v}$:

$$\mathbf{U}_1 = \begin{bmatrix} \mathbf{v} & \mathbf{W} \end{bmatrix}$$

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

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

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

$$\mathbf{U}_2 = \left[\begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & \hat{\mathbf{U}}_1 & \\ 0 & & & \end{array}\right].$$

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

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

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

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

By the Schur Decomposition Theorem, every square matrix is unitarily similar to an upper-triangular matrix: $\mathbf{T} = \mathbf{U}^H\mathbf{A}\mathbf{U}$. It follows that the diagonal elements of $\mathbf{T}$ are the eigenvalues of $\mathbf{A}$. The spectral decomposition theorem is a direct consequence of Schur's theorem.

Theorem 2 (Spectral Theorem)**.** *Suppose that* $\mathbf{A} \in \mathbb{C}^{n\times n}$ *is a normal matrix* ($\mathbf{A}^H\mathbf{A} = \mathbf{A}\mathbf{A}^H$)*. Then* $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^H$ *where* $\mathbf{U} \in \mathbb{C}^{n\times n}$ *is a unitary matrix whose columns are eigenvalues of* $\mathbf{A}$ *and* $\mathbf{\Lambda}$ *is a diagonal matrix whose diagonal elements are the eigenvalues of* $\mathbf{A}$*. Furthermore, the eigenvalues of a Hermitian matrix are real.*

Proof. As a consequence of Shur's Theorem, $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H$ where $\mathbf{T}$ is an upper triangular matrix and $\mathbf{U}$ is a unitary matrix. It follows that

$$\begin{aligned}\mathbf{A}^H\mathbf{A} &= \mathbf{U}\mathbf{T}^H\mathbf{U}^H\mathbf{U}\mathbf{T}\mathbf{U}^H = \mathbf{U}\mathbf{T}^H\mathbf{T}\mathbf{U}^H\\ \mathbf{A}\mathbf{A}^H &= \mathbf{U}\mathbf{T}\mathbf{U}^H\mathbf{U}\mathbf{T}^H\mathbf{U}^H = \mathbf{U}\mathbf{T}\mathbf{T}^H\mathbf{U}^H\end{aligned}$$

Hence, $\mathbf{T}^H\mathbf{T} = \mathbf{T}\mathbf{T}^H$ looks like

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

Let's look just at the diagonal elements of the product $\mathbf{T}^H\mathbf{T} = \mathbf{T}\mathbf{T}^H$. Starting with the $(1,1)$-element:

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

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

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

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

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

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

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

Continuing like this, it follows that $\mathbf{T}$ is a diagonal matrix. End of proof.

Let's reconsider the case when $\mathbf{A}$ is a Hermitian matrix. Although the above proof is valid, it is useful to look at another proof. As a consequence of Shur's Theorem, $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^H$ where $\mathbf{T}$ is an upper triangular matrix and $\mathbf{U}$ is a unitary matrix. It follows from $\mathbf{A} = \mathbf{A}^H$ that

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

Hence, $\mathbf{T} = \mathbf{T}^H$. So, $\mathbf{T}$ is a diagonal matrix. Furthermore, because the diagonal elements of $\mathbf{T}$ are eigenvalues λ, it follows that $\lambda = \overline{\lambda}$ and hence the eigenvalues of a Hermitian matrix are real. □

In other words, a Hermitian matrix $\mathbf{A}$ is unitarily similar to a diagonal matrix $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^H$ where $\mathbf{Q}$ is a unitary matrix of the eigenvectors. Likewise, the $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$ for a (real) symmetrc matrix $\mathbf{A}$. 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 $\mathbf{q}_k \mathbf{q}_k^H$ are an orthogonal projection matrices for unit eigenvectors $\mathbf{q}_k$. This decomposition is known as *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 arise in application as the discrete approximation to the Laplacian and as convolution operators. They are unitarily similar to diagonal matrices $\mathbf{\Lambda} = \mathbf{F}\mathbf{C}\mathbf{F}^H$ 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 $\mathbf{\Lambda}$. We solve the simpler problem and then perform another change of basis. With the invention of the fast Fourier transform, the two changes of the bases $\mathbf{F}$ and $\mathbf{F}^H$ are a relatively quick steps. ◀

Singular value decomposition

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

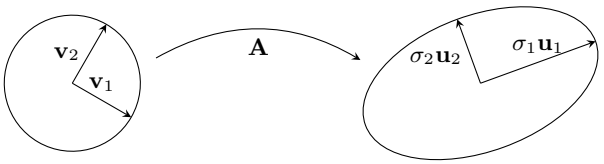
$$\mathbf{U}^H \mathbf{A} \mathbf{V} = \mathbf{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_p)$$

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

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

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

Singular value decomposition has an intuitive geometric interpretation: $\mathbf{AV} = \mathbf{U\Sigma}$. Any matrix $\mathbf{A}$ maps a sphere $\mathbf{V}$ 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.



1.4 Measuring vectors and matrices

Inner products

An *inner product* on the vector space V is any map $(\cdot,\cdot)$ from $V \times V$ into $\mathbb{C}$ with the properties:

1. it is linear with respect to the vectors of V

$$(a\mathbf{u} + b\mathbf{v}, \mathbf{w}) = a\,(\mathbf{u}, \mathbf{w}) + b\,(\mathbf{v}, \mathbf{w}) \text{ for all } \mathbf{u}, \mathbf{v}, \mathbf{w} \in V \text{ and } a, b \in \mathbb{C};$$

2. it is Hermitian

$$(\mathbf{u}, \mathbf{v}) = \overline{(\mathbf{v}, \mathbf{u})};$$

3. it is positive definite

$$(\mathbf{u}, \mathbf{u}) \geq 0 \quad \text{and} \quad (\mathbf{u}, \mathbf{u}) = 0 \text{ iff } \mathbf{u} = 0.$$

Two vectors $\mathbf{u}$ and $\mathbf{v}$ are *orthogonal* if $(\mathbf{u}, \mathbf{v}) = 0$. Two important examples of inner products include the *Euclidean inner product*

$$(\mathbf{u}, \mathbf{v})_2 = \mathbf{u}^T\mathbf{v}$$

and the *energy inner product*

$$(\mathbf{u}, \mathbf{v})_{\mathbf{A}} = \mathbf{u}^T\mathbf{A}\mathbf{v}$$

where $\mathbf{A}$ is symmetric, positive definite.

Vector norms

A *vector norm* on the vector space V is any map $\|\cdot\|$ from V into $\mathbb{R}$ with the properties:

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

Important norms include

- the *Hölder norm* (l^p)

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

- the l^1*-norm* ($p = 1$):

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

- the *Euclidean norm* ($p = 2$):

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

- the l^∞*-norm* ($p \to \infty$)

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

- and the *energy norm*:

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

First note that all vector norms satisfy the *Cauchy–Schwarz inequality* $(\mathbf{x}, \mathbf{y})_2 \leq \|\mathbf{x}\|_2\|\mathbf{y}\|_2$. To show this, note that for any α

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

By taking $\alpha = (\mathbf{x}, \mathbf{y})\,/\|\mathbf{y}\|^2$ it follows that

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

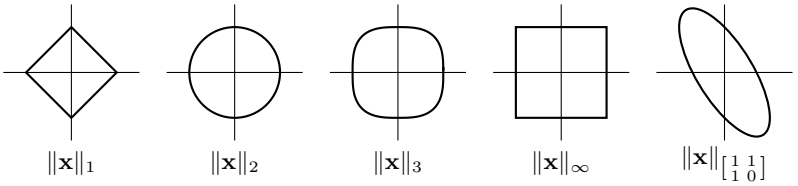


Figure 1.1: Unit circles in different norms on $\mathbb{R}^2$.

Also note that the Euclidean norm is invariant under orthogonal transformations $\|\mathbf{Qx}\|_2 = \|\mathbf{x}\|_2$. This result is clear from the geometric interpretation of the Euclidean norm and orthogonal transformation. But you can cross check it with simple algebra:

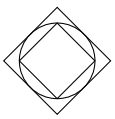
$$\|\mathbf{Qx}\|_2^2 = \|\mathbf{x}^T\mathbf{Q}^T\mathbf{Qx}\|_2 = \|\mathbf{x}^T\mathbf{x}\|_2 = \|\mathbf{x}\|^2.$$

Two norms $\|\cdot\|_\alpha$ and $\|\cdot\|_\beta$ on a vector space V are *equivalent* if there exist constants $0 < c, C < \infty$ 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 may often choose a convenient norm with which to work and the results will hold with respect to the other norms.

Example. Let's show the equivalence of the l^1- and l^2-norms.



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

Now, let's show the equivalence of the l^1- and l^∞-norms.

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

Matrix norms

A *matrix norm* is map $\|\cdot\|$ from $\mathbb{R}^{m\times n}$ into $\mathbb{R}$ with the properties:

1. $\|\mathbf{A}\| \geq 0$ and $\|\mathbf{A}\| = 0$ if and only if $\mathbf{A} = 0$ (positivity)
2. $\|\alpha\mathbf{A}\| = |\alpha|\|\mathbf{A}\|$ (homogeneity)

3. $\|\mathbf{A}+\mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$ (triangle inequality)

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

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

A matrix norm $\|\cdot\|$ is said to be *compatible* or *consistent* with a vector norm $\|\cdot\|$ if

$$\|\mathbf{Ax}\| \leq \|\mathbf{A}\|\|\mathbf{x}\| \quad \text{for all } \mathbf{x} \in \mathbb{R}^n.$$

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*:

$$\|\mathbf{A}\| = \sup_{\mathbf{x}\neq 0} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} = \sup_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\|. \tag{1.1}$$

The induced norm gives the most that a matrix can stretch a vector (in a vector norm).

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

Proof.

1. $\|\mathbf{Ax}\| = \dfrac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|}\|\mathbf{x}\| \leq \left(\sup\limits_{\mathbf{x}\neq 0} \dfrac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|}\right)\|\mathbf{x}\| = |||\mathbf{A}|||\|\mathbf{x}\|$

2. $|||\mathbf{I}||| = \sup\limits_{\|\mathbf{x}\|=1} \|\mathbf{x}\| = 1$

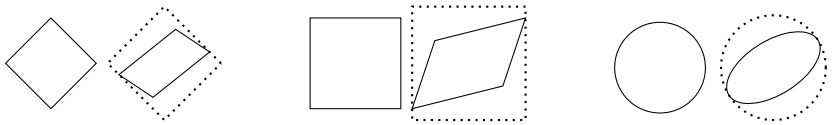
3. $|||\mathbf{AB}||| = \sup\limits_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\| \leq \sup\limits_{\|\mathbf{x}\|=1} |||\mathbf{A}|||\|\mathbf{Bx}\| = |||\mathbf{A}|||\,|||\mathbf{B}|||$ □

Important induced matrix norms include

- *l^p-norms*

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

- *l^1-norm* ($p = 1$). Consider the unit ball for the l^1-norm. The square ($\mathbf{x}$) is mapped to a parallelogram ($\mathbf{Ax}$) under a linear transformation ($\mathbf{A}$). The sup $\|\mathbf{Ax}\|_1$ must come from one of the vertices of the of the parallelogram. See Figure 1.2a on the next page. The vertices of the parallelogram are mapped from the corners of the square which corresponding to one of the standard basis elements $\pm\boldsymbol{\xi}_j$. For example, in three-dimensions $(\pm 1, 0, 0)$, $(0, \pm 1, 0)$



(a) l^1 unit ball (b) l^∞ unit ball (c) l^2-unit ball

Figure 1.2: Mapping of unit balls by $\frac{1}{4}\left[\begin{smallmatrix}1 & 1\\ 1 & 3\end{smallmatrix}\right]$.

or $(0, 0, \pm 1)$. Then $\mathbf{A}\boldsymbol{\xi}_j$ returns (plus or minus) the jth column of $\mathbf{A}$. Therefore,

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

- *l^∞-norm* ($p \to \infty$). Consider the unit ball for the l^∞-norm. As before the square ($\mathbf{x}$) is mapped to a parallelogram ($\mathbf{Ax}$) under a linear transformation ($\mathbf{A}$) and the sup $\|\mathbf{Ax}\|_1$ must come originally from the corners of the square. See Figure 1.2b. This time the corners are at $(\pm 1, \pm 1, \ldots, \pm 1)$.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} \pm 1 \\ \pm 1 \\ \pm 1 \end{bmatrix} = \begin{bmatrix} \pm a \pm b \pm c \\ \pm d \pm e \pm f \\ \pm g \pm h \pm i \end{bmatrix}.$$

  By taking the maximum largest element, we have

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

- *l^2-norm* or the *spectral norm* ($p = 2$). A circle ($\mathbf{x}$) is mapped to an ellipse ($\mathbf{Ax}$) and sup $\|\mathbf{Ax}\|_2$ corresponds to the largest radius of the ellipse. See Figure 1.2c. So,

$$\|\mathbf{A}\|_2 = \sigma_{\max}(\mathbf{A}).$$

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

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

Theorem 4. *$\|\mathbf{A}\|_2$ equals the largest singular value of $\mathbf{A}$. Furthermore, 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 an easy picture proof given in Figure 1.2c on the preceding page. A circle is mapped to an ellipse, and the induce l^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 that picture proof, here's an algebraic proof. $\mathbf{A}^H\mathbf{A}$ is hermitian, so there exists a unitary matrix $\mathbf{U}$ such that

$$\mathbf{U}^H\mathbf{A}^H\mathbf{A}\mathbf{U} = \operatorname{diag}(\mu_1, \dots \mu_n)$$

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

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

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

Theorem 5. *$\rho(\mathbf{A}) \le \|\mathbf{A}\|$ where $\|\cdot\|$ is an induced norm.*

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

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

Theorem 6. *Furthermore, for symmetric matrices, $\|\mathbf{A}\|_{max} \le \rho(\mathbf{A})$ where the max norm $\|\mathbf{A}\|_{max} = \max_{i,j} |a_{ij}|$.*

Proof. Keep in mind that a linear transformation maps a circle to an ellipse. A standard basis vector $\boldsymbol{\xi}_j$ is mapped to the jth column of $\mathbf{A}$. So, $\|\mathbf{A}\|_{\max}$ must be equal to or smaller than the largest component $\mathbf{A}\boldsymbol{\xi}_j$ for some $\boldsymbol{\xi}_j$. The largest component of $\mathbf{A}\boldsymbol{\xi}_j$ must be equal to or smaller than the length of $\mathbf{A}\boldsymbol{\xi}_j$. So, $\|\mathbf{A}\|_{\max} \le \max_j \|\mathbf{A}\boldsymbol{\xi}_j\|_1 \le \sigma_{\max}(\mathbf{A})$. □

Example. We can find a 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 $\rho(\mathbf{A}) \approx 10.03$. ◀

Quadratic forms

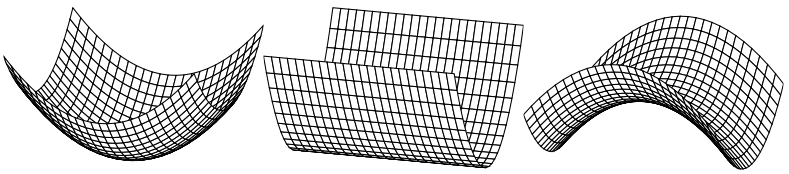
The quadratic form of a real, symmetric $n \times n$ matrix $\mathbf{A}$ is defined as $q(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x}$. In component form,

$$q(x_1, x_2, \ldots, x_n) = \sum_{i,j=1}^{n} a_{ij}x_ix_j$$

In the case of a 2×2 matrix,

$$\begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = ax^2 + 2bxy + cy^2.$$

A matrix $\mathbf{A}$ is said to be *positive definite*, if $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq 0$. Likewise, a matrix $\mathbf{A}$ is said to be negative definite, if $\mathbf{x}^T\mathbf{A}\mathbf{x} < 0$ for all $\mathbf{x} \neq 0$. A matrix $\mathbf{A}$ is positive semidefinite if $\mathbf{x}^T\mathbf{A}\mathbf{x} \geq 0$ and negative semidefinite if $\mathbf{x}^T\mathbf{A}\mathbf{x} \leq 0$. Otherwise, the matrix is indefinite. Quadratic forms $q(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x}$:



Positive Definite Positive Semidefinite Indefinite

Theorem 7. *A symmetric matrix is positive-definite if and only if it has only positive eigenvalues.*

Proof. Suppose that $\mathbf{A}$ is positive-definite. Then $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for any vector $\mathbf{x}$. If $\mathbf{x}$ is an eigenvector, $\mathbf{x}^T\mathbf{A}\mathbf{x} = \mathbf{x}^T\lambda\mathbf{x} = \lambda\|\mathbf{x}\|^2$. It follows that $\lambda > 0$.

On the other hand, suppose that $\mathbf{A}$ has only positive eigenvalues λ_i. Because $\mathbf{A}$ is symmetric, any vector $\mathbf{x}$ can be written as a linear combinations of orthogonal eigenvectors $\mathbf{x} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n$. Then

$$\begin{aligned}\mathbf{x}^T\mathbf{A}\mathbf{x} &= (c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n)^T (c_1\lambda_1\mathbf{v}_1 + c_2\lambda_2\mathbf{v}_2 + \cdots + c_n\lambda_n\mathbf{v}_n) \\ &= c_1^2\lambda_1\|\mathbf{v}_1\|^2 + c_2^2\lambda_2\|\mathbf{v}_2\|^2 + \cdots + c_n^2\lambda_n\|\mathbf{v}_n\|^2\end{aligned}$$

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

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

$$q_B(\mathbf{x}) = \mathbf{x}^T\mathbf{B}\mathbf{x} = \mathbf{x}^T\mathbf{S}^T\mathbf{A}\mathbf{S}\mathbf{x} = (\mathbf{S}\mathbf{x})^T\mathbf{A}\mathbf{S}\mathbf{S} = \mathbf{y}^T\mathbf{A}\mathbf{y} = q_A(\mathbf{y})$$

for $\mathbf{y} = \mathbf{S}\mathbf{x}$. Because the eigenvectors of a symmetric matrices are orthogonal, we can express the quadratic form in terms of the diagonal matrix of eigenvalues in the eigenvector basis: $q(x_1, x_2, \ldots, x_n) = \sum_{i=1}^n \lambda_i x_i^2$.

1.5 Well-posedness

In engineering, the sciences and mathematics, it is convenient to solve a problem by using a black box solution, by letting MATLAB for example take care of the tedious computation. Before questioning the the legitimacy of the black box, how do we know whether a problem has a meaningful solution? The problem may have no solution. The problem may have multiple solutions. Perhaps the problem is so sensitive to change that black box cannot replicate the output if the input changes by the even the slightest amount. French mathematician Jacques Hadamard introduced the notion of mathematical well-posedness in 1923. A problem is called *well-posed* if

1. a solution for the problem exists (existence);
2. the solution is unique (uniqueness); and
3. this solution is stable under small perturbations in the data (stability).

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

Stability

Let $\mathbf{b} = \mathbf{A}\mathbf{x}$. Suppose that we perturb the input data $\mathbf{x}$ by a vector $\delta\mathbf{x}$, perhaps due to round-off error or errors in measurement.

1. How does this perturbation affect our numerical computation of $\mathbf{b}$?
2. Alternatively, how does the perturbation $\mathbf{A} + \delta\mathbf{A}$ affect our computation?
3. Or, if we are solving $\mathbf{A}\mathbf{x} = \mathbf{b}$, what is the effect of error in $\mathbf{b}$?

As we will see, 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

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

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

$$\|\delta\mathbf{b}\| = \|\mathbf{A}\delta\mathbf{x}\| \leq \|\mathbf{A}\|\|\delta\mathbf{x}\|. \tag{1.2}$$

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

$$\|\mathbf{x}\| = \|\mathbf{A}^{-1}\mathbf{b}\| \leq \|\mathbf{A}^{-1}\|\|\mathbf{b}\|. \tag{1.3}$$

Combining (1.2) and (1.3),

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

The factor $\|\mathbf{A}^{-1}\|\|\mathbf{A}\|$ is called the *condition number* of the matrix $\mathbf{A}$ and is denoted as $\kappa(\mathbf{A})$. The condition number of the l^p-norm of $\mathbf{A}$ is denoted as $\kappa_p(\mathbf{A})$. When the condition number is small, a small perturbation $\delta\mathbf{x}$ on the $\mathbf{x}$ 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

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

it follows that

$$\kappa(\mathbf{A}) = \frac{\sup\limits_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|}{\inf\limits_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|}.$$

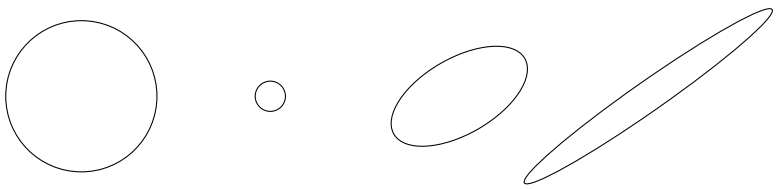
Therefore, a geometric interpretation of the condition number is

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

and the condition number in the Euclidean norm is

$$\kappa_2(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}} = \frac{|\text{semi-major axis}|}{|\text{semi-minor axis}|}.$$

The condition number is the ratio of radii of the semi-major to semi-minor axes of an ellipsoid image of a matrix:



$\kappa_2 = 1$ $\kappa_2 = 1$ $\kappa_2 = 2$ $\kappa_2 = 10$

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 very appealing, because multiplying by them does not introduce numerical instability.

Now, let's consider the second question: how does the perturbation $\mathbf{A} + \delta\mathbf{A}$ affect our computation? Consider the singular value decomposition of $\mathbf{\Sigma} = \mathbf{U}^H\mathbf{A}\mathbf{V}$. Perturbation of $\mathbf{A}$ says

$$\mathbf{U}^H(\mathbf{A} + \delta\mathbf{A})\mathbf{V} = \mathbf{\Sigma} + \delta\mathbf{\Sigma}$$

and hence

$$\mathbf{U}^H\delta\mathbf{A}\mathbf{V} = \delta\mathbf{\Sigma}$$

Because $\mathbf{U}$ and $\mathbf{V}$ are unitary, they are l^2-norm preserving, so

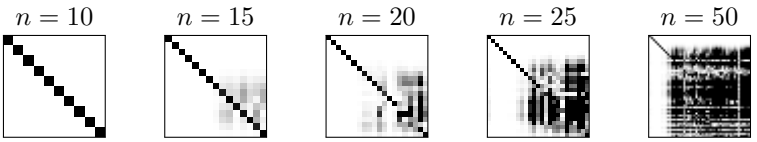
$$\|\delta\mathbf{A}\|_2 = \|\delta\mathbf{\Sigma}\|_2.$$

Perturbations in a matrix cause perturbations of roughly the same size in its singular values. See Watkins [2002] for a detailed analysis of simultaneous perturbations in both the matrix and the input vectors.

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

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

and the elements of a general Hilbert matrix are $h_{ij} = (i + j - 1)^{-1}$.



Density plots for the matrices `hilb(n)\hilb(n)` for $n = 10$, 15, 20, 25, and 50. White represents a zero value, black represents a value of one or greater and intermediate values are depicted in shades of gray. An identity matrix is represented by a diagonal of black in a field of white such as for $n = 10$. Whereas Gaussian elimination appears to be at least moderately successful for $n \leq 10$, Gaussian elimination results in substantial round-off error for even moderately low dimensions such as $n = 15$. At $n = 15$ MATLAB provides the warning message

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.567841e-018. ◀
```

M The MATLAB function `cond(A,2)` or simply `cond(A)` computes the condition number of `A` in the l^2-norm by computing the SVD. Calculating the SVD is computation intensive, so this norm is best suited for smaller matrices. The l^1- and l^∞-condition numbers are given by `cond(A,1)` and `cond(A,inf)` which must compute A^{-1}. Although not as useful as the condition number in the l^2-norm, they are faster to compute. However, computing the inverse of a matrix is still resource intensive. The functions `condest(A)` and `rcond(A)` are faster still and estimate $\kappa_1(A)$ and $1/\kappa_1(A)$. When you use MATLAB to solve a linear system of equations, MATLAB computes `rcond(A)` and will warn you if it thinks that the system is ill-conditioned.

1.6 Geometric interpretation of linear algebra

This chapter reviewed several fundamental concepts of linear algebra. To help reinforce intuitive understanding, here is a summary of the geometric interpretation of basic linear algebra.

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

A diagonal matrix stretches along coordinate axis. A unit upper triangular matrix (or unit lower triangular matrix) shears like sliding the cards on top of one another in deck of cards. A projection matrix squashes all of the vectors in some direction to a pancake. If a vector is already in the pancake column space, then that vector 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 row of a matrix. If there is an even number of exchanges, the signature is even and orientation is right-handed. Otherwise, it is left-handed. A determinant is the signed volume of the parallelepiped spanned by the column vectors of a square matrix. A matrix maps a square to a parallelepiped and the absolute value of the determinant is simply the change in volume. The sign of the determinant is the orientation of the associated permutation. A Jacobian determinant is used calculus to measure the local change volume caused by a change of variables.

An eigenvector of a matrix is any vector that does not change direction by the matrix. An eigenvalue the amount 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. The 2-condition number ratio of semi-major to semi-minor axes of ellipsoid. A basis is the underlying set of vectors on which we create a coordinate system. The standard basis consists of the orthogonal unit vectors. Sometimes, it is more convenient to use a different basis.

A vector norm is the length of a vector—that is, the distance to the origin. Often, we think of distance in only terms of a Euclidean norm. But other norms are useful. We can extend the concept of a norm to a vector. An induced norm is the most any vector could be stretched by the matrix. Norms also help us measure the condition of a matrix. An ill-conditioned matrix a circle is mapped to a very eccentric ellipse. The 2-condition number is the ratio of the semi-major to semi-minor radii.

Exercises

1.1. Suppose that $\mathbf{A} \in \mathbb{C}^{n\times n}$ is nonsingular and $c \in \mathbb{C}$. Show that if $\mathbf{x}$ is an eigenvector of $\mathbf{A}$ with eigenvalue λ, then $\mathbf{x}$ is also an eigenvector of $\mathbf{A}^{-1}$ with eigenvalue $1/\lambda$. Also, show that $\mathbf{x}$ is an eigenvector of $\mathbf{A} - c\mathbf{I}$ with eigenvalue $\lambda - c$.

1.2. Prove that similar matrices have the same spectrum of eigenvalues.

✍ 1.3. Krylov subspaces are important tools in the computation of eigenvalues of large matrices. An order-r Krylov subspace generated by the $n \times n$ matrix $\mathbf{A}$ and a vector $\mathbf{x}$ is $\mathcal{K}_r(\mathbf{A}, \mathbf{x})$ is the subspace spanned by

$$\mathbf{x}, \mathbf{A}\mathbf{x}, \mathbf{A}^2\mathbf{x}, \ldots, \mathbf{A}^{r-1}\mathbf{x}.$$

(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 1×1 (0,1)-matrices, namely, 0 and 1. So, half are invertible. There are sixteen 2×2 matrices whose entries are 1s and 0s:

$$\begin{array}{rl} \text{Singular:} & \left[\begin{smallmatrix}0&0\\0&0\end{smallmatrix}\right], \left[\begin{smallmatrix}0&1\\0&0\end{smallmatrix}\right], \left[\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right], \left[\begin{smallmatrix}1&1\\0&0\end{smallmatrix}\right], \left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right], \left[\begin{smallmatrix}0&0\\1&0\end{smallmatrix}\right], \left[\begin{smallmatrix}0&0\\1&1\end{smallmatrix}\right], \left[\begin{smallmatrix}0&1\\0&1\end{smallmatrix}\right], \left[\begin{smallmatrix}1&0\\1&0\end{smallmatrix}\right], \left[\begin{smallmatrix}1&1\\1&1\end{smallmatrix}\right] \\ \text{Invertible:} & \left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right], \left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right], \left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right], \left[\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right], \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right], \left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right] \end{array}$$

So, three-eighths are invertible. Determine how many $n\times n$ (0,1)-matrices are invertible. Plot the ratio of invertible matrices as a function of size n for $n = 1, 2, \ldots, 20$ and explain the results. *Note:* there are 2^{n^2} $n \times n$ (0,1)-matrices—roughly, 10^{19} 8×8 matrices. To check each 8×8 matrix using a fast supercomputer (500 TFLOPS) would take over 2000 years. Instead, we can approximate the number by taking a random sample. For n=1 to 20, use MATLAB to test a large number (perhaps 10,000) of

random (0,1)-matrices. You can make an $n \times n$ random (0,1)-matrix in MATLAB using `double(rand(n)>.5)`.

✍ 1.5. Consider the $n \times n$ matrix often used to approximate to the second-derivative operator

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

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

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

are eigenvectors of $\mathbf{D}_n$ where $k = 1, \ldots, n$. (Note that the column vectors $\mathbf{x}_k$ divided by $\sqrt{(n+1)/2}$ form an orthogonal matrix $\mathbf{F}$. The spectral decomposition

$$\mathbf{D}_n\mathbf{F} = \mathbf{\Lambda}\mathbf{F}$$

gives us the discrete sine transform of $\mathbf{D}_n$. We'll come back to this idea later, but you don't need to use it for the proof.)

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

(c) Determine the l^2-condition number of the matrix. Discuss the behavior as $n \to \infty$.

(d) Confirm your answers using MATLAB's `eig` command. Hint: An easy way to input $\mathbf{D}_N$ in MATLAB is to use the `diag` function:

```
n = 25;
D = diag(ones([n-1 1]),1) ...
      - 2*diag(ones([n 1]),0) + diag(ones([n-1 1]),-1);
```

1.6. Prove that the induced norm (1.1) is a matrix norm.

1.7. By hand, compute the SVD of

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

(a) $\mathbf{A}$ maps $\mathbb{R}^3$ into $\mathbb{R}^2$. From the SVD determine the nullspace of $\mathbf{A}$.

(b) What vector is magnified the most in the l^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 matrix norm is defined as $\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} a_{ij}^2}$. This norm is particularly useful in image processing.

(a) Prove that the Frobenius matrix norm is in fact a matrix norm by satisfying the three properties of matrix norms.

(b) Determine $\|\mathbf{Q}\|_F$ where $\mathbf{Q}$ is an $n \times n$ orthogonal matrix.

(c) Prove that the Frobenius matrix norm is invariant under orthogonal transformations $\|\mathbf{QA}\|_F = \|\mathbf{A}\|_F$.

(d) Prove that $\|\mathbf{A}\|_F = \sqrt{\sum_i \sigma_i^2}$, where σ_i is the ith singular value of $\mathbf{A}$.

(e) Prove that $\|\mathbf{A}\|_F^2$ equals the trace of $\mathbf{A}^T\mathbf{A}$.

(f) Prove that the Frobenius matrix norm is compatible with the Euclidean vector norm.

(g) Prove that the Frobenius matrix norm is also submultiplicative.

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

1.10. Suppose that $\mathbf{B}$ is similar to $\mathbf{A}$. That is, $\mathbf{B} = \mathbf{S}^{-1}\mathbf{AS}$. Consider the perturbation $\mathbf{B} + \boldsymbol{\delta}\mathbf{B} = \mathbf{S}^{-1}(\mathbf{A} + \boldsymbol{\delta}\mathbf{A})\mathbf{S}$. Show that

(a) $\dfrac{1}{\kappa(\mathbf{S})}\|\mathbf{A}\| \leq \|\mathbf{B}\| \leq \kappa(\mathbf{S})\|\mathbf{A}\|$

(b) $\dfrac{1}{\kappa(\mathbf{S})^2}\dfrac{\|\boldsymbol{\delta}\mathbf{A}\|}{\|\mathbf{A}\|} \leq \dfrac{\|\boldsymbol{\delta}\mathbf{B}\|}{\|\mathbf{B}\|} \leq \kappa(\mathbf{S})^2\dfrac{\|\boldsymbol{\delta}\mathbf{A}\|}{\|\mathbf{A}\|}$

CHAPTER 2

Direct Methods for Linear Systems

In this chapter, we discuss direct methods for solving the problem $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is invertible. We discuss methods for solving $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is not invertible in Chapter 3. And we discuss iterative methods for solving $\mathbf{Ax} = \mathbf{b}$ in Chapter 5.

M There are several ways to solve $\mathbf{Ax} = \mathbf{b}$ in MATLAB. For a 2000×2000 system (on typical computer), `A\b` takes 0.4 seconds and `inv(A)*b` takes 1.2 seconds. The `mldivide` (\) command optimally solves the system by first determining the structure of `A` and then following the precedence:

1. sparse and diagonal: divide by diagonal elements
2. sparse, square and banded: use a banded solver
3. upper or lower triangular matrix: back or forward substitution
4. a permutation of a triangular matrix: permuted back substitution
5. symmetric, positive definite: Cholesky decomposition
6. sparse, symmetric, positive-definite: sparse Cholesky
7. symmetric: LDL^T factorization
8. Hessenberg
9. square: LU decomposition
10. not square: QR decomposition using Householder reflections

2.1 Gaussian elimination

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.

These elementary row operations leave a matrix in row equivalent form. That is, elementary row operations do not fundamentally change the underlying system of equations but instead return an equivalent system of equations. We can solve $\mathbf{Ax} = \mathbf{b}$ by using elementary row operations on the augmented matrix $\left[\mathbf{A} \mid \mathbf{b}\right]$ to get a reduced row echelon matrix.

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

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

$$\begin{bmatrix} \times & \times & \times & \times & | \times \\ \times & \times & \times & \times & | \times \\ \times & \times & \times & \times & | \times \\ \times & \times & \times & \times & | \times \end{bmatrix} \to \begin{bmatrix} \times & \times & \times & \times & | \times \\ & \times & \times & \times & | \times \\ & \times & \times & \times & | \times \\ & \times & \times & \times & | \times \end{bmatrix} \to \begin{bmatrix} \times & \times & \times & \times & | \times \\ & \times & \times & \times & | \times \\ & & \times & \times & | \times \\ & & \times & \times & | \times \end{bmatrix} \to \begin{bmatrix} \times & \times & \times & \times & | \times \\ & \times & \times & \times & | \times \\ & & \times & \times & | \times \\ & & & \times & | \times \end{bmatrix}$$

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

$$\begin{bmatrix} \times & \times & \times & \times & | \times \\ & \times & \times & \times & | \times \\ & & \times & \times & | \times \\ & & & \times & | \times \end{bmatrix} \to \begin{bmatrix} \times & \times & \times & & | \times \\ & \times & \times & & | \times \\ & & \times & & | \times \\ & & & \times & | \times \end{bmatrix} \to \begin{bmatrix} \times & \times & & & | \times \\ & \times & & & | \times \\ & & \times & & | \times \\ & & & \times & | \times \end{bmatrix} \to \begin{bmatrix} \times & & & & | \times \\ & \times & & & | \times \\ & & \times & & | \times \\ & & & \times & | \times \end{bmatrix}$$

M The MATLAB command `rref` returns the reduced row echelon form.

The first stage (forward elimination) requires the most effort because we need to compute using entire rows. We need to operate on n^2 elements to zero out the elements below the first pivot, $(n-1)^2$ elements to zero out the elements below the second pivot, and so forth. The second stage (backward elimination) is relatively fast because we only need to work on one column at each iteration. Zeroing out the elements above the last pivot, requires changing n elements, zeroing out the elements above the second pivot requires $n-1$ elements, and so forth. By saving the intermediate terms of forward elimination, we can express a matrix $\mathbf{A}$ as the product

$$\mathbf{A} = \mathbf{LU}$$

where $\mathbf{L}$ is a unit lower triangular matrix (with ones along the diagonal) and $\mathbf{U}$ is an upper triangular matrix. If such a decomposition exists, it is unique. Such a decomposition allows us to solve the problem $\mathbf{Ax} = \mathbf{b}$ in three steps:

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

This process is called *Gaussian elimination*. Two-stage Gaussian elimination combines step 1, called *LU decomposition*, with step 2. In this case, the $\mathbf{L}$ matrix is never explicitly formulated. But, as we will see there is an advantage to formulate $\mathbf{L}$ explicitly.

Solving triangular systems

Steps 2 and 3, solving triangular systems, are relatively straight-forward. We'll start with them and then come back to step 1. Step 2

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

can be implemented using forward elimination (starting from the top and working down). From the ith row

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

we have that

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

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

Now, let's implement step 3

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

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

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

we have that

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

As in step 1, we can overwrite the array $\mathbf{b}$ with the values $\mathbf{x}$.

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

LU decomposition

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

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

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

if we simply overwrite the elements of $\mathbf{A}$. For example,

$$\begin{bmatrix} 2 & 4 & 2 & 3 \\ -2 & -5 & -3 & -2 \\ 4 & 7 & 6 & 8 \\ 6 & 10 & 1 & 12 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 4 & 2 & 3 \\ -1 & -1 & -1 & 1 \\ 2 & -1 & 2 & 2 \\ 3 & -2 & -5 & 3 \end{bmatrix} \longrightarrow$$

$$\begin{bmatrix} 2 & 4 & 2 & 3 \\ -1 & -1 & -1 & 1 \\ 2 & 1 & 3 & 1 \\ 3 & 2 & -3 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 4 & 2 & 3 \\ -1 & -1 & -1 & 1 \\ 2 & 1 & 3 & 1 \\ 3 & 2 & -1 & 2 \end{bmatrix}.$$

Let's implement this algorithm. Starting with the first column we move right. For the jth column, using row operations we zero out each element in that column below the jth row. And here's the switch, we backfill those zeros with the values from the $\mathbf{L}$ matrix.

$$\left.\begin{array}{l} \text{for } j = 1, \ldots, n \\ \quad \left[\begin{array}{l} \text{for } i = j+1, \ldots, n \\ \quad \left[\begin{array}{l} m \leftarrow a_{ij}/a_{jj} \\ \text{for } k = j+1, \ldots, n \\ \quad \left[\; a_{ik} \leftarrow a_{ik} - m a_{jk} \right. \\ a_{ij} \leftarrow m \end{array}\right. \end{array}\right. \end{array}\right\} \text{LU decomposition}$$

$$\left.\begin{array}{l} \text{for } i = 2, \ldots, n \\ \quad \left[\; b_i \leftarrow b_i - \sum_{j=1}^{i-1} a_{ij} b_j \right. \end{array}\right\} \text{forward elimination}$$

$$\left.\begin{array}{l} \text{for } i = n, n-1, \cdots, 1 \\ \quad \left[\; b_i \leftarrow \left(b_i - \sum_{j=i+1}^{n} a_{ij} b_j \right) / a_{ii} \right. \end{array}\right\} \text{backward elimination}$$

The MATLAB code is

```
function b = lu_decomp(A,b)
n = length(A);
for j=1:n
  for i=j+1:n
    m = A(i,j)/A(j,j);
    A(i,j+1:n) = A(i,j+1:n) - m*A(j,j+1:n);
    A(i,j) = m;
  end
end
for i=2:n
  b(i) = b(i) - A(i,1:i-1)*b(1:i-1);
end
for i=n:-1:1
  b(i) = ( b(i) - A(i,i+1:n)*b(i+1:n) )/A(i,i);
end
```

Operation count

We can count the number of additions and multiplications to gauge the complexity of Gaussian elimination. To produce the LU decomposition:

$$\begin{aligned}\sum_{j=1}^{n}\sum_{i=j+1}^{n}\left(1+\sum_{k=j+1}^{n}2\right) &= \sum_{j=1}^{n}\sum_{i=j+1}^{n}2(n-j)+1\\ &= \sum_{j=1}^{n}2(n-j)^2+(n-j)\\ &= \sum_{j=0}^{n-1}2j^2+j \approx \frac{2}{3}n^3+\frac{1}{2}n^2\end{aligned}$$

where we use the approximation that $\sum \approx \int$ when n is large. Forward and backward elimination each require:

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

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.

Often we want to solve the problem $\mathbf{A}\mathbf{x}(t) = \mathbf{b}(t)$ in which the source term $\mathbf{b}(t)$ changes in time. 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.

Partial pivoting

If the pivot a_{ii} is zero at any step, Gaussian elimination clearly fails, because we are dividing by zero. In practice, if a pivot is close to zero, this algorithm may be also unstable because roundoff errors may be amplified.

Example. Consider the following matrix and its LU decomposition without row exchanges

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

Let $\varepsilon = 10^{-15}$, or about five times machine epsilon. The condition number of this matrix is $\kappa_2 \approx 10.3$, so it is numerically pretty stable. Because

of round-off, the computed LU decomposition is

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

The original matrix $\mathbf{A}$ and the effective matrix $\tilde{\mathbf{A}}$ differ in the bottom right elements. In truth, the bottom right element of $\mathbf{A}$ cannot easily be determined because of rounding error in $8 + 40/\varepsilon - 40/\varepsilon$. For $\mathbf{b} = (-5, 10, 0)^T$, $\mathbf{A}^{-1}\mathbf{b} \approx (5, 0, -5)^T$ whereas $\tilde{\mathbf{A}}^{-1}\mathbf{b} = (3, 2, -5)^T$, an error of about $(2, -2, 0)$. The error came from dividing by a small number—it would have been better to divide by a larger number. ◀

The fix is to use partial or complete pivoting in each iteration. In *partial pivoting*, we interchange the rows of the matrix to put the maximum element in the pivot position a_{ii}. Find r such that

$$|a_{ri}| = \max_{i \le k \le n} |a_{ki}|$$

and interchange rows i and r. For row operations, we left-multiply

$$\mathbf{PA} = \mathbf{LU}$$

where $\mathbf{P}$ is a permutation matrix. From pivoting, $\mathbf{L}$ has the property that $|l_{ij}| \le 1$ for all i and j.

Complete pivoting

In *complete pivoting*, we permute the rows and the columns of the submatrix to put the maximum element in the pivot position a_{ii}. That is, find r, c such that

$$|a_{rc}| = \max_{i \le k \le n, j \le l \le n} |a_{kl}|$$

and interchange rows i and r and columns i and l. For column operations, we right-multiply

$$\mathbf{PAQ} = \mathbf{LU}$$

where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrix. Applied to the problem $\mathbf{Ax} = \mathbf{b}$ complete pivoting yields

$$\mathbf{LU}(\mathbf{Q}^{-1}\mathbf{x}) = \mathbf{P}^{-1}\mathbf{b}.$$

Since permutation matrices are orthogonal, we equivalently have that

$$\mathbf{LU}(\mathbf{Q}^T\mathbf{x}) = \mathbf{P}^T\mathbf{b}.$$

Generally, complete pivoting is unnecessary and partial pivoting suffices.

2.2 Cramer's rule is not your friend

Some math textbooks introduce Cramer's rule as a means of solving linear system $\mathbf{Ax} = \mathbf{b}$. The ith component of $\mathbf{x}$ is expressed as the ratio of two determinants:

$$x_i = \frac{\det(\mathbf{A}_i)}{\det(\mathbf{A})}$$

where $\mathbf{A}_i$ is the matrix formed by replacing the ith column of $\mathbf{A}$ with $\mathbf{b}$. Cramer's rule applied to a system of n equations requires on the order of $(n+1)!$ flops—to compute $n+1$ determinants each requiring $n!$ operations. A fast PC can compute 17 gigaflops per second. To solve a relatively small system of twenty equations using Cramer's rule would require almost 100 years! On the other hand, it would require less than a microsecond using Gaussian elimination. The bottom line is that Cramer's rule is important for some analytical problems, but it is rather limited for numerical problems.

2.3 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. First we will prove this fact then we will examine how to implement it numerically. The proof has several pieces.

Theorem 8. *A inverse of a unit lower triangular matrix is a unit lower triangular matrix.*

Proof. It is easy to confirm the block matrix identity

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

The hypothesis is certainly true for 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. □

Theorem 9. *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.*

Proof. $\mathbf{A}$ 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

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

Note that $\mathbf{M}^{-1}\mathbf{A}\mathbf{M}^{-T}$ is a symmetric matrix and by Theorem 8 $\mathbf{M}^{-1}\mathbf{L}\mathbf{D}$ is a lower triangular matrix with diagonal $\mathbf{D}$. Hence, $\mathbf{M}^{-1}\mathbf{A}\mathbf{M}^{-T}$ must be the diagonal matrix $\mathbf{D}$. So,

$$\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{M}^T = \mathbf{M}^{-1}\mathbf{A}\mathbf{M}^{-T}\mathbf{M}^T = \mathbf{L}\mathbf{M}^{-1}\mathbf{A}$$

from which it follows that $\mathbf{L}\mathbf{M}^{-1} = \mathbf{I}$. So, $\mathbf{L} = \mathbf{M}$. □

Theorem 10. *If* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *is symmetric, positive-definite, and* $\mathbf{X} \in \mathbb{R}^{n\times k}$ *with* $\operatorname{rank}(\mathbf{X}) = k$*, then* $\mathbf{B} = \mathbf{X}^T\mathbf{A}\mathbf{X} \in \mathbb{R}^{k\times k}$ *is symmetric, positive definite—that is,* $\mathbf{x}^T\mathbf{B}\mathbf{x} \geq 0$.

Proof. Take $\mathbf{x} \in \mathbb{R}^k$. Then

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

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

We say that $\mathbf{B}$ is *congruent* to $\mathbf{A}$ if there exists an invertible $\mathbf{X}$ such that $\mathbf{B} = \mathbf{X}^T\mathbf{A}\mathbf{X}$.

Corollary 1. *A symmetric, positive-definite matrix* $\mathbf{A}$ *has the decomposition* $\mathbf{R}^T\mathbf{R}$ *where* $\mathbf{R} = \mathbf{L}\mathbf{D}^{1/2}$*. This is called the Cholesky decomposition.*

Proof. By Theorem 9, $\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^T$ where $\mathbf{L}$ is a unit lower triangular matrix. And by Theorem 10, $\mathbf{D} = \mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T}$ is positive definite. Therefore, eigenvalues of $\mathbf{D}$ (the diagonal elements of $\mathbf{D}$) are positive. So, we can define $\mathbf{R}^T = \mathbf{L}\mathbf{D}^{1/2}$. □

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 $\mathbf{A}$:

$$\begin{bmatrix} r_{11} & & & \\ r_{21} & r_{22} & & \\ \vdots & & \ddots & \\ r_{n1} & r_{n2} & \dots & r_{nn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{21} & \dots & r_{n1} \\ & r_{22} & & r_{n2} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{21} & a_{22} & \dots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

We start with a_{11}:

$$r_{11}r_{11} = a_{11} \qquad \Longrightarrow \qquad r_{11} = \sqrt{a_{11}}$$

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

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:

$$r_{11}r_{i1} = a_{1i} \qquad \Longrightarrow \qquad r_{1i} = a_{1i}/r_{i1}$$

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

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

$$\sum_{k=1}^{i} r_{ik}r_{ik} = a_{ii} \qquad \Longrightarrow \qquad r_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} r_{ik}^2 \right)^{\frac{1}{2}}$$

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

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

$$\sum_{k=1}^{i} r_{jk}r_{ik} = a_{ji} \qquad \Longrightarrow \qquad r_{ji} = \frac{1}{r_{ii}} \left(a_{ji} - \sum_{k=1}^{i-1} r_{jk}r_{ik} \right)$$

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

We can write this algorithm as the following pseudo-code:

$$\begin{array}{l} \text{for } i = 1, \ldots, n \\ \left[\begin{array}{l} r_{ii} \leftarrow \left(a_{ii} - \displaystyle\sum_{k=1}^{i-1} a_{ik}^2 \right)^{1/2} \\ \text{for } j = 2, \ldots, i \\ \quad \left[\; r_{ji} \leftarrow \dfrac{1}{r_{ii}} \left(a_{ji} - \displaystyle\sum_{k=1}^{i-1} r_{jk}r_{ik} \right) \right. \end{array} \right. \end{array}$$

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

2.4 Sparse matrices

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

Ⓜ The MATLAB command `spy(A)` returns the sparsity plot of a matrix $\mathbf{A}$. The command `spy` (without an argument) returns an Easter egg.

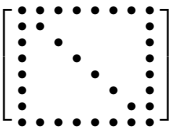
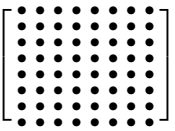
Banded matrices

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

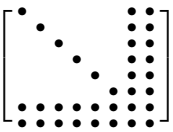
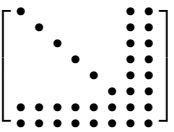
If a banded matrix $\mathbf{A}$ has a lower bandwidth p and and an upper bandwidth q, then the LU decomposition of $\mathbf{A}$ produces a lower triangular matrix $\mathbf{L}$ with a lower bandwidth of p and an upper bandwidth of q. Furthermore, the number of operations is approximately $2npq$ flops.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

Preserving sparsity

Consider the simple example of the following sparse matrix:

 has the LU decomposition 

with a tremendous amount of fill-in. Now, consider reordering the rows and columns first by moving the first row to the bottom and the first column to the end. That is $\tilde{\mathbf{A}} = \mathbf{PAP}$. Then

 has the LU decomposition  .

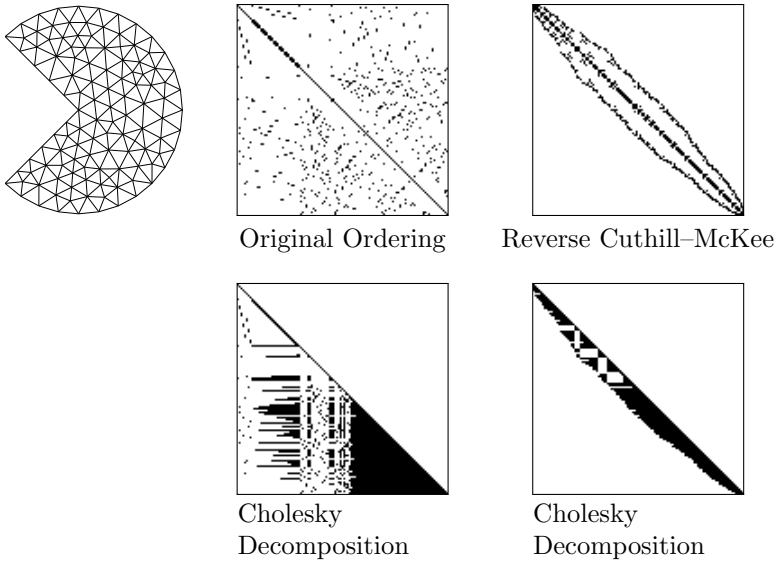


Figure 2.1: The sparsity plot for the Laplacian operator in a finite element solution for a pacman domain and the reverse Cuthill–McKee sorting along with the Cholesky decomposition for both.

For an $n \times n$ 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.

Cuthill–McKee resorting

We can lower the bandwidth and thereby reduce the amount of fill in by reordering (permuting) the rows and columns to try to get all nonzeros as close to the diagonal as possible. The Cuthill–McKee algorithm is one such method. Consider a symmetric matrix $\mathbf{A}$. We can associate the sparse matrix with a *oriented graph* which connects *vertices* using *paths*. The number of connections (paths) that a vertex has to neighboring vertices is called the *degree of the vertex*. The Cuthill–McKee algorithm works as follows:

0. Choose a vertex with a low number of connections. Relabel this vertex as 1.

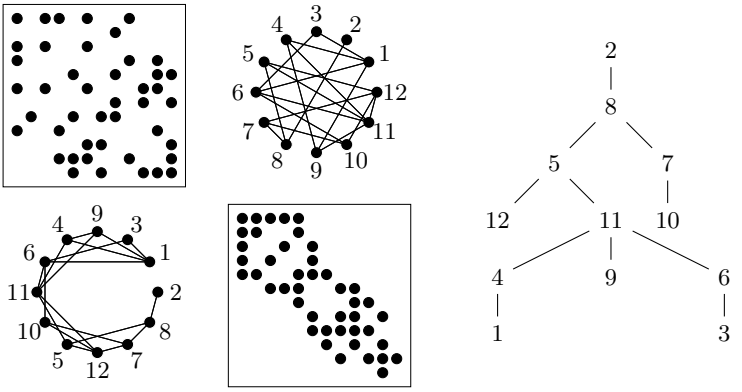


Figure 2.2: Cuthill–McKee ordering $\{2, 8, 5, 7, 12, 11, 10, 4, 9, 6, 1, 3\}$

1. Progressively relabel each of the neighboring vertices.
2. Now, relabel each of the vertices neighboring the vertices from step 1 skipping any vertex that has already been relabeled.
3. Continue to the next tier until all the vertices have been relabeled.

See Figure 2.2 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: $\tilde{\mathbf{A}} = \mathbf{PAP}$.

In practice, one typically uses the *reverse Cuthill–McKee* algorithm which simply reverses the ordering of the relabeled vertices. Other resorting methods include the minimum degree and the nested dissection.

M The MATLAB function `symrcm` returns the permutation vector using the reverse Cuthill–McKee algorithm for symmetric matrices. The command `symamd` returns the permutation vector using the symmetric approximate minimum degree permutation algorithm.

How MATLAB stores a sparse matrix

Suppose that we have the 4×6 full matrix

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

If the number of nonzero elements is small we could store the 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, MATLAB goes a step further and uses *compressed column format* (CCF). Rather than saving the explicit columns numbers, CCF data structure only saves pointers to where the new columns begin

index	1		3	4		6			9	10				14
row	1	4	2	3	4	1	3	4	2	1	2	3	4	
value	11	41	22	33	43	14	34	44	25	16	26	36	46	

Ⓜ The MATLAB command `help sparfun` returns a list of MATLAB's functions for working with sparse matrices.

Exercises

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

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

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

2.2. Consider a 1000×1000 (or larger) tridiagonal matrix from problem 1.5. Let $\mathbf{b}$ be a column vector of 1000 ones. Compare how long it takes MATLAB to solve $\mathbf{Ax} = \mathbf{b}$ using the backslash operator, when $\mathbf{A}$ is inputted as a full matrix versus a sparse matrix. Hint: To input $\mathbf{A}$ as a full matrix use

```
A = diag(ones([n 1]),0) ...
     - diag(ones([n-1 1]),1) - diag(ones([n-1 1]),-1);
```

To input $\mathbf{A}$ as a sparse matrix use

```
A = spdiags(repmat([-1 2 -1],[n 1]), [-1 0 1], n, n);
```

To measure time, use the `tic` function to start the timer and the `toc` function to output the elapsed time. For example,

```
tic; A\b; toc
```

Repeat the problem using $n \times n$ tridiagonal matrices for different n and plot use a log-log plot to demonstrate the complexity of the algorithm as a function of n. For example, take $n = 100, 200, 400, 800$, and so forth.

M MATLAB compiles some functions into memory when they are initially called. The first call of a function may take considerably more time than subsequent ones. Keep this in mind when comparing speeds of different functions.

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

✍ 2.4. LU decomposition can be used to efficiently compute the determinant of a matrix. This is how MATLAB does it. Write a program to find a determinant using LU decomposition.

✍ 2.5. Use the reverse Cuthill–McKee algorithm to reorder the following symmetric matrix into an matrix with a smaller bandwidth

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

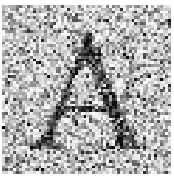
Specifically, give the permutation matrix $\mathbf{P}$ and the reordered matrix $\tilde{\mathbf{A}} = \mathbf{PAP}$. Note: $\bullet$ denotes a nonzero element and $\circ$ denotes a zero element.

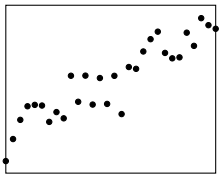
CHAPTER 3

Least Squares Problem and SVD

3.1 Overdetermined systems

Often, we want to find the solution to a problem with noisy input data. Such problems include curve fitting, image recognition, and statistical modeling. We may have abundant but inconsistent information. Hence, the problem is ill-posed. The solution is to filter out the noise, leaving us with consistent data. For example, it is not difficult for us to mentally filter out the noise and recognize the letter on the left or the slope of the line on the right.





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 (range) of $\mathbf{A}$, that is, 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 $\hat{\mathbf{x}}$, so that $\hat{\mathbf{b}} = \mathbf{A}\hat{\mathbf{x}}$ is closest to $\mathbf{b}$. This means that we want to choose $\hat{\mathbf{x}}$ that minimizes the norm of the *residual* $\hat{\mathbf{r}} = \mathbf{b} - \tilde{\mathbf{b}} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}$.

So, which norm? The l^1-, l^2- and l^∞-norms are all relevant norms, but the l^2-norm is Goldilocks' baby bear of norms. The l^1-norm minimizes the importance of outliers and the l^∞-norm only looks at the outliers. In addition, some problems are ill-posed in the l^1- and l^∞-norms. So, the l^2-norm seems to be the natural choice. In fact, the l^2-norm actually gives us more—perhaps the nicest property of the l^2-norm is that the

problem of finding a solution that minimizes the l^2-norm of the residual is a linear problem.

Recently, the l^1-norm has gained some popularity in research, because it handles noisy data better than the l^2-norm by minimizing the influence of outliers (Candè and Wakin [2008]). But because the minimization problem in the l^1-norm is nonlinear, more complex solvers must be used.

3.2 Normal equation

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

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

Rearranging the terms on the left-hand side, we have that

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

and then expanding

$$\|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2^2 + \|\mathbf{A}\mathbf{y}\|_2^2 + 2\mathbf{y}\mathbf{A}^H\mathbf{A}\mathbf{x} - 2\mathbf{y}^H\mathbf{A}^H\mathbf{b} \geq \|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2^2$$

which says

$$\|\mathbf{A}\mathbf{y}\|_2^2 + 2\mathbf{y}^H\mathbf{A}^H\mathbf{A}\hat{\mathbf{x}} - 2\mathbf{y}^H\mathbf{A}^H\mathbf{b} \geq 0.$$

Since $\|\mathbf{A}\mathbf{y}\|_2^2 \geq 0$ for any $\mathbf{y}$, it follows that

$$2\mathbf{y}^H\mathbf{A}^H\mathbf{A}\hat{\mathbf{x}} - 2\mathbf{y}^H\mathbf{A}^H\mathbf{b} \geq 0$$

or equivalently

$$\mathbf{y}^H(\mathbf{A}^H\mathbf{A}\hat{\mathbf{x}} - \mathbf{A}^H\mathbf{b}) \geq 0.$$

Because this expression must hold for any $\mathbf{y}$ and because $\mathbf{A}^H\mathbf{A}\hat{\mathbf{x}} - \mathbf{A}^H\mathbf{x}$ is independent of $\mathbf{y}$, it follows that

$$\mathbf{A}^H\mathbf{A}\hat{\mathbf{x}} - \mathbf{A}^H\mathbf{b} = 0$$

and we get the *normal equation*

$$\mathbf{A}^H\mathbf{A}\hat{\mathbf{x}} = \mathbf{A}^H\mathbf{b}.$$

Here's a faster way to see this. The residual $\|\mathbf{r}(\hat{\mathbf{x}})\|_2$ is minimized when the gradient of $\|\mathbf{r}(\hat{\mathbf{x}})\|_2$ is zero, or equivalently, when the gradient of $\|\mathbf{r}(\hat{\mathbf{x}})\|_2^2$ is zero. We have that

$$\|\mathbf{r}(\hat{\mathbf{x}})\|_2^2 = \|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2^2 = \|\mathbf{A}\hat{\mathbf{x}}\|_2^2 - 2\hat{\mathbf{x}}^H\mathbf{A}^H\mathbf{b} + \|\mathbf{b}\|_2^2,$$

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

$$0 = 2\mathbf{A}^H\mathbf{A}\hat{\mathbf{x}} - 2\mathbf{A}^H\mathbf{b}.$$

Therefore,

$$\mathbf{A}^H\mathbf{A}\hat{\mathbf{x}} = \mathbf{A}^H\mathbf{b}.$$

Let's see what the normal equation tells us by looking at the error $\mathbf{e} = \mathbf{x} - \hat{\mathbf{x}}$ and the residual $\mathbf{Ae} = \mathbf{b} - \hat{\mathbf{b}}$. The solution to the normal equation is

$$\hat{\mathbf{x}} = (\mathbf{A}^H\mathbf{A})^{-1}\mathbf{A}^H\mathbf{b}$$

from which

$$\hat{\mathbf{b}} = \mathbf{A}\hat{\mathbf{x}} = \mathbf{A}(\mathbf{A}^H\mathbf{A})^{-1}\mathbf{A}^H\mathbf{b} = \mathbf{P}_{\mathbf{A}}\mathbf{b}$$

where $\mathbf{P}_{\mathbf{A}}$ is the orthogonal projection matrix into the range (column space) of $\mathbf{A}$. The residual is simply

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

where $\mathbf{P}_{N(\mathbf{A}^T)}$ is the orthogonal projection matrix into the left nullspace of $\mathbf{A}$. The normal equation gets its name because the residual $\mathbf{b} - \mathbf{Ax}$ is normal to the range 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. By furthermore zeroing out $\mathbf{x}_{\text{null}}$, we'll get a unique solution. We'll come back to this idea later in the chapter.

Often, the normal equation can make a difficult problem even more difficult because the condition number $\kappa((\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T)$ may be as much as $\kappa(\mathbf{A})^3$, and a moderately ill-conditioned matrix leads to a very ill-conditioned problem. For now, we'll 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.

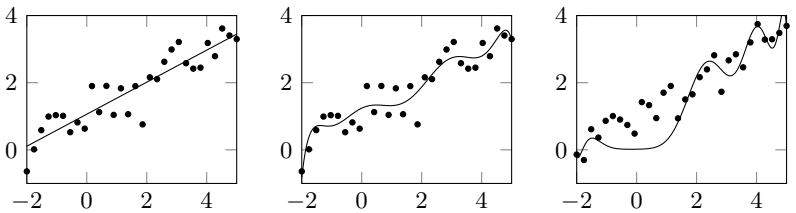
Example. Consider the problem of fitting an nth-degree polynomial

$$y = c_0 + c_1x + c_2x^2 + \cdots + c_nx^n$$

to data $(x_0, y_0), (x_2, y_2), \ldots, (x_m, y_m)$. Given $n+1$ distinct points, we can determine an nth-degree polynomial by solving the Vandermonde system $\mathbf{Vc} = \mathbf{y}$:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

If we have more data than unknowns, we instead solve the normal equation $\mathbf{V}^T\mathbf{V}\mathbf{c} = \mathbf{V}^T\mathbf{y}$. For the moment, ignore the cardinal sin of data mining: overfitting. By choosing polynomials of higher degree we are able to get a closer match by reducing the residual. Numerically, this is true up to a point. The Vandermonde matrix $\mathbf{V}$ becomes ill-conditioned for large dimensions, and it just gets worse for $(\mathbf{V}^T\mathbf{V})^{-1}\mathbf{V}$. Eventually, computation is overwhelmed by round-off error. The following graphs show the least squares fit with 1-, 8-, and 13-degree polynomials using the normal equation. The residuals $\|\mathbf{r}\|_2$ are 2.46, 2.04, and 4.17, respectively. The residuals should continue to decrease as higher degree polynomials are used because we can get a better fit, but eventually numerical error wins out.



The Vandermonde matrix is ill-conditioned when it is large, the solution falls apart at degree 13. The condition numbers of the Vandermonde matrix are roughly 3, 10^5 and 10^{10}. The condition numbers of the normal matrix $\mathbf{V}^T\mathbf{V}$ are roughly 11, 10^{11} and 10^{20}. Least squares fit with 1-, 8-, and 13-degree polynomials using the normal equation.

The ill-conditioning of the Vandermonde matrix can be understood by looking at the graphs of the monomials $\{1, x, x^2, x^3, \cdots, x^n\}$. As n becomes large the monomials are difficult to distinguish. In other words, these basis vectors are far from orthogonal. One solution is to pick an orthogonal basis such as Legendre polynomials. This topic will be discussed in Chapter 10. ◀

3.3 QR decomposition

Every real $m \times n$ matrix $\mathbf{A}$ can be written as the product of an $n \times n$ 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 $\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}$. 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 on the left-hand side we simply have the Cholesky decomposition $\mathbf{R}^T\mathbf{R}$ of $\mathbf{A}^T\mathbf{A}$. By rearranging the terms of this equality we have

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

which says that $\mathbf{v} = \mathbf{Rx} - \mathbf{Q}^T\mathbf{b}$ is in the nullspace of $\mathbf{R}^T$ (the left-nullspace of $\mathbf{R}$):

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

So, $\mathbf{Rx}$ must equal $\mathbf{Q}^T\mathbf{b}$ in the first n elements. Otherwise, it can be anything.

Let's take a second look. Suppose that we have the overdetermined system $\mathbf{Ax} = \mathbf{b}$:

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

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

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

We discard the (last three) inconsistent equations and solve the system

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

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

Gram–Schmidt orthogonalization

The Gram–Schmidt process finds an orthonormal basis for a subspace by successively projecting out the nonorthogonal components. Recall that the orthogonal projection matrix is

$$\mathbf{P}_{\mathbf{A}} = \mathbf{A}(\mathbf{A}^H\mathbf{A})^{-1}\mathbf{A}^H.$$

A one-dimensional subspace span$\{\mathbf{v}\}$ has the simpler projection matrix

$$\mathbf{P}_{\mathbf{v}} = \frac{\mathbf{v}\mathbf{v}^H}{\mathbf{v}^H\mathbf{v}}.$$

For a unit length vector $\mathbf{q}$, the projection matrix is simply $\mathbf{P}_{\mathbf{q}} = \mathbf{q}\mathbf{q}^H$.

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

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 $\mathbf{q}_2$ so 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 $\mathbf{v}_2$. Take

$$\mathbf{w}_2 = \mathbf{v}_2 - \mathbf{P}_{\mathbf{q}_1}\mathbf{v}_2$$

and noting that

$$\mathbf{P}_{\mathbf{q}_j}\mathbf{v}_k = \mathbf{q}_j^H\mathbf{q}_j\mathbf{v}_k = (\mathbf{q}_j, \mathbf{v}_k)_2\,\mathbf{q}_j$$

we have that

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

This step finds the closest vector to $\mathbf{v}_2$ (in the l^2-norm) which 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 span$\{\mathbf{q}_1, \mathbf{q}_2\}$ components of $\mathbf{v}_3$ from $\mathbf{v}_3$ and normalize.

$$\begin{aligned}\mathbf{w}_3 &= \mathbf{v}_3 - \mathbf{P}_{\mathbf{q}_1}\mathbf{v}_3 - \mathbf{P}_{\mathbf{q}_2}\mathbf{v}_3\\ &= \mathbf{v}_3 - (\mathbf{v}_3, \mathbf{q}_1)_2\,\mathbf{q}_1 - (\mathbf{q}_2, \mathbf{v}_3)_2\,\mathbf{q}_2\end{aligned}$$

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 kth 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 $\mathbf{q}_4, \ldots, \mathbf{q}_n$.

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

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

Givens rotations

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

A rotation in the plane $\mathbb{R}^2$ is given by

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

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

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

by taking

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

Applying the right rotation matrix to a 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 $\mathbf{A} \in \mathbb{R}^5$ 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

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

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

For example, consider a 4×4 matrix

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

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

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

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

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

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

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

So,

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

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 MATLAB function `givens(x,y)` returns the Givens rotation matrix for vector (x, y).

Householder reflections

While Givens rotations zero out one element at time, *Householder reflections* speed things up by fixing a whole column at a time

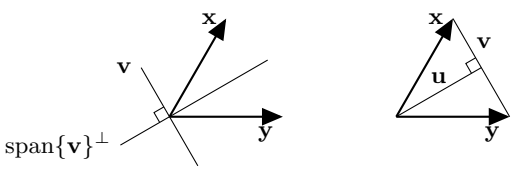
$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \to \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{bmatrix}.$$

We'll start with the first column. Take a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ and let $\mathbf{y} = (y_1, 0, \ldots, 0)^T$ with $\|\mathbf{y}\|_2 = \|\mathbf{x}\|_2$. That is, $\mathbf{y} = \pm\|\mathbf{x}\|_2\mathbf{e}_1$ where the sign is chosen to agree with the first component of $\mathbf{x}$. Let's find the orthogonal matrix $\mathbf{Q}$ such that $\mathbf{Qx} = \mathbf{y}$. We can do this using geometry.

Let

$$\mathbf{u} = (\mathbf{x} + \mathbf{y})/2 \quad \text{and} \quad \mathbf{v} = (\mathbf{x} - \mathbf{y})/2.$$

Note that $\text{span}\{\mathbf{u}, \mathbf{v}\} = \text{span}\{\mathbf{x}, \mathbf{y}\}$ and $\mathbf{u} \perp \mathbf{v}$. Furthermore, we can get $\mathbf{y}$ by subtracting twice $\mathbf{v}$ from $\mathbf{x}$.



Of course, this is the same as saying that 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*

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

Since we want $\mathbf{y}$ to be $\pm\|\mathbf{x}_1\|\mathbf{e}_1$, we take $\mathbf{v} = \mathbf{x} \pm \|\mathbf{x}\|\mathbf{e}_1$. (Technically, we should have taken $\mathbf{v} = (\mathbf{x} \pm \|\mathbf{x}\|\mathbf{e}_1)/2$, but $\mathbf{P}_{\mathbf{v}}$ would have removed the one-half later.) Geometrically, the Householder matrix $\mathbf{H}$ reflects a vector across the subspace $\text{span}\{\mathbf{v}\}^{\perp}$.

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

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

where $\mathbf{H}_{n-1}$ is the $(n-1)\times(n-1)$ Householder matrix matrix that maps a vector $\mathbf{x}$ to the vector $\pm\|\mathbf{x}\|_2\mathbf{e}_1$.

In the kth step, we use the Householder matrix

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

where $\mathbf{H}_{n-k}$ is the $(n-k)\times(n-k)$ Householder matrix that maps an $(n-k)$-dimensional vector $\mathbf{x}$ to an $(n-k)$-dimensional vector $\pm\|\mathbf{x}\|_2\mathbf{e}_1$.

Ⓜ The MATLAB function `qr` implements LAPACK routines to compute the QR factorization of a matrix using Householder reflection.

3.4 Least squares problem (again)

Let's reexamine the least squares problem in 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 $m \geq n$. Find the $\mathbf{x} \in \mathbb{R}^n$ that minimizes the residual

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

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

$$\|\mathbf{r}\|_2 = \|\mathbf{Q}\mathbf{R}\mathbf{x} - \mathbf{b}\|_2.$$

Applying an orthogonal transform to $\mathbf{r}$ doesn't change $\|\mathbf{r}\|_2$, so

$$\|\mathbf{r}\|_2 = \|\mathbf{Q}^T\mathbf{r}\|_2 = \|\mathbf{R}\mathbf{x} - \mathbf{Q}^T\mathbf{b}\|_2$$

Letting $\mathbf{s} = \mathbf{R}\mathbf{x} - \mathbf{Q}^T\mathbf{b}$, the $\mathbf{x}$ that minimizes the residual $\mathbf{r}$ also minimizes the residual $\mathbf{s}$. This means that we can instead consider the an equivalent upper triangular system

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

where $\mathbf{c} = \mathbf{Q}^T\mathbf{b}$. Writing $\mathbf{c} = \begin{bmatrix} \hat{\mathbf{c}} \\ \mathbf{d} \end{bmatrix}$ where $\hat{\mathbf{c}} \in \mathbb{R}^n$, the residual $\mathbf{s}$ is

$$\mathbf{s} = \begin{bmatrix} \hat{\mathbf{c}} \\ \mathbf{d} \end{bmatrix} - \begin{bmatrix} \hat{\mathbf{R}} \\ \mathbf{0} \end{bmatrix} \mathbf{x} = \begin{bmatrix} \hat{\mathbf{c}} - \hat{\mathbf{R}}\mathbf{x} \\ \mathbf{d} \end{bmatrix}.$$

So,

$$\|\mathbf{s}\|_2^2 = \sum_{i=1}^{n} s_i^2 = \|\hat{\mathbf{c}} - \hat{\mathbf{R}}\mathbf{x}\|_2^2 + \|\mathbf{d}\|_2^2.$$

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

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

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

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

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

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

and it follows that $\|\mathbf{s}\|^2$ is minimized for any vector $\mathbf{x}_2$ when

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

So, we don't have a unique solution. Instead, we'll find the "best" solution in the next section.

3.5 Underdetermined (and inconsistent) systems

Up until now we've assumed that if $\mathbf{Ax} = \mathbf{b}$ had a least squares solution, then it had a unique least squares solution. This assumption is not always true. If the nullspace of $\mathbf{A}$ has more than the zero vector, then $\mathbf{A}^T\mathbf{A}$ is not invertible (and $\hat{\mathbf{R}}$ from the previous section does not have full rank). In this case, the problem has infinitely many solutions. So then, how do we choose which of the infinitely many solutions is the "best" solution? One way is to take the solution $\mathbf{x}$ with the smallest norm $\|\mathbf{x}\|$. This approach is known as *Tikhonov regularization*. This is especially good in problems where $\mathbf{x}$ is has a zero mean (which happen frequently in statistics). Change our original problem

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

to the new problem

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

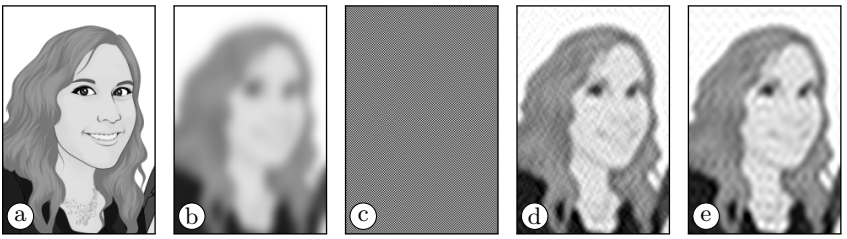
where $\alpha > 0$ is some regularizing parameter. The result is a solution that fits $\mathbf{x}$ to $\mathbf{b}$ but penalizes solutions with large norms. As before, we solve $\nabla\Phi(\mathbf{x}) = 0$ where the gradient is with respect to $\mathbf{x}$. We get

$$2\mathbf{A}^T\mathbf{Ax} - 2\mathbf{A}^T\mathbf{b} + 2\alpha^2\mathbf{x} = 0$$

from which we ca solve for $\mathbf{x}$ to get

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

Example. Photographs often contain motion or focal blur and noise. Let $\mathbf{x}$ be the original image and $\mathbf{b} = \mathbf{Ax}+\mathbf{n}$ be the observed image where $\mathbf{A}$ is the blurring matrix and $\mathbf{n}$ is a noise vector. The blurring matrix $\mathbf{A}$ does not have a unique inverse and therefore the problem is ill-posed. Compare the following images:



(a) Original image: $\mathbf{x}$. (b) Blurring and noise have been added to image A: $\mathbf{b} = \mathbf{A}\mathbf{x} + \mathbf{n}$. (c) Least-square solution: $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$. This method clearly fails. (d) Regularized least-square solution: $\mathbf{R}_\alpha\mathbf{b}$. The winner! (e) Solution using the pseudoinverse: $\mathbf{A}^+\mathbf{b}$. Not as good. ◀

3.6 Singular value decomposition

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

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

we minimize $\|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2$ by zeroing out forcing $\mathbf{b}_{\text{left null}} = \mathbf{0}$ and we ensure a unique solution $\|\mathbf{x}\|_2$ by forcing $\mathbf{x}_{\text{null}} = \mathbf{0}$. In the first case, we are zeroing out the left null space; in the second case, we are zeroing out the null space. We can use singular value decomposition to do precisely this. A matrix's singular value decomposition is $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices and $\mathbf{\Sigma}$ is a diagonal matrix of singular values. The singular values σ_i are always in descending order

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0.$$

This ordering is a natural result of the SVD algorithm, which we study in the next chapter after we develop the tools for finding eigenvalues. In this section, we look at applications of singular value decomposition.

The SVD of an $m \times n$ matrix $\mathbf{A}$ produces an $m \times m$ matrix $\mathbf{U}$, an $m \times n$ diagonal matrix $\mathbf{\Sigma}$, 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 $\mathbf{A}$ is a rectangular matrix with $m > n$, only the first n left singular matrices are needed to reconstruct $\mathbf{A}$. In this case, it's more efficient to use the *thin* or *economy* SVD. An $m \times n$ matrix $\mathbf{A}$ produces an $m \times n$ matrix $\mathbf{U}$, an $n \times n$ diagonal matrix $\mathbf{\Sigma}$, and an $n \times n$ matrix $\mathbf{V}^T$. See Figure 3.1.

If $\mathbf{A}$ has rank r, then

$$\begin{aligned} \text{the first } r \text{ columns of } \mathbf{U} &\text{ span the column space of } \mathbf{A}; \\ \text{the last } m-r \text{ columns of } \mathbf{U} &\text{ span the left null space of } \mathbf{A}; \\ \text{the first } r \text{ columns of } \mathbf{V} &\text{ span the row space of } \mathbf{A}; \\ \text{the last } n-r \text{ columns of } \mathbf{V} &\text{ span the null space of } \mathbf{A}. \end{aligned}$$

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

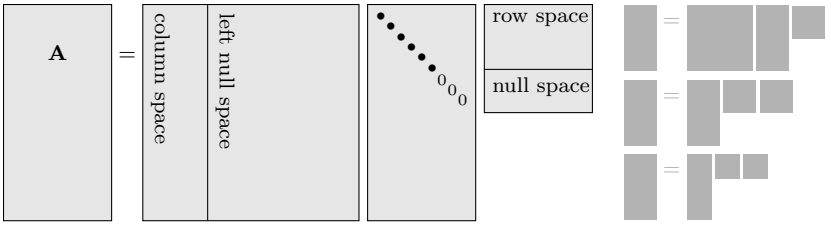


Figure 3.1: Left: Antomy of the SVD. Right: SVD (top), the *thin* or *economy* SVD (middle), and the *compact* or *truncated* SVD (bottom).

Often for large matrices, the small singular values do not contribute much to $\mathbf{A}$. By keeping the s largest singular values and the corresponding singular vectors, we get the closest rank s approximation to $\mathbf{A}$ in the Frobenius norm. The *truncated* SVD of an $m \times n$ matrix $\mathbf{A}$ produces an $m \times s$ matrix $\mathbf{U}_s$, an $s \times s$ diagonal matrix $\mathbf{\Sigma}_s$, and an $s \times n$ matrix $\mathbf{V}_s^T$. A truncated SVD is useful for

- reducing memory and storage of $\mathbf{A}$;
- making computations faster;
- simplifying a problem by projecting the data to a lower dimensional subspace; and
- regularizing a problem by reducing the condition number to σ_1/σ_s.

Ⓜ The MATLAB command `[u,s,v]=svd(A)` returns the SVD decomposition of a matrix `A` and `svd(A,0)` returns the "economy" version of the SVD. The MATLAB command `svds(A,k)` returns the first `k` singular values and associated singular vectors.

Pseudoinverse

Suppose that $\mathbf{A}$ is an arbitrary $m \times n$ matrix with the singular value decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. If $\mathbf{A}$ is nonsingular, then the solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ is

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T\mathbf{b}$$

where $\mathbf{\Sigma}^{-1}$ is simply the diagonal matrix of the reciprocal of the singular values. The *Moore–Penrose pseudoinverse* $\mathbf{A}^+$ of $\mathbf{A}$ is an $n \times m$ matrix defined as

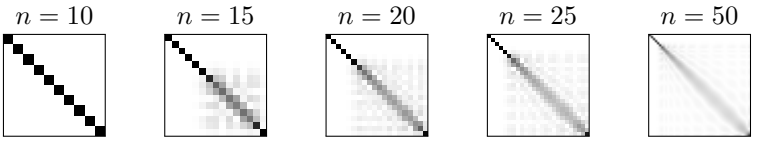
$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^T$$

where $\mathbf{\Sigma}^+$ is a diagonal matrix of the reciprocals of the nonzero elements of $\mathbf{\Sigma}$, leaving the zero elements unchanged. This has the effect of zeroing out both the nullspace and the left nullspace of $\mathbf{A}$. Recall that the singular vectors corresponding to the zero singular values are in the nullspace and left nullspace of $\mathbf{A}$. By zeroing these out, not only does the pseudoinverse ensure a solution, it ensures a unique solution. Namely, for an $m \times n$ matrix $\mathbf{A}$ with rank r,

$$\begin{array}{lll} \text{if } m = n = r & \text{then} & \mathbf{A}^+ = \mathbf{A}^{-1}, \\ \text{if } m > n = r & \text{then} & \mathbf{x} = \mathbf{A}^+\mathbf{b} \text{ minimizes } \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2 \\ & & \text{and } \mathbf{A}^+ = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T, \\ \text{if } n > m = r & \text{then} & \mathbf{x} = \mathbf{A}^+\mathbf{b} \text{ gives the rowspace projection} \\ & & \text{and } \mathbf{A}^+ = \mathbf{A}(\mathbf{A}\mathbf{A}^T)^{-1}. \end{array}$$

We also can use the pseudoinverse to numerically regularize unstable (ill-conditioned) problems by removing the small singular values by using a truncated SVD which zeros out elements below some tolerance α. By doing so, we lower the condition number to $\kappa_2(\mathbf{A}) = \sigma_1/\alpha$ and solve a problem in a lower dimensional subspace.

Example. Consider the ill-conditioned Hilbert matrix $\mathbf{H}$ where $h_{ij} = (i + j - 1)^{-1}$.



The density plots above show the matrices `pinv(hilb(n))*hilb(n)` for $n = 10$, 15, 20, 25, and 50. White represents a zero value, black represents a value of one or greater and intermediate values are depicted in shades of gray. While the pseudoinverse solution has noticeable error when $n > 10$, the failure is much more graceful the solution using Gaussian elimination. See the example on page 23. The pseudoinverse truncates all singular values less than 10^{-15} which happens near σ_8, effectively reducing the problem to a using a rank 8 matrix. ◀

Ⓜ The MATLAB function `pinv` computes the Moore–Penrose pseudoinverse by computing the SVD using a default tolerance of `max(size(A))*norm(A)*eps`, that is, machine epsilon scaled by the largest singular value times the largest dimension of `A`.

Tikhonov regularization

Tikhonov regularization introduced the section 3.5 and the pseudoinverse discussed above both provides a means to solve the underdetermined problem. Let's examine the connection between the two. Recall the Tikhonov regularization

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

If $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$, then

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

where $\boldsymbol{\Sigma}_\alpha^{-1}$ is a diagonal matrix with diagonal components

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

It is useful to rewrite these components as

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

where $w_\alpha(\sigma_i^2)$ is the *Wiener weight*. From this we see that Tikhonov regularization is simply a smooth filter to the singular values. In comparison, the truncated SVD filter

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

exhibits a sharp cut-off at α. See Figure 3.2 on the next page. Because of this, Tikhonov regularization may outperform the pseudoinverse.

Principal component analysis

Principle component analysis (PCA) is a statistical method that uses singular value decomposition to generate a lower dimensional approximation of a matrix $\mathbf{A}$. Let $\mathbf{A}$ be an $m \times n$ matrix, then $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$ can be written as the sum of rank-one, mutually orthogonal *principal component* $\mathbf{E}_k = \sigma_k \mathbf{u}_k \mathbf{v}_k^T$:

$$\mathbf{A} = \mathbf{E}_1 + \mathbf{E}_2 + \cdots + \mathbf{E}_r + \cdots + \mathbf{E}_p$$

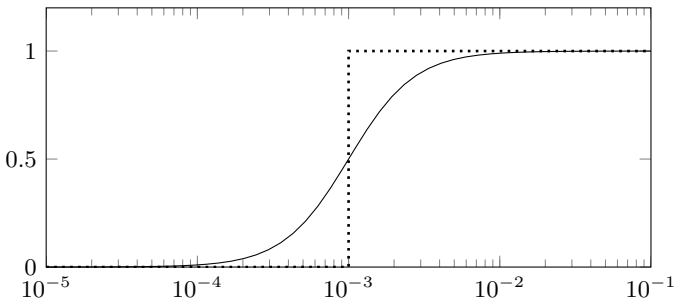


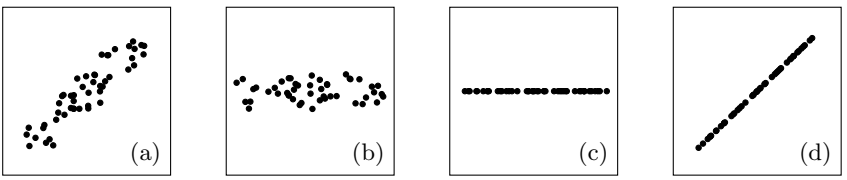
Figure 3.2: Plot of Tikhonov filter $w_\alpha(\sigma)$ for $\alpha = 10^{-3}$ with truncated SVD filter $w_\alpha^{\text{TSVD}}(\sigma)$ with singular values zeroed out below 10^{-3}.

where $p = \min(m, n)$. We can get a rank $r < p$ approximation by truncating the sum

$$\mathbf{A}_r \approx \mathbf{E}_1 + \mathbf{E}_2 + \cdots + \mathbf{E}_r.$$

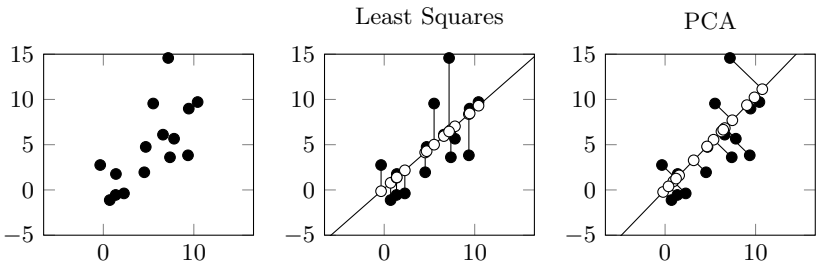
.

Example. Suppose that the heights $\mathbf{h}$ and weights $\mathbf{w}$ of 50 people are measured. Let $\mathbf{A}$ be the 50×2 standardized data matrix with $\mathbf{a}_1 = \mathbf{h} - \bar{\mathbf{h}}$ (the mean-centered height) and $\mathbf{a}_2 = \mathbf{w} - \bar{\mathbf{w}}$ (the mean-centered weight). The matrix $\mathbf{A}^T\mathbf{A}$ is the covariance matrix. If the data is linearly correlated, there is a variable $\mathbf{s}$ that incorporates both a person's height and weight. This variable is in the direction of the first right singular value $\mathbf{u}_1$ and has length σ_1. There is a second variable $\mathbf{s}^\perp$ that accounts for the variation in the data. Taking a rank one approximation $\mathbf{A}_1 = \sigma_1 \mathbf{u}_1 \mathbf{v}_1^T$, eliminates the $\mathbf{s}^\perp$.



(a) The data is correlated and lies for the most part along $\mathbf{u}_1$. (b) The matrix $\mathbf{AV}$ acts as a rotation of the heights and weights onto the x and y axes. (c) By keeping only the first principal component, we discard the y offsets. (d) Back in our original basis, all the data lies along the $\mathbf{s}$ direction. ◀

Note that unlike least squares fit in which the data is projected "vertically" onto the height-weight line, PCA projects the data orthogonally onto the height-weight line. Original data ●. Center: Orthogonal projection of the y-components into the column space of the 15×2 Vandermonde matrix. Right: Orthogonal projection into the first left singular vector. Note the difference in the solutions using least squares and PCA.



Data compression

A raster image is simply a matrix whose entries denote pixel intensity. Perhaps the most natural way to compare two images $\mathbf{A}$ and $\mathbf{B}$ is with a pixel-by-pixel comparison. The l^2-norm over all pixels is given by the Frobenius norm $\|\mathbf{A}-\mathbf{B}\|_F$, which equals the sum of the principal values of $\mathbf{A}-\mathbf{B}$. The image $\mathbf{A}$ can be written as the sum of its orthogonal principal components $\mathbf{A} = \mathbf{E}_1 + \mathbf{E}_2 + \cdots + \mathbf{E}_p$. If we let $\mathbf{A}_r = \mathbf{E}_1 + \mathbf{E}_2 + \cdots + \mathbf{E}_r$ with $0 < r \leq p$, then

$$\|\mathbf{A} - \mathbf{A}_r\|_F = \|\mathbf{E}_{r+1} + \cdots + \mathbf{E}_p\|_F = \sqrt{\sigma_{r+1}^2 + \cdots + \sigma_p^2}$$

An $m \times n$ matrix $\mathbf{A}$ has mn elements. To store the truncated SVD $\mathbf{A}_r = \mathbf{U}_r \mathbf{\Sigma}_r \mathbf{V}_r^T$ requires mr elements for the $\mathbf{U}_r$ and nr elements for $\mathbf{\Sigma}_r \mathbf{V}_r$. So, the mn-element image can be compressed to $(m+n)r$ elements. See Figure 3.3 on the facing page.

Note that images are rarely (if ever) compressed using singular value decomposition simply because there are much better and much faster algorithms such as lossy compression used in jpeg and lossless compression used in png files.

Image recognition

Automated methods for image recognition and detection are important areas of research. One such method, the so-called eigenface method, developed for facial recognition, employs singular value decomposition to

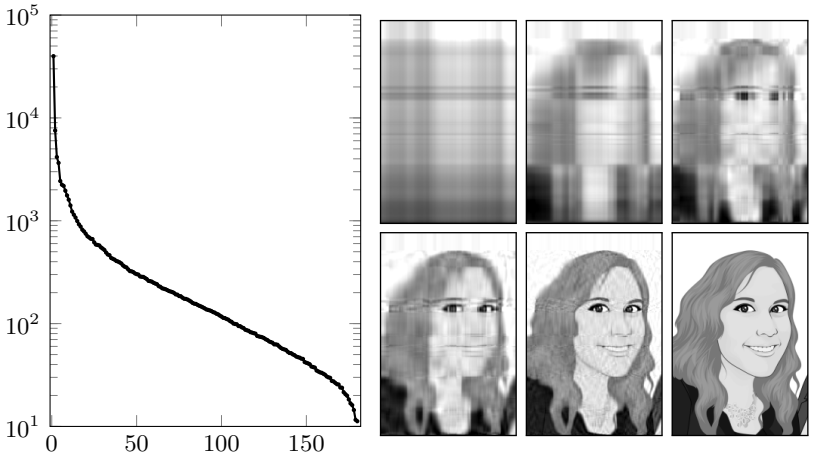


Figure 3.3: Left: The 180 singular values for the 270×180 grayscale image in the lower right. Right: Image approximations for 1, 3, 5, 10, 25, and all 180 singular values.

combine a database of training images to produce a smaller database of orthogonal images called eigenfaces (Sirovich and Kirby [1987]).

A set of n $p \times q$-pixel grayscale training images span an n-dimensional subspace of an $m = pq$-dimensional vector space. Let $\mathbf{t}_j$ be the $m \times 1$ vector formed by reshaping the jth $p \times q$ array. Furthermore, by subtracting the mean $\bar{\mathbf{t}}$ (mean centering) and dividing by the standard deviation $\mathbf{s}$ (standardization), we can reduce the condition number. Let $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{bmatrix}$ with

$$\mathbf{a}_j = \operatorname{diag}^+(\mathbf{s})(\mathbf{t}_j - \bar{\mathbf{t}}).$$

Consider the singular value decomposition $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$. The projection of $\mathbf{A}$ onto the column space of $\mathbf{U}$ is

$$\mathbf{A} = \mathbf{U}(\mathbf{U}^T\mathbf{U})^{-1}\mathbf{U}^T\mathbf{A} = \mathbf{U}\mathbf{W}$$

where $\mathbf{W} = \mathbf{U}^T\mathbf{A} = \boldsymbol{\Sigma}\mathbf{V}^T$ is the signature matrix. The rows of $\mathbf{W}$ are the components of the projection $\mathbf{A}$ onto $\mathbf{U}$. Of course, the columns of $\mathbf{A}$ are already in the column space of $\mathbf{U}$, but the idea will be the same later when we consider the projection of $\mathbf{A}$ into a subspace of $\mathbf{U}$. The only thing we need to store is the signature matrix $\mathbf{W}$ and eigenface matrix $\mathbf{U}$. One of the benefits of the eigenface method is that although we use n training images, we can use substantially lower dimensional subspace

by discarding the singular vectors corresponding to the smallest singular values.

Now, suppose that we are given a new image $\mathbf{t}^*$ and we want to find its closest match in the set of training images. To recognize a new image, we will project the image vector onto the subspace spanned by $\mathbf{U}$ and select the closest image vector from the training set. To *recognize* the new image $\mathbf{t}^*$ we compare the signature vector $\mathbf{U}^T(\mathbf{t}^* - \bar{\mathbf{t}})$ with the columns of $\mathbf{W}$. The closest matching image from the training set is the image j that minimizes the error

$$\varepsilon_j = \|\mathbf{U}^T\mathbf{t}^* - \mathbf{W}_j\|_2.$$

The reconstructed image

$$\mathbf{s}^* = \operatorname{diag}(\mathbf{s})(\mathbf{U}\mathbf{U}^T\mathbf{t}^* + \bar{\mathbf{t}})$$

may also be examined.

Latent semantic indexing

Suppose that you want to determine which documents in some library or database are the best match to a given query. One way to do this is to build a dictionary of all possible terms and identify the documents with vectors in a vector space. The jth element of a vector $\mathbf{d}_i$ tells us the number of times that term j appears in document i. Given a query $\mathbf{q}$, we can find the most relevant documents to the query by finding the closest vector $\mathbf{d}_i$ to $\mathbf{q}$

$$\cos\theta_i = \frac{(\mathbf{q}, \mathbf{d}_i)}{\|\mathbf{q}\|\|\mathbf{d}_i\|}$$

Latent semantic indexing, developed by Susan Dumais of Bell Labs in the 1990s, extends this idea using singular value decomposition to remove statistical noise. Suppose we have m terms and n documents. Let the $m \times n$ term-by-document matrix

$$\mathbf{A} = \begin{bmatrix} \hat{\mathbf{d}}_1 & \hat{\mathbf{d}}_2 & \cdots & \hat{\mathbf{d}}_n \end{bmatrix}$$

where $\hat{\mathbf{d}}_i = \mathbf{d}_i/\|\mathbf{d}_i\|_2$ and $\hat{\mathbf{q}} = \mathbf{q}/\|\mathbf{q}\|_2$ are normalized vectors. Instead, of simply finding

$$\max \mathbf{A}^T\hat{\mathbf{q}}$$

as we did above, we will take

$$\max \mathbf{A}_k^T\hat{\mathbf{q}}$$

where $\mathbf{A}_k = \sum_{i=1}^k \sigma_n \mathbf{u}_n \mathbf{v}_n^T$ is formed using the truncated SVD of $\mathbf{A}$ to reduce the rank to k. By reducing the dimension, the documents and terms are now grouped as concepts.

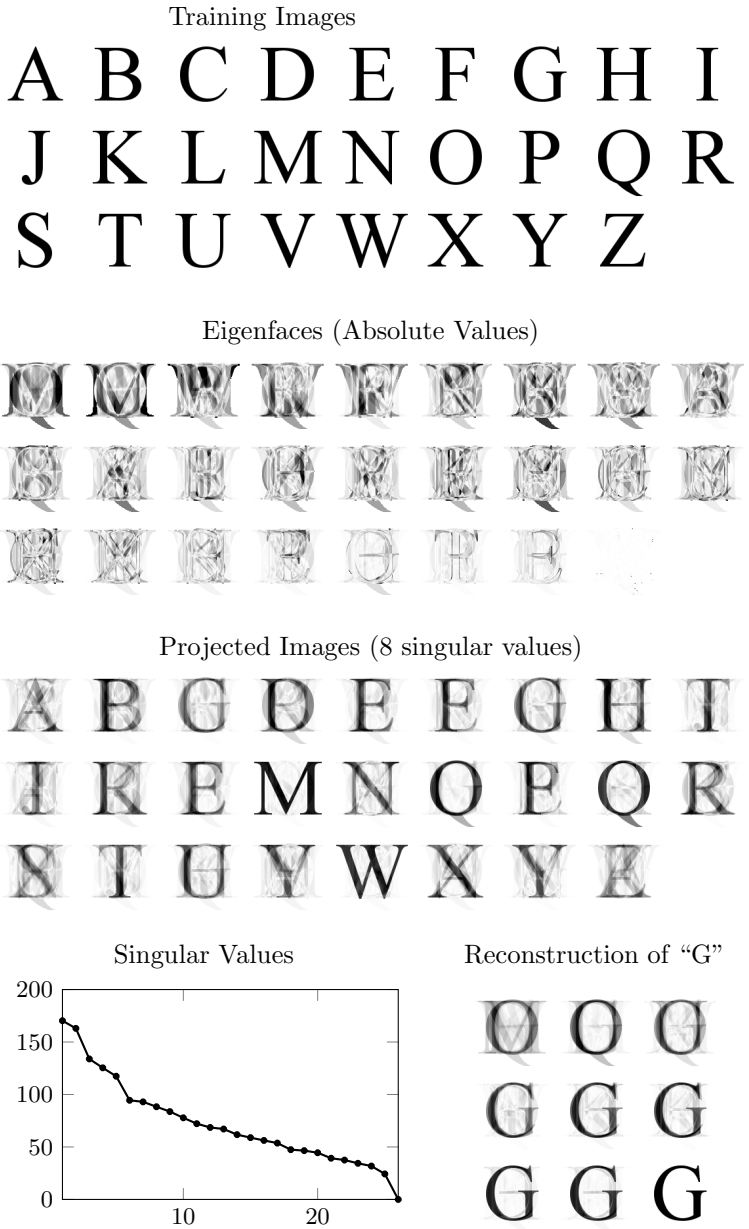


Figure 3.4: These 120×120 pixel images span a 26-dimensional subspace of $\mathbb{R}^{14400}$. Eigenfaces (ordered by their associated singular values) form an orthogonal basis for the subspace spanned by the training images.

Exercises

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

3.2. Find the point(s) $(x, y) \in \mathbb{R}^2$ on the line $y = 1$ closest to the origin using the l^∞-norm.

3.3. Why is it unwise to use the normal equation $\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}$ to solve the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ when $\mathbf{A}$ has a large condition number?

3.4. The continuous L^2-norm is

$$(f, g) = \int_0^1 f(x)g(x)\,\mathrm{d}x.$$

Use the definition of the angle subtended by vectors

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

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

3.5. Prove that the Moore–Penrose pseudoinverse satisfies the properties

(a) $\mathbf{A}\mathbf{A}^+\mathbf{A} = \mathbf{A}$

(b) $\mathbf{A}^+\mathbf{A}\mathbf{A}^+ = \mathbf{A}^+$

(c) $(\mathbf{A}\mathbf{A}^+)^T = \mathbf{A}\mathbf{A}^+$

(d) $(\mathbf{A}^+\mathbf{A})^T = \mathbf{A}^+\mathbf{A}$

✍ 3.6. The Filippelli problem was contrived to test statistical software packages. The NIST "Filip" Statistical Reference Dataset

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

consists of 82 (x, y) data points along with a certified polynomial of degree 10 in x

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

Find the parameters $\{\beta_0, \beta_1, \beta_2, \ldots, \beta_{10}\}$ by constructing a Vandermonde matrix $\mathbf{X}$ with

$$\mathbf{X}_{ij} = x_i^{p-j}$$

$$\mathbf{X} = \begin{bmatrix} x_1^{10} & x_1^9 & \dots & x_1 & 1 \\ x_2^{10} & x_2^9 & \dots & x_2 & 1 \\ & \dots\dots\dots\dots & & & \\ x_{82}^{10} & x_{82}^9 & \dots & x_{82} & 1 \end{bmatrix}$$

using the normal equation `(X'*X)\(X'*y)` and QR decomposition. Determine the residuals and compare the coefficients obtained with the coefficients from the NIST web page. Plot the polynomials as continuous curves along with the data points as dots. What happens if you mean center the data first by subtracting $\bar{x}$ from the x-data?

Hint: The MATLAB function `vander` uses the following code to build the Vandermonde matrix:

```
n = 10;
V(:,1) = ones(length(x),1);
for j = 1:n
   V(:,j+1) = x.*V(:,j);
end
```

To construct a polynomial of order 10 using 200 points between x_1 and x_{82}, you can use

```
xp = linspace(min(x),max(x),200);
yp = vander(xp,n)*beta;
```

3.7. Implement the eigenface method. The training images will consist of the capital letters: `A`, `B`, ...`Z`. The test images to recognize are the following: the letter `A` with garble, the character `B` in a different font,a letter `K` offset from center, a slightly skew letter `M`, the letter `P` with noise, and a "pixelated" letter `S`.

(a) Use the training images to create an orthonormal basis $\mathbf{U}$ and signature matrix $\mathbf{W}$. For each test image, plot the original image $\mathbf{s}$ along with the reconstructed image $\mathbf{s}^*$. Compute the errors ε_j to determine which training images give the closest matches to the test images. Typically, if the error fails to be below a given threshold, image detection is said to have failed. Explain why the eigenface method worked in choosing the correct training image or why it failed.

(b) Singular values that are zero contain no information and singular values that are close to zero provide only minor adjustments to A. Plot the singular values. What does the graph say about the effect of

using all 26 eigenvectors versus a smaller number of them as a basis for the eigenface basis?

(c) By zeroing out the singular values that are close to zero, we reduce storage and computing requirements and produce a close approximation to the original matrix $\mathbf{A}$. Use the subspace spanned by the first 5 columns of $\mathbf{U}$ by using a truncated SVD. Compute the projection $\mathbf{A}_5$ of $\mathbf{A}$ into this subspace by computing $\mathbf{A}_5 = \mathbf{U}_5\mathbf{\Sigma}_5\mathbf{V}_5^T$. Compare a few projected letters with the originals to see the effect of the projection.

(d) Repeat part (b) using this reduced subspace.

Hint: The following MATLAB code may be helpful.

- To import and reshape the image files:

```
for i = 1:26
  letter(:,:,i) = imread('alphabet/' 'A'+i-1 '.png']);
  A(:,i) = reshape(letter(:,:,i),[],1);
end
```

- To reshape and view the images (for example, the letter "E"):

```
E = reshape(A(:,5),[120 120])
image(E*256); colormap(gray(256)); axis square;
```

- To implement the truncated SVD:

```
[u,s,v] = svds(A,5);
```

CHAPTER 4

Eigenvalue Problems

Using pencil and paper, one often finds a the eigenvalues of a matrix by determining the zeros of the characteristic polynomial $\det(\mathbf{A} - \lambda\mathbf{I})$. This approach is useful for small matrices, but we cannot hope to apply it to a general matrix larger than 4×4. Abel's impossibility theorem states that there are no general solutions to polynomial equations of degree five or higher using a finite number of algebraic operations (additions, multiplications and radicals). As a consequence, there can be no direct method for determining eigenvalues of a general matrix of rank five or higher. We must find the eigenvalues using iterative methods instead.

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

Ⓜ The MATLAB command `roots` finds the roots of a polynomial $p(x)$ by computing `eig` for the companion matrix of $p(x)$.

Before we discuss the methods, let's look at estimation and stability of eigenvalues.

4.1 Eigenvalue sensitivity

Suppose that $\mathbf{A}$ is semisimple with the diagonalization $\mathbf{\Lambda} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S}$ and suppose that $\delta\mathbf{A}$ is some change in $\mathbf{A}$. Then the perturbation in the eigenvalues is

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

and hence

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

Taking the matrix norm

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

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

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

$$\mathbf{y}^H\mathbf{A} = \lambda\mathbf{y}^H.$$

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

$$\delta\mathbf{A}\mathbf{x} + \mathbf{A}\delta\mathbf{x} = \delta\lambda\mathbf{x} + \lambda\delta\mathbf{x}.$$

Multiplying by the left eigenvector

$$\mathbf{y}^H\delta\mathbf{A}\mathbf{x} + \mathbf{y}^H\mathbf{A}\delta\mathbf{x} = \mathbf{y}^H\delta\lambda\mathbf{x} + \mathbf{y}^H\lambda\delta\mathbf{x}.$$

Since $\mathbf{y}^H\mathbf{A}\delta\mathbf{x} = \mathbf{y}^H\lambda\delta\mathbf{x}$, we have that

$$\mathbf{y}^H\delta\mathbf{A}\mathbf{x} = \mathbf{y}^H\delta\lambda\mathbf{x}$$

and so

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

Therefore,

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

We define the *eigenvalue condition number* to be

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

that is, one over the cosine of the angle between the right and left eigenvectors.

Ⓜ The MATLAB function `condeig` returns the eigenvalue condition number.

Finding the eigenvalues of a matrix $\mathbf{A}$ is the same as determining for which λ is $\mathbf{A} - \lambda\mathbf{I}$ singular. Numerically, asking this question is the wrong approach. Instead, a we should ask when $\mathbf{A} - \lambda\mathbf{I}$ is close to singular, that

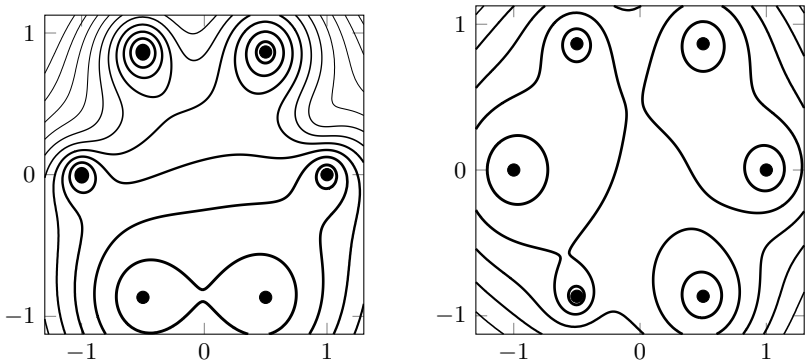


Figure 4.1: Contours of ε-pseudospectra of two similar matrices. Each contour represents a change of $\varepsilon = 0.0005$. For each matrix the 2-condition number is on the order of 10^6 and the condition number of the matrix of eigenvectors is on the order of 10^3.

is, when $\|(\mathbf{A} - \lambda\mathbf{I})^{-1}\|$ is large. In this case, is is useful to talk about the pseudospectrum of the matrix $\mathbf{A}$. Recall that we define the spectrum of a matrix $\mathbf{A}$ as the set of its eigenvalues

$$\lambda(\mathbf{A}) = \{z \in \mathbb{C} \mid \mathbf{A} - z\mathbf{I} \text{ is singular.}\}.$$

We define the ε-pseudospectrum of $\mathbf{A}$ as the set

$$\lambda_\varepsilon(\mathbf{A}) = \{z \in \mathbb{C} \mid \|(\mathbf{A} - z\mathbf{I})^{-1}\| \geq \frac{1}{\varepsilon}\}.$$

In other words, z is an eigenvalue of $\mathbf{A} + \mathbf{E}$ with $\|\mathbf{E}\| < \varepsilon$. For the Euclidean norm,

$$\lambda_\varepsilon(\mathbf{A}) = \{z \in \mathbb{C} \mid \sigma_{\min}(\mathbf{A} - z\mathbf{I}) \leq \varepsilon\}.$$

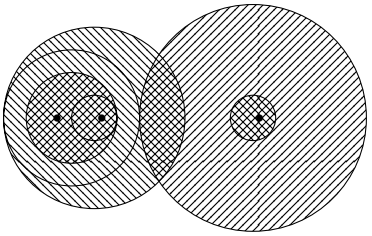
See Figure 4.1.

4.2 Eigenvalue estimation

Theorem 11 (Gershgorin Circle Theorem)**.** *Let* $\mathbf{A} \in \mathbb{C}^{n\times n}$*. Then*

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

The sets $\mathcal{R}_i$ *are called Gershgorin circles.*



$$\mathbf{A} = \begin{bmatrix} 10 & 2 & 3 \\ 1 & 2 & 1 \\ 0 & 1 & 3 \end{bmatrix}$$

Figure 4.2: Gershgorin circles in the complex plane and the eigenvalues of $\mathbf{A}$. The shaded regions show the row circles ▨ and column circles ▧. Eigenvalues (roughly $\{10.3, 3.3, 1.4\}$) are in the intersections ▩.

Proof. Let's split $\mathbf{A}$ by the diagonal and off-diagonal elements:

$$\mathbf{A} = \mathbf{D} + \mathbf{M} \quad \text{where} \quad \mathbf{D} = \operatorname{diag} \mathbf{A}.$$

For each $\lambda \in \sigma(\mathbf{A})$ with $\lambda \neq a_{ii}$, we introduce the matrix

$$\mathbf{B}_\lambda = \mathbf{A} - \lambda \mathbf{I} = (\mathbf{D} - \lambda \mathbf{I}) + \mathbf{M}.$$

There is a nonzero vector $\mathbf{x} \in \mathbb{C}^m$ (an eigenvector of $\mathbf{A}$) in the nullspace of $\mathbf{B}_\lambda$, so

$$\mathbf{B}_\lambda \mathbf{x} = ((\mathbf{D} - \lambda \mathbf{I}) + \mathbf{M})\mathbf{x} = 0.$$

Equivalently,

$$\mathbf{x} = (\mathbf{D} - \lambda \mathbf{I})^{-1} \mathbf{M} \mathbf{x}.$$

In the $\|\cdot\|_\infty$ 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,

$$1 \leq \|(\mathbf{D} - \lambda \mathbf{I})^{-1} \mathbf{M}\|_\infty = \sup_{1 \leq k \leq n} \sum_{\substack{j=1 \\ j \neq k}}^{n} \frac{|a_{jk}|}{|a_{kk} - \lambda|}$$

Because $\mathbf{M}$ is the matrix with off-diagonal elements of $\mathbf{A}$ and $(\mathbf{D} - \lambda \mathbf{I})^{-1}$ is a diagonal matrix with elements $(a_{jk} - \lambda)^{-1}$. So,

$$1 \leq \sum_{\substack{j=1 \\ j \neq k}}^{n} \frac{|a_{jk}|}{|a_{kk} - \lambda|}$$

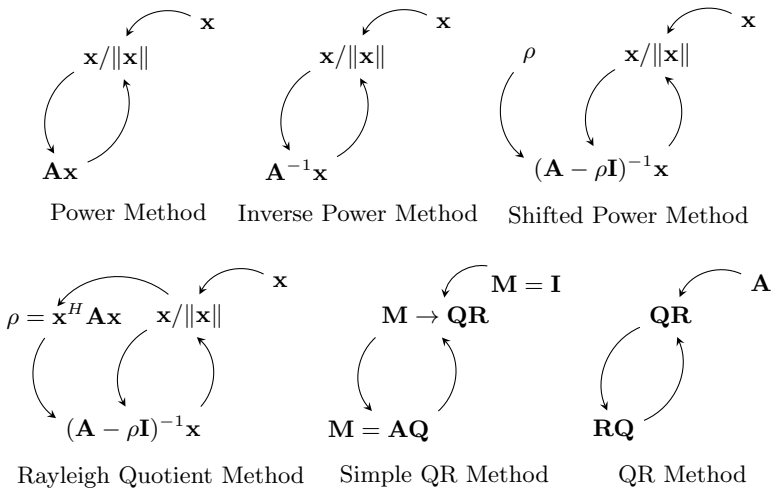


Figure 4.3: Power method and some of its variations.

for some $1 \leq k \leq n$ from which it follows that

$$|a_{kk} - \lambda| \leq \sum_{\substack{j=1 \\ j \neq k}}^{n} |a_{jk}|$$

for that specific k. Hence, $\lambda \in \mathcal{R}_k$. $\square$

Of course, in the the above theorem we could have used the $\|\cdot\|_1$ norm instead. This approach would give column circles instead of row circles:

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

Theorem 12. *Let* $\mathbf{A} \in \mathbb{C}^{n \times n}$. *Then* $\sigma(\mathbf{A}) \subseteq \mathcal{S}_{\mathcal{R}} \cap \mathcal{S}_{\mathcal{C}}$.

4.3 The power method

Power iteration

Let $\mathbf{A} \in \mathbb{C}^{n \times n}$ be a diagonalizable matrix with eigenvalues $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$. We say that λ_1 is the *dominant eigenvalue* of $\mathbf{A}$. For some initial guess $\mathbf{x}^{(0)} \neq 0$, let

$$\mathbf{x}^{(k)} = \mathbf{A}\mathbf{x}^{(k-1)}.$$

Then $x^{(k)}$ will converge to the eigenvector $\mathbf{x}_1$ corresponding to dominant eigenvalue λ_1.

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

$$\mathbf{x}^{(0)} = \alpha_1\mathbf{x}_1 + \alpha_2\mathbf{x}_2 + \cdots + \alpha_n.$$

By applying $\mathbf{A}$ to this initial vector, we have

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

And after k iterations

$$\begin{aligned}\mathbf{A}^k\mathbf{x}^{(k)} &= \alpha_1\lambda_1^k\mathbf{x}_1 + \alpha_2\lambda_2^k\mathbf{x}_2 + \cdots + \alpha_n\lambda_n^k\mathbf{x}_n \\ &= \alpha_1\lambda_1^k\left(\mathbf{x}_1 + \frac{\alpha_2}{\alpha_1}\left(\frac{\lambda_2}{\lambda_1}\right)^k\mathbf{x}_2 + \cdots + \frac{\alpha_n}{\alpha_1}\left(\frac{\lambda_n}{\lambda_1}\right)^k\mathbf{x}_n\right) \\ &\to \lambda_1^k\alpha_1\mathbf{x}_1\end{aligned}$$

To prevent over or underflow, we normalize $\mathbf{x}^{(k)}$ at each iteration so that $\|\mathbf{x}^{(k)}\| = 1$:

$$\begin{aligned}\mathbf{z}^{(k)} &= \mathbf{A}\mathbf{q}^{(k-1)} \\ \mathbf{q}^{(k)} &= \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|} \\ \lambda_1^{(k)} &= \mathbf{q}^{(k)H}\mathbf{A}\mathbf{q}^{(k)}\end{aligned}$$

Stop when the error

$$\left\|\mathbf{e}^{(k)}\right\| = \left\|\mathbf{q}^{(k-1)} - \operatorname{sign}(q_1^{(k-1)}q_1^{(k)})\mathbf{q}^{(k)}\right\| \quad \text{or} \quad 1 - |\mathbf{q}^{(k-1)H}\mathbf{q}^{(k)}|$$

is within your prescribed tolerance.

The power method converges linearly as $O(|\lambda_2/\lambda_1|^k)$. Without loss of generality, take $\{\mathbf{x}_i\}$ with $\|\mathbf{x}_i\| = 1$ for $i = 1, \ldots, n$. Then the error at the kth iterate is

$$\begin{aligned}\left\|\mathbf{q}^{(k)} - \mathbf{x}_1\right\| &= \left\|\mathbf{x}_1 + \sum_{i=2}^n\left[\frac{\alpha_i}{\alpha_1}\left(\frac{\lambda_2}{\lambda_1}\right)^k\mathbf{x}_i\right] - \mathbf{x}_1\right\| = \left\|\sum_{i=2}^n\left[\frac{\alpha_i}{\alpha_1}\left(\frac{\lambda_2}{\lambda_1}\right)^k\mathbf{x}_i\right]\right\| \\ &= \left|\frac{\lambda_2}{\lambda_1}\right|^k\left\|\sum_{i=2}^n\frac{\alpha_i}{\alpha_1}\mathbf{x}_i\right\| \le C\left|\frac{\lambda_2}{\lambda_1}\right|^k\end{aligned}$$

For the Euclidean norm $C = \left(\sum_{i=2}^n(\alpha_i/\alpha_1)^2\right)^{1/2}$.

Inverse and shifted power method

The power method itself only gets us the dominant eigenvector. Getting this much may be enough for many problems (see problem 4.4), but sometimes we may need some or all of them. We can extend the power method to get other eigenvectors.

Suppose that the spectrum of $\mathbf{A}$ is

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

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

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

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 if $\rho \in \mathbb{C}$, the spectrum of $\mathbf{A} - \rho\mathbf{I}$ is

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

with the same associated eigenspace.

This reveals a clever approach to get the other eigenvectors called *shift-and-invert*. We make a good guess of the value of one of the eigenvalues, say $\rho \approx \lambda_i$ for some i. Now apply power iteration to $(\mathbf{A} - \rho\mathbf{I})^{-1}$. We will recover the eigenvector associated with the eigenvalue λ_i. The ratio of convergence is $|\lambda_i - \rho|/|\lambda_k - \rho|$ where λ_k is the next closest eigenvalue to ρ.

For each iteration, we must apply $\mathbf{q}^{(k+1)} = (\mathbf{A} - \rho\mathbf{I})^{-1}\mathbf{q}^{(k)}$. In practice, we solve

$$(\mathbf{A} - \rho\mathbf{I})\mathbf{z}^{(k+1)} = \mathbf{q}^{(k)} \quad \text{with} \quad \mathbf{q}^{(k)} = \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|}$$

Solving this system using Gaussian elimination requires $\frac{2}{3}n^3$ flops for LU-decomposition and then only $2n^2$ flops for each subsequent iteration.

Rayleigh quotient iteration

The rate of convergence for the shift-and-invert method depends on having a good initial guess for the shift ρ. It would seem that we could improve the method by refining the shift ρ at each iteration. One way we can do this is use the eigenvalue equation

$$\mathbf{A}\mathbf{x}^{(k)} = \rho^{(k)}\mathbf{x}^{(k)}.$$

Unless $\rho^{(k)}$ and $\mathbf{x}^{(k)}$ are already eigenvalues of $\mathbf{A}$, this equation is inconsistent. Instead, we will find the "best" $\rho^{(k)}$. That is, let's determine $\rho^{(k)}$ that minimizes the 2-norm of the residual

$$\|\mathbf{r}\|_2 = \|\mathbf{A}\mathbf{x}^{(k)} - \rho\mathbf{x}^{(k)}\|_2$$

At the minimum $\frac{\mathrm{d}}{\mathrm{d}\rho}\|\mathbf{r}\|_2^2 = 0$:

$$\begin{aligned} 0 &= \tfrac{\mathrm{d}}{\mathrm{d}\rho}\mathbf{A}\mathbf{x}^{(k)} - \rho\mathbf{x}^{(k)}\|_2^2 \\ &= \tfrac{\mathrm{d}}{\mathrm{d}\rho}\|\mathbf{A}\mathbf{x}^{(k)}\|_2^2 - 2\rho\mathbf{x}^H\mathbf{A}\mathbf{x} + \rho^2\|\mathbf{x}\|_2^2 \\ &= -2\mathbf{x}^H\mathbf{A}\mathbf{x} + 2\rho\mathbf{x}^H\mathbf{x} \end{aligned}$$

Therefore, the "best" choice for ρ is

$$\rho = \frac{\mathbf{x}^H\mathbf{A}\mathbf{x}}{\mathbf{x}^H\mathbf{x}}$$

This number, called the *Rayleigh quotient*, can be thought of as an eigenvalue approximation. By the Courant–Fischer theorem

$$\lambda_{\min} \le \rho \le \lambda_{\max}$$

for symmetric, positive-definite matrix.

Combining the Rayleigh quotient approximation to the eigenvalues with the shift-and-invert method we get for each iteration k:

$$\begin{aligned} \rho^{(k)} &= \frac{\mathbf{x}^{(k)H}\mathbf{A}\mathbf{x}^{(k)}}{\mathbf{x}^{(k)H}\mathbf{x}^{(k)}} \\ (\mathbf{A} - \rho^{(k)}\mathbf{I})\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} \\ \mathbf{x}^{(k+1)} &= \frac{\mathbf{x}^{(k+1)}}{\|\mathbf{x}^{(k+1)}\|} \end{aligned}$$

Rayleigh quotient iteration is locally cubically convergent if $\mathbf{A}$ is Hermitian and quadratically convergent otherwise, for an appropriate initial guess for the eigenvector. See Demmel [1997]. Like many nonlinear iterative methods (such as Newton's method), Rayleigh iteration exhibits basins of attraction and sensitivity to initial conditions. See Figure 4.4 on the facing page.

4.4 The QR method

QR iteration

QR iteration is an extension of the power method to get all the eigenvectors simultaneously rather than one at a time. Recall the idea of the

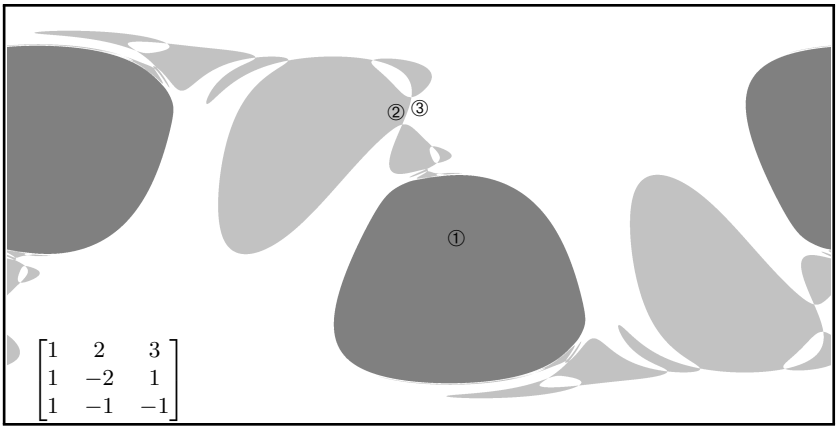


Figure 4.4: Basins of attraction for Rayleigh iteration in the (φ, θ)-plane. Initial guesses in ■ regions converge to the eigenvector at ①, guesses in ■ regions converge to the eigenvector at ②, and guesses in □ regions converge to the eigenvector at ③.

power method. Let $\mathbf{A} \in \mathbb{R}^{n \times n}$. Choose an initial vector $\mathbf{x}^{(1)}$. 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.

Now, suppose we start with n initial vectors and we want to get all n eigenvectors $\{\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n\}$ with $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$. Let's develop a method so that the first vector sequence converges to $\mathbf{v}_1$, the second vector sequence to 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 as they span $\mathbb{R}^n$. So, let's take the standard unit vectors $\{\mathbf{x}_1^{(0)}, \mathbf{x}_2^{(0)}, \ldots, \mathbf{x}_n^{(0)}\} = \mathbf{I} = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n\}$ as our initial vectors.

It should be clear that if we apply the power method directly to $\mathbf{I}$, all the columns will converge to the dominant eigenvector unless one happens to already be an eigenvector. Let's fix this problem and get all the eigenvectors. We won't do anything to the first column $\mathbf{x}_1^{(k)}$ except normalize it to $\mathbf{q}_1^{(k)}$. It converges to the dominant eigenvector $\mathbf{v}_1$, and within several iterations it gives 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

$\text{span}\{\mathbf{x}_1^{(k)}\}^\perp$. Since $\mathbf{q}_1^{(k)}$ 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 growth $\mathbf{q}_2^{(k)}$ is now dominated by the $\mathbf{v}_2$. Let $\mathbf{q}_3^{(k)}$ be the orthogonal projection of $\mathbf{x}_3^{(k)}$ into $\text{span}\{\mathbf{x}_1^{(k)}, \mathbf{x}_2^{(k)}\}^\perp = \text{span}\{\mathbf{q}_1^{(k)}, \mathbf{q}_2^{(k)}\}^\perp$. Since $\mathbf{q}_1^{(k)}$ is close to $\mathbf{v}_1$ and $\mathbf{q}_2^{(k)}$ is close to $\mathbf{v}_2$, this projection will ensure that $\mathbf{q}_3^{(k)}$ 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 $\mathbf{A}$: $\mathbf{M}_k = \mathbf{A}\mathbf{Q}_{k-1}$
2. Take the QR-decomposition of the resulting matrix: $\mathbf{Q}_k\mathbf{R}_k = \mathbf{M}_k$

We can write the kth iteration as

$$\mathbf{Q}_k\mathbf{R}_k = \qquad \mathbf{M}_k = \qquad \mathbf{A}\mathbf{Q}_{k-1}. \tag{4.1}$$

It's not yet clear what this equation means. Let's change the basis to $\mathbf{Q}_{k-1}$. To do so, we simply need to "rotate" by $\mathbf{Q}_{k-1}^{-1} = \mathbf{Q}_{k-1}^T$. Multiplying (4.1) by $\mathbf{Q}_{k-1}^T$ gives us

$$\mathbf{Q}_{k-1}^T\mathbf{Q}_k\mathbf{R}_k = \mathbf{Q}_{k-1}^T\mathbf{M}_k = \mathbf{Q}_{k-1}^T\mathbf{A}\mathbf{Q}_{k-1}. \tag{4.2}$$

This says that $\mathbf{Q}_{k-1}^T\mathbf{Q}_k\mathbf{R}_k$ is unitarily similar to $\mathbf{A}$. So, $\mathbf{Q}_{k-1}^T\mathbf{Q}_k\mathbf{R}_k$ has the same eigenvalues as $\mathbf{A}$. If the iteration converges, then $\mathbf{Q}_{k-1} \to \mathbf{Q}_k$ and so $\mathbf{Q}_{k-1}^T\mathbf{Q}_k \to \mathbf{I}$ and we are left with $\mathbf{R}_k$ on the left-hand side. The matrix $\mathbf{R}_k$ 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 $\mathbf{Q}_{k-1}$ to $\mathbf{Q}_k$. That is, let $\mathbf{Q}_k = \hat{\mathbf{Q}}_k\mathbf{Q}^{k-1}$. Then

$$\mathbf{Q}_k = \hat{\mathbf{Q}}_k\hat{\mathbf{Q}}_{k-1}\cdots\hat{\mathbf{Q}}_2\hat{\mathbf{Q}}_1$$

and $\hat{\mathbf{Q}}_1 = \mathbf{Q}_0 = \mathbf{I}$. And (4.2) is equivalent to

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

where $\hat{\mathbf{M}}_k = \mathbf{Q}_{k-1}^T\mathbf{M}_k$. From (4.2) we have that $\mathbf{R}_k\mathbf{Q}_{k-1}^T = \mathbf{Q}_k^T\mathbf{A}$, and hence $\mathbf{R}_{k-1}\mathbf{Q}_{k-2}^T = \mathbf{Q}_{k-1}^T\mathbf{A}$. So,

$$\hat{\mathbf{Q}}_k\mathbf{R}_k = \hat{\mathbf{M}}_k = \mathbf{R}_{k-1}\mathbf{Q}_{k-2}^T\mathbf{Q}_{k-1} = \mathbf{R}_{k-1}\hat{\mathbf{Q}}_{k-1}$$

We are left with the simple implementation. For each iteration,

$$\begin{cases} \text{Factor} & \mathbf{T}_k \to \mathbf{Q}_k\mathbf{R}_k \\ \text{Form} & \mathbf{T}_{k+1} \leftarrow \mathbf{R}_k\mathbf{Q}_k \end{cases}$$

with $\mathbf{T}_0 = \mathbf{A}$.

The QR algorithm provides us with the Schur decomposition of a matrix

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

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

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

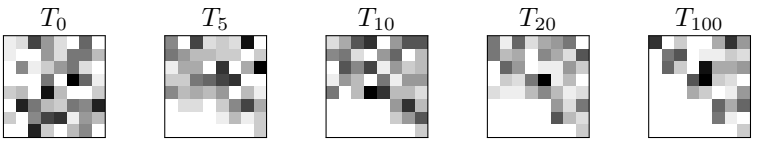
Continuing...

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

Because $(\mathbf{Q}_0 \cdots \mathbf{Q}_k)$ 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 QR method applied to a matrix at iteration T_0, T_5, T_{10}, T_{20}, and T_{100}:



Convergence of the QR method

Because the QR-method uses the power method, convergence is generally slow. The diagonal elements of 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

$$\lim_{k \to \infty} R_k = \begin{bmatrix} \lambda_1 & r_{12} & \cdots & r_{1n} \\ 0 & \lambda_2 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

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 can improve the QR algorithm by doing simple things: first reduce the matrix to upper Hessenberg form and then shift the matrix in each iteration to speed up convergence.

4.5 Speeding up the QR method

The QR decomposition takes $O(n^3)$ operations. We need to repeat it $O(n)$ times to get the n eigenvalues. Hence the net cost is $O(n^4)$. We can speed up the QR method using upper Hessenberg matrices, shifting, and deflation.

QR algorithm with upper Hessenberg matrices

Recall that it is impossible in general (because of Abel's theorem) to determine the Schur decomposition of a matrix $\mathbf{A}$, i.e., an upper triangular matrix which is unitarily similar to a matrix $\mathbf{A}$. But we can get an almost upper triangular matrix. We can find an upper Hessenberg matrix unitarily similar to $\mathbf{A}$ in a $O(n^3)$ operations. 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 is $O(n^3)$. 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 get the QR decomposition. So, a QR step applied to an upper Hessenberg matrix requires at most $O(n^2)$ operations. For a Hermitian matrix, each QR step take $O(n)$ operations because we are working on a tridiagonal matrix.

Let's outline how to get an upper Hessenberg matrix. We can do this using Householder reflectors in a manner similar to QR decomposition. We first construct a Householder reflector $\mathbf{Q}_1$ to zero out the elements below the first subdiagonal element. To do this, partition $\mathbf{A}$ as

$$\mathbf{A} = \begin{bmatrix} a_{11} & \mathbf{c}^T \\ \mathbf{b} & \hat{\mathbf{A}} \end{bmatrix}$$

and let $\mathbf{Q}_1$ be

$$\mathbf{Q}_1 = \begin{bmatrix} 1 & \mathbf{0}^T \\ \mathbf{0} & \hat{\mathbf{Q}}_1 \end{bmatrix}$$

where $\hat{\mathbf{Q}}_1$ is the reflector that takes

$$\hat{\mathbf{Q}}_1\mathbf{b} = (-\|\mathbf{b}\|_2, 0, \ldots, 0)^T$$

Because $\hat{\mathbf{Q}}_1$ is a Householder reflector, $\hat{\mathbf{Q}}_1 = \hat{\mathbf{Q}}_1^{-1} = \hat{\mathbf{Q}}_1^T$. Note that $\mathbf{Q}_1$ is a block matrix with the identity matrix in the upper-left block. So, left-multiplying a matrix by $\mathbf{Q}_1$ does not change the first row and right-multiplying a matrix by $\mathbf{Q}_1$ does not change the first column.

$$\begin{aligned}
\mathbf{A}_1 = \mathbf{Q}_1\mathbf{A}\mathbf{Q}_1^T &= \begin{bmatrix} 1 & & \\ & \hat{\mathbf{Q}}_1 & \\ & & \end{bmatrix} \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} 1 & & \\ & \hat{\mathbf{Q}}_1 & \\ & & \end{bmatrix} \\
&= \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} 1 & & \\ & \hat{\mathbf{Q}}_1 & \\ & & \end{bmatrix} \\
&= \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \end{bmatrix}
\end{aligned}$$

We construct a similar Householder reflector $\mathbf{Q}_2$ to zero out the elements below the second subdiagonal.

$$\begin{aligned}
\mathbf{A}_2 = \mathbf{Q}_2\mathbf{A}\mathbf{Q}_2^T &= \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \end{bmatrix} \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \end{bmatrix} \\
&= \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \\ & & \bullet & \bullet & \bullet \\ & & \bullet & \bullet & \bullet \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & \hat{\mathbf{Q}}_2 & \\ & & & \end{bmatrix} \\
&= \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet & \bullet \\ & & \bullet & \bullet & \bullet \\ & & \bullet & \bullet & \bullet \end{bmatrix}
\end{aligned}$$

Continuing in this manner, we will form an upper Hessenberg matrix that is unitarily similar to $\mathbf{A}$. Since it is similar, it has the same eigenvalues. Of course, if $\mathbf{A}$ is Hermitian then, the upper Hessenberg matrix is a tridiagonal matrix.

Ⓜ The MATLAB command `hess` provides the unitarily similar upper Hessenberg form of a matrix using LAPACK.

Shifted QR iteration

Recall that if the eigenvalues of $\mathbf{A}$ are $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$, then the eigenvalues of $\mathbf{A} - \rho\mathbf{I}$ are $\{\lambda_1 - \rho, \lambda_2 - \rho, \ldots, \lambda_n - \rho\}$. The diagonal elements

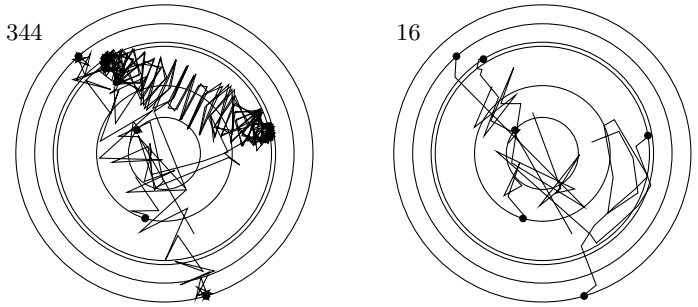


Figure 4.5: Convergence of the QR method (left) and the shifted QR method (right) to eigenvalues of a 6×6 complex matrix. The eigenvectors are depicted by $\bullet$ and lines show the paths of convergence. The regular QR method takes 365 iterations to converge to an error of 10^{-3} and the shifted QR method requires only 16 iterations. Notice that convergence is slower to eigenvalues whose magnitudes are close.

of $\mathbf{A}_k$ converge to the eigenvalues linearly with errors of the subdiagonals given by

$$|a_{i,i-1}^{(k)}| = O\left(\left|\frac{\lambda_i}{\lambda_{i-1}}\right|^k\right).$$

We can speed up convergence by applying QR iteration to $\mathbf{A} - \rho\mathbf{I}$ and choosing ρ close to an eigenvalue. This will speed up convergence of one of the eigenvalues. Once that eigenvalue has converged, we move to another eigenvalue.

Of course, the best choice for ρ would be an eigenvalue. The next best choice for ρ is an approximation to the eigenvalue. This is the same idea that we used in picking the shift in the Rayleigh iteration. Since the diagonal elements of $\mathbf{A}_k$ converge to the eigenvalues of $\mathbf{A}$, they are the natural choices for ρ.

Start with the lower right corner $i = n$. Recall that this diagonal element will converge to the smallest eigenvalue. At each iteration:

1. Set $\rho = a_{ii}^{(k)}$.
2. Factor $\mathbf{A}_k - \rho\mathbf{I} \to \mathbf{Q}_{k+1}\mathbf{R}_{k+1}$
3. Restore $\mathbf{A}_{k+1} \leftarrow \mathbf{R}_{k+1}\mathbf{Q}_{k+1} + \rho\mathbf{I}$

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

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

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

The convergence with each shift is

$$\frac{\lambda_i - \rho}{\lambda_{i-1} - \rho}$$

where $\rho \approx \lambda_i$. Overall, we get quadratic convergence. For Hermitian matrices, where the eigenvectors are orthogonal, we get cubic convergence. This type of shifting is called *Rayleigh quotient shifting*.

Occasionally, Rayleigh quotient shifting will fail to converge. For example, consider

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

In this case, we take $\rho = 2$. The actual eigenvalues of $\mathbf{A}$ are $\lambda = 1$ and 3. The shifted matrix is the orthogonal matrix

$$\mathbf{A} - \rho\mathbf{I} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

with eigenvalues ± 1 are each of equal size. Each eigenvector pulls equal amounts and the algorithm cannot decided to which eigenvalue to go.

One fix to this problem is to instead use the *Wilkinson shift*. The Wilkinson shift defines ρ to be the eigenvalue the submatrix

$$\begin{bmatrix} a_{i-1,i-1}^{(m-1)} & a_{i-1,i}^{(m-1)} \\ a_{i,i-1}^{(m-1)} & a_{i,i}^{(m-1)} \end{bmatrix}$$

that is closest to $a_{i,i}^{(m-1)}$. The eigenvalues of a 2×2 matrix are easily computed by using the quadratic formula to solve the characteristic equation.

Deflation

Recall that the total cost is $\frac{7}{3}n^3$ for the QR decomposition and $O(n)$ multiplication of $\mathbf{RQ}$. We can speed up the method by using *deflation*. As soon as $|a_{m,m-1}| <$ tolerance, the eigenvalue $\lambda_m \approx a_{m,m}$ and we move the shift ρ to $a_{m-1,m-1}$ to recover λ_m. To reduce the number of computations at the next step, we can consider the $(m-1) \times (m-1)$ principal submatrix obtained by removing the last row and last column. We continue QR decomposition on this smaller matrix. And continue like this.

Once a submatrix of size 1×1 has been isolated. It contains an eigenvalue. Complex eigenvalues of real matrices are extracted in pairs.

4.6 Implicit QR

At each step of the shifted QR method, we compute

$$(\mathbf{A} - \rho\mathbf{I}) = \mathbf{QR}, \qquad \hat{\mathbf{A}} = \mathbf{RQ} + \rho\mathbf{I} \tag{4.3}$$

where $\mathbf{A}$ is in proper upper Hessenberg form. This gives us the similarity transform

$$\hat{\mathbf{A}} = \mathbf{Q}^T\mathbf{AQ}. \tag{4.4}$$

In practice, we don't need (or necessarily want) to compute the QR decomposition (4.3) explicitly. When the shift ρ is close to an eigenvalue (which is what we want it to be) the QR method is prone to round-off and hence may be unstable. The solution is to never explicitly subtract off $\rho\mathbf{I}$ and in fact 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, it is no more difficult than 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 apply the same Givens rotation $\mathbf{Q}_1^T$ to our original matrix $\mathbf{A}$.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ & a_{32} & a_{33} & a_{34} & a_{35} \\ & & a_{43} & a_{44} & a_{45} \\ & & & a_{44} & a_{55} \end{bmatrix} \rightarrow \begin{bmatrix} * & * & * & * & * \\ \rho s & x+\rho c & * & * & * \\ & a_{32} & a_{33} & a_{34} & a_{35} \\ & & a_{43} & a_{44} & a_{45} \\ & & & a_{44} & a_{55} \end{bmatrix}$$

This time element $(2,1)$ is changed from a_{21} to ρs and element $(2,2)$ is changed from a_{22} to $x + \rho c$ where s and c are the Givens rotations $\sin\theta$ and $\cos\theta$. Now, right-multiply by $\mathbf{Q}_1$ to complete the similarity transform

$$\rightarrow \begin{bmatrix} * & * & * & * & * \\ sx & cx & * & * & * \\ sa_{32} & ca_{32} & a_{33} & a_{34} & a_{35} \\ & & a_{43} & a_{44} & a_{45} \\ & & & a_{44} & a_{55} \end{bmatrix}$$

In the next step we need a Givens rotation $\mathbf{Q}_2^T$ to zero out a_{32}. Exactly the same $\mathbf{Q}_2^T$ works for both $\mathbf{A}$ and $\mathbf{A} - \rho\mathbf{I}$. The same is true for $\mathbf{Q}_3^T$ and so on.

So, all we really 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. 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 the Implicit Q theorem.

Theorem 13 (Implicit Q theorem)**.** *If $\mathbf{H} = \mathbf{Q}^H\mathbf{A}\mathbf{Q}$ be the reduction of a matrix $\mathbf{A}$ to Hessenberg form and the elements in the lower diagonal of $\mathbf{H}$ are non-zero, then $\mathbf{Q}$ and $\mathbf{H}$ are uniquely determined by the first column of $\mathbf{Q}$ (up to a sign).*

Proof.

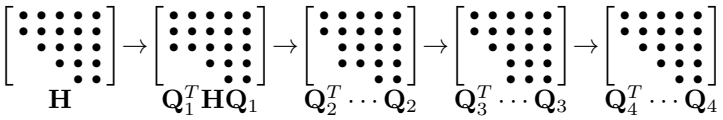
$$\mathbf{A} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}, \quad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

If $\mathbf{A}$ is $n \times n$, then $\mathbf{A}$ has n^2 degrees of freedom $\bullet$. Also, $(n-2)(n-1)/2$ elements of $\mathbf{H}$ are zero, so $\mathbf{H}$ 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 $\mathbf{Q}$. 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 us $n(n-1)/2$ 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 $\tilde{\mathbf{H}} = \mathbf{H} - \rho\mathbf{I}$ with the first column as $\begin{bmatrix}\tilde{h}_{11} & \tilde{h}_{21} & 0 & \cdots & 0\end{bmatrix}^T$.
2. Determine the Householder (or Givens) transformation $\mathbf{Q}^T$ that zeros out $\tilde{a}_{21}$.
3. Compute $\mathbf{H} \leftarrow \mathbf{Q}^T\mathbf{H}\mathbf{Q}$.
4. Now, use Householder (or Givens) transformations to reduce the new $\mathbf{H}$ into upper Hessenberg form by "chasing the bulge":
   a) For $i = 1, \cdots, n-2$ determine the Householder (or Givens) transformation $\mathbf{Q}_i$ that zeros out element $(i+2, i)$ using element $(i+1, i)$.
   b) Compute $\mathbf{A} \leftarrow \mathbf{Q}^T\mathbf{H}\mathbf{Q}$.



Multiple steps

By this point the QR method bears little resemblance to the underlying power method. But just as we could speed up convergence of the power method by taking multiple steps ($\mathbf{A}^2\mathbf{x}$ versus $\mathbf{A}\mathbf{x}$ for example), we can speed up QR method using multiple steps. Namely, we can take $(\mathbf{H} - \rho_1\mathbf{I})(\mathbf{H} - \rho_2\mathbf{I}) \cdots (\mathbf{H} - \rho_k\mathbf{I})$ in one QR step. In practice, k is typically $1, 2, 4$ or 6. Note that the bulge that we need to chase is size k.

Taking multiple steps is especially appealing when working with real matrices that have complex eigenvalues because we can avoid complex arithmetic (cutting computation by half). We can formulate a double-step QR method with $\rho_1 = \bar{\rho}_2 = \rho$. In this case $\tilde{\mathbf{A}} = \mathbf{H}^2 - 2\,\mathrm{Re}(\rho)\mathbf{H} + |\rho|^2\mathbf{I}$.

M The MATLAB command `eig` computes the eigenvalues and eigenvectors of a matrix with a LAPACK library that implements the shifted QR method.

4.7 Getting the eigenvectors

Up to this point we have neglected explicitly computing the eigenvectors for the QR method. In fact, keeping track of the rotator or reflector

matrices $\mathbf{Q}_i$ that accumulate is inefficient. Once we have found the eigenvalues (or more precisely approximations of eigenvalues), computing the eigenvectors is typically a simple task. In this section, we will look at two methods.

Shifted power method

We start again from the upper Hessenberg matrix form of the matrix $\mathbf{H} = \mathbf{Q}^T\mathbf{A}\mathbf{Q}$. Let ρ be our approximation to an eigenvalue. Then using the shifted power method

$$(\mathbf{H} - \rho\mathbf{I})\mathbf{z}^{(k+1)} = \mathbf{q}^{(k)} \quad \text{with} \quad \mathbf{q}^{(k)} = \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|}$$

will return an approximation to the associated eigenvector $\mathbf{q}$ of $\mathbf{H}$. When ρ is a good approximation, only one iteration is typically needed. The eigenvector to $\mathbf{A}$ is then $\mathbf{Q}^T\mathbf{q}$.

Using the Schur form

To get the eigenvectors using the Schur form $\mathbf{Q}^H\mathbf{A}\mathbf{Q} = \mathbf{T}$ without accumulating the matrix $\mathbf{Q}$, we can use an *ultimate shift* strategy. First, we use the QR method to find only the eigenvalues. Once we have the approximate eigenvalues, we rerun the QR method using these eigenvalues as shifts and accumulate the matrix $\mathbf{Q}$, using two steps per shift to ensure convergence. If the matrix is symmetric, then $\mathbf{T}$ is diagonal and $\mathbf{Q}$ gives us the eigenvectors.

Let's look at the nonsymmetric case. Suppose that λ is the kth eigenvalue of $\mathbf{T}$. The upper triangular matrix $\mathbf{T}$ can be written as

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

where $\mathbf{T}_{11}$ is a $(k-1)\times(k-1)$ upper triangular matrix and $\mathbf{T}_{33}$ is an $(n-k)\times(n-k)$ triangular matrix. Furthermore, suppose that λ is a simple eigenvalue so that $\lambda \notin \lambda(\mathbf{T}_{11}) \cup \lambda(\mathbf{T}_{33})$. Let $\mathbf{y} = \begin{bmatrix}\mathbf{y}_{k-1} & y & \mathbf{y}_{n-k}\end{bmatrix}^T$. Then eigenvector problem $(\mathbf{T} - \lambda\mathbf{I})\mathbf{y} = 0$ can be written as

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

If λ is simple, $\mathbf{T}_{11} - \lambda\mathbf{I}_{k-1}$ and $\mathbf{T}_{33} - \lambda\mathbf{I}_{n-k}$ are both nonsingular. So, $\mathbf{y}_{n-k} = \mathbf{0}$ and $\mathbf{y}_{k-1} = y(\mathbf{T}_{11} - \lambda\mathbf{I}_{k-1})^{-1}\mathbf{v}$. Since y is arbitrary, we will

set it to one. Then

$$\mathbf{y} = \begin{bmatrix} (\mathbf{T}_{11} - \lambda \mathbf{I}_{k-1})^{-1}\mathbf{v} \\ \mathbf{1} \\ \mathbf{0} \end{bmatrix}$$

which can be evaluated using Gaussian elimination. The eigenvector for $\mathbf{A}$ is then $\mathbf{x} = \mathbf{Q}\mathbf{y}$.

4.8 Arnoldi method

Suppose that $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a large, sparse matrix, and assume that we only need a few of the largest (or smallest eigenvalues). The QR method is inefficient when n is large (much larger than say 1000) and we only want to find a few eigenvalues, because it operates on all $\mathbb{R}^n$ and it fills in the sparse structure. We can do better by restricting ourselves to a subspace of $\mathbb{R}^n$. By using a subspace that is one-tenth the size of the $\mathbb{R}^n$, we'll get an algorithm that is almost one-thousand times faster. Of course, we need to find a subspace that includes (or at least almost includes) the eigenvectors we are looking for.

The power method in section 4.3 preserves the sparsity of the matrix. But we only get one eigenvalue at a time because we throw away information about the other eigenvectors. Let's reexamine the power method. Take an initial guess $\mathbf{q}$, which for a semisimple matrix can be expressed in terms of an eigenvector basis $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$:

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

For $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$, we see that (or at least hope that) the subspace

$$\operatorname{span}\{\mathbf{q}, \mathbf{A}\mathbf{q}, \mathbf{A}^2\mathbf{q}, \ldots, \mathbf{A}^{m-1}\mathbf{q}\}$$

is close (or at least relatively close) to the subspace spanned by the first m eigenvectors

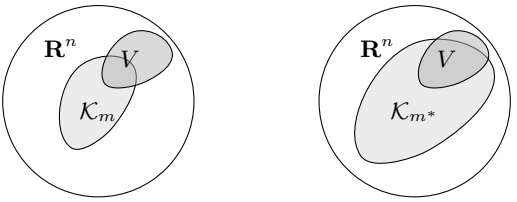
$$\operatorname{span}\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \ldots, \mathbf{v}_m\}.$$

The *Krylov subspace* $\mathcal{K}_m(\mathbf{A}, \mathbf{q})$ is the subspace spanned by

$$\mathbf{q}, \mathbf{A}\mathbf{q}, \mathbf{A}^2\mathbf{q}, \ldots, \mathbf{A}^{m-1}\mathbf{q}.$$

These vectors are the columns of the Krylov matrix $\mathbf{K}_m$. Remember that the Krylov subspace we are using is much smaller than $\mathbb{R}^n$, so it is unlikely that is will actually 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 $\mathcal{K}_m(\mathbf{A}, \mathbf{q})$.

The m-dimensional Krylov subspace $\mathcal{K}_m \subset \mathbb{R}^n$ does not (necessarily) contain the eigenspace V—only its projection into $\mathcal{K}_m$. By making the Krylov subspace larger ($\mathcal{K}_{m^*}$ with $m^* > m$), we get a better approximation at the cost of more computing time:



We can cast the problem in several weak forms.

Original problem Find $\mathbf{x}$ such that $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$

Galerkin Find $\mathbf{x}$ such that $(\mathbf{v}, \mathbf{A}\mathbf{x}) = (\mathbf{v}, \lambda\mathbf{x})$ for all $\mathbf{v} \in \mathcal{K}_m(\mathbf{A}, \mathbf{q})$

Fixed-point Find $\mathbf{x}$ such that $(\mathbf{v}, \mathbf{A}\mathbf{v})$ is a fixed point for $\mathbf{v} \in \mathcal{K}_m(\mathbf{A}, \mathbf{q})$

Minimization Find $\mathbf{x}$ that minimizes $\|\mathbf{A}\mathbf{x} - \lambda\mathbf{x}\|$ for $\mathbf{x} \in \mathcal{K}_m(\mathbf{A}, \mathbf{q})$

The weak forms recast the problem as a projection into a smaller subspace. We say that $(\mathbf{x}_i, \lambda_i)$ is a *Ritz pair* if it satisfies the Galerkin problem.

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 $\mathbf{K}_m$ point in nearly the same direction. So, $\mathbf{K}_m$ is ill-conditioned. To fix the ill-conditioning of $\mathbf{K}_m$, we need to replace the vectors $\mathbf{q}, \mathbf{A}\mathbf{q}, \ldots, \mathbf{A}^{m-1}\mathbf{q}$ with orthonormal set $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m$. The process of generating the orthonormal basis for $\mathcal{K}_m(\mathbf{A}, \mathbf{q})$, called the *Arnoldi process*. The Arnoldi process is similar to the Gram-Schmidt process, except we don't need to start with a set of m basis vectors already in hand. Instead, we only need $\mathbf{q}$. Arnoldi method combines the power method and Gram-Schmidt to get the first m eigenvectors approximations.

First, normalize the first vector

$$\mathbf{q}_1 \leftarrow \mathbf{q}_1/\|\mathbf{q}_1\|.$$

On the next step take

$$\mathbf{q}_2 = \mathbf{A}\mathbf{q}_1 - (\mathbf{A}\mathbf{q}_1, \mathbf{q}_1)\,\mathbf{q}_1 = \mathbf{A}\mathbf{q}_1 - h_{11}\mathbf{q}_1 \quad \text{and} \quad \tilde{\mathbf{q}}_2 \leftarrow \mathbf{q}_2/\|\mathbf{q}_2\|.$$

And, on the subsequent steps take

$$\mathbf{q}_{k+1} = \mathbf{A}\mathbf{q}_k - \sum_{j=1}^{k} \mathbf{q}_j h_{jk} \quad \text{and} \quad \mathbf{q}_{k+1} \leftarrow \mathbf{q}_{k+1}/\|\mathbf{q}_{k+1}\|$$

where h_{jk} is the Gram-Schmidt coefficient

$$h_{jk} = (\mathbf{A}\mathbf{q}_k, \mathbf{q}_j)\,.$$

In practice, we implement this step using the modified Gram-Schmidt (with reorthogonalization). Complete the step by normalizing $\mathbf{q}_{k+1}$

$$\mathbf{q}_{k+1} = \mathbf{q}_{k+1}/h_{k+1,k} \quad \text{where} \quad h_{k+1,k} = \|\mathbf{q}_{k+1}\|_2.$$

The Arnoldi process can be written as the following algorithm

Guess an initial $\mathbf{q}^{(1)}$ with $\|\mathbf{q}^{(1)}\| = 1$
for $k = 2, \ldots, m$
  $\mathbf{q}^{(k)} \leftarrow \mathbf{A}\mathbf{q}^{(k-1)}$
  $\mathbf{q}^{(k)} \leftarrow \mathbf{q}^{(k)}/\|\mathbf{q}^{(k)}\|$
  for $j = 1, \ldots, k-1$
    $h_{j,k-1} \leftarrow \left(\mathbf{q}^{(j)}, \mathbf{q}^{(k)}\right)$
    $\mathbf{q}^{(k)} \leftarrow \mathbf{q}^{(k)} - h_{j,k-1}\mathbf{q}^{(j)}$
  $h_{k,k-1} \leftarrow \|\mathbf{q}^{(k)}\|$
  $\mathbf{q}^{(k)} \leftarrow \mathbf{q}^{(k)}/h_{k,k-1}$

In matrix representation, we have that

$$\mathbf{A}\mathbf{Q}_m = \mathbf{Q}_{m+1}\mathbf{H}_{m+1,m}$$

where

$$\mathbf{Q}_m = \begin{bmatrix} \mathbf{q}_1 & \cdots & \mathbf{q}_m \end{bmatrix} \in \mathbb{C}^{n\times m}$$

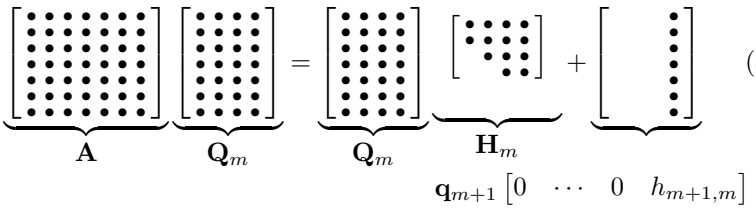
and

$$\mathbf{H}_{m+1,m} = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ & \bullet & \bullet & \bullet \\ & & \bullet & \bullet \\ & & & \bullet \end{bmatrix} \in \mathbb{C}^{(m+1)\times m}$$

is a non-square upper-Hessenberg matrix. Note we can complete the $\mathbf{H}_{m+1,m}$ by adding a column $\mathbf{e}_1$ to get a upper triangular matrix

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

in which case $\mathbf{K}_m = \mathbf{Q}_m\mathbf{R}$ is the QR decomposition of the Krylov matrix $\mathbf{K}_m$. We can also rewrite the Arnoldi decomposition by separating out the bottom row of $\mathbf{H}_{m+1,m}$ to get



(4.5)

Because $\mathbf{Q}_m^T$ is orthogonal to $\mathbf{q}_{m+1}$, multiplying (4.5) by $\mathbf{Q}_m^T$ zeros out the last term and leaves us with

$$\mathbf{Q}_m^T\mathbf{A}\mathbf{Q}_m = \mathbf{H}_m$$

which says that in the projection, the upper Hessenberg matrix $\mathbf{H}_m$ has the same eigenvalues as $\mathbf{A}$, i.e., the Ritz values.

More precisely, if $(\mathbf{x}, \mu)$ is an eigenpair of $\mathbf{H}_m$, then $\mathbf{v} = \mathbf{Q}_m\mathbf{x}$ satisfies

$$\begin{aligned}
\|\mathbf{A}\mathbf{v} - \mu\mathbf{v}\|_2 &= \|\mathbf{A}\mathbf{Q}_m\mathbf{x} - \mu\mathbf{Q}_m\mathbf{x}\|_2 \\
&= \|(\mathbf{A}\mathbf{Q}_m - \mathbf{Q}_m\mathbf{H}_m)\mathbf{x}\|_2 \\
&= h_{m+1,m}\|\mathbf{e}_{m+1}^T\mathbf{x}\|_2 \\
&= h_{m+1,m}|x_{m+1}|
\end{aligned}$$

The residual norm $h_{m+1,m}|x_{m+1}|$ is called the *Ritz estimate*. If $(\mu, \mathbf{v})$ is a good approximation to an eigenpair of $\mathbf{A}$, then the Ritz estimate is small. See Figure 4.6 on the next page.

It is mathematically intuitive that the success of Arnoldi iteration depends on choosing a good starting vector $\mathbf{q}$. If $\mathbf{q}$ is chosen at random, it will likely have significant components in all eigenvector directions. So, the Krylov subspace is not a good match for the space spanned by the first k eigenvectors. If we were able to (magically) start with

$$\mathbf{q} \in \operatorname{span}\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}, \quad \text{i.e.,} \quad \mathbf{q} = c_1\mathbf{v}_1 + c_1\mathbf{v}_1 + \cdots + c_1\mathbf{v}_k$$

then the Arnoldi process stops after k steps with

$$\operatorname{span}\{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_k\} = \operatorname{span}\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}.$$

Realistically, how do we get a good starting vector $\mathbf{q}$ for the Krylov subspace?

One solution is to do Arnoldi process itself to get a good starting vector and then restart using this better guess. This method is called

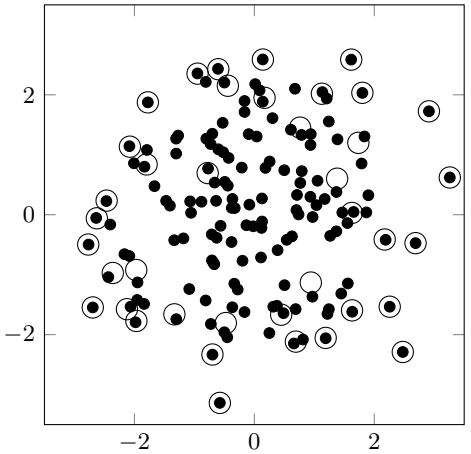


Figure 4.6: Location of the Ritz values ○ for $\mathbf{K}_{40}$ and the eigenvalues • in the complex plane for a 144×144 complex tridiagonal matrix starting with a random initial vector. Notice that the Ritz values are closest for larger magnitude eigenvalues.

the *implicitly restarted Arnoldi process*, Suppose that we want to get the k eigenvectors corresponding to the largest eigenvalues. Let's take the Arnoldi subspace to have $m = j + k \approx 2k$ dimensions. First, run the Arnoldi process to get m Arnoldi vectors: $\mathbf{Q}_m$ and $\mathbf{H}_m$. Now, we will suppress (or filter) the components of the eigenvectors of $\lambda_{k+1}, \ldots, \lambda_m$ out of 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. For each shift, we use the approximate Ritz value $\{\mu_{k+1}, \ldots, \mu_m\}$. We use the accumulated unitary matrix $\mathbf{V}_m$ from the j QR steps to change the basis to our Arnoldi subspace. Namely,

$$\begin{aligned}\mathbf{H} &\leftarrow \mathbf{V}_m^H \mathbf{H}_m \mathbf{V}_m \\ \mathbf{Q} &\leftarrow \mathbf{Q}_m \mathbf{V}_m\end{aligned}$$

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 process starting with $\mathbf{q}_{k+1}$. We continue like this until convergence.

Ⓜ The MATLAB command `eigs` computes several eigenvalues of a sparse matrix using implicitly restarted Arnoldi process.

The Arnoldi method applied to a symmetric, sparse matrix is called

the *Lanczos method*. Symmetric matrices have two clear advantages over nonsymmetric matrices. The eigenvectors are orthogonal and the upper Hessenberg matrix is tridiagonal, so the complexity is only $O(n)$.

4.9 Singular value decomposition

Recall that the singular value decomposition of an $m \times n$ matrix $\mathbf{A}$ is $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$ where $\mathbf{U}$ is an $m \times m$ unitary matrix of eigenvectors of $\mathbf{A}\mathbf{A}^T$, $\boldsymbol{\Sigma}$ 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 $\mathbf{A}\mathbf{A}^T$. While there are fast methods for computing the eigenvalues of symmetric matrices, it is typically best not to use $\mathbf{A}\mathbf{A}^T$ or $\mathbf{A}^T\mathbf{A}$ to compute the singular values. Instead, we'll look at the Golub–Kahan–Reinisch algorithm for the SVD—there are a number of others. The algorithm consists of two parts: transformation to upper bidiagonal form and the applying the QR algorithm.

Step 1. Compute

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

where $\mathbf{B}$ is an upper bidiagonal matrix. To get such a matrix,we use a series of Householder reflections (or Givens rotations):

$$\overset{\mathbf{A}}{\begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix}} \xrightarrow{\tilde{\mathbf{U}}_1^T \cdots} \begin{bmatrix} \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix} \xrightarrow{\cdots \tilde{\mathbf{V}}_1} \begin{bmatrix} \bullet & \bullet & & \\ & \bullet & \bullet & \bullet \\ & & \bullet & \bullet \\ & & \bullet & \bullet \\ & & \bullet & \bullet \\ & & \bullet & \bullet \end{bmatrix}$$

$$\xrightarrow{\tilde{\mathbf{U}}_2^T \cdots} \begin{bmatrix} \bullet & \bullet & & \\ & \bullet & \bullet & \bullet \\ & & \bullet & \bullet \\ & & \bullet & \bullet \\ & & \bullet & \bullet \\ & & \bullet & \bullet \end{bmatrix} \xrightarrow{\cdots \tilde{\mathbf{V}}_2} \begin{bmatrix} \bullet & \bullet & & \\ & \bullet & \bullet & \\ & & \bullet & \bullet \\ & & \bullet & \bullet \\ & & \bullet & \bullet \\ & & \bullet & \bullet \end{bmatrix}$$

$$\xrightarrow{\tilde{\mathbf{U}}_3^T \cdots} \begin{bmatrix} \bullet & \bullet & & \\ & \bullet & \bullet & \\ & & \bullet & \bullet \\ & & & \bullet \\ & & & \bullet \\ & & & \bullet \end{bmatrix} \xrightarrow{\cdots \tilde{\mathbf{V}}_3} \begin{bmatrix} \bullet & \bullet & & \\ & \bullet & \bullet & \\ & & \bullet & \bullet \\ & & & \bullet \\ & & & \\ & & & \end{bmatrix}$$

After $m-1$ steps

$$\tilde{\mathbf{U}} = \tilde{\mathbf{U}}_1\tilde{\mathbf{U}}_2\cdots\tilde{\mathbf{U}}_{m-1} \quad \text{and} \quad \tilde{\mathbf{V}} = \tilde{\mathbf{V}}_1\tilde{\mathbf{V}}_2\cdots\tilde{\mathbf{V}}_{n-2}$$

Step 2. Now, we use the QR method on $\mathbf{B}$ to get the singular values $\boldsymbol{\Sigma}$. One cycle (implicit QR on $\mathbf{B}^T\mathbf{B}$):

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

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

$$\mathbf{Q} : \begin{bmatrix} b_{11}^2 - \rho & b_{11}b_{12} \end{bmatrix} \to \begin{bmatrix} * & 0 \end{bmatrix}$$

3. "Chase the bulge" $(\mathbf{U} \cdots \mathbf{V}^T)$ using Givens rotations

$$\mathbf{U} : \begin{bmatrix} b_{i,i} \\ b_{i+1,i} \end{bmatrix} \to \begin{bmatrix} * \\ 0 \end{bmatrix} \qquad \mathbf{V} : \begin{bmatrix} b_{i,i+1} & b_{i,i+2} \end{bmatrix} \to \begin{bmatrix} * & 0 \end{bmatrix}.$$

4. Deflation: if $|b_{n-1,n}| < \varepsilon$, use upper left $n-1 \times n-1$ matrix.

Computing the SVD of a sparse matrix

To find the singular values and singular vectors of a large (sparse) matrix, we can use the Arnoldi method (or more specifically the Lanczos method) of finding eigenvalues. Take $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$. Recall that the singular values of $\mathbf{A}$ are $\sigma(\mathbf{A}) = \sqrt{\lambda(\mathbf{A}^H\mathbf{A})}$, that the left-singular vectors $\mathbf{u}$ are the eigenvectors of $\mathbf{A}\mathbf{A}^H$, and that the right-singular vectors $\mathbf{v}$ are the eigenvectors of $\mathbf{A}^H\mathbf{A}$. We have that

$$\underbrace{\begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{A}^H & \mathbf{0} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \lambda \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}. \tag{4.6}$$

To see this, note that (4.6) is equivalent to

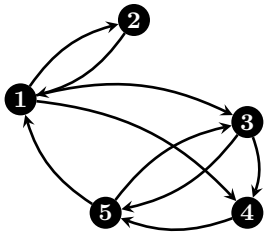
$$\left.\begin{aligned} \mathbf{A}\mathbf{v} &= \lambda\mathbf{u} \\ \mathbf{A}^H\mathbf{u} &= \lambda\mathbf{v} \end{aligned}\right\} \quad \Rightarrow \quad \left\{\begin{aligned} \mathbf{A}^H\mathbf{A}\mathbf{v} &= \lambda\mathbf{A}^H\mathbf{u} = \lambda^2\mathbf{v} \\ \mathbf{A}\mathbf{A}^H\mathbf{u} &= \lambda\mathbf{A}\mathbf{v} = \lambda^2\mathbf{u} \end{aligned}\right. .$$

$\mathbf{M}$ is a Hermitian matrix, so we can apply the Lanczos method to get the largest eigenvalues.

Exercises

4.1. Explain why the the eigenvalues of a normal matrix are perfectly well-conditioned, that is, the eigenvalue condition number of a normal matrix is one.

✍ 4.2. Consider the $n \times n$ matrix $\mathbf{A}$ where the elements are normally distributed random numbers with variance 1. Conjecture about the spectrum of eigenvalues $\lambda(A)$ for arbitrary n. Hint: It is useful to plot the eigenvalues in the complex plane using `plot(eig(randn(n)),'.')`. Use `hold on` and loop several times until a pattern emerges.



$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 1 & 0 \end{bmatrix}$$

Figure 4.7: A graph and its associated Markov matrix $\mathbf{H}$.

✍ 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 use MATLAB's `eig` function to compute the true values.

4.4. *Google's PageRank.* For a given keyword, an internet search engine will likely return hundreds of web pages. The PageRank algorithm (Patent No. US 7,058,628) sorts web pages using eigenvector centrality. Imagine that millions of people are browsing the web, clicking hyperlinks at random. We may wonder how many are on a specific web page at any time. This probability is the PageRank.

The *hyperlink* matrix of the internet is a weighted *incident matrix* $\mathbf{H}$ with elements

$$h_{ij} = \begin{cases} 1/o_j, & \text{if there is a hyperlink from page } j \text{ to page } i \\ 0, & \text{otherwise} \end{cases}$$

where o_j is the number of outgoing links from page j. $\mathbf{H}$ is a large, sparse $n \times n$ matrix where the number of webpages n is on the order of billions. For webpages j without any outgoing links, we'll pick a random page from any of the n possible webpages. We do this by setting all values in column j to $1/n$. Define the Markov matrix

$$\mathbf{S} = \mathbf{H} + \mathbf{1}\mathbf{a}^T$$

where $\mathbf{1}$ is a $n \times 1$ vector of ones and $\mathbf{a}$ is the $n \times 1$ vector with

$$a_j = \begin{cases} 1/n, & \text{if page } j \text{ has no outgoing links} \\ 0, & \text{otherwise.} \end{cases}$$

Occasionally, someone may decide not to follow any link on the webpage that he or she is currently viewing and instead opens an arbitrary webpage. To model such a behavior, we use the mollified Markov matrix $\mathbf{G}$ by setting

$$\mathbf{G} = p\mathbf{S} + (1-p)\mathbf{E}$$

where p is the likelihood of following the links on the page and $\mathbf{E}$ is an $n \times n$ matrix of with elements $e_{ij} = 1/n$. A typical value for p is 0.85.

A vector $\mathbf{x}$ is a *probability vector* if its column sum equals one, i.e., $\|\mathbf{x}\|_1 = 1$. The ith element of $\mathbf{x}$ is the probability of being on page i, and ith element of $\mathbf{Gx}$ is the probability of being on page i a short time later. Since $\mathbf{G}$ is a Markov matrix, $\lim_{k\to\infty} \mathbf{G}^k$ will converge to a steady state operator. By the Perron–Frobenius theorem, the eigenvector corresponding to the largest eigenvalue ($\lambda = 1$) gives the steady-state probability of being on a given webpage. This eigenvector is the PageRank.

Google uses the power method to find the PageRank $\mathbf{x}^{(k+1)} \leftarrow \mathbf{Gx}^{(k)}$. Remember that $\mathbf{H}$ was originally a very sparse matrix requiring $O(n)$ operations for every matrix-vector multiplication, but $\mathbf{G}$ is a full matrix requiring $O(n^2)$ operations. So, it is much more efficient to reformulate the problem in terms of $\mathbf{H}$. Because $\|\mathbf{x}^{(k)}\|_1 = 1$, it follows that $\mathbf{Ex} = n^{-1}\mathbf{1}$. So,

$$\mathbf{x}^{(k+1)} \leftarrow \mathbf{Gx}^{(k)} = p\mathbf{Hx}^{(k)} + p\mathbf{1}(\mathbf{a}^T\mathbf{x}^{(k)}) + (1-p)n^{-1}\mathbf{1}. \tag{4.7}$$

1. The *connectivity* matrix for the internet is the matrix $\mathbf{C}$ where the elements

$$c_{ij} = \begin{cases} 1, & \text{if there is a hyperlink from page } j \text{ to page } i \\ 0, & \text{otherwise.} \end{cases}$$

   The MAT-file contains the connectivity matrix `A` for the AFIT website and the corresponding array `U` of url names. It can be loaded into the MATLAB workspace using the `load` command. Use the MATLAB command `spy` to examine the sparsity pattern of the matrix `C`.

2. Use the power method (4.7) to compute the PageRank of the AFIT website and use this to sort the urls. Take $p = 0.85$. The MATLAB command `[Y,I] = sort(X)` will sort the elements of `X` in ascending order and also return the index array `I`. Hint: use the 1-norm instead of the 2-norm.

3. The largest eigenvalue of $\mathbf{G}$ is 1. Compute the magnitude of the second largest eigenvalue λ_2 by determining the rate of convergence of the power method. You can do this by rerunning the power

method to get the error $\varepsilon_k = \|\mathbf{x} - \mathbf{x}^{(k)}\|_1$. Since $\varepsilon_k \approx |\lambda_2|^k$, the slope of $\log \varepsilon_k$ will give you $|\lambda_2|$. Use the MATLAB command `eigs(G,2)` to check your results.

For further discussion see Moler [2004].

✍ 4.5. Use Rayleigh iteration to find the eigenvalues of

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 6 & 4 \\ 3 & 0 & 3 & 1 \\ 6 & 3 & 8 & 8 \\ 4 & 1 & 8 & 2 \end{bmatrix}.$$

Check your results using MATLAB's `eig` command. Confirm that you are getting cubic rate of convergence for a symmetric matrix.

✍ 4.6. Write a program which implements the implicit QR method using single-step Rayleigh-shifting with deflation. Test the method on

$$\mathbf{A} = \mathbf{R}\Lambda\mathbf{R}^{-1} \quad \text{where } \mathbf{R} = \texttt{randn(20)} \text{ and } \mathbf{\Lambda} = \texttt{diag(1:20)}$$

and on $\mathbf{D}_{20}$ from problem 1.5. Use a tolerance of $\varepsilon = 10^{-12}$. Compare your approximate eigenvalues with the exact eigenvalues. Hint: Recall that the implicit QR-method is summarized as

1. Compute the Hessenberg matrix that is unitarily similar to $\mathbf{A}$.
2. One QR-cycle is
   a) Take $\rho = a_{nn}$ (Rayleigh Shifting)
   b) Compute $\mathbf{QAQ}^T$ for Givens rotation $\mathbf{Q} : \begin{bmatrix} a_{11} - \rho \\ a_{21} \end{bmatrix} \to \begin{bmatrix} * \\ 0 \end{bmatrix}$.
   c) "Chase the bulge" $(\mathbf{Q} \cdots \mathbf{Q}^T)$ using Givens rotations $\mathbf{Q} : \begin{bmatrix} a_{i+1,i} \\ a_{i+2,i} \end{bmatrix} \to \begin{bmatrix} * \\ 0 \end{bmatrix}$.
   d) Deflation: if $|a_{n,n-1}| < \varepsilon$, use upper left $n-1 \times n-1$ matrix.

Hint: 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).

4.7. The Arnoldi method can also be used with some modification to get the smallest eigenvalues. Outline such an approach.

4.8. Make a diagram showing the connections between concepts in this chapter: power method, inverse power method, shifted power method,

Rayleigh iteration, QR method, reduction to Hessenberg form, shifted QR method, deflation, multi-step QR method, implicit QR method, Arnoldi method, Lanczos method and singular value decomposition.

CHAPTER 5

Iterative Methods for Linear Systems

For small and moderate sized matrices, Gaussian elimination is fast. Even for a 1000×1000 matrix Gaussian elimination takes less than a second on a desktop computer. Suppose instead that we want to solve a partial differential equation in three dimensions. At each time step, we make a discrete approximation to the solution using 100 grid-points in each dimension. Such a problem requires that we solve a system of a million equations at each time step. Now, using Gaussian elimination on a desktop computer would take over thirty years. Fortunately, such a system is sparse—only 0.001% of the elements in the matrix are nonzero. If the matrix has a narrow bandwidth, Gaussian elimination is still fast. But a system that has no exploitable feature other than its sparsity can still be attacked using an iterative method. Gaussian elimination takes $O(n^3)$ operations, but multiplication takes only $O(n^2)$ operations and as little as $O(n)$ operations if the system is sparse. An iterative method will beat a direct method as long as the number of iterations needed for convergence is much smaller than n.

There's another reason when iterative solvers are important. While direct solvers must solve each problem from scratch, if we have a good initial guess $\mathbf{x}^{(0)}$, convergence of an iterative method can be even faster. Suppose that we are solving a time dependent partial differential equation. At each new time step the solution does not change much from the solution computed at an earlier time step. If we are using a direct solver, hopefully we've been smart and have the LU or Cholesky decomposition, so we only need $O(n^2)$ operations to compute the solution. Iterative methods, on the other hand, can use the the solution at the previous

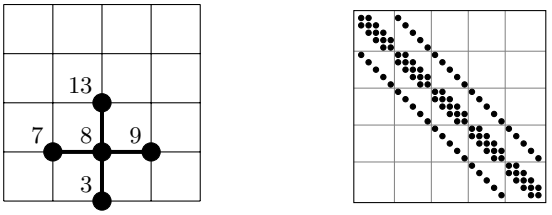


Figure 5.1: Finite-difference stencil for the discrete two-dimensional Laplacian on a 5×5 mesh. By labeling the points in lexicon fashion, we can construct a 25×25 block tridiagonal matrix.

time step as a good initial guess for the solution at the new time step.

Also, sometimes you may not need the accuracy of a solution using a direct solver. For example, a finite-difference approximation to a partial differential equation may already have substantial truncation error. In this case, it may be significantly faster to get below this error threshold than to try to get close to machine precision.

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

Example. Consider the two-dimensional Poisson equation

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x, y)$$

with Dirichlet boundary conditions $f(x, y) = 0$ for $x = 0$, $x = 1$, $y = 0$ and $y = 1$. The Poisson equation can be used to model several physical problems:

- the steady-state heat distribution $u(x, y)$ given a source $f(x, y)$
- the steady-state electrical charge distribution $u(x, y)$ given a source $f(x, y)$
- the shape of an elastic membrane $z = u(x, y)$ given a load $f(x, y)$

Numerically, a simple, yet effective means of solving the problem is to discretize the partial derivative. Consider partitioning a square domain into n intervals of length h. By Taylor series expansion we have that at each grid point (x_i, y_j) the discrete Laplacian operator can be represented by

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} \approx \frac{4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}}{h^2}$$

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 $1 \le i, j \le n$ with $u_{ij} \equiv u(x_i, y_j)$ and $f_{ij} \equiv u(x_i, y_j)$. In matrix–vector notation, $\mathbf{Au} = \mathbf{f}$ where the column vectors

$$\begin{aligned} \mathbf{u} &= \begin{bmatrix} u_{11} & u_{21} & \cdots & u_{n1} & u_{12} & \cdots & u_{nn} \end{bmatrix}^T \\ \mathbf{f} &= \begin{bmatrix} f_{11} & f_{21} & \cdots & f_{n1} & f_{12} & \cdots & f_{nn} \end{bmatrix}^T . \end{aligned}$$

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. ◀

5.1 Jacobi and Gauss–Seidel methods

Suppose we want to solve $\mathbf{Ax} = \mathbf{b}$. In index format

$$\sum_{j=1}^{n} a_{ij} x_j = b_i .$$

We solve for x_i

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j \right) .$$

The elements of the unknown $\mathbf{x}$ appear on both sides of the equation. One way to solve this problem is to implement this solution iteratively as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right) .$$

In this method, called the *Jacobi method*, each element of $\mathbf{x}$ is updated independently, which makes vectorization easy.

Another idea is to use the newest and best approximations available, overwriting the elements of $\mathbf{x}$. That is,

for $i = 1, \ldots, n$

$$\left[\; x_i \leftarrow \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j \right) \right.$$

Formally,

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(-\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} + b_i \right). \tag{5.1}$$

This is the *Gauss–Seidel method*. Unlike the Jacobi method, which needs to store two copies of $\mathbf{x}$, the Gauss–Seidel method allows us to use the same array for each iteration, overwriting each element sequentially.

At this point, we should ask two questions:

1. When do the Jacobi and Gauss–Seidel methods converge?
2. How quickly do they converge?

To answer these questions, let's look at the Jacobi and Gauss–Seidel methods with a bit more generality. The Jacobi and Gauss–Seidel methods are examples of linear iterative methods. Consider using the splitting $\mathbf{A} = \mathbf{P} - \mathbf{N}$ for some $\mathbf{P}$ and $\mathbf{N}$. In this case, the linear system $\mathbf{Ax} = \mathbf{b}$ is the same as

$$\mathbf{Px} = \mathbf{Nx} + \mathbf{b}. \tag{5.2}$$

Given $\mathbf{x}^{(0)}$, we can compute $\mathbf{x}^{(k)}$ by solving the system

$$\mathbf{Px}^{(k+1)} = \mathbf{Nx}^{(k)} + \mathbf{b}. \tag{5.3}$$

We call $\mathbf{P}$ the *preconditioner*. Let's see why it's called this. Suppose that we wanted 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 the new system is $\kappa(\mathbf{P}^{-1}\mathbf{A})$. If we can choose $\mathbf{P} \approx \mathbf{A}$, then we can get a well-conditioned system. For a general linear iterative method $\mathbf{Px}^{(k+1)} = \mathbf{Nx}^{(k)} + \mathbf{b}$ to be a good method

1. the preconditioner $\mathbf{P}$ should be easy to invert; and
2. the method should converge quickly.

One type of preconditioner that is often used for a general matrix is the incomplete LU factorization (ILU). In this case we take $\mathbf{P} = \mathbf{LU}$ so that certain elements of $\mathbf{P}$ matches the corresponding elements of $\mathbf{A}$. That is, we choose the preconditioner $\mathbf{P}$ so that $\mathbf{P} \approx \mathbf{A}$ and with an LU-decomposition that is much faster to compute.

Let's return to Jacobi and Gauss–Seidel methods. The Jacobi method splits the matrix into diagonal and off-diagonal matrices:

$\mathbf{A} = \mathbf{P} - \mathbf{N} = \mathbf{D} + \mathbf{M}$:

For an initial guess $\mathbf{x}^{(0)}$,

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

The Gauss–Seidel method splits the matrix into lower-triangular and strictly upper-triangular matrices:

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

For an initial guess $\mathbf{x}^{(0)}$, for each $k = 0, 1, 2, \ldots$

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

The error at the kth iteration is

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

Subtracting (5.3) from (5.2), gives us

$$\mathbf{P}\mathbf{e}^{(k+1)} = \mathbf{N}\mathbf{e}^{(k)}. \tag{5.4}$$

Equivalently,

$$\mathbf{e}^{(k+1)} = \mathbf{P}^{-1}\mathbf{N}\mathbf{e}^{(k)}. \tag{5.5}$$

An iterative method creates a *contraction mapping* $\mathbf{e}^{(k+1)} = \mathbf{P}^{-1}\mathbf{N}\mathbf{e}^{(k)}$ so that $\mathbf{e}^{(k)} \to \mathbf{0}$ as $k \to \infty$. In terms of the initial error $\mathbf{e}^{(0)}$

$$\mathbf{e}^{(k+1)} = (\mathbf{P}^{-1}\mathbf{N})^k \mathbf{e}^{(0)} \tag{5.6}$$

and by submultiplicitivity it follows that

$$\|\mathbf{e}^{(k+1)}\| \leq \|(\mathbf{P}^{-1}\mathbf{N})\|^k \|\mathbf{e}^{(0)}\|.$$

This is true for *every* induced norm. Recall from Theorem 5 that all induced matrix norms are bounded below by the spectral radius. Hence,

$$\|\mathbf{e}^{(k+1)}\| \leq \rho(\mathbf{P}^{-1}\mathbf{N})^k \|\mathbf{e}^{(0)}\|.$$

Typically, we can expect a convergence rate of about $\rho(\mathbf{P}^{-1}\mathbf{N})$ for an iterative method.

Theorem 14. *The iteration* $\mathbf{P}\mathbf{x}^{(k+1)} = \mathbf{N}\mathbf{x}^{(k)} + \mathbf{b}$ *converges if and only if the spectral radius* $\rho(\mathbf{P}^{-1}\mathbf{N}) < 1$.

Theorem 15. *A matrix* $\mathbf{A}$ *is diagonally dominant, if*

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \,.$$

If a matrix is diagonally dominant, then the Jacobi and Gauss–Seidel methods converge.

Proof. We'll prove convergence for the Jacobi method. The proof for the Gauss–Seidel method can be found in Kincaid and Cheney [2001].

$$\rho(\mathbf{D}^{-1}\mathbf{M}) \leq \|\mathbf{D}^{-1}\mathbf{M}\|_\infty = \max_i \sum_{\substack{j=1 \\ j \neq i}}^{n} \left| \frac{a_{ij}}{a_{ii}} \right| .$$

It follows that $\rho(\mathbf{D}^{-1}\mathbf{M}) < 1$, so the Jacobi method converges. □

Example. Let's examine the convergence using the $n \times n$ matrix

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} . \tag{5.7}$$

For the Jacobi method (from Exercise 1.5) the spectral radius is

$$\rho(\mathbf{P}^{-1}\mathbf{N}) = \cos \frac{\pi}{n+1} \approx 1 - \frac{1}{2} \left(\frac{\pi}{n+1} \right)^2 .$$

As $n \to \infty$, the asymptotic convergence rate

$$-\log \text{error} \approx -\log \big(\rho(\mathbf{P}^{-1}\mathbf{N}) \big) \approx \frac{1}{2} \left(\frac{\pi}{n+1} \right)^2$$

where we use the approximation $\log(1-z) \approx z + \frac{1}{2}z^2 + \cdots$. For the Gauss–Seidel method, the spectral radius is

$$\rho(\mathbf{P}^{-1}\mathbf{N}) = \left(\cos \frac{\pi}{n+1} \right)^2 \approx 1 - \left(\frac{\pi}{n+1} \right)^2 .$$

As $n \to \infty$, the asymptotic convergence rate

$$-\log \text{error} \approx \left(\frac{\pi}{n+1} \right)^2 .$$

If $\mathbf{A}$ is a rather small 21×21 matrix, then the rate of convergence for the Jacobi method is 0.99 and for the Gauss–Seidel, it is 0.98. Hence, we can expect roughly 450 iterations and 225 iterations respectively to get the error to 0.01 of the initial error. Pretty lousy. And for a larger matrix, it's much worse. What can we do to speed up convergence? ◀

5.2 Successive over-relaxation

The preconditioners in both the Jacobi and Gauss–Seidel methods are easy to invert. Now, our goal is to speed up convergence by somehow choosing a splitting that minimizes $\rho(\mathbf{P}^{-1}\mathbf{N})$. The *residual* at the kth 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 $\mathbf{x} = \mathbf{P}^{-1}(\mathbf{N}\mathbf{x} + \mathbf{b})$ as

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{P}^{-1}(\mathbf{N}\mathbf{x}^{(k)} + \mathbf{b}) \\ &= \mathbf{x}^{(k)} + \mathbf{P}^{-1}\mathbf{r}^{(k)}.\end{aligned}$$

We call $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{\delta}$ for some vector $\boldsymbol{\delta}$ the *relaxation* of $\mathbf{x}$. The word "relaxation" makes perfect sense, when you think of $\mathbf{x}^{(k)}$ converging to a steady-state from a perturbed state.

Consider the Jacobi method for which the preconditioner $\mathbf{P} = \mathbf{D}$. Arguably we can speed things up by weighting the preconditioner. Take $\mathbf{P} = \omega^{-1}\mathbf{D}$ instead. Then, we have a*weighted Jacobi method* with the splitting

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

For our example problem (see Figure 5.2 on the next page)

$$\lambda_k = 1 + \frac{1}{\omega}\left(\cos\frac{k\pi}{n+1} - 1\right).$$

The Jacobi method simultaneously relaxes $\mathbf{x}$, while the Gauss–Seidel method successively relaxes $\mathbf{x}$. The idea behind the *successive over-relaxation* (SOR) method is that convergence is faster if we go *beyond* the Gauss–Seidel correction. The algorithm for the SOR method is

for $i = 1, \ldots, n$

$$\left[\begin{array}{l} \delta \leftarrow \dfrac{1}{a_{ii}}\left(b_i - \sum_{j\neq 1} a_{ij}x_j\right) - x_i \\ x_i \leftarrow x_i + \omega\delta \end{array}\right.$$

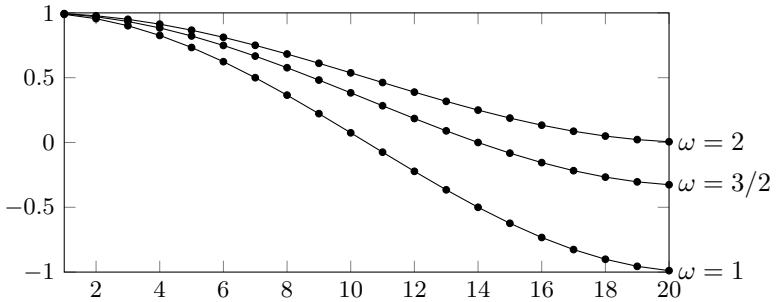


Figure 5.2: Plot of the eigenvalues of the weighted Jacobi method for a 20×20 tridiagonal matrix $\mathbf{A}$ given in (5.7).

The constant ω is called the relaxation factor. When $\omega = 1$, the SOR method is simply the Gauss–Seidel method. In general, the SOR method has the splitting

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

For an initial guess $\mathbf{x}^{(0)}$, for each $k = 0, 1, 2, \ldots$

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

The trick is to choose an ω so that the SOR method converges quickly. That is, choose the ω that minimizes $\rho(\mathbf{P}_\omega^{-1}\mathbf{N}_\omega)$ where $\mathbf{P}_\omega = \mathbf{D} + \omega\mathbf{L}$ and $\mathbf{N}_\omega = (1-\omega)\mathbf{D} - \omega\mathbf{U}$. The choice is problem specific and not necessarily easy.

Example. The optimal choice of ω for the matrix (5.7) is

$$\omega_{\text{opt}} = \frac{2 - 2\sqrt{1 - \lambda_{\max}^2}}{\lambda_{\max}^2} \tag{5.8}$$

where

$$\lambda_{\max} = \cos\frac{\pi}{n+1}$$

is the spectral radius of the Jacobi matrix. For the SOR, we have (at best)

$$\lambda_n = \left(\cos\frac{\pi}{n+1}\right)^2 \Bigg/ \left(1 + \sin\frac{\pi}{n+1}\right)^2.$$

If $\mathbf{A}$ is 21×21 (as before), then the rate of convergence for the Jacobi method is 0.75. Every single SOR step is the equivalent of 30 Jacobi steps and it now takes 15 iterations to get the error to 0.01 of the initial error. ◀

Example. Relaxation can be viewed as finding the steady-state solution to an analogous time-dependent problem. To see this, let's consider the one-dimensional Poisson equation

$$-\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = f(x).$$

We discretized the second derivative operator by

$$-\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} \approx \frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2}$$

to get a system of n equations

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{h^2} = f_i$$

represented by the system $\mathbf{Au} = \mathbf{b}$. We can solve this system iteratively

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

which can be rewritten as the equivalent

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

Note the similarity between this system and the forward-Euler method

$$\mathbf{u}^{(k+1)} - \mathbf{u}^{(k)} = (\Delta t)\left(\mathbf{b} - \mathbf{A}\mathbf{u}^{(k)}\right)$$

to solve the continuous time-dependent heat equation with source term

$$\frac{\partial u(t,x)}{\partial t} = f + \frac{\partial^2 u(t,x)}{\partial x^2}.$$

◀

5.3 Multigrid

One reason that he Poisson equation takes so many iterations is that the stencil for the discrete Laplacian only "talks" to adjacent meshpoints, we are only able to update information using nearest neighbors. It takes

n operations to pass information from one side of the domain to the other if the discretization has n meshpoints in each dimension. The result, as we have seen, is slow convergence of the iterative method. Of course, we could use a larger stencil. But then we are adding more nonzero elements to our sparse matrix. An alternative approach is to use a multigrid method.

Eigenmodes (of the heat equation) with large negative eigenvalues decay rapidly leaving only slowly decaying small eigenmodes. In a discrete framework, we can understand this as "high frequencies are very local, low frequencies are very global." Our discretization of the Laplacian can only "see" (and influence) the nearest neighbors, so it takes a long time (many iterations) for the effect of relaxation to work and error to be minimized for low frequencies. Of course if we changed the mesh to a coarse mesh, then we should expect the low frequency information to move faster since the grid spacing is wider.

The multigrid method embodies the following idea:

- Solve the problem on a course mesh to kill off the residual from the low frequencies.
- Solve the problem on a fine mesh for high accuracy.
- Use the solution on the course mesh as a starting guess for the solution on the fine mesh (and vice versa).

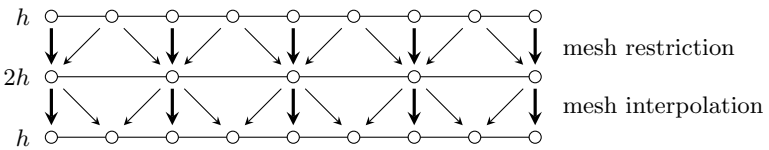
Consider the simple one-dimensional Poisson equation discussed in the previous example. Let $\mathbf{A}_h$ have a meshsize h and let $\mathbf{A}_{2h}$ be $\mathbf{A}_h$ restricted to a meshsize $2h$. One single cycle (with mesh h) can be implemented as follows

1. Iterate a few times on $\mathbf{A}_h\mathbf{u}_h = \mathbf{b}_h$ using weighted Jacobi, Gauss–Seidel, or SOR. This will kill off the residual $\mathbf{r}_h = \mathbf{b}_h - \mathbf{A}_h\mathbf{u}_h$ for the higher frequencies (high eigenvalues) and smooth out the solution.
2. Restrict the residual $\mathbf{r}_h$ to a coarse grid by taking $\mathbf{r}_{2h} = \mathbf{R}_h^{2h}\mathbf{r}_h$ where the restriction matrix is defined as

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

Then $\mathbf{r}_h \to \mathbf{r}_{2h}$ maps a $2n+1$ vector to an $n+1$ vector:

$$\left.\begin{array}{c} \frac{1}{4}(3r_0 + r_1) \\ \frac{1}{4}(r_{2j-1} + 2r_{2j} + r_{2j+1}) \\ \frac{1}{4}(r_{2n-1} + 3r_{2n}) \end{array}\right\} \to \left\{\begin{array}{ll} r_0 & \\ r_j & \text{for} \quad 0 < j < n \\ r_n & \end{array}\right.$$



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, $\mathbf{A}_{2h} = \mathbf{R}_h^{2h}\mathbf{A}_h$.

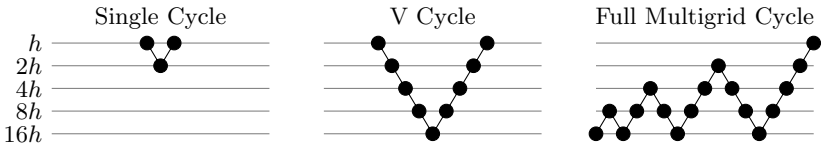
4. Interpolate back to $\mathbf{e}_h = \mathbf{I}_{2h}^h \mathbf{e}_{2h}$ where the interpolation matrix $\mathbf{I}_{2h}^h = 2\left(\mathbf{R}_h^{2h}\right)^T$:

$$\mathbf{e}_h = \frac{1}{2}\begin{bmatrix} 2e_0 & e_0 + e_1 & 2e_1 & e_1 + e_2 & \cdots & e_{n-1} + e_n & 2e_n \end{bmatrix}^T$$

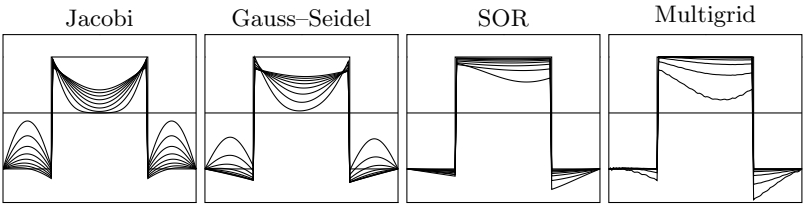
5. Add $\mathbf{e}_h$ to $\mathbf{u}_h$

6. Iterate $\mathbf{A}_h\mathbf{u}_h = \mathbf{b}_h$

Note that the error $\mathbf{e}_h = \mathbf{I}_{2h}^h(\mathbf{R}_h^{2h}\mathbf{A})^{-1}\mathbf{R}_h^{2h}\mathbf{r} = \mathbf{S}\mathbf{r}$. The multigrid cycle matrix $\mathbf{S}$ is a projection matrix. In practice, multiple cycles are used when restricting to coarser and coarse meshes before interpolating back, and multigrid may be started with a coarse grid. Here are different cycles used in the multigrid method to restrict and interpolate between the orignal mesh h and a coarser mesh $16h$:



Example. The Poisson equation $u'' = f(x)$ is solved using the Jacobi, Gauss–Seidel, SOR with relaxation factor $\omega = 1.75$, and multigrid using three cycles of SOR ($\omega = 1.75$). The source function is a combination of the derivative of Dirac delta distribution: $f(x) = \delta'(x - \frac{1}{2}) - \delta'(x + \frac{1}{2})$.



Starting with $\mathbf{u}^{(0)} = \frac{1}{2}$, each plot shows the solution over the domain $(-1, 1)$ at every tenth iteration for eight iterations. The Jacobi and Gauss–Seidel methods both exhibit a significant residual. ◀

5.4 Solving the minimization problem

We've already seen in Chapter 3, that we can solve the problem $\mathbf{Ax} = \mathbf{b}$ by solving an equivalent minimization problem. We also encountered a minimization problem when we wanted solve the eigenvalue problem for sparse matrices by finding the solution that minimized the error in the Krylov subspace. The Poisson equation $-\Delta u = f$ can also be re-expressed as a the problem of finding the solution u that minimizes the energy functional $\int_\Omega \frac{1}{2}|\nabla u|^2 - uf \,\mathrm{d}V$. In this section, we'll see how to formulate an iterative method to solve the minimization problem.

Steepest descent method

One of the difficulties of the SOR method is that you need to know the optimal relaxation parameter ω. If $\mathbf{A}$ is a symmetric, positive-definite matrix, there is a better approach by recasting the problem as a minimization problem.

Theorem 16. *If $\mathbf{A}$ is symmetric, positive definite, then the solution to the equation $\mathbf{Ax} = \mathbf{b}$ is the unique minimizer of the quadratic form*

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

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

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

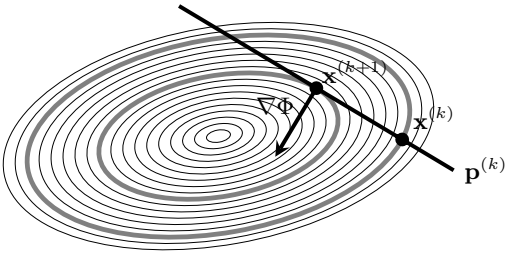


Figure 5.3: Optimal vector $\mathbf{x}^{(k+1)}$ along direction $\mathbf{p}^{(k)}$.

because $\mathbf{A}$ is symmetric

$$= \tfrac{1}{2}(\mathbf{y}-\mathbf{x})^T\mathbf{A}(\mathbf{y}-\mathbf{x}) > 0$$

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

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 kth subsequent estimate

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \tag{5.9}$$

for some direction $\mathbf{p}^{(k)}$. The error at the kth step is $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$. If we want to get to the bottom of the "bowl" only knowing the local topography at $\mathbf{x}^{(k)}$, then the best path would be to take the steepest descent, i.e., the direction of the negative gradient. The gradient of Φ at $\mathbf{x}^{(k)}$ is

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

That is, the negative gradient is simply the residual. This means that the kth subsequent estimate (5.9) can be rewritten as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}. \tag{5.10}$$

For now let's consider the more general iterative method (5.9) for an arbitrary direction vector $\mathbf{p}^{(k)}$, keeping in mind that $\mathbf{p}^{(k)} = \mathbf{r}^{(k)}$ for the method of steepest descent, and let's find α_k. Since evaluation of $\mathbf{A}\mathbf{x}^{(k)}$ to get the residual can require a lot of computer processing, we try to reduce the number of times we make a change in direction. We will only change direction when we are closest to $\mathbf{x}$ along our current direction. A vector $\mathbf{x}$ is said to be *optimal* with respect to a direction $\mathbf{p} \neq 0$ if

$$\Phi(\mathbf{x}) \leq \Phi(\mathbf{x} + \gamma\mathbf{p}), \qquad \text{for all } \gamma \in \mathbb{R}.$$

That is, $\mathbf{x}$ is optimal with respect to $\mathbf{p}$ if Φ is increases when we move away from $\mathbf{x}$ along the direction $\mathbf{p}$. Put another way, $\mathbf{x}$ is optimal with respect to a direction $\mathbf{p}$ if the directional derivative of Φ at $\mathbf{x}$ in the direction $\mathbf{p}$ is zero, i.e., $\nabla\Phi(\mathbf{x}) \cdot \mathbf{p} = 0$. If $\mathbf{x}$ is optimal with respect to any direction in a vector space V, we say the $\mathbf{x}$ is optimal with respect to V. Looking at (5.10), we want to know when $\mathbf{x}^{(k+1)}$ is optimal with respect to a direction $\mathbf{r}^{(k)}$.

Because $\nabla\Phi(\mathbf{x}^{(k+1)}) = -\mathbf{r}^{(k+1)}$, it follows that $\mathbf{x}^{(k+1)}$ is optimal with respect to $\mathbf{p}^{(k)}$—in the direction of $\mathbf{r}^{(k)}$—if and only if $\mathbf{r}^{(k+1)} \cdot \mathbf{p}^{(k)} = 0$. That is, if and only if $\mathbf{r}^{(k+1)}$ is orthogonal to $\mathbf{p}^{(k)}$.

$$\begin{aligned}
\mathbf{r}^{(k+1)} &= \mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)} \\
&= \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)}) \qquad \text{from (5.9)} \\
&= \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)} \\
&= \mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)}. \qquad (5.11)
\end{aligned}$$

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

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

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

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

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

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

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 $\mathbf{x}$ or we are close enough. That's the idea. Let's write it out as an algorithm.

Choose $\mathbf{x}^{(0)}$
for $k = 0, \ldots$
$\quad \mathbf{r}^{(k)} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$
$\quad$ if $\|\mathbf{r}^{(k)}\| <$ tolerance, then break
$\quad \alpha_k \leftarrow {\mathbf{r}^{(k)}}^T \mathbf{r}^{(k)} / {\mathbf{r}^{(k)}}^T \mathbf{A}\mathbf{r}^{(k)}$
$\quad \mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}$

Gauss–Seidel and SOR as descent methods

Let's go back an look at the general descent method (5.9)

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

where $\mathbf{x}^{(k)}$ is optimal with repect to the direction $\mathbf{p}^{(k)}$ when

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

Consider taking the direction $\mathbf{p}^{(k)}$ successively be along each of the coordinate axes $\boldsymbol{\xi}_i$. For example, if $\mathbf{A}$ is 3×3, we take

$$\mathbf{p}^{(1)} = \boldsymbol{\xi}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \ \mathbf{p}^{(2)} = \boldsymbol{\xi}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \ \mathbf{p}^{(3)} = \boldsymbol{\xi}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \ \mathbf{p}^{(4)} = \boldsymbol{\xi}_1, \ldots$$

Then

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \frac{{\boldsymbol{\xi}_i}^T \mathbf{r}^{(k)}}{{\boldsymbol{\xi}_i}^T \mathbf{A}\boldsymbol{\xi}_i} \boldsymbol{\xi}_i \quad \text{with} \quad i = k \,(\mathrm{mod}\, n) + 1.$$

We can rewrite this expression in component form by first noting that

$${\boldsymbol{\xi}_i}^T \mathbf{r}^{(k)} = b_i - \sum_{j=1}^{n} a_{ij} x_j \quad \text{and} \quad {\boldsymbol{\xi}_i}^T \mathbf{A}\boldsymbol{\xi}_i = a_{ii},$$

and then noting that $\mathbf{x}^{(k)} + \alpha_k \boldsymbol{\xi}_i$ only updates the ith 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 (5.1). 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 we have the SOR method. See Figure 5.4 on the following page.

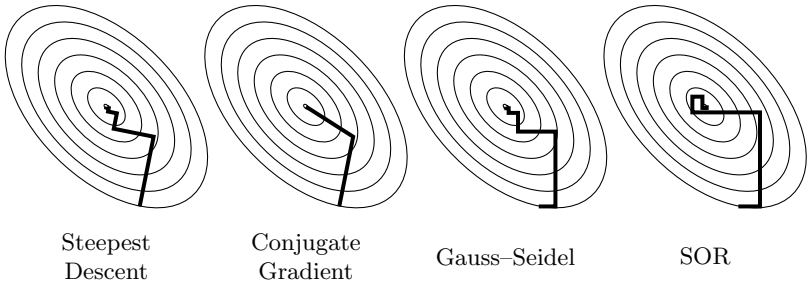
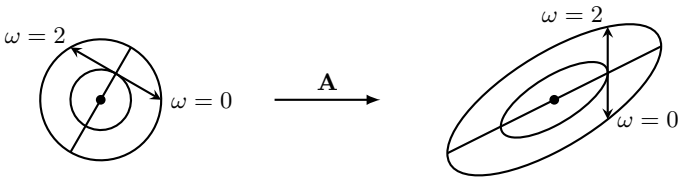


Figure 5.4: Solution using the method of steepest descent (left), conjugate gradient method, Gauss–Seidel, and SOR (right) on level curves of Φ.

Theorem 17. *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 16, if $\mathbf{A}$ is a symmetric, positive-definite matrix, then Φ has a unique minimizer. As a descent method the Gauss–Seidel converges to the bottom of the "bowl." We can consider the "bowl" as a circular paraboloid that is sheared (and stretched) along a coordinate axis (this follows directly from QR-decomposition).



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

Conjugate gradient method

While locally the steepest descent does gives the best choice of directions, globally it doesn't, especially when the "bowl" is very eccentric (high condition number). So, in practice the steepest descent method doesn't work particularly well. Let's look at a global method by taking optimal directions $\mathbf{p}$ that are not necessarily along the residuals.

What went wrong with the gradient method? Each new search direction $\mathbf{p}^{(k)}$ was determined only using information from most recent search direction $\mathbf{p}^{(k-1)}$. As before let

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

for some direction $\mathbf{p}^{(k)}$. Suppose that $\mathbf{x}^{(k+1)}$ is optimal with respect to $\mathbf{p}^{(k)}$ (as we did with the gradient method):

$$\mathbf{p}^{(k)} \perp \nabla\Phi(\mathbf{x}^{(k+1)}) \quad \text{or equivalently} \quad \mathbf{p}^{(k)} \perp \mathbf{r}^{(k+1)}.$$

Let's impose the further condition that $\mathbf{x}^{(k+1)}$ is also optimal with respect to $\mathbf{p}^{(j)}$ for all $j \leq k$. That is,

$$\mathbf{r}^{(k+1)} \perp \mathbf{p}^{(j)} \quad \text{for all} \quad j \leq k.$$

Let's pause a minute to examine what this new optimality conditions really means. If $\mathbf{x}^{(k+1)}$ is optimal with respect to each of the directions $\mathbf{p}^{(k)}$, $\mathbf{p}^{(k-1)}, \ldots \mathbf{p}^{(0)}$, then $\mathbf{x}^{(k+1)}$ is optimal with respect to the entire $\text{span}\{\mathbf{p}^{(k)}, \ldots, \mathbf{p}^{(0)}\}$ by linearity of the gradient. As we iterate, the subspace spanned by the directions $\mathbf{p}^{(k)}$, $\mathbf{p}^{(k-1)}, \ldots, \mathbf{p}^{(0)}$ will continue to grow as long as each new direction is linearly independent. It sounds pretty good. Let's find the directions $\mathbf{p}^{(k)}$, $\mathbf{p}^{(k-1)}, \ldots, \mathbf{p}^{(0)}$ and show that they are in fact linearly independent.

From

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

and our optimality condition we have that

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

for all $j \leq k$. So,

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

For $j = k$ it follows that

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

And, because $\left(\mathbf{r}^{(k)}, \mathbf{p}^{(j)}\right)_2 = 0$ for all $j < k$, we must have that

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

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

$$\left(\mathbf{p}^{(k)}, \mathbf{p}^{(j)}\right)_{\mathbf{A}} \equiv \mathbf{p}^{(j)^T}\mathbf{A}\mathbf{p}^{(k)} = 0, \quad \text{for} \quad j = 0, 1, \ldots, k-1. \tag{5.12}$$

As before, let's take the first direction equal to the gradient at $\mathbf{x}^{(0)}$, i.e., $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$. Then each subsequent direction will be the A-conjugate to this gradient. Hence, the name *conjugate gradient method*.

It still makes sense to take the directions as close to the residuals as possible, yet still being mutually conjugate. Take

$$\mathbf{p}^{(k+1)} = \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)}. \tag{5.13}$$

where β_k is chosen so that

$$\left(\mathbf{p}^{(k+1)}, \mathbf{p}^{(j)}\right)_{\mathbf{A}} = 0, \qquad j = 0, 1, \ldots, k. \tag{5.14}$$

It may seem almost magical that by choosing $\mathbf{p}^{(k+1)}$ with explicit dependence only on $\mathbf{p}^{(k)}$ we in fact ensure that $\mathbf{p}^{(k+1)}$ is A-conjugae to all $\mathbf{p}^{(j)}$ for $j \leq k$. Let's see why it works. We'll dispell the magic by using Krylov subspaces in the next section. It follows from multiplying (5.13) by $\mathbf{A}\mathbf{p}^{(k)}$ that for $j = k$ we must have that

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

Now, let's show that this choice for β_k actually ensures that (5.14) in fact holds for all $j < k$. Let

$$V_k = \text{span}\{\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(k-1)}\}.$$

Since $\mathbf{p}^{(0)} = \mathbf{r}^{(0)}$, it follows from (5.13) that

$$V_k = \text{span}\{\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(k-1)}\}.$$

By our optimality condition we must have that $V_k \perp \mathbf{p}^{(k+1)}$. Also,

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

So, $\mathbf{A}\mathbf{p}^{(k-1)} \in V_{k+1}$ and hence $\mathbf{A}\mathbf{p}^{(j)} \in V_{k+1}$ for all $j < k$. Therefore, $\mathbf{A}\mathbf{p}^{(j)} \perp \mathbf{p}^{(k+1)}$ or equivalently $\left(\mathbf{p}^{(k+1)}, \mathbf{p}^{(j)}\right)_{\mathbf{A}} = 0$ for $j = 0, 1, \ldots, k-1$. Note that what we are really doing is performing an Arnoldi iteration to generate an $\mathbf{A}$-conjugate set of directions.

Here's the algorithm for the conjugate gradient method:

Choose $\mathbf{x}^{(0)}$ and take $k = 0$
$\mathbf{p}^{(0)} \leftarrow \mathbf{r}^{(0)} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
while $\|\mathbf{r}^{(k)}\| >$ tolerance

$$\left[\begin{array}{l} \alpha_k \leftarrow \dfrac{{\mathbf{r}^{(k)}}^T \mathbf{p}^{(k)}}{{\mathbf{p}^{(k)}}^T \mathbf{A}\mathbf{p}^{(k)}} \\ \mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \\ \mathbf{r}^{(k+1)} \leftarrow \mathbf{r}^{(k)} - \alpha_k \mathbf{A}\mathbf{p}^{(k)} \\ \beta_k \leftarrow \dfrac{{\mathbf{r}^{(k+1)}}^T \mathbf{A}\mathbf{p}^{(k)}}{{\mathbf{p}^{(k)}}^T \mathbf{A}\mathbf{p}^{(k)}} \\ \mathbf{p}^{(k+1)} \leftarrow \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)} \\ k \leftarrow k + 1 \end{array}\right.$$

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 not necessary to run the method for the full n iterations. Instead, we typically consider the conjugate gradient as an iterative method and stop after $k \ll n$ iterations.

The conjugate gradient method converges very quickly if the condition number is close to one. We can use a symmetric, positive-definite preconditioner $\mathbf{P}$ such that $\kappa(\mathbf{P}^{-1}\mathbf{A}) < \kappa(\mathbf{A})$. For a simple Jacobi preconditioner (diagonal scaling), we take $\mathbf{P} = \mathbf{D}$. The algorithm for the preconditioned conjugate gradient method replaces

$$\begin{array}{l} \beta_k \leftarrow \dfrac{{\mathbf{r}^{(k+1)}}^T \mathbf{A}\mathbf{p}^{(k)}}{{\mathbf{p}^{(k)}}^T \mathbf{A}\mathbf{p}^{(k)}} \\ \mathbf{p}^{(k+1)} \leftarrow \mathbf{r}^{(k+1)} - \beta_k \mathbf{p}^{(k)} \end{array} \qquad \text{with} \qquad \begin{array}{l} \text{Solve } \mathbf{P}\mathbf{z}^{(k+1)} = \mathbf{r}^{(k)} \\ \beta_k \leftarrow \dfrac{{\mathbf{z}^{(k+1)}}^T \mathbf{A}\mathbf{p}^{(k)}}{{\mathbf{p}^{(k)}}^T \mathbf{A}\mathbf{p}^{(k)}} \\ \mathbf{p}^{(k+1)} \leftarrow \mathbf{z}^{(k+1)} - \beta_k \mathbf{p}^{(k)} \end{array}$$

in the conjugate gradient method above.

Ⓜ The MATLAB function `pcg` 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 the general iterative method

$$\mathbf{P}\mathbf{x}^{(k+1)} = \mathbf{N}\mathbf{x}^{(k)} + \mathbf{b}.$$

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

$$\begin{aligned}\mathbf{x}^{(k)} - \mathbf{x}^{(0)} = \mathbf{e}^{(k)} - \mathbf{e}^{(0)} &= \left[(\mathbf{I} - \mathbf{P}^{-1}\mathbf{A})^k - \mathbf{I}\right] \mathbf{e}^{(0)} \\ &= \left[\sum_{j=1}^{k} \binom{k}{j} (-1)^j (\mathbf{P}^{-1}\mathbf{A})^j\right] \mathbf{e}^{(0)}.\end{aligned}$$

This says that $\mathbf{x}^{(k)} - \mathbf{x}^{(0)}$ lies in the Krylov space

$$\mathcal{K}_k(\mathbf{P}^{-1}\mathbf{A}, \mathbf{z}) = \operatorname{span}\left\{\mathbf{z}, \mathbf{P}^{-1}\mathbf{A}\mathbf{z}, \ldots, (\mathbf{P}^{-1}\mathbf{A})^k \mathbf{z}\right\}.$$

with $\mathbf{z} = \mathbf{P}^{-1}\mathbf{A}\mathbf{e}^{(0)} = \mathbf{P}^{-1}\mathbf{r}^{(0)}$.

The steepest descent method is also a Krylov method. Equation (5.11) says that the residuals are elements of the Krylov subspace

$$\mathcal{K}_k(\mathbf{A}, \mathbf{r}^{(0)}) = \operatorname{span}\left\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}, \ldots, \mathbf{A}^k \mathbf{r}^{(0)}\right\}.$$

In the previous sections, we considered a generic iterative method

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} \\ &= \mathbf{x}^{(0)} + \sum_{j=1}^{k} \alpha_j \mathbf{p}^{(j)}\end{aligned}$$

for some directions $\{\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(k)}\}$. For the method of steepest descent, 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

$$V_{k+1} = \operatorname{span}\{\mathbf{p}^{(0)}, \mathbf{p}^{(1)}, \ldots, \mathbf{p}^{(k)}\} = \operatorname{span}\{\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(k)}\}.$$

Furthermore, we noted that the V_{k+1} is actually a Krylov subspace

$$V_{k+1} = \operatorname{span}\{\mathbf{r}^{(0)}, \mathbf{A}\mathbf{r}^{(0)}, \ldots, \mathbf{A}^k \mathbf{r}^{(0)}\}.$$

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 $\boldsymbol{\delta}^{(k)} \in \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{r}^{(0)})$. For an initial guess $\mathbf{x}^{(0)}$, let's consider two methods for advancing $\mathbf{x}^{(k)}$ with $\boldsymbol{\delta}^{(k)} \in \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{r}^{(0)})$

1. Choose $\mathbf{x}^{(k+1)}$ that minimizes the error $\|\mathbf{e}^{(k+1)}\|_{\mathbf{A}}$.
2. Choose $\mathbf{x}^{(k+1)}$ that minimizes the residual $\|\mathbf{r}^{(k+1)}\|_2$.

The first approach is often called the Arnoldi method or the Full Orthogonalization Method (FOM). When $\mathbf{A}$ is a symmetric, positive-definite matrix, FOM is equivalent to the conjugate gradient method. The second approach is known as the general minimal residual method (GMRES). It is called simply the minimal residual method (MINRES) when $\mathbf{A}$ is symmetric.

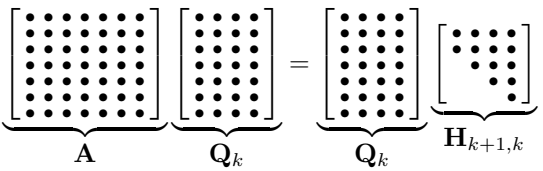
Before looking at either of the two methods, let's recall the Arnoldi process from the previous chapter. The Arnoldi process is a variation of the Gram-Schmidt method that generates an orthogonal basis for a Krylov subspace. By choosing an orthogonal basis, we ensure that the system is well-conditioned. Starting with the residual $\mathbf{r}^{(0)}$, take

$$\mathbf{q}^{(1)} \leftarrow \mathbf{r}^{(0)}/\|\mathbf{r}^{(0)}\|$$

and set $h_{11} = \|\mathbf{r}^{(0)}\|$. At the kth iteration take

$$\begin{aligned} h_{ik} &= (\mathbf{q}_i, \mathbf{A}\mathbf{q}_k) \quad \text{for} \quad i = 1, \ldots, k \\ \mathbf{q}^{(k+1)} &= \mathbf{A}\mathbf{q}^{(k)} - \sum_{i=1}^{k} h_{ik}\mathbf{q}^{(k)} \\ \mathbf{q}^{(k+1)} &\leftarrow \mathbf{q}^{(k+1)}/h_{k+1,k} \quad \text{with} \quad h_{k+1,k} = \|\mathbf{q}_{k+1}\|. \end{aligned}$$

We can summarize the method as $\mathbf{A}\mathbf{Q}_k = \mathbf{Q}_{k+1}\mathbf{H}_{k+1,k}$:



(5.15)

By separating the bottom row of $\mathbf{H}_{k+1,k}$ and the last column of $\mathbf{Q}_{k+1}$, we have that

$$\mathbf{A}\mathbf{Q}_k = \mathbf{Q}_k\mathbf{H}_k + h_{k+1,k}\mathbf{q}_{k+1}\boldsymbol{\xi}_k^T \tag{5.16}$$

where $\boldsymbol{\xi}_k = (0, \ldots, 0, 1)$. Note that if $\mathbf{A}$ is Hermitian (or real symmetric), then $\mathbf{H}_k$ is tridiagonal. In this case, the Arnoldi method simplifies and is called the Lanczos method.

Full orthogonalization method (FOM)

Let's examine the first method: choose $\mathbf{x}^{(k+1)}$ that minimizes the error $\|\mathbf{e}^{(k+1)}\|_{\mathbf{A}}$. At the kth 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

$$\|\mathbf{e}^{(k+1)}\|_{\mathbf{A}}^2 = \left(\mathbf{A}\mathbf{e}^{(k+1)}, \mathbf{e}^{(k+1)}\right)_2 = \mathbf{e}^{(k+1)^T}\mathbf{A}\mathbf{e}^{(k+1)}.$$

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 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 gradient is zero:

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

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 (5.16) 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 $\boldsymbol{\xi}_1 = (1, 0, \ldots, 0)$. Note that norm of the residual

$$\|\mathbf{r}^{(k)}\|_2 = \|\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}\|_2 = |h_{k+1,h}| \cdot |\boldsymbol{\xi}_k^T\mathbf{z}^{(k)}| = |h_{k+1,h}| \cdot |(z_k)^{(k)}|$$

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

It may not be obvious, but FOM outlined above is really just the conjugate gradient method when $\mathbf{A}$ is a symmetric, positive-definite matrix. Start with

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{x}^{(0)} + \mathbf{Q}_k\mathbf{z}^{(k)} \\ &= \mathbf{x}^{(0)} + \mathbf{Q}_k\mathbf{H}_k^{-1}\|\mathbf{r}^{(0)}\|_2\boldsymbol{\xi}_k.\end{aligned}$$

Taking the LU-decomposition of $\mathbf{H}_k$ we have that

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \mathbf{Q}_k\mathbf{U}_k^{-1}\mathbf{L}_k^{-1}\|\mathbf{r}^{(0)}\|_2\boldsymbol{\xi}_k = \mathbf{x}^{(0)} + \mathbf{P}_k\mathbf{L}_k^{-1}\|\mathbf{r}^{(0)}\|_2\boldsymbol{\xi}_k$$

where the columns of $\mathbf{P}_k$ are the directions $\mathbf{p}^{(k)}$. For a symmetric matrix $\mathbf{A}$, we have that

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

Hence, $\mathbf{P}_k^T\mathbf{A}\mathbf{P}_k$ is a diagonal matrix and it follows that

$$\left(\mathbf{p}^{(i)}, \mathbf{A}\mathbf{p}^{(j)}\right)_2 = \left(\mathbf{p}^{(i)}, \mathbf{p}^{(j)}\right)_{\mathbf{A}} = 0 \quad \text{for} \quad i \neq j,$$

which says that the directions are A-conjugate.

General minimized residual (GMRES)

This time we will choose $\mathbf{x}^{(k+1)}$ that minimizes the residual $\|\mathbf{r}^{(k+1)}\|_2$. As before we take

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(0)} + \mathbf{Q}_k\mathbf{z}^{(k)}$$

for some vector $\mathbf{z}^{(k)}$ to be determined. From $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$, we have

$$\begin{aligned}\mathbf{r}^{(k+1)} &= \mathbf{r}^{(0)} - \mathbf{A}\mathbf{Q}_k\mathbf{z}^{(k)} \\ &= \mathbf{Q}_{k+1}\mathbf{Q}_{k+1}^T\mathbf{r}^{(0)} - \mathbf{A}\mathbf{Q}_k\mathbf{z}^{(k)} \\ &= \mathbf{Q}_{k+1}(\|\mathbf{r}^{(0)}\|_2\boldsymbol{\xi}_1 - \mathbf{H}_{k+1,k}\mathbf{z}^{(k)})\end{aligned}$$

where $\boldsymbol{\xi}_1 = (1, 0, \ldots, 0)$. We now have the problem of finding $\mathbf{z}^{(k)}$ to minimize

$$\|\mathbf{r}^{(k+1)}\|_2 = \left\| \ \|\mathbf{r}^{(0)}\|_2\boldsymbol{\xi}_1 - \mathbf{H}_{k+1,k}\mathbf{z}^{(k)} \ \right\|_2.$$

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

$$\mathbf{H}_{k+1,k}\mathbf{z}^{(k)} = \|\mathbf{r}^{(0)}\|_2\boldsymbol{\xi}_1 = \mathbf{b}$$

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 sub-diagonal elements, giving us

$$\hat{\mathbf{R}}\mathbf{z}^{(k)} = \hat{\mathbf{Q}}^T\mathbf{b}.$$

The residual $\|\mathbf{r}^{(k+1)}\|_2$ is given by the bottom element of $\hat{\mathbf{Q}}^T\mathbf{b}$ which turns out to be

$$\|\mathbf{r}^{(0)}\|_2(\boldsymbol{\xi}_1^T\hat{\mathbf{Q}}\boldsymbol{\xi}_{k+1}) = \|\mathbf{r}^{(0)}\|_2\hat{q}_{1,k+1}$$

If $\|\mathbf{r}^{(k)}\|_2 < \varepsilon\|\mathbf{r}^{(0)}\|_2$ for some tolerance ε we stop. Otherwise, we continue by finding the next basis vector $\mathbf{q}^{(k+2)}$ of the Krylov subspace using the Arnoldi process.

Ⓜ The MATLAB function `gmres` implements the generalized minimum residual method and `minres` implements the minimum residual method.

Exercises

✍ 5.1. Consider the 2-by-2 matrix

$$\mathbf{A} = \begin{bmatrix} 1 & \sigma \\ -\sigma & 1 \end{bmatrix}.$$

Under what conditions will Gauss–Seidel converge? For what range ω will the SOR method converge? What is the optimal choice of ω?

5.2. Show that if $\mathbf{A} \in \mathbb{R}^{n\times n}$ is symmetric, positive-definite and has k distinct eigenvalues, then the conjugate gradient method does not require more than $k+1$ steps to converge.

✍ 5.3. Consider the three-dimensional Poisson equation $-\Delta u(x,y,z) = f(x,y,z)$ with Dirichlet boundary conditions $u(x,y,z) = 0$ over the unit cube $[0,1]\times[0,1]\times[0,1]$. Just as in the two-dimensional case, we can use a finite-difference approximation to solve the problem, but instead of a five-point stencil, now we'll use a seven-point stencil

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} \approx \frac{6u_{ijk} - u_{i-1,j,k} - u_{i+1,j,k} - u_{i,j-1,k} - u_{i,j+1,k} - u_{i,j,k-1} - u_{i,j,k+1}}{h^2}$$

Suppose that we discretize space using 50 points in each dimension. The finite-difference matrix $\mathbf{A}$ to the problem $\mathbf{Ax} = \mathbf{b}$ is a 125000×125000 sparse matrix. Even though $\mathbf{A}$ has over 15 billion elements, only 0.005% 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

$$u(x,y,z) = \tfrac{1}{2}(x-x^2)(y-y^2)(z-z^2).$$

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. The conjugate gradient method is easy to program, but if you want, you may use MATLAB's `pcg` command. Plot the error of the methods for the first 500 iterations and comment on the convergence.

Hint: An easy way to build the finite difference matrix is by using a Kronecker tensor product. The Kronecker tensor product is defined as

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

The seven-point finite-difference operator is

$$\mathbf{D} \otimes \mathbf{I} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{D} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{I} \otimes \mathbf{D}$$

where $\mathbf{D}$ is a 50×50 the difference matrix (5.7) and $\mathbf{I}$ is a 50×50 identity matrix. It can be implemented in MATLAB with the `kron` command.

```
n = 50; x = (1:n)'/(n+1); dx = 1/(n+1);
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;
```

The Jacobi, Gauss–Seidel and SOR methods can be implemented easily by iterating on `u=u+P\(f-A*u)` by just redefining the preconditioner `P`. You can use the `meshgrid` command to help build your source `f` and then use lexicon ordering `f(:)`.

CHAPTER 6

Fast Fourier Transform

The discrete Fourier transform (DFT) is an important algorithm for signal processing, data compression, and solutions to partial differential equations. It is a critical for technologies such as cell phones, mp3s, and jpegs. One of the reasons the DFT has become so important is that its implementation using the fast Fourier transform (FFT) is efficient.

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 points through the points $(z_j, y_j) \in \mathbb{C}^2$ for $j = 1, \ldots, n-1$. In this case, we simply solve the system $\mathbf{Vc} = \mathbf{y}$ where $\mathbf{V}$ is the Vandermonde matrix to get the coefficients c_j of the polynomial:

$$\begin{bmatrix} 1 & z_0 & z_0^2 & \cdots & z_0^n \\ 1 & z_1 & z_1^2 & \cdots & z_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & z_{n-1} & z_{n-1}^2 & \cdots & z_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Now, suppose that we restrict the points z_j to equally spaced points on the unit circle $|z_j| = 1$ with $z_0 = 1$. That is, take

$$z_j = \mathrm{e}^{-\mathrm{i}2\pi j/n} = \omega_n^j$$

by choosing nodes clockwise around the unit circle staring with $z_0 = 1$. The value $\omega_n = \exp(-\mathrm{i}2\pi/n)$ is called an *nth root of unity* because it solves the equation $z^n = 1$. (In fact, ω_n^k are all nth roots of unity because they all are solutions to $z^n = 1$.) In this case, the Vandermonde matix is

$$\mathbf{F}_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}. \tag{6.1}$$

This matrix is called the *discrete Fourier transform* (DFT). In index notation we have that

$$y_k = \sum_{j=0}^{n-1} c_j \mathrm{e}^{-\mathrm{i}2\pi jk/n} = \sum_{j=0}^{n-1} c_j \omega_n^{jk}.$$

The discrete Fourier transform has several important properties.

- The DFT matrix is a scaled unitary matrix. That is, $\mathbf{F}_n/\sqrt{n}$ is a unitary matrix. This fact follows from a simple application of the geometric series. Recall that if $z \neq 1$, then

  $$\sum_{j=0}^{n-1} z^j = \frac{z^n - 1}{z - 1}.$$

  Then (taking $\overline{\omega}$ to be the complex conjugate of ω) we have that

  $$\sum_{j=0}^{n-1} \omega_n^{jk} \overline{\omega}_n^{jl} = \sum_{j=0}^{n-1} \omega_n^{j(k-l)} = \begin{cases} \dfrac{\omega_n^{n(k-l)} - 1}{\omega_n^{k-l} - 1} = \dfrac{1-1}{\omega_n^{k-l} - 1} = 0, & k \neq l \\ \displaystyle\sum_{j=0}^{n-1} 1 = n, & k = l \end{cases}$$

  It also follows that the *inverse discrete Fourier transform* (IDFT) is simply

  $$\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^H.$$

  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. Also, note that the DFT is symmetric (but not Hermitian).

- The DFT maps a vector from spatial-domain to frequency-domain and the IFT maps it back.

- There are several alternative (but ultimately equivalent) formulations of the DFT. For example,

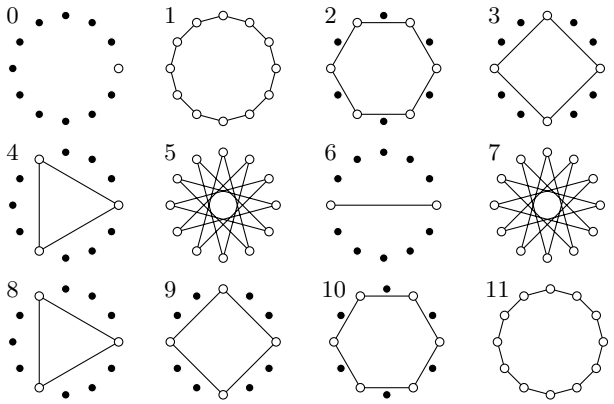


Figure 6.1: Subgroups using the generator ω_{12}^j for $j = 0, 1, \ldots, 11$.

$$\sum_{j=0}^{n-1} f_{j+1}\omega_n^{jk} \quad \text{(MATLAB)} \qquad \frac{1}{\sqrt{n}}\sum_{j=0}^{n-1} f_j\omega_n^{jk} \quad \text{(Mathematica)}$$

$$\frac{1}{n}\sum_{j=0}^{n-1} f_j\omega_n^{jk} \quad \text{(Maple)} \qquad \frac{1}{n}\sum_{j=-n/2+1}^{n/2} f_j\omega_n^{jk}$$

And there are others.

- The kth column (and row) of $\mathbf{F}_n$ forms a subgroup generated by ω_n^k. That is, $\{kj \mid j = 0, 1, 2, \ldots, n-1\}$ forms a group under addition modulo n. For example, take $n = 12$ 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 Also, note that using 10 as a generator gives us the same group, because $10 \equiv -2 \pmod{12}$. In particular note that this subgroup corresponds to the six 6th roots of unity. That is, the subgroup generated by ω_{12}^2 equals the group generated by ω_6^1.

  If the generator k is relatively prime with respect to n, then the number of elements of the group is n. We say the group has order n. For example, both 5 and 7 are relative primes of 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* which is 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 $\omega \equiv \omega_{12}$

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

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 $\gcd(k, n)$. See Figure 6.1 on the preceding page. The number of cosets determines the radix which is the basis for Cooley–Tukey algorithm.

6.2 Fast Fourier transform

There are several FFT algorithms. Each works by exploiting the group structure of the DFT matrix. The typical FFT algorithm is Cooley–Tukey algorithm developed by Cooley and Tukey in 1965. (This algorithm was actually invented by Gauss in 1805.) In this section, we look at the Cooley–Tukey radix-2 algorithm (using powers of two).

Consider the case when n is divisible by two. The DFT $\mathbf{y}$ of $\mathbf{c}$ is

$$y_j = \sum_{k=0}^{n-1} \omega_n^{kj} c_k = \underbrace{\sum_{k=0}^{n/2-1} \omega_n^{2kj} c_{2k}}_{\text{even index}} + \underbrace{\sum_{k=0}^{n/2-1} \omega_n^{(2k+1)j} c_{2k+1}}_{\text{odd index}}.$$

Let $m = n/2$. Since $\omega_n^2 = \omega_m$, we have that

$$y_j = \sum_{k=0}^{m-1} \omega_m^{kj} c'_k + \omega_n^j \sum_{k=0}^{m-1} \omega_m^{kj} c''_k$$

where $c'_k = c_{2k}$ and $c''_k = c_{2k+1}$. So, $\mathbf{y}$ can be computed by using the sum of two smaller DFTs of size $m = n/2$

$$y_j = y'_j + \omega_n^j y''_j$$

Furthermore, we only need to compute $j = 0, \ldots, m-1$ and not $j = m, \ldots, n-1$ because

$$\begin{aligned} \omega_m^{k(m+j)} &= \omega_m^{km} \omega_m^{kj} = \omega_m^{kj} \quad \text{and} \\ \omega_n^{m+j} &= \omega_n^m \omega_n^j = -\omega_n^j. \end{aligned}$$

So,

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

Therefore,

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

To get a better picture of the math behind the algorithm, let's look at the matrix formulation of the radix-2 FFT. The $n = 4$ DFT is

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

Consider the even-odd permutation matrix (the inverse of the perfect shuffle)

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

Then

$$\begin{aligned} \mathbf{F}_4 \mathbf{P}_4^{-1} &= \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & -1 & -\mathrm{i} & \mathrm{i} \\ \hline 1 & 1 & -1 & -1 \\ 1 & -1 & \mathrm{i} & -\mathrm{i} \end{array}\right] = \begin{bmatrix} \mathbf{F}_2 & \mathbf{\Omega}_2 \mathbf{F}_2 \\ \mathbf{F}_2 & -\mathbf{\Omega}_2 \mathbf{F}_2 \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I}_2 & \mathbf{\Omega}_2 \\ \mathbf{I}_2 & -\mathbf{\Omega}_2 \end{bmatrix} \begin{bmatrix} \mathbf{F}_2 & \\ & \mathbf{F}_2 \end{bmatrix} \end{aligned}$$

where

$$\mathbf{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{\Omega}_2 = \begin{bmatrix} 1 & 0 \\ 0 & \mathrm{i} \end{bmatrix} = \operatorname{diag}(1, \omega_4)$$

The matrix

$$\mathbf{B}_4 = \begin{bmatrix} \mathbf{I}_2 & \Omega_2 \\ \mathbf{I}_2 & -\Omega_2 \end{bmatrix}$$

is called the *butterfly matrix*. The DFT is then

$$\begin{aligned} \mathbf{F}_4 &= \mathbf{B}_4 \begin{bmatrix} \mathbf{F}_2 & \\ & \mathbf{F}_2 \end{bmatrix} \mathbf{P}_4 \\ &= \mathbf{B}_4 \begin{bmatrix} \mathbf{F}_2 \mathbf{P}_2^{-1} & \\ & \mathbf{F}_2 \mathbf{P}_2^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{P}_2 & \\ & \mathbf{P}_2 \end{bmatrix} \mathbf{P}_4 = \mathbf{B}_4 \begin{bmatrix} \mathbf{B}_2 & \\ & \mathbf{B}_2 \end{bmatrix} \mathbf{P} \end{aligned}$$

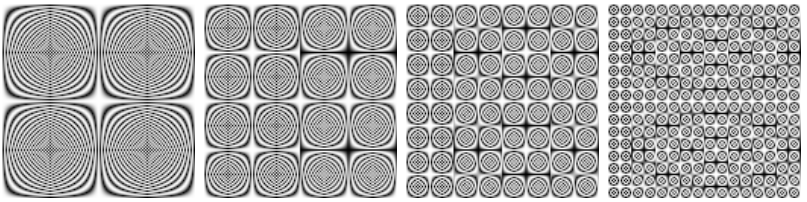


Figure 6.2: $\mathbf{F}_{128}$ (left), after one permutation, after a second permutation, and after third permutation (right) of the columns. Note the emergence of $\mathbf{F}_{64}$, $\mathbf{F}_{32}$ and $\mathbf{F}_{16}$.

where $\mathbf{P}$ is the permutation matrix of successive even/odd rearrangements. The pattern reveals itself for larger matrices as well. See Figure 6.2. For $n = 4$,

$$\mathbf{F}_4\mathbf{x} = (\mathbf{I}_1 \otimes \mathbf{B}_4)(\mathbf{I}_2 \otimes \mathbf{B}_2)\mathbf{P}^T\mathbf{c}.$$

In general

$$\mathbf{F}_n\mathbf{c} = (\mathbf{I}_1 \otimes \mathbf{B}_n)(\mathbf{I}_2 \otimes \mathbf{B}_{n/2}) \cdots (\mathbf{I}_{n/2} \otimes \mathbf{B}_2)\mathbf{P}\mathbf{c} \tag{6.3}$$

where

$$\mathbf{B}_{2n} = \begin{bmatrix} \mathbf{I}_n & \mathbf{\Omega}_n \\ \mathbf{I}_n & -\mathbf{\Omega}_n \end{bmatrix}$$

and $\mathbf{\Omega} = \mathrm{diag}(1, \overline{\omega}_{2n}, \overline{\omega}_{2n}^2, \ldots, \overline{\omega}_{2n}^{n-1})$. 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 *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

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1q}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_p\mathbf{B} & \cdots & a_{pq}\mathbf{B} \end{bmatrix}$$

M The MATLAB function `kron(A,B)` returns the Kronecker product $\mathbf{A} \otimes \mathbf{B}$.

Directly computing the DFT of $\mathbf{c}$ requires $2n^2$ multiplications and additions. We are now able to do compute it using 2 DFTs of length $m = n/2$ and an additional $n/2$ multiplications and n additions. That is a total of $\frac{1}{2}n^2 + \frac{3}{2}n$. But we're not finished yet. Suppose that n is a power of two. Then we can extend this idea recursively by continuing

to decompose to $n/4$, $n/8$, $\ldots$, 1 nodes. For $n = 2^p$, the number of multiplications and additions is

$$\begin{aligned} M_p &= 2M_{p-1} + 2^{p-1} \\ A_p &= 2A_{p-1} + 2^p. \end{aligned}$$

The solution to this difference equation is

$$\begin{aligned} M_p &= p2^{p-1} = \tfrac{1}{2} n \log_2 n \\ A_p &= p2^p = n \log_2 n. \end{aligned}$$

So, the number of operations is $O(\frac{3}{2} n \log_2 n)$. This means that a thousand-point FFT is roughly a hundred times faster than a direct computation $O(2n^2)$. Similar, algorithms work for n power of three, five, etc. (radix-3, radix-5, etc.).

Using (6.1) and (6.2), the radix-2 algorithm can be written as a recursive function in MATLAB (Moler [2004]):

```
function y = fftx(c)
n = length(c);
omega = exp(-2i*pi/n);
if mod(n,2) == 0
% Recursively apply equations (6.2)
  k = (0:n/2-1)';
  u = fftx(c(1:2:n-1));
  v = (omega.^k).*fftx(c(2:2:n));
  y = [u+v; u-v];
else
% Compute the DFT
  j = 0:n-1;
  k = j';
  F = omega.^(k*j);
  y = F*c;
end
```

$$\omega_n = \exp(2\pi \mathrm{i}/n)$$

$$y_j = y'_j + w_n^j y''_j$$

$$y_{j+n/2} = y'_j - w_n^j y''_j$$

$$\mathbf{F}_n = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

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 return to two such algorithms later.

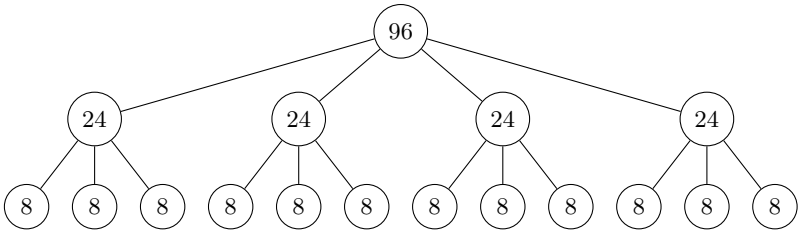


Figure 6.3: Mixed radix factorization down to radix-8.

6.3 Toeplitz and circulant matrices

An $n \times n$ matrix is *Toeplitz* (or diagonal-constant) if each descending diagonal has constant value. That is, a Toeplitz matrix has the form

$$\mathbf{T} = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \cdots & a_{-(n-2)} \\ a_2 & a_1 & a_0 & \cdots & a_{-(n-3)} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \end{bmatrix}$$

for some values $\{a_{-(n-1)}, a_{-(n-2)}, \ldots, a_0, \ldots, a_{(n-1)}\}$.

Ⓜ The MATLAB command `toeplitz` will produce a Toeplitz matrix.

An $n \times n$ Toeplitz matrix is *circulant* if it has the form

$$\mathbf{C} = \begin{bmatrix} c_0 & c_{n-1} & c_{n-2} & \cdots & c_1 \\ c_1 & c_0 & c_{n-1} & \cdots & c_2 \\ c_2 & c_1 & c_0 & \cdots & c_3 \\ \vdots & \vdots & \vdots & & \vdots \\ c_{n-1} & c_{n-2} & c_{n-3} & \cdots & c_0 \end{bmatrix}$$

for some values $\{c_0, c_1, \ldots, c_{n-1}\}$. Circulant matrices has several important properties:

- $\mathbf{C}$ is a Krylov matrix formed by the downshift permutation matrix

$$\mathbf{R} = \begin{bmatrix} \boldsymbol{\xi}_2 & \boldsymbol{\xi}_2 & \cdots & \boldsymbol{\xi}_n & \boldsymbol{\xi}_1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

That is,

$$\mathbf{C} = \begin{bmatrix} \mathbf{c} & \mathbf{Rc} & \mathbf{R}^2\mathbf{c} & \cdots & \mathbf{R}^{n-1}\mathbf{c} \end{bmatrix} \text{ with } \mathbf{c} = \begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix}^T.$$

- $\mathbf{C}$ can be written as $\mathbf{C} = c_0\mathbf{I} + c_1\mathbf{R} + c_2\mathbf{R}^2 + \cdots + c_{n-1}\mathbf{R}^{n-1}$.
- $\mathbf{R}$ is itself a circulant matrix

Ⓜ A circulant matrix can be built in MATLAB from a vector `v` using `toeplitz(v,circshift(flipud(v),1))`.

Theorem 18. *An $n \times n$ 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 $n \times n$ discrete Fourier transform.*

Proof. Let $\mathbf{R}$ be an $n \times n$ downshift permutation matrix and let $\mathbf{w}_k$ be the kth column of the $n \times n$ DFT matrix $\mathbf{F}$. That is to say, let $\mathbf{w}_k = \begin{bmatrix} 1 & \omega^k & \omega^{2k} & \cdots & \omega^{(n-1)k} \end{bmatrix}^T$ where $\omega = \exp(-2\pi i/n)$. It follows that $\mathbf{Rv}_k = \omega^k\mathbf{v}_k$—that is, ω^k is an eigenvalue of $\mathbf{R}$ and $\mathbf{v}_k$ is its associated eigenvector. Therefore, the downshift permutation matrix has the diagonalization $\mathbf{R} = \mathbf{F}^{-1}\mathbf{WF}$ where the diagonal matrix

$$\mathbf{W} = \operatorname{diag}(1, \omega, \omega^2 \ldots, \omega^{n-1}).$$

Take $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$. Then

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

and it follows that

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

□

Fast convolution

The convolution of two functions can be described as their shifted inner products. For continuous functions $f(t)$ and $g(t)$, we have that the convolution

$$(f * g)(t) = \int_{-\infty}^{\infty} f(s)g(t-s)\,\mathrm{d}s$$

When $\int_{-\infty}^{\infty} f(s)\,\mathrm{d}s = 1$, you can think of $f * g$ as a moving average of g using f as a window function. For discrete periodic functions $\mathbf{u}$ and $\mathbf{v}$, the kth element of the circular convolution $\mathbf{u} * \mathbf{v}$ is simply

$$(\mathbf{u} * \mathbf{v})_k = \sum_{j=0}^{n-1} u_j v_{k-j \,(\mathrm{mod}\, n)}.$$

Example. While the notion of a convolution may at first seem foreign to many, we are actually taught convolutions in grade school. Consider the product of 123 and 241. We compute the product using a convolution

$$\begin{array}{cccccl}
 & & 1 & 2 & 3 & \\
\times & & 2 & 4 & 1 & \\
\hline
 & & 1 & 2 & 3 & \\
 & 4 & 8 & 12 & & \\
2 & 4 & 6 & & & \\
\hline
2 & 8 & 15 & 14 & 3 & \quad \text{(and carrying...)} \\
\hline
2 & 9 & 6 & 4 & 3 &
\end{array}$$

So, $123 \times 241 = 29643$. ◀

The circular convolution of $\mathbf{u}$ and $\mathbf{v}$ is expressed in matrix notation as $\mathbf{u} * \mathbf{v} = \mathbf{C_u v}$ where $\mathbf{C_u}$ is the circulant matrix constructed from $\mathbf{u}$:

$$\begin{bmatrix} u_0 & u_4 & u_3 & u_2 & u_1 \\ u_1 & u_0 & u_4 & u_3 & u_2 \\ u_2 & u_1 & u_0 & u_4 & u_3 \\ u_3 & u_2 & u_1 & u_0 & u_4 \\ u_4 & u_3 & u_2 & u_3 & u_0 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}.$$

Therefore, from Theorem 18

$$\mathbf{u} * \mathbf{v} = \mathbf{C_u v} = \mathbf{F}^{-1} \operatorname{diag}(\mathbf{Fu})\mathbf{Fv}.$$

In practice,

$$\mathbf{u} * \mathbf{v} = \mathbf{F}^{-1}(\mathbf{Fu}) \circ (\mathbf{Fv}) \tag{6.4}$$

where $\circ$ denotes the Hadamard product. The Hadamard product (or Schur product) is a fancier way of saying the element-wise product.

Ⓜ In MATLAB, the convolution of `u` and `v` is simply `ifft(fft(u).*fft(v))`.

We can compute the convolution simply by computing the IDFT of the component-wise product of the DFTs of $\mathbf{u}$ and $\mathbf{v}$. This evaluation requires computation of three FFTs. Instead of the $2n^2$ operations required to perform the convolution directly, it only take $2n \log_2 n$ operations to compute the convolution using (6.4).

Fast Toeplitz multiplication

From the previous section, we saw that a circulant matrix can be multiplied quickly by using FFTs. We can do the same thing for a general Toeplitz matrix by padding it out to become a circulant matrix. Consider $\mathbf{v} = \mathbf{T}\mathbf{u}$:

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

where we discard the $*$ terms to get the solution $\mathbf{v}$. 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 so that an m-length DFT is convenient, e.g., m is a power of two or some other product of small primes. For example,

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

We never need to construct the full matrix $\mathbf{C}$—instead, we perform a fast convolution using the first column of $\mathbf{C}$. Unfortunately, we can't use this approach to solve a system—only mutliply.

Example. Consider the $n \times n$ Toeplitz matrix whose first column is given by the n-long vector $\mathbf{u}$ and whose first row is given by the $(n-1)$-long vector $\mathbf{v}$. Suppose that we also pad the matrix with zeros to get a $2^p \times 2^p$ circulant matrix for some p so that we can efficiently use our radix-2 function `fftx`. The MATLAB code is

```
function y = fasttoeplitz(u,v,x)
n = length(x);
m = 2^(ceil(log2(n)))-n;
x1 = [u; zeros([m 1]); v(end:-1:2)];
x2 = [x; zeros([m+n-1 1])];
v = ifftx(fftx(x1).*fftx(x2));
y = real(v(1:n));
```

◀

6.4 Bluestein and Rader factorization

The Cooley–Tukey algorithm only works on DFT for vectors of length n where n is a composite. And it only works well recursively if n is a composite of small primes. Luckily there are methods when n is a large prime number.

Bluestein (Chirp-z) factorization

Consider the DFT $\mathbf{y} = \mathbf{Fc}$. By using the identity $(j-k)^2 = j^2 - 2jk + k^2$, we can replace jk in the DFT by $\frac{1}{2}j^2 + \frac{1}{2}(j-k)^2 + \frac{1}{2}k^2$):

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

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 $\mathbf{F} = \mathbf{WTW}$ where the diagonal matrix $\mathbf{W} = \operatorname{diag}(\mathbf{w})$ with

$$\mathbf{w} = \begin{bmatrix} 1 & \omega_{2n} & \omega_{2n}^4 & \omega_{2n}^9 & \dots & \omega_n^{(n-1)^2} \end{bmatrix}$$

and $\mathbf{T}$ is a symmetric Toeplitz matrix whose the first row is given by $\overline{\mathbf{w}}$. For example, for $\mathbf{F}_5$

$$\mathbf{w} = \begin{bmatrix} 1 & \omega & \omega^4 & \omega^9 & \omega^6 \end{bmatrix} \quad \text{and} \quad \overline{\mathbf{w}} = \begin{bmatrix} 1 & \omega^9 & \omega^6 & \omega^1 & \omega^4 \end{bmatrix}$$

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, $\mathbf{y}$ equals $\mathbf{w} \circ (\overline{\mathbf{w}} * (\mathbf{w} \circ \mathbf{c}))$ where $\circ$ denotes the Hadamard product.

The Toeplitz product can be computed as in the previous section. We need to compute three DFTs over a vector of at least $2n-1$ elements. And we can pad the vectors with zeros to get a new vector of a convenient length. For $n = 31$, we have that $2n - 1 = 61$, which can be padded out to say $2^6 = 64$.

```
function y = bluestein(x)
n = length(x);
w = exp((1i*pi/n)*((0:n-1).^2))';
y = w.*fasttoeplitz(conj(w),conj(w(2:end)),w.*x);
```

Rader factorization

The Bluestein factorization rewrites the DFT as a Toeplitz matrix which requires the evaluation of three DFT of at least twice the size of the original. Rader factorization permutes the rows and the columns of the DFT matrix to get a matrix

$$\begin{bmatrix} 1 & \boldsymbol{\xi}^T \\ \boldsymbol{\xi} & \mathbf{C}_{n-1} \end{bmatrix}$$

where $\mathbf{C}_{n-1}$ is an $(n-1) \times (n-1)$ circulant matrix and $\boldsymbol{\xi}$ is a vector of ones. Multiplication by the circulant matrix $\mathbf{C}_{n-1}$ requires three DFTs of size at least $n-1$. Unlike Bluestein factorization, Rader factorization only works when n is a prime number.

Let's see how we can do this. The fact that there is such a permutation is result of number theory. We take the following theorems as given and simply demonstrate them with an example.

Theorem 19. *If n is prime, then there is an integer r with $2 \leq r \leq n-1$ such that*

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

The integer r is called a primitive root (or generator) modulo n.

Theorem 20. *There exists and integer s such that $sr \pmod n = 1$ for prime n. We call s the inverse of r and denote it by r^{-1}.*

Theorem 21. *If r is a primitive root, then so is r^{-1}.*

Example. Consider $n = 13$. Then $r = 2$ is a primitive root modulo 13 because

$$2^j \pmod{13} = \{2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1\} \quad \text{for} \quad j = 1, \ldots, 12.$$

The integers $r = 6, 7, 11$ are also primitive roots modulo 13. The integer 7 is the inverse of 2 because $2 \cdot 7 = 14 = 1 \pmod{13}$. ◀

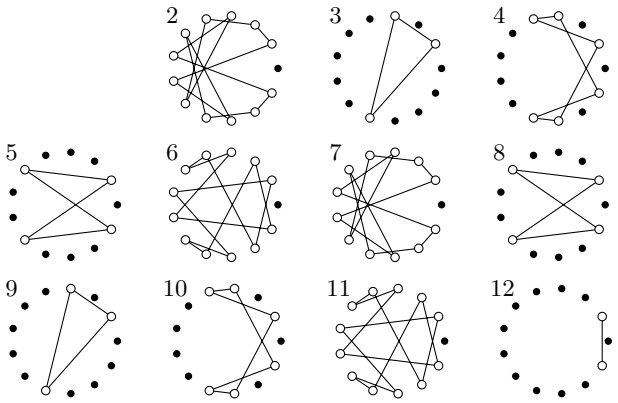


Figure 6.4: Primitive roots modulo 13 are 2, 6, 7, and 11.

Suppose that we have the discrete Fourier transform

$$y_k = \sum_{j=0}^{n-1} c_j \omega_n^{jk}$$

with n prime. By Theorems 19–21, we can take the permutation $j \to r^j$ and $k \to r^{-k}$ for $j, k = 1, \ldots, n-1$ giving us

$$y_0 = \sum_{j=0}^{n-1} c_j$$

$$y_{(r^{-k})} = c_0 + \sum_{j=1}^{n-1} c_{(r^j)} \omega_n^{(r^{j-k})}.$$

So,

$$\mathbf{y} = \mathbf{F}_n \mathbf{c} = \mathbf{P}_{-r}^{-1} \begin{bmatrix} 1 & \mathbf{e}^T \\ \mathbf{e} & \mathbf{C}_{n-1} \end{bmatrix} \mathbf{P}_r \mathbf{c}$$

where $\mathbf{C}_{n-1}$ is the circulant matrix whose first row is given

$$\begin{bmatrix} 1 & \omega_n^r & \omega_n^{r^2} & \cdots & \omega_n^{r^{n-2}} \end{bmatrix}$$

and where $\mathbf{P}_r$ is the permutation matrix determined by

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

with primitive root r. While $n-1$ is even and hence it can be factored, it may be advantageous to pad the matrix $\mathbf{C}$ with zeros to make the evaluation of the three DFTs more efficient.

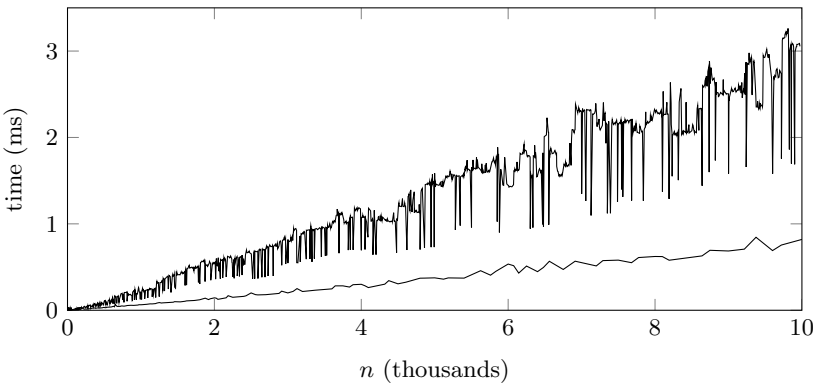


Figure 6.5: Computation time for DFTs of vectors of length n using MATLAB's `fft`. The lower curve shows composite numbers of primes 2, 3, and 5 such as $720 = 2^4 \cdot 3^2 \cdot 5$. The upper curve shows prime numbers.

6.5 The FFT in MATLAB

Ⓜ The MATLAB function `fft` implements the Fastest Fourier Transform in the West (FFTW) library. The inverse FFT can be computed using `ifft` and multidimensional FFTs can be computed using `fftn` and `ifftn`.

The FFTW library has several FFT algorithms and chooses the fastest based on heuristics and trial. The first time that the library is called it may try several different FFT algorithms for the same problem and thereafter use the fastest. The FFTW library decomposes the problem using the composite Cooley–Tukey algorithm recursively until the problem can be solved using fixed-size codelets, which use split-radix, Cooley–Tukey, and a prime factor algorithm. If n is a prime number the FFTW library decomposes the problem using Rader decomposition and then uses the Cooley–Tukey algorithm to compute the three $(n-1)$-point DFTs. FFTW saves the fastest approach for a given array size as the so-called *wisdom* to be used when an FFT is needed for another array of the same size. If a problem requires several repeated FFTs, it may be advantageous for FFTW to try all of the possible algorithms initially to see which is fastest. Other times (for example, if you only need to perform the FFT once), it may be better to have FFTW estimate the fastest approach.

Ⓜ The MATLAB command `fftw` can be used to tune the FFTW parameters. By default, MATLAB uses `estimate` to tune FFTW.

Note that MATLAB defines the DFT as $F_j^{(1)} = \sum_{j=0}^{n-1} f_{j+1}\omega_n^{jk}$ while

some problems may define the DFT as $F_j^{(2)} = \frac{1}{n}\sum_{j=-n/2+1}^{n/2} f_j \omega_n^{jk}$. The zero-frequency component of $F^{(1)}$ is given by $F_0^{(1)}$, whereas the zero-frequency component of $F^{(2)}$ is given by $F_{n/2}^{(2)}$.

Ⓜ The MATLAB functions `fftshift` and `ifftshift` shift the zero frequency component to the center of the array.

6.6 Applications

A typical use of the DFT is to construct the frequency components of a time-varying or space-varying function such as a sound wave. There are several other important uses for the DFT. We will look at some of them in this section.

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 $[0,1]\times[0,1]\times[0,1]$ from Problem 5.3 in previous chapter. We solved the problem $\mathbf{Au} = \mathbf{f}$ by 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

$$\mathbf{D} = \frac{1}{(\Delta x)^2}\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}.$$

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. Recall that the eigenvalues of $\mathbf{D}$ are

$$\lambda_k = \left(2 - 2\cos\frac{k\pi}{n+1}\right)/(\Delta x)^2 \quad \text{with} \quad k = 1,\ldots,n$$

and the eigenvectors are given by

$$\mathbf{x}_k = \left[\sin\left(\frac{k\pi}{n+1}\right) \quad \sin\left(\frac{2k\pi}{n+1}\right) \quad \sin\left(\frac{3k\pi}{n+1}\right) \quad \cdots \quad \sin\left(\frac{nk\pi}{n+1}\right)\right]^T.$$

Consider two arbitrary $n \times n$ matrices $\mathbf{M}$ and $\mathbf{N}$ and the $n \times n$ identity matrix $\mathbf{I}$. If $\mathbf{M}\mathbf{x}_i = \lambda_i \mathbf{x}_i$ and $\mathbf{M}\mathbf{y}_i = \mu_i \mathbf{y}_i$, then

$$(\mathbf{M} \otimes \mathbf{N})(\mathbf{x}_i \otimes \mathbf{y}_i) = \lambda_i \mu_j (\mathbf{x}_i \otimes \mathbf{y}_i).$$

Furthermore, the matrix $(\mathbf{I} \otimes \mathbf{M}) + (\mathbf{N} \otimes \mathbf{I})$ has eigenvalues $\{\lambda_i + \lambda_j\}$ with $1 \le i, j \le n$. From this it follows that the eigenvalues of $\mathbf{A}$ are

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

and the i, j, k-component of eigenvalues $\mathbf{x}_{\zeta\xi\eta}$ is

$$\sin\left(\frac{i\zeta\pi}{n+1}\right)\sin\left(\frac{j\xi\pi}{n+1}\right)\sin\left(\frac{k\eta\pi}{n+1}\right)$$

with integers $1 \le \zeta, \xi, \eta \le n$. From the diagonalization of the operator $\mathbf{A}$

$$\mathbf{A}\mathbf{u} = \mathbf{f} \quad \Rightarrow \quad \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}\mathbf{u} = \mathbf{f}$$

we have the solution

$$\mathbf{u} = \mathbf{A}^{-1}\mathbf{f} \quad \Rightarrow \quad \mathbf{u} = \mathbf{S}\mathbf{\Lambda}^{-1}\mathbf{S}^{-1}\mathbf{f}.$$

The operator $\mathbf{S}$ is called the discrete sine transform (DST). The DST of $\mathbf{f}$ in component form is

$$\hat{f}_{lmn} = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} f_{ijk}\sin\left(\frac{i\zeta\pi}{n+1}\right)\sin\left(\frac{j\xi\pi}{n+1}\right)\sin\left(\frac{k\eta\pi}{n+1}\right). \tag{6.5}$$

We can use the Fast Fourier Transform to compute the discrete sine transform by first noting that

$$\sin\frac{k\pi}{n} = \frac{\mathrm{e}^{\mathrm{i}\pi/n} - \mathrm{e}^{-\mathrm{i}k\pi/n}}{2\mathrm{i}}$$

and extending the input function (which is zero at the endpoints) to make a periodic function. Because the indices of the DFT run from 0 to $n-1$ whereas the indices of the DST run from 1 to n, we'll need to add a zero to the input vector $\mathbf{f}$ to account for the constant (zero frequency) term. So, the discrete sine transform of

$$(f_1, f_2, \ldots, f_n)$$

can be computed by taking components 2 through $N+1$ of the disrete Fourier transform of

$$(0, f_1, f_2, \ldots, f_n, 0, -f_n, \ldots, -f_2, -f_1).$$

Because we now need to compute a $(2n+2)$-point FFT, so it is fastest to take $n = 2^p - 1$ for some p. The DST $\mathbf{F}$ is a real symmetric, scaled orthogonal matrix, so its inverse

$$\mathbf{F}^{-1} = \frac{2}{n+1}\mathbf{F}.$$

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. The following MATLAB code solves Problem 5.3 using a fast Poisson solver. The function `dft3` computes the three-dimensional DST (6.5) by computing the DST over each dimension. The function `dftx3` computes a DST in one-dimension for a three-dimensional array. The `real` command in the function `dftx3` is only needed to remove the round-off error from the FFT. The program takes less than a third of a second to run for $n = 50$.

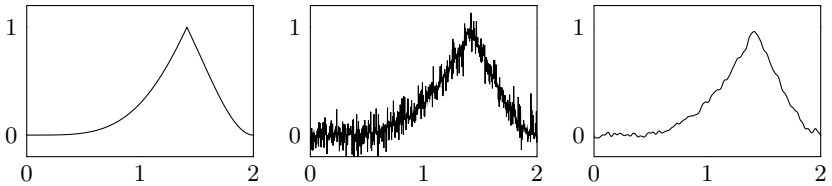
```
function fastpoissonsolver
n = 50;
x = (1:n)'/(n+1); dx = 1/(n+1);
[x,y,z] = meshgrid(x);
ei = @(k) 2 - 2*cos(k*pi/(n+1));
[ix,iy,iz] = meshgrid(1:n);
lambda = (ei(ix)+ei(iy)+ei(iz))./(dx^2);
f = 2*((x-x.^2).*(y-y.^2) + ...
    (x-x.^2).*(z-z.^2)+(y-y.^2).*(z-z.^2));
u = dst3(dst3(f)./lambda);

function a = dst3(a)
a = dstx3(shiftdim(a,1));
a = dstx3(shiftdim(a,1));
a = dstx3(shiftdim(a,1));

function a = dstx3(a)
n = size(a);
y = [zeros(1,n(2),n(3));a;zeros(1,n(2),n(3));-a(end:-1:1,:,:)];
y = fft(y);
a = real(0.5i*y(2:n(1)+1,:,:)*(sqrt(2/(n(1)+1))));
```

Typically, it is much more convenient to operate on the basis functions directly to solve a partial differential equation instead of first discretizing in space. We'll discuss this approach in the next section.

Data filtering

Suppose that we have noisy data. High frequencies (high derivatives) can be filtered and functions can be regularized by using a convolution

with a smooth function such as a Gaussian distribution. A convolution requires $O(n^2)$ operations to apply directly, but only $O(n)$ operations if performed in the Fourier domain. Of course, it will require two $O(n \log n)$ operations to put it in the Fourier domain and bring it back, but when n is large, the overhead is relatively small. For example, noise was added to the original data on the left below to get the center figure. A Gaussian distribution was used as a kernel of the convolution to filter the noise to get the figure on the right.



Lossy data compression

A photo is often saved using jpeg compression, which divides the image into 8×8 blocks and performs discrete cosine transforms (DCT) on each of the blocks. Similarly, audio can be saved using a DCT compression of blocks as an mp3.

The Fourier transform of a smooth function decays rapidly in the frequency domain. So, for significantly smooth data, high frequency information can be discarded without introducing substantial error. Since the DFT is a scaled unitary matrix, the l^2-norm and the Frobenius norm are preserved by the DFT. This means that the error $\|\mathbf{e}\|_2 = \|\mathbf{Fe}\|_2$ and $\|\mathbf{e}\|_F = \|\mathbf{Fe}\|_F$. By clipping or zeroing out the high frequencies in the Fourier transformed data, we can still keep relevant information. When the data is not smooth, significant information resides in the higher frequencies and discarding it can result in substantial error. The so-called Gibb's phenomenon associated lossy compression of data with discontinuities manifests itself as ringing artifacts at edges.

See Figure 6.6 on the following page. A 384×256-pixel raw image on the top left is compressed to 25%, 10% and 1% of its original image size by taking its two-dimensional Fourier transform and discarding high frequency information. Because the original image has high-contrast edges, the reconstructed images (top row) are affected by the Gibbs phenomenon. The difference between the upper-right portion of original image and reconstructed images is shown in the bottom row.

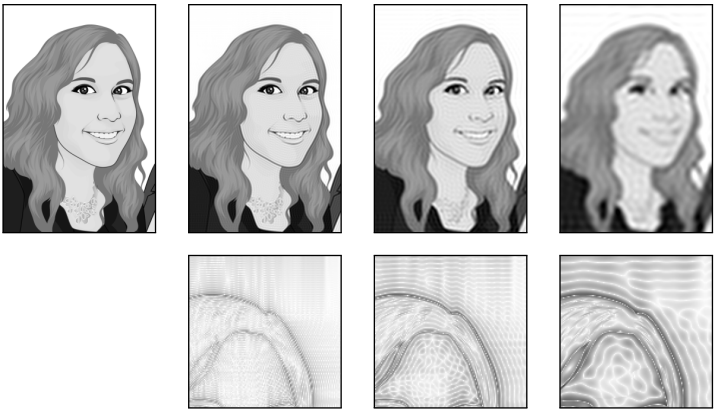


Figure 6.6: Lossy image compression by zeroing high Fourier frequencies

Exercises

✍ 6.1. Modify the MATLAB program `fftx` on page 131 to implement an FFT algorithm suitable for powers of 3. Test your program with MATLAB's `fft` program on a matrix of size $3^4 = 81$.

6.2. Show that permutation matrix

$$\mathbf{P} = (\mathbf{I}_1 \otimes \mathbf{P}_n)(\mathbf{I}_2 \otimes \mathbf{P}_{n/2}) \cdots (\mathbf{I}_{n/4} \otimes \mathbf{P}_4).$$

used in the radix-2 decomposition of the DFT in (6.3) can be implemented by simply reversing the binary index. For example, for $n = 8$

$$\mathbf{P} \begin{bmatrix} a_{0=000} \\ a_{1=001} \\ a_{2=010} \\ a_{3=011} \\ a_{4=100} \\ a_{5=101} \\ a_{6=110} \\ a_{7=111} \end{bmatrix} = \begin{bmatrix} a_{0=000} \\ a_{4=100} \\ a_{2=010} \\ a_{6=110} \\ a_{1=001} \\ a_{5=101} \\ a_{3=011} \\ a_{7=111} \end{bmatrix}.$$

✍ 6.3. *Fast multiplication of large integers.* Consider two polynomials

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

By taking $x = 10$, the $p(x)$ and $q(x)$ return the decimal numbers $p(10)$ and $q(10)$, respectively.

By choosing the basis $\{x^{2n}, \ldots, x, 1\}$, the polynomials can represented by coefficient vectors

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

where $\mathbf{p}$ and $\mathbf{q}$ are padded with n zeros. The products $p(x)q(x)$ and $p(10)q(10)$ are represented by

$$(p_0q_0, p_0q_1 + p_1q_0, \ldots, p_{n-1}q_n + p_nq_{n-1}, p_0q_0) \tag{6.6}$$

where the jth element is given by

$$\sum_{i=0}^{j} p_i q_{j-i}.$$

For example, for $n = 2$

$$\begin{bmatrix} q_0 & & \\ q_1 & q_0 & \\ q_2 & q_1 & q_0 \\ & q_2 & q_1 \\ & & q_2 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix}.$$

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 jth 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 the taking the inverse discrete Fourier transform to transform them back.

4. Carrying the digits so that each element only contains a single digit. For example, multiplying 35 and 19 with the array representations $(5, 3, 0, 0, 0)$ and $(9, 1, 0, 0, 0)$ results in $(45, 32, 3, 0, 0)$, which must be corrected to give $(5, 6, 6, 0, 0)$ or 665.

The benefit of this approach is that we can reduce the number of operations from $O(n^2)$ to $O(n \log n)$.

Write and implement an algorithm that uses FFTs to compute the product of the primes

$$\begin{aligned} &3490529510847650949147849619903 8\ldots \\ &\qquad \ldots 9813341776463849338784399082057 7 \end{aligned}$$

and

$$\begin{aligned} &3276913299326670954996198819083 44\ldots \\ &\qquad \ldots 6141317764296799294253979828853 3 \end{aligned}$$

Note: 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. Hint: MATLAB typically works with floating point numbers, so you can't easily input a large integer directly. Instead, you 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]`.

Numerical Methods for Analysis

CHAPTER 7

Preliminaries

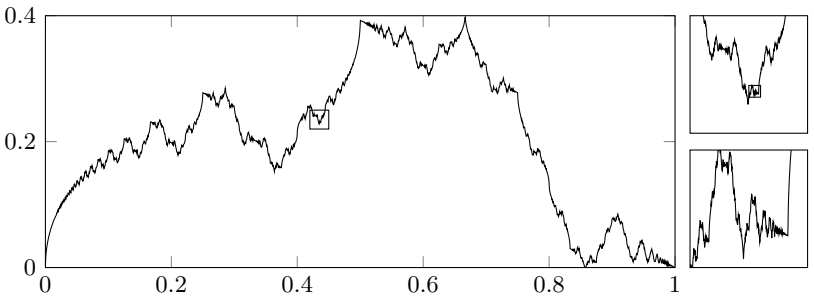
This chapter provides a short introduction to several key ideas that we will use and grow throughout the course. Specifically, we examine how functions, problems, methods, and computational error impact getting a good numerical solution.

7.1 Well-behaved functions

Perhaps the easiest way to introduce well-behaved functions is with examples of functions that are not so well-behaved.

Example. Weierstrass functions are examples mathematical monsters. They are everywhere continuous and almost nowhere differentiable. The following Weierstrass function was proposed by Riemann in the 1860s

$$\sum_{n=1}^{\infty} \frac{\sin(\pi n^2 x)}{\pi n^2}.$$



It 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 $-1/2$ (Hardy [1916], Thim [2003]). It is also one of the earliest examples of a fractal, a set with repeating, self-similarity at every scale and having non-integer dimensions. ◀

Example. We can classify arithmetic operations using a hyperoperation sequence. The simplest arithmetic operation is succession, $x+1$, which gives us the number that comes after x. Next, addition of two numbers $x+y$, which is the succession of x applied y times. Then, multiplication of two numbers xy, which is x added to itself y times. Followed by exponentiation of two numbers x^y, which is x multiplied by 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 $x \uparrow\uparrow y$. Tetration can create big numbers quickly. For example, computing ${}^x x$ we have ${}^1 1 = 1$, ${}^2 2 = 16$, and ${}^3 3$ is a number with over three trillion digits. ◀

Mathematically almost all functions are pathological. But most functions that are important for applications are nice. To help describe the limiting behavior of nice functions we have Landau notation.

Landau notation

Landau notation ("big-O" and "little-o") is used to describe the limiting behavior of a function:

$$\begin{aligned} f(x) \in O(g(x)) \quad &\text{if} \quad \lim_{x\to\infty} f(x)/g(x) \text{ is bounded.} \\ f(x) \in o(g(x)) \quad &\text{if} \quad \lim_{x\to\infty} f(x)/g(x) = 0. \end{aligned}$$

One often says that $f(x)$ is $O(g(x))$ or $f(x) = O(g(x))$ 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 discuss the complexity of an algorithm. Gaussian elimination, used to solve a system of n linear equations, requires roughly $\frac{2}{3}n^3 + 2n^2$ operations (additions and multiplications). So, Gaussian elimination is $O(n^3)$.

Big-O and little-o can also be used to simply notation of infinitesimal asymptotics. In this case,

$$\begin{aligned} f(x) \in O(g(x)) \quad &\text{if} \quad \lim_{x\to 0} f(x)/g(x) \text{ is bounded.} \\ f(x) \in o(g(x)) \quad &\text{if} \quad \lim_{x\to 0} f(x)/g(x) = 0. \end{aligned}$$

For example, $e^x = 1 + x + O(x^2)$ and $e^x = 1 + x + o(x)$ when $x \ll 1$. We often use Big-O notation to discuss truncation error of finite-difference approximation to differentiation. For a small offset h, the Taylor series approximation

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

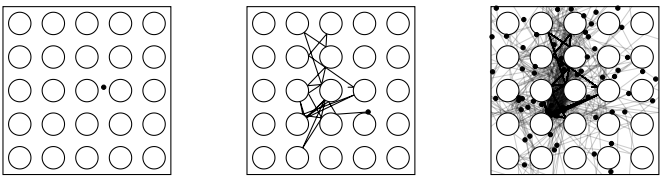
From this it follows that

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

where we've taken $O(h^2)/h = O(h)$. So, forward differencing provides an $O(h)$ approximation (or first-order approximation) to the derivative.

7.2 Well-posed problems

Example. Just as there are ill-behaved functions, there are also ill-behaved problems. Mathematician Nick Trefethen once offered $100 as a prize to solving ten numerical mathematics problems to ten significant digits (Trefethen [2002]). One of them was "A photon moving at speed 1 in the xy-plane starts at t = 0 at (x, y) = (0.5, 0.1) heading due east. Around every integer lattice point (i, j) in the plane, a circular mirror of radius 1/3 has been erected. How far from the origin is the photon at t = 10?" The solution can be found analytically using some geometry of intersection of lines and circles. At each intersection we need to evaluate a square root. And, with each numerical evaluation round-off error is added into the solution, compounding over and over again, losing about a unit of precision with every reflection. While the original problem asked for the solution at $t = 10$ to 100-digits of accuracy, we could easily think about what happens at $t = 20$. The figure on the left below shows the starting position.



The figure in the middle, shows the solution for a photon at $t = 20$. Instead of just one photon, let's solve for 100 photons each with a very slight initial velocity angle discrepancy 10^{-16} (machine floating-point precision). To put it in perspective, if these photons left from a point on the

sun, they would all land on earth in an area covered by period at the end of this sentence. The figure on the right shows the solution. It's hard to tell which is the right one and whether our original solution in the middle was in fact the correct one. ◀

In this section we'll consider what makes a problem a "nice" problem. Consider the problem:

$$F(y, x) = 0 \text{ for some input data } x \text{ and some output data } y. \tag{7.1}$$

F might be a function of the variable y and $x = \{c_n, c_{n-1}, \cdots, c_0\}$ may be a vector of coefficients of F:

$$c_n y^n + c_{n-1} y^{n-1} + \cdots + c_1 y + c_0 = 0.$$

Or, $F(y, x) = 0$ could be a differential equation where y is some function and x is the initial condition or the boundary condition

$$y' - 5y = 0 \quad \text{with} \quad y(0) = x.$$

Or, $F(y, x) = 0$ could be a definite integral where x is a function and $y \in \mathbb{R}$:

$$y = \int_0^1 \sin(t)\, \mathrm{d}t$$

In a simple (thousand-word) picture, our problem is the following:

$$\boxed{\begin{array}{c}\text{input}\\ \text{data}\end{array}} \longrightarrow \boxed{\begin{array}{c}\text{black}\\ \text{box}\end{array}} \longrightarrow \boxed{\begin{array}{c}\text{output}\\ \text{data}\end{array}}$$

We can classify the problem in three ways

1. *Direct problem*: F and x are given and y is unknown. We know the black box and the input—determine the output.

2. *Inverse problem*: F and y are given and x is unknown. We know the black box and the output—determine the input.

3. *Identification problem*: y and x are given and F is unknown. We know the input and the output—determine what is happening inside the black box.

A direct problem is typically easier to solve than an inverse problem, which itself is easier 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 black box is so sensitive to change that we cannot replicate the output if the input changes by the even the slightest amount. In general, how do we know whether a problem has a meaningful solution? To answer this question, French mathematician Jacques Hadamard introduced the notion of mathematical well-posedness in 1923. A problem is called *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 fails to hold, the problem is called *ill-posed*.

Condition

Even if a problem is well-posed, the output may still be sensitive to the input data. Let's examine the idea of stability. We say that the output y *depends continuously* on the input x if "small" perturbations in the input yields "small" changes in the output. That is to say that 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. Continuous dependence says that there is some η such that

$$\text{if } \|\delta x\| \leq \eta, \text{ then } \|\delta y\| \leq \kappa \|\delta x\|$$

for some κ and some norm $\|\cdot\|$.

We call κ the condition number. It is simply the ratio of the change in the output data to the change in the input data. We define the *absolute condition number*

$$\kappa_{\text{abs}}(x) = \sup \left\{ \frac{\|\delta y\|}{\|\delta x\|} \right\}.$$

If $y \neq 0$ and $x \neq 0$, we define the *relative condition number*

$$\kappa(x) = \sup \left\{ \frac{\|\delta y\|/\|y\|}{\|\delta x\|/\|x\|} \right\}.$$

A problem is *ill-conditioned* if $\kappa(x)$ is large for any x and *well-conditioned* if it is small for all x.

Condition of the 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

$$f(x+\delta x) = f(x) + f'(x)\delta x + o(\|\delta x\|) \quad \text{for} \quad \delta x \to 0$$

where f' is the Jacobian matrix. Hence,

$$\delta y = f'(x)\delta x + o(\|\delta x\|).$$

So, the absolute condition number is

$$\kappa_{\text{abs}}(x) = \|f'(x)\| \quad \text{for} \quad \delta x \to 0.$$

Similarly, from $\delta y \approx f'(x)\delta x$ and $y = f(x)$, we get that the relative condition number is

$$\kappa(x) = \|f'(x)\| \frac{\|x\|}{\|f(x)\|} \quad \text{for} \quad \delta x \to 0.$$

Example. (Addition/subtraction) Consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by $f(a,b) = a+b$. The gradient of $f(a,b)$ is

$$f' = \left(\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b}\right) = (1,1).$$

In the 1-norm ($\|\mathbf{v}\|_1 = \sum_{i=1}^n |v_i|$):

$$\kappa(x) \approx \|f'(x)\| \frac{\|x\|}{\|f(x)\|} = \|(1,1)\| \frac{\|(a,b)^T\|}{\|a+b\|} = \frac{|a|+|b|}{|a+b|}$$

So, if a and b have the same sign then $\kappa(x) \approx 1$ and addition is a well-conditioned operator. But if a and b have opposite signs and are (relatively) close, then $\kappa \gg 1$. Subtraction can lead to cancellation of significant digits. For example, $\kappa(x) \approx 1$ for $a = 1000$ and $b = 999$, but $\kappa(x) \approx 2000$ but $a = 1000$ and $b = -999$. ◀

Example. (Quadratic Equation) The solutions to the quadratic equation $x^2 - 2px + 1 = 0$ are

$$f_\pm(p) = x_\pm = p \pm \sqrt{p^2-1}$$

where $f : \mathbb{R} \to \mathbb{R}^2$. The Jacobian matrix is given by

$$(f'_+(p), f'_-(p))^T = \left(1 + \frac{p}{\sqrt{p^2-1}}, 1 - \frac{p}{\sqrt{p^2-1}}\right)^T$$

Using the l^2-norm, we have that

$$\kappa(p) \approx \|f'(p)\|_2 \frac{\|p\|_2}{\|f(p)\|_2} = \sqrt{2 + 2\frac{p^2}{p^2-1}} \frac{|p|}{\sqrt{4p^2-2}} = \frac{|p|}{\sqrt{p^2-1}}$$

For large enough p (greater than 2, let's say) the condition number is small. But if $p \approx 1$, the solution is ill-conditioned.

The problem is not ill-posed because we can rewrite the solution in a well-conditioned way:

$$f = (f_+(t), f_-(t)) = (t, t^{-1}) \quad \text{where} \quad t(p) = p + \sqrt{p^2-1}$$

In this case, the Jabobian is

$$f' = (f'_+(t), f'_-(t))^T = \left(1, -t^{-2}\right)^T$$

Using the l^2-norm, we have that

$$\kappa(p) \approx \|f'(t)\|_2 \frac{\|t\|_2}{\|f(t)\|_2} = \sqrt{1+t^{-4}} \frac{|t|}{\sqrt{t^2+t^{-2}}} = 1$$

So, the solution is well-conditioned. ◀

Condition of the inverse problem

Consider $y = \varphi(x)$ where $\varphi : \mathbb{R} \to \mathbb{R}$ and φ is invertible in a neighborhood of x. Suppose that we know the output y and want to determine the input x. In this case, the implicit function theorem says that

$$x = \varphi^{-1}(y)$$

and $(\varphi^{-1})'(y) = [\varphi'(x)]^{-1}$. The absolute condition number of the problem $x = \varphi^{-1}(y)$ is

$$\kappa_{\text{abs}} \approx \|(\varphi^{-1})'(x)\| = \|\varphi'(x)^{-1}\|$$

and the relative condition number is

$$\kappa \approx \|\varphi'(x)^{-1}\| \frac{\|\varphi(x)\|}{\|x\|}$$

for $y \neq 0$. The problem is ill-posed if $\varphi(x)$ is a double root. It is ill-conditioned if $\varphi'(x)$ is small and well-conditioned it $\varphi'(x)$ is large.

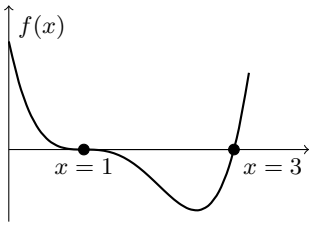


Figure 7.1: $f(x) = (x-1)^3(x-3)$ has zeros at $x_1 = 1$ an $x_2 = 3$. The condition number of f in a neighborhood of x_1 is $\kappa_{\text{abs}}(x) = \infty$. The condition number of f in a neighborhood of x_2 is 8. The problem of finding the zeros of f is well-conditioned at x_2 and ill-conditioned at x_1.

7.3 Well-posed methods

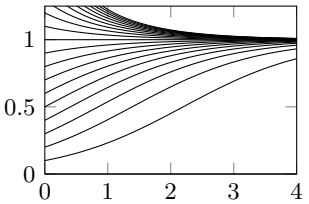
Just as we saw pathological functions and pathological problems, there are also pathological solutions, even to well-behaved problems.

Example. The logistic differential equation is a simple model of population dynamics $s(t)$ with a limited carrying capacity:

$$\frac{\mathrm{d}s}{\mathrm{d}t} = s(1-s). \tag{7.2}$$

When the population s is small, $\mathrm{d}s/\mathrm{d}t \approx s$ and population growth is proportional to the population. As the population approaches and exceeds the carrying capacity $s = 1$, starvation occurs causing negative population growth. The solution to this differential equation is

$$s(t) = \left[1 - \left(1 - \frac{1}{s(0)}\right)\mathrm{e}^{-t}\right]^{-1}$$



The solution is well-behaved and $s(t) \to 1$ as $t \to \infty$, reaching an equilibrium between reproduction and starvation.

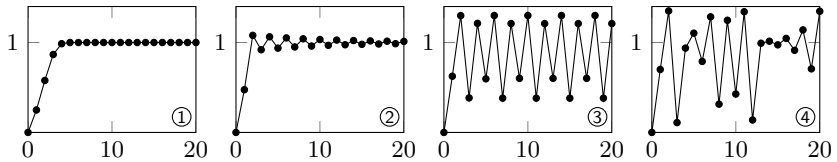
A simple numerical method of solving the differential equation is to use the forward Euler approximation to the derivative and then iteratively solve the resulting difference equation. In this case, we have

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

where Δt is the time step between subsequent evolutions of the solution $s^{(k)}$. By rescaling $s^{(k)} \mapsto (\Delta t)/(1+\Delta t)x^{(k)}$ and solving for $x^{(k+1)}$, we get an equivalent equation called the logistic map:

$$x^{(k+1)} = rx^{(k)}(1 - x^{(k)}) \quad \text{where} \quad r = (1 + \Delta t). \tag{7.4}$$

The equation has the equilibrium solution $x = 1 - r^{-1}$. Let's examine the solution to (7.3) for $x(0) = 1/4$ with Δt given by (1) 1, (2) 1.9, (3) 2.5, and (4) 2.8:



When $\Delta t = 1$, the solution to (7.3) closely matches the solution to (7.2). When $\Delta t = 1.9$, the numerical solution exhibits transient oscillations and converges to equilibrium solution to (7.2). When $\Delta t = 2.5$, the solution converges to a limit cycle with period 4. As Δt increases, so does the periodicity of the limit cycle. When $\Delta t = 2.8$, the numerical solution is completely chaotic. As Δt continues to increase, eventually the solution becomes unstable and blows up. ◀

Again, consider the well-posed problem (7.1):

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

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 and y_n as the approximate solution.

$$\boxed{\text{approximate input}} \longrightarrow \boxed{\text{numerical method}} \longrightarrow \boxed{\text{approximate output}}$$

We say that the numerical method is *well-posed* if for any n

1. there is a solution y_n corresponding to the input x_n; (existence)
2. the computation of y_n is unique for each input x_n; (uniqueness)
3. y_n depends continuously on the x_n. (stability)

Consistency

Assume that the approximate input $x_n \to x$ as $n \to \infty$. We say that the numerical method (7.5) is *consistent* if the numerical method approaches the original problem as $n \to \infty$. In this case it can be made as close to the original problem as we want (by refining mesh size, for example). Specifically, for every $\varepsilon > 0$ there is an N such that

$$\|F_n(y_n, x_n) - F(y, x)\| < \varepsilon \quad \text{for all} \quad n > N.$$

Stability

Suppose we perturb the input data by δx_n and let δy_n be the corresponding perturbation in the solution determined by the numerical method

$$F_n(y_n + \delta y_n, x_n + \delta x_n) = 0.$$

We say that the method is *numerically stable* if a small error or perturbation in the approximate input data produces a "small" perturbation in the output. That is,

$$\text{for every } \delta x_n \quad \text{with} \quad \|\delta x_n\| \le \eta, \qquad \|\delta y_n\| \le \kappa \|\delta x_n\| \quad \text{for some } \kappa.$$

As before we define the absolute condition number as

$$\kappa_{\text{abs},n}(x) = \sup \left\{ \frac{\|\delta y_n\|}{\|\delta x_n\|} \right\}.$$

And, if $y_n \neq 0$ and $x_n \neq 0$, we define the relative condition number as

$$\kappa_n(x) = \sup \left\{ \frac{\|\delta y_n\| / \|y_n\|}{\|\delta x_n\| / \|x_n\|} \right\}.$$

Numerical convergence

One would hope that for any good numerical method, the approximate solution y_n is a better approximation to y as n gets larger. We say that a numerical method is *convergent* if the numerical solution can be made as close to the analytical solution as we want (by refining the mesh size and limiting round-off error, for example). That is, for every $\varepsilon > 0$ there is a $\delta > 0$ such that for all sufficiently large n,

$$\|y(x) - y_n(x + \delta x_n)\| \le \varepsilon \quad \text{whenever} \quad \|\delta x_n\| < \delta.$$

It should come as no surprise that consistency and stability imply convergence. The following theorem, known as the Lax–Richtmyer Equivalence Theorem, is so important that it is often called the Fundamental Theorem of Numerical Analysis.

Theorem 22. *A consistent method is stable if and only if it is convergent.*

Proof. This proof follows the proof in Quarteroni et al. [2007]. 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 F_n is differentiable and that the Jacobian $\partial F_n / \partial y$ is invertible and bounded for all

n. (The assumption of differentiability can be weakened to being Lipschitz by altering the proof.) Taylor series expansion with respect to x_n gives

$$F_n(y(x), x) = F_n(y_n(x), x) + \left.\frac{\partial F_n}{\partial y}\right|_{\xi, x} (y(x) - y_n(x))$$

for some ξ in the neighborhood of $y(x)$ and $y_n(x)$. Therefore,

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

We can replace $F_n(y_n(x), x)$ with $F(y(x), x)$ in the expression above, because both of them are zero. So,

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

It follows that

$$\|y(x) - y_n(x)\| \leq M \, \|F_n(y(x), x) - F(y(x), x)\|$$

where $M = \|(\partial F_n/\partial y)^{-1}\|$. Because the method is consistent, for every $\varepsilon > 0$ there is an N such that

$$\|F_n(y_n, x_n) - F(y, x)\| < \varepsilon \quad \text{for all} \quad n > N.$$

So,

$$\|y(x) - y_n(x)\| \leq M\varepsilon.$$

First, we will show convergence implies stability. If numerical method is consistence and convergent, then

$$\begin{aligned}
\|\delta y_n\| &= \|y_n(x) - y_n(x + \delta x_n)\| \\
&= \|y_n(x) - y(x) + y(x) - y_n(x + \delta x_n)\| \\
&\leq \|y_n(x) - y(x)\| + \|y(x) - y_n(x + \delta x_n)\| \\
&\leq \underbrace{M\varepsilon}_{\text{consistency}} + \underbrace{\varepsilon'}_{\text{convergence}}
\end{aligned}$$

By choosing $\varepsilon, \varepsilon' \leq \|\delta x_n\|$, then

$$\|\delta y_n\| \leq (M + 1)\|\delta x_n\|.$$

So, the method is stable. Now, we show stability implies convergence.

$$\begin{aligned}
\|y(x) - y_n(x + \delta x_n)\| &= \|y(x) - y_n(x) + y_n(x) - y_n(x + \delta x_n)\| \\
&\leq \|y(x) - y_n(x)\| + \|y_n(x) - y_n(x + \delta x_n)\| \\
&\leq \underbrace{M\varepsilon}_{\text{consistency}} + \underbrace{\kappa\|\delta x_n\|}_{\text{stability}}
\end{aligned}$$

By choosing $\|\delta x_n\| \leq \varepsilon/\kappa$, then

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

so the method is convergent. □

Summary

Problem	$F(y, x) = 0$
Numerical method	$F_n(y_n, x_n) = 0$
Consistency	$F_n(y, x) \to F(y, x) = 0$ as $n \to \infty$
Stability	If $\|\delta x_n\| \leq \eta$, then $\|\delta y_n\| \leq \kappa\|\delta x_n\|$ for some κ.
Convergence	For any ε there are η and N such that if $n > N$ and $\|\delta x_n\| \leq \eta$, then $\|y(x) - y_n(x + \delta x_n)\| \leq \varepsilon$.
Lax–Richtmyer equivalence	Consistency & stability $\to$ convergence

7.4 Floating-point arithmetic

Most modern programming languages support several numerical data types: 64-bit and 32-bit floating-point numbers for real (and complex) data; 64-bit, 32-bit and 16-bit integers and (unsigned integers) for whole numbers; 8-bit characters for ASCII letters, and 1-bit logicals for "true" and "false" information. A complex number is typically stored as an array of two floating-point numbers. Other data types exist including arbitrary-precision arithmetic (also called multiple-precision and bignum), which is often used in cryptography and symbolic arithmetic used by systems like Mathematica and Maple.

It is important to emphasize that you can't mix data types. You must convert and keep track of what data type you are using. For example, in C `4.0/5.0 = 0.800` but `4/5 = 0`. In Mathematica , which uses symbolic math and floating-point math, entering `2*Sin[Pi/6] - 1` returns 0 but `2*Sin[Pi/6.0] - 1` returns -1.11022×10^{-16}.

Floating-point arithmetic is crucial in scientific computing, and it is useful to have a basic working understanding of it. 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 and 64 bits (8 bytes) with 1 sign bit, 11 exponent bits and 52 fraction mantissa bits to represent a double-precision number. 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.

s	e	m

s	e	m

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	10000011	00010100000000000000000

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

1	01111110	11000000000000000000000

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

The mantissa is normalized between 0 and 1 and the leading "hidden bit" in the mantissa is implicitly a one. This idea of *normalizing* earns the floating-point system an extra bit of precision.

Note that only combinations of powers of 2 can be expressed exactly. For example, 0.825 can be expressed exactly as $(1/2 + 1/4 + 1/16)$, but 0.1 has no finite binary expression. So, only a finite set of numbers can be represented exactly. All other numbers must be approximated. The *machine epsilon* or *unit roundoff* is the distance between 1 and the next exactly representable number greater than one.

Ⓜ The MATLAB variable `eps` returns the double precision machine epsilon of $2^{-52} \approx 10^{-16}$. The MATLAB function `eps(n)` returns the double-precision round-off error at n. For example, `eps(0)` returns $2^{-1074} \approx 10^{-323}$. Of course, `eps(1)` is the same as `eps`.

IEEE 754 also specifies quadruple-precision numbers using 128 bits—one sign bit, 15 exponent bits, and 112 mantissa bits to given about 33 significant digits in decimal. IEEE 754 even has a specification for 256-bit octuple-precision floating point numbers, but it is rarely implemented, in part because of arbitary-precision software libraries.

Round-off error

Example. (Rump's Catastrophic Cancellation.) Consider the following problem

$$y = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

where $a = 77617$ and $b = 33096$.

The MATLAB code

```
a = 77617; b = 33096;
y = 333.75*b^6+a^2*(11*a^2*b^2-b^6-121*b^4-2)+5.5*b^8+a/(2*b);
```

returns $y \approx -1.181 \times 10^{21}$. Programmed in C (using the gcc compiler), I get 4.386×10^{20}. If I write out the the powers as products (for example, `b^6` as `b*b*b*b*b*b`), I get $y \approx 1.173$. What's going on?

If we rewrite the problem as

$$\begin{aligned} z &= 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) \\ x &= 5.5b^8 \\ y &= z + x + \frac{a}{2b} \end{aligned}$$

then

$$\begin{aligned} z &= -7917111340668961361101134701524942850 \\ x &= +7917111340668961361101134701524942848 \end{aligned}$$

From $z + x = 2$ we have that $y = -\frac{54767}{66192} \approx -0.827396$. Recall that the condition number for addition is

$$\kappa = \frac{|z| + |x|}{|z + x|} \approx 10^{36}.$$

◀

Example. Patriot surface-to-air missiles were used during the Gulf War to intercept Iraqi Scud missiles. In February 1991, a missile battery in Dhahran, Saudi Arabia, failed to track an incoming Scud missile. The Scud missile subsequently hit an army barracks. The Patriot missile system measured time in tenths of seconds. The internal clock was based on a 1970s design which used a 24-bit fixed-point numbers. The decimal 0.1 cannot be exactly represented in binary. The round-off error introduced by a 24-bit register (by keeping only 24 bits) is approximately 9.5×10^{-8}. So, every second the clock lost 9.5×10^{-7} seconds. The Patriot computer had been running for 100 hours, and therefore the clock was off by about 0.34 seconds. A Scud missile travels more that half of a kilometer in this time. Although the system detected an incoming missile, because of the round-off error the Scud missile was outside of the predicted range-gate and the initial detection was believed to be spurious (Blair et al. [1992]).

◀

Underflow, overflow and `NaN`

Because that the number of bits reserved for the exponent of a floating point are limited, floating point arithmetic has an upper bound (closest to infinity) and lower bound (closest to zero). Any computation outside of this bound results in either an overflow producing `inf` or an underflow producing `0`.

Ⓜ The MATLAB command `realmax` returns the largest floating-point number—one bit less than 2^{1024} or about 1.8×10^{308}. For command `realmin` returns the smallest normalized floating-point number—2^{-1022} or about 2.2×10^{-308}.

IEEE 754 uses gradual underflow (versus abrupt underflow), by using denormalized numbers. The smallest normalized floating point number 2^{-1022} (and 2^{-126}) gradually underflows by

$$(-1)^s \times (0 + \text{mantissa}) \times 2^{-b}$$

and loses bits in the mantissa. This means that numbers can be as small as `eps(0) = realmin*eps(1)/2` before complete underflow.

Example. A spectacular example of overflow happened in 1996 when the European Space Agency (after spending 10 years and $7 billion) launched the first Arianne 5 rocket. The rocket crashed within 40 seconds. The error occurred when a 64-bit floating point number was converted into a 16-bit signed integer. The number was larger than 2^{15} resulting in an overflow. The guidance system shut down, ultimately leading to an explosion and destroying the rocket and cargo valued at $500 million. ◀

Some expressions (and limits) 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: while mathematically 0^0 is not defined, IEEE 764 returns 1.

Ⓜ 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. For example, compare `plot([1 2 2 2 3],[1 2 NaN 1 2])` with `plot([1 2 2 2 3],[1 2 2 1 2])`.

7.5 Computational error

Computational error is the sum of rounding error caused by limited precision computation and truncation error resulting from a finite dimensional

approximation to the original problem.

Rounding error

Let $\mathrm{fl}(x)$ denote the floating-point approximation of a number x. The relative error is

$$\frac{x - \mathrm{fl}(x)}{x} = \varepsilon$$

where $|\varepsilon| \leq \mathrm{eps}$ (machine epsilon). We equivalently have that

$$\mathrm{fl}(x) = (1 + \varepsilon)x.$$

Let's consider the propagation of error caused by floating-point arithmetic. Consider $a + b$. The relative error E in the floating-point computation is given by $(a + b)(1 + E)$:

$$(a + b)(1 + E) = \mathrm{fl}[\mathrm{fl}(a) + \mathrm{fl}(b)] = [a(1 + \varepsilon_1) + b(1 + \varepsilon_2)](1 + \varepsilon_3)$$

where ε_i denotes the round-off error. Expanding the expression on the left-hand side and only keeping the linear terms gives us

$$(a + b)(1 + E) = a + b + a\varepsilon_1 + b\varepsilon_2 + (a + b)\varepsilon_3.$$

Solving for E will give us the relative error in the computation:

$$E = \frac{a\varepsilon_1 + b\varepsilon_2 + (a + b)\varepsilon_3}{a + b}$$

By taking $|\varepsilon_i| \leq \mathrm{eps}$ (machine epsilon), we have that

$$|E| \leq \frac{|a| + |b| + |a + b|}{|a + b|}\mathrm{eps}$$

Note that if a approximately equals (but is not equal to) $-b$ and both numbers are large, then $E \gg \varepsilon$ leading to catastrophic cancellation.

We can extend this example to a general sum. For example, for $a+b+c$ we have

$$\begin{aligned}(a + b + c)(1 + E) &= \mathrm{fl}\{\mathrm{fl}[\mathrm{fl}(a) + \mathrm{fl}(b)] + \mathrm{fl}(c)\} \\ &= [a(1 + \varepsilon_1) + b(1 + \varepsilon_2)](1 + \varepsilon_4) + c(1 + \varepsilon_3)](1 + \varepsilon_5) \\ &= a\varepsilon_1 + b\varepsilon_2 + c\varepsilon_3 + (a + b)\varepsilon_4 + (a + b + c)\varepsilon_5\end{aligned}$$

giving a bound on the relative error:

$$\begin{aligned}E &\leq \frac{|a| + |b| + |c| + |a + b| + |a + b + c|}{|a + b + c|}\mathrm{eps} \\ &\leq \frac{3|a| + 3|b| + 2|c|}{|a + b + c|}\mathrm{eps}.\end{aligned}$$

For n digits added sequentially $x_1 + x_2 \cdots + x_n$ there are $2n-1$ floating-point operations giving a crude upper bound on the absolute error

$$E \le [(n+1)|x_1| + (n+1)|x_2| + n|x_3| + \cdots + 2|x_n|]\,\mathrm{eps},$$

The actual error is often much smaller. To minimize the propagation of error, one should add smaller numbers first. If we model the errors ε_i as independent, uniformly distributed random variables $\varepsilon_i \in (-\mathrm{eps}/2, \mathrm{eps}/2)$, then the central limit theorem says that the sum of the errors should be normally distributed with mean 0 and variance $\mathrm{eps}^2 n/3$. So, with 99% probability the relative error

$$|E| \le 1.5\sqrt{n}\,\mathrm{eps}.$$

However, if errors are simply truncated (and not rounded) so that $\varepsilon_i \in (0, \mathrm{eps})$, then the central limit theorem says that the sum of the errors should be normally distributed with mean $\mathrm{eps}/2$ and variance $\mathrm{eps}^2 n/3$. So, with 99% probability

$$\left|E - \frac{n}{2}\mathrm{eps}\right| \le 1.5\sqrt{n}\,\mathrm{eps}.$$

Example. In 1982 the Vancouver stock exchange introduced an index starting with a value of 1000.000. After each transaction, the index was recomputed and truncated (not rounded) to three decimal places. After 22 months with roughly 2800 transactions per day the index was at 524.881. The index should have been 1098.892. With an estimated 1.3 million transactions and $\mathrm{eps} = 0.001$, error propagation by simply truncating is roughly 640 while the error propagation by rounding is 1.7. ◀

Example. When $x \approx 0$ the quantity $\mathrm{e}^x \approx 1$, and calculating $\mathrm{e}^x - 1$ can be affected by rounding error. We can avoid rounding error when $x = o(\mathrm{eps})$ by instead computing

$$\mathrm{e}^x - 1 = (1 + x + \tfrac{1}{2}x^2 + \tfrac{1}{3}x^3 + \cdots) - 1 = x + \tfrac{1}{2}x^2 + O(\mathrm{eps}^3). \qquad ◀$$

Ⓜ The MATLAB commands `expm1` and `logp1` compute $\mathrm{e}^x - 1$ and $\log(x+1)$ more precisely than `exp(x)-1` and `log(x+1)` when $x = o(\mathrm{eps})$.

Truncation error

We can compute the approximation to the derivative of a function using a first-order finite-difference approximation. Consider the Taylor series

expansion of

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

for $\xi \in (x, x+h)$. Then

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

and we have the finite-difference approximation

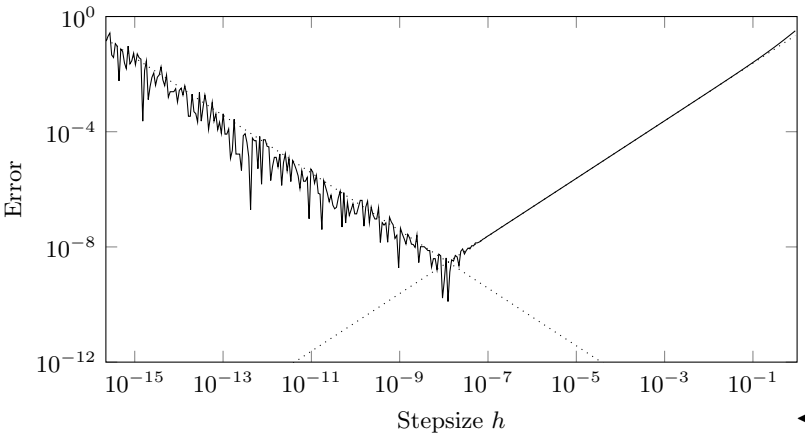
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

with truncation error bounded by $e_T = \frac{1}{2}hM$ where $M = \sup_\xi |f''(\xi)|$. The round-off error is bounded by $e_R = 2\,\mathrm{eps}/h$. The total error given by $E(h) = e_T + e_R$ is minimized when $E'(h) = 0$:

$$E' = \tfrac{1}{2}M - 2\,\mathrm{eps}/h^2 = 0$$

So, the error is minimum at $h \approx 2\sqrt{\mathrm{eps}/M}$.

Example. Consider first-order finite-difference approximation of $f'(x)$ the function $f(x) = \sin(x)$ at $x = 0.5$. We should expect a minimum total error of about 10^{-8} near $h = 10^{-8}$:



◀

Exercises

✍ 7.1. By combining the Taylor series expansions

$$f(x-h) = f(x) - hf'(x) + \tfrac{1}{2}h^2 f''(x) - \tfrac{1}{6}f'''(\xi)h^3$$

for $\xi \in (x-h, x)$ and

$$f(x+h) = f(x) + hf'(x) + \tfrac{1}{2}h^2 f''(x) + \tfrac{1}{6}f'''(\xi)h^3$$

for $\xi \in (x, x+h)$, we get the central difference approximation

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

with truncation error bounded by $e_T = \frac{1}{3}h^2 M$ 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 $f(x) = \exp(x)$ 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 the complex valued $f(x+\mathrm{i}h)$ and take its imaginary part. Estimate the total error as a function of h using this method and then confirm your estimate by plotting the total error as a function of the stepsize h for $f(x) = \exp(x)$ at $x = 0$.

✍ 7.2. 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.

CHAPTER 8

Solutions to Nonlinear Equations

Suppose that we want to find the solution to $f(x) = 2x - \tan x = 0$. It is a transcendental equation and cannot be solved algebraically. In this case, we rely on a numerical methods to generate a solution. But, it isn't just nonalgebraic equations that require numerical solutions. In general, roots of polynomials of degree five or greater can only be determined numerically due to Abel's impossibility theorem. In this chapter, we examine techniques to solve nonlinear equations and systems of equations.

8.1 Bisection method

Consider a real-valued function $f(x)$ which is continuous but not necessarily differentiable. Perhaps the simplest method of determining a zero x^* of $f(x)$ is to use the bisection method. Consider two points a and b such that $f(a)$ and $f(b)$ have opposite signs. We call the interval $[a, b]$ a *bracket*. Suppose $f(a) < 0$ and $f(b) > 0$. By the intermediate value theorem, the function f has a zero in the interval $[a, b]$. Take $c = (a + b)/2$. If $f(c) > 0$, then the zero is in the interval $[a, c]$. Otherwise, the zero is in the interval $[c, b]$. We continue iteratively subdividing the appropriate interval into smaller intervals until we are within some tolerance. Expressed as a MATLAB function:

```
function z = bisection(f,a,b,tolerance)
while abs(b-a) > tolerance
  c = (a+b)/2;
  if sign(f(c)) == sign(f(a))
    b = c;
  else
    a = c;
end
z = (a+b)/2;
```

How good is the bisection method? That is to say, when does the bisection method work and when it works, how fast do we get a good result? To answer these question, let's define the rate of convergence.

Convergence

A sequence $\{x^{(k)}\}$ is said to converge to x^* with order p if there exists a constant $C > 0$ such that the subsequent error $\left\|e^{(k+1)}\right\| \leq C \left\|e^{(k)}\right\|^p$:

$$\left\|x^{(k+1)} - x^*\right\| \leq C\left\|x^{(k)} - x^*\right\|^p$$

for all k sufficiently large. A numerical method that generates such a sequence is said to be of *order* p. If $p = 1$, we say that the method converges linearly. If the method is linear, then the convergence factor $C < 1$ for convergence. If $C = C_k$ varies with each iteration and $\lim_{k\to\infty} C_k = 0$, then we say that the method is superlinear. If $\lim_{k\to\infty} C_k = 1$, then we say that the method is sublinear. If $p = 2$, we say convergence is quadratic. If $p = 3$, convergence is cubic, and so on.

Let a search interval be $\mathrm{I}_0 = [a, b]$. A method is *locally convergent* if any $x^{(0)}$ close enough to the root x^* converges to x^*. A method is *globally convergent* if the method converges to x^* for any choice of $x^{(0)}$ in an interval $[a, b]$.

Let's go back to the bisection method. The size of the original bracket is $|\mathrm{I}^{(0)}| = |b - a|$ and the size of each subsequent bracket is half the size of the previous one. The absolute error at the kth step is $e^{(k)} = x^{(k)} - x^*$ and

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

So, the bisection method appears to converge linearly with a convergence factor of $C = \frac{1}{2}$. But, we can't really say that the bisection method converges linearly, because while the absolute error in the bisection method is a monotonically decreasing sequence, it is not always true that $e^{(k)} \leq Ce^{(k-1)}$. So, the bisection method is not technically of order one. But maybe this is nit picking.

How many iterations do we need to take to get the error $|e^{(k)}| < \varepsilon$? Since $e^{(k)} \leq 2^{-k}|b - a|$, we have that

$$k = \log_2\left(\frac{b - a}{\varepsilon}\right).$$

So, in order to gain a digit of precision in the solution, we would need to take about $\log_2(10) \approx 3.3$ bisections.

What have we learned? The $\lim_{k\to\infty} |e^{(k)}| = 0$ for any $x^{(0)} \in [a, b]$, which says that the bisection method is globally convergent. (This is

good.) But convergence is relatively slow. (This is bad.) So, the bisection method is a good method to get close enough to a solution in order to employ a faster (but only) locally convergent method such as Newton's method.

8.2 Newton's method

To get a better method, we need to include more information about the function $f(x)$ in our method. As before, we take x^* to be a zero of the function f, so that $f(x^*) = 0$. Suppose that x is sufficiently close to x^*. Take the Taylor series expansion of f about the point x to evaluate $f(x^*)$. That is,

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

where $\xi \in (x^*, x)$. Solving for x^* we have that

$$x^* = x - \frac{f(x)}{f'(x)} + o(x^* - x).$$

Take $x^{(0)} = x$ and take the linearized equation

$$x^{(k+1)} = x^{(k)} - \frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)}. \tag{8.1}$$

This method is called *Newton's method* or the *Newton–Raphson method*. Let's take a look at the convergence of Newton's method. The kth iterate is $x^{(k)} = x^* + e^{(k)}$ where $e^{(k)}$ is the error. Subtracting x^* from both sides of Newton's method gives

$$\begin{aligned} e^{(k+1)} &= e^{(k)} - \frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)} \\ &= \frac{f'\left(x^{(k)}\right)e^{(k)} - f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)} \\ &= \frac{f'\left(x^* + e^{(k)}\right)e^{(k)} - f\left(x^* + e^{(k)}\right)}{f'\left(x^* + e^{(k)}\right)}. \end{aligned}$$

Computing the Taylor expansion of each term and canceling terms gives us the following expression for $e^{(k)}k + 1$:

$$\frac{\frac{1}{2}f''(x^*)(e^{(k)})^2 + \frac{1}{3}f'''(x^*)(e^{(k)})^3 + \frac{1}{8}f''''(x^*)(e^{(k)})^4 + o\left((e^{(k)})^4\right)}{f'(x^*) + f''(x^*)e^{(k)} + \frac{1}{2}f'''(x^*)(e^{(k)})^2 + o\left((e^{(k)})^2\right)}. \tag{8.2}$$

If $f'(x^*) \neq 0$, then

$$e^{(k+1)} \approx \frac{1}{2}\frac{f''(x^*)}{f'(x^*)}{e^{(k)}}^2.$$

Therefore, Newton's method converges quadratically. This means that with every iteration we effectively double the number of correct digits in the solution, so for a good starting guess we should get an answer which is within machine precision in about 5 iterations.

Note that if $f'(x^*) = 0$ in $f''(x^*) \neq 0$ (a double root) then we have

$$e^{(k+1)} \approx \tfrac{1}{2}e^{(k)}$$

So, we have only first order convergence. Moreover, if $f^{(j)}(x^*) = 0$ for $j = 0, \ldots, k-1$ and $f^{(n)}(x^*) \neq 0$ (a root with multiplicity n), then from (8.2), the method converges linearly with

$$e^{(k+1)} \approx \left(1 - \tfrac{1}{n}\right) e^{(k)}.$$

When n is large, convergence can be slow. To speed up convergence, in the case of multiple root, we can try taking a larger step:

$$x^{(k+1)} = x^{(k)} - m\frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)}$$

for some $m > 1$. In this case, if $f^{(j)}(x^*) = 0$ for $j = 0, \ldots, n-1$ and $f^{(n)}(x^*) \neq 0$, then computing the error as in (8.2) we have that $e^{(k+1)}$ equals

$$\frac{\frac{n-m}{n!}f^{(n)}(x^*)(e^{(k)})^n + \frac{n+1-m}{(n+1)!}f^{(n+1)}(x^*)(e^{(k)})^{n+1} + O((e^{(k)})^{n+2})}{\frac{1}{(n-1)!}f^{(n)}(x^*)(e^{(k)})^{n-1} + O((e^{(k)})^n)}.$$

By taking $m = n$ we once again get quadratic convergence

$$e^{(k+1)} \approx \frac{1}{n(n+1)}f^{(n)}(x^*)(e^{(k)})^2.$$

Example. The Babylonian method is the earliest known algorithm to compute the square root of a number: $x^{(k+1)} = (x^{(k)} + a/x^{(k)})/2$. One way to derive the Babylonian method is as a special case of Newton's method. For $f(x) = x^2 - a$, Newton's method is

$$x^{(k+1)} = x^{(k)} - \frac{(x^{(k)})^2 - a}{2x^{(k)}} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right).$$

Because Newton's method has quadratic convergence, we can expect it to be fast. Let's compute $\sqrt{2}$ with a starting guess of 1:

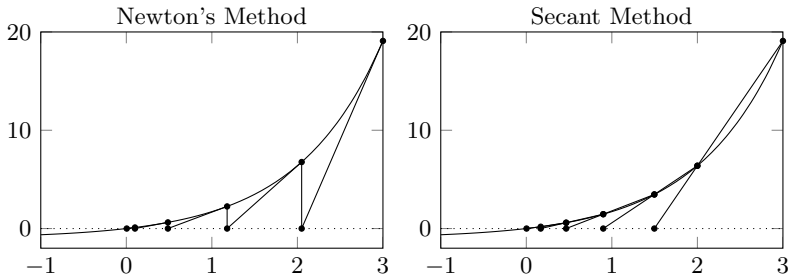


Figure 8.1: Methods for finding the zeros of $f(x) = \exp(x) - 1$.

1.0000000000000000000000000000000000000000000000000...
1.5000000000000000000000000000000000000000000000000...
1.416666666666666666666666666666666666666666666666666...
1.41421568627450980392156862745098039215686274509803...
1.414213562374689910626295578890134910116559622115 74...
1.4142135623730950488016 896235025302436149819257 7619...
1.41421356237309504880168872420969807856967187537723...

The correct digits are in bold face. The first iteration gets one correct digit. The second iteration gets three. Then six. Then 12, 25, and finally 48 at the sixth iteration. The number of correct decimals doubles with each iteration, as we should expect for a quadratic method. ◀

Secant method

As stated earlier, Newton's method finds the point of intersection of the line of slope $f'(x^{(k)})$ passing through the point $\left(x^{(k)}, f(x^{(k)})\right)$. See Figure 8.1. 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:

$$x^{(k+1)} = x^{(k)} - f(x^{(k)})/q^{(k)} \tag{8.3}$$

By taking the constant slope defined by two subsequent iterates

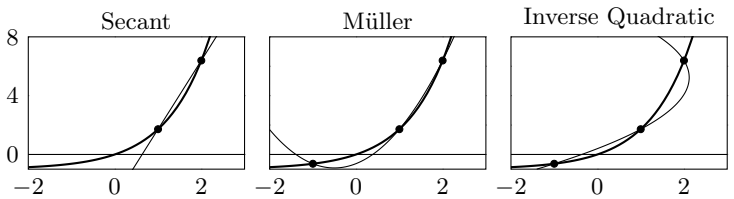
$$q^{(k)} = \frac{f\left(x^{(k)}\right) - f\left(x^{(k-1)}\right)}{x^{(k)} - x^{(k-1)}} \tag{8.4}$$

we have the *secant method*. Note that the secant method requires two initial values. The secant method converges with order of about 1.63.

Higher order methods

We can generate methods that converge faster than order two. For instance, there are several higher order extensions of Newton's method called Householder methods. But, because the same net result can be achieved with multiple iterations of a superlinear method, it is not often necessary to go through the trouble of implementing higher order methods. Two iterations of the second-order Newton's method is a fourth-order method.

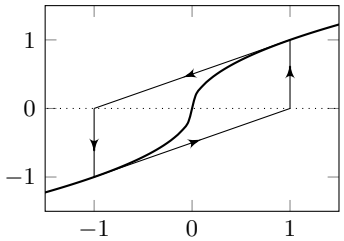
The secant method uses simple linear interpolation to find the $x^{(k+1)}$. One way to improve convergence is to interpolate using a quadratic function instead. Consider the parabola $y = ax^2 + bx + c$ which passes though the points $\big(x^{(k-2)}, f(x^{(k-2)})\big)$, $\big((x^{(k-1)}, f(x^{(k-1)})\big)$ and $\big((x^{(k)}, f(x^{(k)})\big)$. Because the parabola likely crosses the x-axis at two points, we'll need to choose the best one. This method, called *Müller's method*, converges with order of about 1.84. A similar method called *inverse quadratic approximation* interpolates using the parabola $x = ay^2 + by + c$. This parabola crosses the x-axis ($y = 0$) at $x^{(k+1)} = c$. Of course, these methods need three points rather than just two needed for the secant method or one needed for Newton's method.



Dekker–Brent method

Newton's method along with the secant method are not globally convergent. When using these methods, an iterate can converge to a different root, it can go off to infinity, or it can get trapped in cycles.

Example. Consider the function $f(x) = \text{sign}(x)\sqrt{|x|}$. The function is continuous every and differential every except at $x = 0$.



Newton's method gives the iteration

$$x^{(k+1)} = x^{(k)} - 2x^{(k)} = -x^{(k)}.$$

The iteration clearly does not converge, instead oscillating between $x^{(0)}$ and $-x^{(0)}$ for any non-zero starting guess. ◀

The bisection method is globally convergent (and therefore robust), but it is only linearly convergent. The secant (and inverse quadratic interpolation) method is superlinearly convergent (with convergence rate approaching 1.63) but it is only locally convergent. The Dekker–Brent Method is a hybridization of the bisection methods and the secant methods and takes advantage of the strengths of both methods.

0. Start with a bracket.
1. Perform one secant (or inverse quadratic interpolation) iteration to get $x^{(k)}$. But if the secant step goes outside the bracketing interval, use bisection instead to get $x^{(k)}$.
2. Refine the bracketing interval using $x^{(k)}$
3. Repeat until convergence.

Ⓜ The MATLAB 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.

8.3 Fixed-point iterations

If I type any number into a calculator and then hit the cosine button over and over again eventually the display stops changing and I get `0.7390851`. We call this number a *fixed point*. A fixed point of a function $\phi(x)$ is any point x such that $x = \phi(x)$. The simple geometric interpretation of a fixed point is the intersection of the line $y = x$ and the curve $y = \phi(x)$. Newton's method can be studied as examples of a broader class of numerical methods—fixed-point iterations:

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

In fact, for any function $f : [a, b] \to \mathbb{R}$, we can always write the problem $f(x) = 0$ as a fixed-point iteration $x = \phi(x)$ where $\phi(x) = x - f(x)$. But

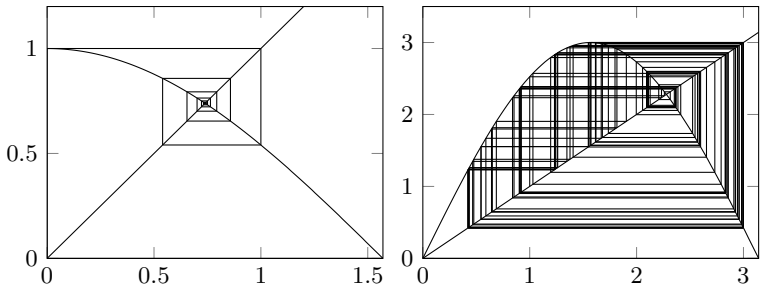


Figure 8.2: The fixed point iteration for $\phi(x) = \cos x$ (which converges) and $\phi(x) = 3\sin x$ (which does not converge).

we are not guaranteed that the fixed-point iteration will converge. See Figure 8.2.

We will say that $\phi(x)$ is a *contraction mapping* over the interval $[a, b]$ if ϕ is differentiable function with $\phi : [a, b] \to [a, b]$ 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 $K < 1$ of a metric space onto itself. In this generalization, the following contraction mapping theorem is known as the *Banach fixed-point theorem*.

Theorem 23. *Consider the sequence $x^{(k+1)} = \phi(x^{(k)})$ for some $x^{(0)} \in [a, b]$. If the iteration function ϕ is a contraction map over $[a, b]$, then ϕ has a unique fixed-point $x^* \in [a, b]$ and $\{x^{(k)}\}$ converges it for any $x^{(0)} \in [a, b]$.*

Proof. The contraction mapping ϕ is a continuous map from $[a, b]$ to $[a, b]$, so ϕ has at least one fixed point. Now, suppose that there are at least two distinct values $x^*, y^* \in [a, b]$ such that $\phi(x^*) = x^*$ and $\phi(y^*) = y^*$. The first-order Taylor approximation of ϕ expanded about x^* and evaluated at y^* is

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

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 ϕ expanded about x^* and evaluated at $x^{(k)}$ is

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

for some $\xi^{(k)} \in [x^*, x^{(k)}]$. By definition $x^{(k+1)} = \phi(x^{(k)})$ and so it follows that

$$x^{(k+1)} - x^* = \phi(x^{(k)}) - \phi(x^*) = (x^{(k)} - x^*)\phi'(\xi^{(k)}).$$

So,

$$|x^{(k+1)} - x^*| \le K|x^{(k)} - x^*| \le K^{k+1}|x^{(0)} - x^*| \to 0 \quad \text{as} \quad k \to \infty. \quad \square$$

The set of points $\{x^{(0)}, \phi(x^{(0)}), \phi^2(x^{(0)}), \phi^3(x^{(0)}), \dots\}$ of the iterated function ϕ is called the forward *orbit* of $x^{(0)}$. A fixed point x^* is an *attractive fixed point* if there is an interval (a, b) about x^* such that if $x^{(0)} \in (a, b)$ then $\phi(x^{(k)}) \in (a, b)$ for all $k \ge 0$ and $\phi(x^{(k)}) \to x^*$ as $k \to \infty$. A fixed point x^* is a *repulsive fixed point* if there is an interval (a, b) about x^* such that if $x^{(0)} \in (a, b)$ then there is some $k > 0$ such that $\phi(x^{(k)}) \notin (a, b)$. The fixed point x^* is attractive if $|\phi'(x^*)| < 1$ and repulsive if $|\phi'(x^*)| > 1$.

When fixed-point iteration converges, the error at each step diminishes. We can computer the ratio of the subsequent errors $e^{(k)}$:

$$\begin{aligned}\lim_{k\to\infty} \frac{e^{(k+1)}}{e^{(k)}} &= \lim_{k\to\infty} \frac{x^{(k+1)} - x^*}{x^{(k)} - x^*} \\ &= \lim_{k\to\infty} \frac{\phi(x^{(k)}) - \phi(x^*)}{x^{(k)} - x^*} = \lim_{k\to\infty} \phi'(\xi^{(k)}) = \phi'(x^*).\end{aligned}$$

The value $|\phi'(x^*)|$ is called the *asymptotic convergence factor*. In general, we have following theorem:

Theorem 24. *If $\phi \in C^{p+1}$ in some neighborhood of x^* and $\phi^{(j)}(x^*) = 0$ for $1 \le j \le p$ and $\phi^{(p+1)}(x^*) \ne 0$, then the fixed-point method with function ϕ has order $p+1$ with asymptotic convergence factor $\frac{1}{(p+1)!}\phi^{(p+1)}(x^*)$.*

Proof. The Taylor series expansion of ϕ around $x = x^*$ is

$$\begin{aligned}\phi(x^{(k)}) &= \phi(x^*) + \sum_{j=1}^{p} \frac{\phi^{(j)}(x^*)}{j!}(x^{(k)} - x^*)^j + \frac{\phi^{(p+1)}(\xi)}{(p+1)!}(x^{(k)} - x^*)^{p+1} \\ &= \phi(x^*) + \frac{\phi^{(p+1)}(\xi)}{(p+1)!}(x^{(k)} - x^*)^{p+1}\end{aligned}$$

for some $\xi \in [x^{(k)}, x^*]$. By definition $x^{(k+1)} = \phi(x^{(k)})$ and $x^* = \phi(x^*)$, so

$$x^{(k+1)} - x^* = \frac{\phi^{(p+1)}(\xi)}{(p+1)!}(x^{(k)} - x^*)^{p+1}.$$

Therefore,

$$\lim_{k\to\infty} \frac{e^{(k+1)}}{(e^{(k)})^{p+1}} = \lim_{k\to\infty} \frac{x^{(k+1)} - x^*}{(x^{(k)} - x^*)^{p+1}} = \lim_{k\to\infty} \frac{\phi^{(p+1)}(\xi)}{(p+1)!} = \frac{\phi^{(p+1)}(x^*)}{(p+1)!}.$$

$\square$

Speeding up the fixed-point method

We can modify a linearly convergent fixed-point method $x^{(k+1)} = \phi(x^{(k)})$ to get faster convergence. To do this, we'll consider Newton's method where $f(x) = x - \phi(x) = 0$:

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - \phi(x^{(k)})}{1 - \phi'(x^{(k)})}. \tag{8.6}$$

Just as we approximated the the slope $f'(x)$ in Newton's method to get the secant method, we can approximate $\phi'(x^{(k)})$:

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

Substituting this expression in for $\phi'(x^{(k)})$ gives us

$$x^{(k+1)} = x^{(k)} - \frac{(\phi(x^{(k)}) - x^{(k)})^2}{\phi(\phi(x^{(k)})) - 2\phi(x^{(k)}) + x^{(k)}} \tag{8.7}$$

This expression is known as *Aitken's extrapolation formula*. Even if a method is not convergent (for example, $x^{(k+1)} = 3\sin(x^{(k)})$), Aitken's method converges.

8.4 Stopping criteria

We should stop iteration once the error $e^{(k)} = x^* - x^{(k)}$ is smaller than some given tolerance ε. But because we don't know the value of x^*, we can't compute $e^{(k)}$. Instead, we could use the residual $|f(x^{(k)})|$ or the increment $|x^{(k+1)} - x^{(k)}|$ to estimate the error, stopping when one of them falls below some tolerance. Which one do we use and when?

- Stopping when the residual $|f(x^{(k)})| \leq \varepsilon$. Recall that the relative error is bounded by the relative condition number. Suppose that $f'(x^*) \neq 0$ (x^* is a simple root). Then

  $$\frac{|e^{(k)}|}{x^*} < \kappa(x^*) = \left| \frac{f(x^{(k)})}{x^* f'(x^*)} \right|$$

  and the error is scaled by $1/|f'(x^*)|$.

  1. If $|f'(x^*)| \approx 1$, then $|e^{(k)}| \approx \varepsilon$ and the residual provides a good estimate.
  2. If $|f'(x^*)| \ll 1$, then $|e^{(k)}|$ could be large with respect to ε. The residual is not reliable.

3. If $|f'(x^*)| \gg 1$, then $|e^{(k)}| \ll \varepsilon$. The residual is too restrictive.

- Stopping when the increment $|x^{(k+1)} - x^{(k)}| < \varepsilon$. The error of the fixed-point method $x^{(k+1)} = \phi(x^{(k)})$ is

$$e^{(k+1)} = x^* - x^{(k)} = \phi(x^*) - \phi(x^{(k)}) \approx \phi'(x^*)e^{(k)}.$$

So,

$$x^{(k+1)} - x^{(k)} = e^{(k+1)} - e^{(k)} \approx (1 - \phi'(x^*))\, e^{(k)}$$

and hence

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

1. If $|\phi'(x^*)| = 0$ (as in Newton's method for a simple root), then $|e^{(k)}| \approx \varepsilon$. The increment is a good estimate. The increment is also a good estimate if $-2 < \phi'(x^*) < 0$.
2. If $\phi'(x^*) \approx 1$, then $|e^{(k)}|$ could be large with respect to ε. The increment is not a reliable test.

8.5 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. We could easily compute the solution using the quadratic formula, but a toy problem like this can help us better understand the dynamics of the fixed-point method. 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. Note that it is equivalent to the logistic map $x^{(k+1)} = rx^{(k)}(1 - x^{(k)})$ introduced in (7.4) by scaling and translating the variables. To see this, first complete the square in the logistic map: $x^{(k+1)} = -r\left(x^{(k)} - \frac{1}{2}\right)^2 + \frac{1}{4}r$. Now, let $-r(x^{(k)} - \frac{1}{2}) \mapsto x^{(k)}$ and simplify to get the quadratic map $x^{(k+1)} = (x^{(k)})^2 + c$ where $c = r(2 - r)/4$.

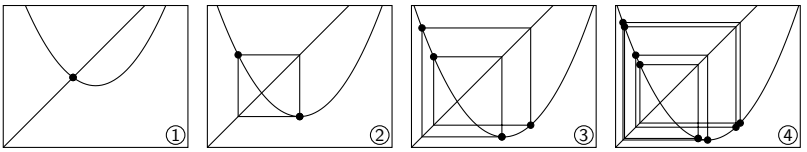
By the fixed-point theorem, as long as $|\phi_c'(x)| < 1$ in a neighborhood of a fixed-point x^*, any initial guess in that neighborhood will converge to that fixed point. Because $|\phi_c'(x)| = |2x|$, the fixed point x^* is attractive whenever it is in the interval $(-\frac{1}{2}, \frac{1}{2})$, which happens when $c \in (-\frac{3}{4}, \frac{1}{4})$. When $c > \frac{1}{4}$, the equation $x^2 - x + c = 0$ no longer has a solution. What about when $c < -\frac{3}{4}$? Let's look at a few cases.

① $c = -1/2$: the orbit of $x^{(0)} = 0$ converges to a fixed point.

② $c = -1$: 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 of period 4.

④ $c = -1.38$: the orbit of $x^{(0)} = 0$ converges to an orbit of period 8.



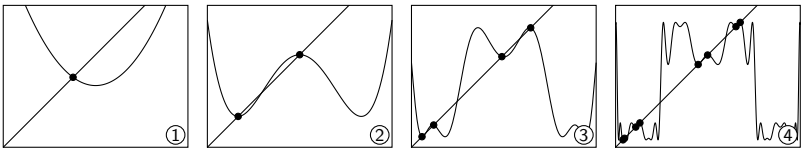
Why does the solution have this periodic limiting behavior? Take the iterated functions $\phi_c^2 = \phi_c \circ \phi_c$, $\phi_c^3 = \phi_c \circ \phi_c \circ \phi_c$, and so on. We note that the forward orbit $\{x^{(0)}, \phi_c(x^{(0)}), \phi_c^2(x^{(0)}), \phi_c^3(x^{(0)}), \dots\}$ can be decomposed 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 $x^{(1)} = \phi_c(x^{(0)})$. Examining the plot of $y = \phi_c^2(x)$ and $y = x$, we see that the function has two fixed-points where $|(\phi_c^2)'(x)| < 1$. Starting with $x^{(0)}$ leads us to one fixed point under the mapping ϕ_c^2 and starting with $x^{(1)}$ leads us to another fixed point. Similarly, we can decompose

$$\begin{aligned}\{x^{(0)}, \phi_c^2(x^{(0)}), \phi_c^4(x^{(0)}), \dots\} \cup \{x^{(1)}, \phi_c^2(x^{(1)}), \phi_c^4(x^{(1)}), \dots\} = \\ \{x^{(0)}, \phi_c^4(x^{(0)}), \phi_c^8(x^{(0)}), \dots\} \cup \{x^{(2)}, \phi_c^4(x^{(2)}), \phi_c^8(x^{(2)}), \dots\} \cup \\ \{x^{(1)}, \phi_c^4(x^{(1)}), \phi_c^8(x^{(1)}), \dots\} \cup \{x^{(3)}, \phi_c^4(x^{(3)}), \phi_c^8(x^{(3)}), \dots\}.\end{aligned}$$

From the plot of $\phi_c^4(x)$, we see that the function 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 $c = -1.75$. Then at $c \approx 1.368$. The rate of period doubling limits a value called the *Feigenbaum contant*. When $c \approx 1.401155$, the period of the limit cycle approaches infinity and the map is said to be *chaotic*. But, as c is made more negative, a finite period cycle re-emerges only to again double and re-double. See the bifurcation diagram in Figure 8.3 on the facing page. Finally, when $x < -2$, the sequence escapes to infinity.

Often examining a more general problem provides greater insight on the specific problem. We have examine the quadratic map when the

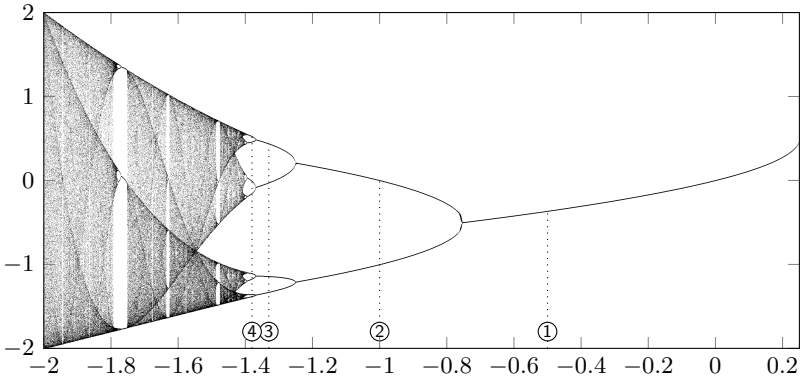


Figure 8.3: Limit cycle $\{\phi_c^\infty(0)\}$ as a function of the c.

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, when does $z^{(k)}$ converge to a fixed-point z^*? By the fixed-point theorem, z attractive when $|\phi_c'(z)| < 1$. That is, when $|z| < \frac{1}{2}$. Let's take the boundary $z = \frac{1}{2}e^{i\theta}$ and find out which values c correspond to this boundary. From the original equation $z^2 - z + c = 0$, we have that $c = z - z^2 = \frac{1}{2}e^{i\theta} - \frac{1}{4}e^{2i\theta}$, which is a cardoid , the shape traced by point on a circle rolling round another circle of the same diameter. There is an interesting related puzzle: "A coin is rolled around a similar coin without slipping. What is the orientation of the coin when it is half-way around the stationary coin?" The answer is at first counter-intuitive, but it is clear in the equation of the cardoid. For values c inside the cardoid, the orbit $\{0, \phi_c(0), \phi_c^2(0), \dots\}$ limits a fixed-point.

If we extend c to include orbits $\{0, \phi_c(0), \phi_c^2(0), \dots\}$ whose limit sets are and bounded, the set of c is called the *Mandelbrot set*. See Figure 8.4 on the next page. At the heart of the Mandelbrot set is the cardoid 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 cardoid 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.

If the Mandelbrot set is the set of parameters c that result in a 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-plane, 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

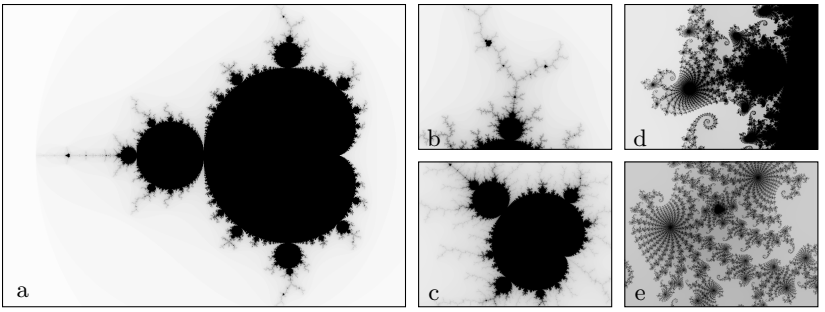


Figure 8.4: The Mandelbrot set. Each subfigure (b–e) is a magnification of the previous ones (a–d).

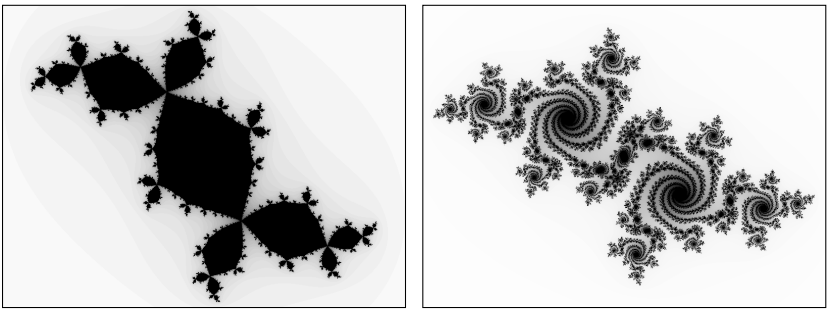


Figure 8.5: Julia sets.

is disconnected and called a *Cantor sets*. See Figure 8.5.

8.6 Roots of polynomials

An nth degree polynomial has exactly n complex roots counting multiplicity. But Abel's impossibility theorem states that there is no general analytical method to finding these roots if the polynomial is degree five or higher. In this case, we must find the roots numerically. While any of the previously discussed methods will work, a better design is one that exploits the polynomial structure. Namely, once we have found one root, we can factor this root out of the polynomial to simplify the next step. This process is called *deflation*.

Horner's deflation

Note that the polynomial $p(x) = x^3 - 6x^2 + 11x - 6$ can be written in the Horner form $x(x(x-6)+11)-1)-6$. This new form allows us to evaluate a polynomial more quickly, requiring $n-1$ multiplications and $n-1$ additions rather than $2n-1$ multiplications and $n-1$ additions. Because of this, it also reduces propagated round-off error. Synthetic multiplication provides us with a convenient means of record-keeping when evaluating a polynomial in Horner form. For a general polynomial

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0,$$

synthetic multiplication is performed by determining b_0 given coefficients $\{a_n, a_{n-1}, \ldots, a_0\}$ 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 $b_k = a_k + zb_{k+1}$ for $k = 0, \cdots, n-1$, and $p(z) = b_0$. Note that if z is a root of $p(x)$ then $b_0 = 0$. To evaluate $p(4)$

$$\begin{array}{c|cccc} 4 & 1 & -6 & 11 & -6 \\ & & 4 & -8 & 6 \\ \hline & 1 & -2 & 3 & 6 \end{array}.$$

So, $p(4) = 6$.

By definition $b_k - zb_{k+1} = a_k$ 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

$$p(x) = (x-z)b_n x^{n-1} + (x-z)b_{n-1}x^{n-2} + \cdots + (x-z)b_1 + b_0.$$

So we have that

$$p(x) = (x-z)q(x) + b_0 \tag{8.8}$$

where

$$q(x) = b_n x^{n-1} + b_{n-1}x^{n-2} + \cdots + b_2 x + b_1.$$

This says that if z is the root of $p(x)$ then (as we noted earlier) $b_0 = p(z) = 0$. 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 this idea in action by factoring $p(x)$ using Ruffini's scheme. For this polynomial, 2 is a root:

$$\begin{array}{c|cccc} 2 & 1 & -6 & 11 & -6 \\ & & 2 & -8 & 6 \\ \hline & 1 & -4 & 3 & 0 \end{array}$$

The factored polynomial $x^2 - 4x + 3$. We can continue—3 is also a root:

$$\begin{array}{c|ccc} 3 & 1 & -4 & 3 \\ & & 3 & -3 \\ \hline & 1 & -1 & 0 \end{array}$$

Leaving us with $x - 1$. So, we find that the roots are 3, 2 and 1.

Newton–Horner method

We can use Newton's method or Müller's method to find a root z and then use Horner's method to deflate the polynomial. Newton's method for polynomials is simple. Note that from (8.8), for some z (not necessarily a root) $p(x) = (x - z)q(x) + b_0$ where $q(x)$ and b_0 are functions by 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. Find $x^{(1)}$ using Newton's method for the polynomial $p(x) = x^3 - 6x^2 + 11x - 6$ starting with $x^{(0)} = 4$. Let's determine $p'(4)$:

$$\begin{array}{r|rrrr} 4 & 1 & -6 & 11 & -6 \\ & & 4 & -8 & 6 \\ \hline & 1 & -2 & 3 & 6 \end{array}$$

$$\begin{array}{r|rrr} 4 & 1 & -2 & 3 \\ & & 4 & 8 \\ \hline & 1 & 2 & 11 \end{array}$$

So, $p'(4) = 11$. In the case of $x^{(0)} = 4$, we have that $x^{(1)} = 4 - \frac{6}{11}$. ◀

Here's the Newton–Horner method for $p(x) = a_n x^n + \cdots + a_1 x + a_0$:

1. Let $\mathbf{a} = \begin{pmatrix} a_n & a_{n-1} & \cdots & a_1 & a_0 \end{pmatrix}$ be the array of coefficients.
2. Choose an initial guess $x^{(0)}$.
3. Use Horner's method on $\mathbf{a}$ using $z = x^{(j)}$ to compute the array of coefficients $\mathbf{b} = \begin{pmatrix} b_n & b_{n-1} & \cdots & b_1 & b_0 \end{pmatrix}$ where $b_n = a_n$ and $b_k = a_k + z b_{k+1}$ for $k = 0, \ldots, n-1$. Then $p(z) = b_0$.
4. Use Horner's method on $\begin{pmatrix} b_n & b_{n-1} & \cdots & b_1 \end{pmatrix}$ using z to compute $q(z)$.
5. Compute one Newton iteration $x^{(j+1)} = x^{(j)} - p(z)/q(z)$.
6. Repeat steps 3–5 until convergence.
7. Polynomial deflation. Set $a_{n-1} \leftarrow b_n$, $a_{n-2} \leftarrow b_{n-1}, \ldots, a_0 \leftarrow b_1$ and take $n \leftarrow n - 1$.
8. Repeat for the remaining roots.

Companion matrix method

Another way to find the roots of a polynomial $p(x)$ is to consider a related problem of finding the eigenvalues of a matrix whose characteristic polynomial is $p(x)$. The *companion matrix* of the polynomial

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1} + x^n$$

is the matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & \cdots & 0 & -c_0 \\ 1 & 0 & \cdots & 0 & -c_1 \\ 0 & 1 & \cdots & 0 & -c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -c_{n-1} \end{bmatrix}.$$

We can compute the characteristic polynomial the companion matrix $\det(\mathbf{A} - \lambda\mathbf{I})$ using the method of cofactors:

$$\begin{vmatrix} \lambda & 0 & \cdots & 0 & c_0 \\ -1 & \lambda & \cdots & 0 & c_1 \\ 0 & -1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & -1 & \lambda + c_{n-1} \end{vmatrix}$$

$$= (-1)^n(\lambda + c_{n-1})\lambda^{n-1} + (-1)^n \begin{vmatrix} \lambda & 0 & \cdots & 0 & c_0 \\ -1 & \lambda & \cdots & 0 & c_1 \\ 0 & -1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & -1 & c_{n-2} \end{vmatrix}$$

which simply equals $p(\lambda) = \lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_2\lambda^2 + c_1\lambda + c_0$.

Ⓜ The MATLAB function `roots` returns the roots of a polynomial by finding the eigenvalues of the companion matrix.

Other methods

There are several other popular methods of finding roots of polynomials such as Laguerre's method and Jenkins–Traub method, which is used in Mathematica's `NSolve` function. These methods typically find a root and then deflate using Horner deflation.

8.7 Systems of nonlinear equations

Suppose that we have a system of equations. Such a system often arises when finding a nonlinear least-squares fit or implementing an implicit solver for a partial differential equation. When we found the roots to a nonlinear equation, we had to contend with non-unique (but isolated) solutions. For systems, it is possible to have non-isolated points, so the well-posedness of the problem becomes more complicated.

Newton's method

Consider a system of n coupled nonlinear equations with n unknowns $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ where $\mathbf{F}$ is a continuously differentiable function. Suppose that we are located at $\mathbf{x}^{(k)}$ near some root $\mathbf{x}^*$. Then we only need to

determine the offset vector $\delta\mathbf{x} = \mathbf{x}^* - \mathbf{x}^{(k)}$ to solve the problem. Taylor series expansion about $\mathbf{x}^{(k)}$ evaluated at $\mathbf{x}^*$ is simply

$$0 = \mathbf{F}(\mathbf{x}^*) = \mathbf{F}(\mathbf{x}^{(k)}) + \mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(k)})\delta\mathbf{x} + O((\delta\mathbf{x})^2)$$

where $\mathbf{J}$ is the Jacobian matrix with elements given by $\mathbf{J}_{ij} = \partial F_i/\partial x_j$. Solving this system of equations (assuming that the Jacobian matrix is nonsingular) gives

$$\delta\mathbf{x} = -\left(\mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{x}^{(k)}) + O((\delta\mathbf{x})^2)$$

and since $\mathbf{x}^* = \mathbf{x}^{(k)} + \delta\mathbf{x}$

$$\mathbf{x}^* = \mathbf{x}^{(k)} - \left(\mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{x}^{(k)}) + O((\delta\mathbf{x})^2).$$

We haven't really solved the problem yet, because we have the $O((\delta\mathbf{x})^2)$ term on the right hand side implicitly. But by sending the error term $O((\delta\mathbf{x})^2)$ to zero, we have solved an approximate linear problem to get an approximate solution

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{x}^{(k)})$$

with the error

$$\|\mathbf{x}^* - \delta\mathbf{x}^{(k)}\| = \|O(\delta\mathbf{x})^2)\|.$$

As we get a better estimate $\delta\mathbf{x}^{(k+1)}$ for $\mathbf{x}^*$, the error term becomes smaller and smaller.

In practice, we do not always need to or want to compute the inverse of the Jacobian directly. Rather we solve the problem:

$$\text{evaluate} \quad \mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(k)}), \tag{8.9a}$$

$$\text{solve} \quad \mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(k)})\boldsymbol{\delta}\mathbf{x}^{(k)} = -\mathbf{F}(\mathbf{x}^{(k)}), \tag{8.9b}$$

$$\text{set} \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \boldsymbol{\delta}\mathbf{x}^{(k)}. \tag{8.9c}$$

If Jacobian matrix $\mathbf{J}_{\mathbf{F}}$ is a dense matrix, the costs are $O(n^2)$ to determine $\mathbf{J}_{\mathbf{F}}$ and $O(n^3)$ to solve the system. For a sparse system, we have $O(n)$ and $O(n^2)$, respectively. To speed up the method there are several things that we can do.

- Reuse the Jacobian—don't compute the Jacobian matrix at every step. Instead, evaluate it at every few steps. By using LU-decomposition to solve (8.9b), we can save and reuse the upper and lower triangular factors, so that each subsequent iteration take $O(n^2)$ operations.

- We can use an iterative solution to solve the linear system (8.9b) such as Jacobi method, Gauss–Seidel, SOR, or the conjugate gradient methods. Iterative methods work by multiplying (instead of factoring), and therefore such methods require only $O(n^2)$ operations for dense matrices (and only $O(n)$ operations for sparse matrices). Because we are only computing an approximate solution to start with, it is not necessary to get "full" convergence of the iterative method. (These methods are discussed in detail in Chapter 5.)

- Rather than compute the Jacobian matrix analytically, we can make a finite difference approximation. By definition $J_{ij} = \partial F_i/\partial x_j$, so the jth column of the Jacobian matrix is simply the derivative of $\mathbf{F}$ with respect to x_j. We can, for example, take the second order approximation of the jth column of $\mathbf{J}$ is

$$[J^{(k)}]_j \approx \frac{\mathbf{F}(\mathbf{x}^{(k)} + h_j^{(k)}\mathbf{e}_j) - \mathbf{F}(\mathbf{x}^{(k)} - h_j^{(k)}\mathbf{e}_j)}{2h_j^{(k)}}$$

where ${h_j}^{(k)}$ is the step size and $\mathbf{e}_j$ is the jth vector in the canonical basis of $\mathbb{R}^n$. That is,

$$[J^{(k)}]_{ij} \approx \frac{F_i(x_1, \ldots, x_j + h_j^{(k)}, \ldots, x_n) - F_i(x_1, \ldots, x_j - h_j^{(k)}, \ldots, x_n)}{2h_j^{(k)}}$$

To choose the step-size $h_j^{(k)}$ we need to weigh truncation error against round-off error. Recall Figure **??** on page ??. For a well-conditioned problem, we can typically take the step size h_j approximately equal to the cube root of machine epsilon. The total error

$$e_{\text{total}} \approx \underbrace{Mh^2}_{\text{truncation}} + \underbrace{2\,\text{eps}/h}_{\text{round-off}}$$

(where eps is machine epsilon and M is some constant) is minimized when its derivative with respect to h is 0. Then

$$2Mh - 2h^{-2}\text{eps} = 0 \qquad \text{and so} \qquad h = \sqrt[3]{\text{eps}/M}.$$

Example. Consider roots of the polynomial system

$$\begin{aligned} x^3 - 3xy^2 - 1 &= 0 \\ y^3 - 3x^2y &= 0. \end{aligned}$$

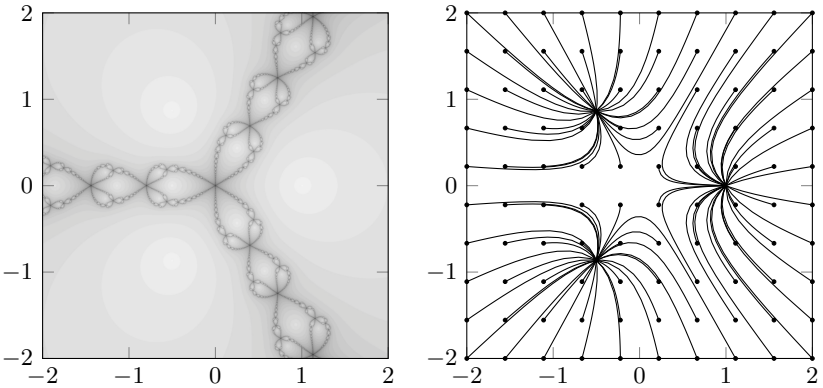


Figure 8.6: Newton fractal (left) for roots of $z^3 - 1$. Darker shades of gray require more iterations to converge. Homotopy continuation (right) for the same problem.

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 every initial guess either converges to a root or fails to converge. The set of points which eventually converges to a specific root is called a *basin of attraction*. The union of the boundaries of each basin of attraction is called the *Julia set*. See Figure 8.6. ◀

Secant-like methods

Recall that the secant method used two subsequent iterates $x^{(k)}$ and $x^{(k-1)}$ to compute the approximate slope for an approximation of Newton's method. Just as we could extend Newton's method to a system of equations, one might try to extend the secant method to a system of equations also. For comparison, recall (8.1) and (8.3). Formally, we replace the $n \times n$ Jacobian matrix $\mathbf{J}$ in Newton's method

$$\mathbf{J}^{(k)}\boldsymbol{\delta}\mathbf{x}^{(k)} = -\mathbf{F}(\mathbf{x}^{(k)}) \quad \text{where} \quad \boldsymbol{\delta}\mathbf{x}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}$$

with an approximate $n \times n$ Jacobian matrix $\tilde{\mathbf{J}}^{(k)}$ where

$$\tilde{\mathbf{J}}^{(k)}\boldsymbol{\delta}\mathbf{x}^{(k)} = \mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{F}(\mathbf{x}^{(k-1)}). \tag{8.10}$$

For comparison, recall the secant method slope (8.4). Equation (8.10) is an identification problem: given output $\boldsymbol{\delta}\mathbf{F}^{(k)} = \mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{F}(\mathbf{x}^{(k-1)})$

along with input $\boldsymbol{\delta}\mathbf{x}^{(k)}$, find the operator $\tilde{\mathbf{J}}^{(k)}$. Unfortunately, $\tilde{\mathbf{J}}^{(k)}$ has n^2 unknowns and we are only given n equations—a bit of a problem. It would seem reasonable that $\tilde{\mathbf{J}}^{(k)}$ should be close to the previous $\tilde{\mathbf{J}}^{(k-1)}$, so perhaps we can approximate $\tilde{\mathbf{J}}^{(k)}$ using $\tilde{\mathbf{J}}^{(k-1)}$ with (8.10) as a constraint. That is, we want to minimize $\|\tilde{\mathbf{J}}^{(k)} - \tilde{\mathbf{J}}^{(k-1)}\|$ subject to the constraint that $\tilde{\mathbf{J}}^{(k)}\boldsymbol{\delta}\mathbf{x}^{(k)} = \boldsymbol{\delta}\mathbf{F}^{(k)}$. What norm $\|\cdot\|$ do we choose? It seems that a good choice of norms would be the Frobenius norm, because the Frobenius norm of a matrix $\mathbf{A}$ is a simple extension of the l^2-norm for vectors $\|\mathbf{A}\|_F = (\sum_{i,j} a_{ij}^2)^{1/2}$, and minimizing this norm results in a linear problem.

To minimize a functional subject to a constraint we can use the method of Lagrange multipliers. Take $\lambda \in \mathbb{R}^n$ and define

$$L(\tilde{\mathbf{J}}^{(k)}, \lambda) = \|\tilde{\mathbf{J}}^{(k)} - \tilde{\mathbf{J}}^{(k-1)}\|_F^2 + \lambda^T(\tilde{\mathbf{J}}^{(k)}\boldsymbol{\delta}\mathbf{x}^{(k)} - \boldsymbol{\delta}\mathbf{F}^{(k)}).$$

Setting $\nabla L(\tilde{\mathbf{J}}^{(k)}, \lambda) = 0$ (the gradient is with respect to the n^2 elements of $\tilde{\mathbf{J}}^{(k)}$ and the n elements of λ) gives us

$$2(\tilde{\mathbf{J}}^{(k)} - \tilde{\mathbf{J}}^{(k-1)}) + \underbrace{\lambda(\boldsymbol{\delta}\mathbf{x}^{(k)})^T}_{\text{rank 1 matrix}} = 0 \tag{8.11}$$

$$\tilde{\mathbf{J}}^{(k)}\boldsymbol{\delta}\mathbf{x}^{(k)} = \boldsymbol{\delta}\mathbf{F}^{(k)}. \tag{8.12}$$

We now just need to eliminate λ. Right multiply the first equation by $\boldsymbol{\delta}\mathbf{x}^{(k)}$. Then

$$2\left(\tilde{\mathbf{J}}^{(k)}\boldsymbol{\delta}\mathbf{x}^{(k)} - \tilde{\mathbf{J}}^{(k-1)}\boldsymbol{\delta}\mathbf{x}^{(k)}\right) + \lambda(\boldsymbol{\delta}\mathbf{x}^{(k)})^T\boldsymbol{\delta}\mathbf{x}^{(k)} = 0.$$

It follows after substituting in the constraint equation (8.12) that

$$\lambda = -\frac{2(\boldsymbol{\delta}\mathbf{F}^{(k)} - \tilde{\mathbf{J}}^{(k-1)}\boldsymbol{\delta}\mathbf{x}^{(k)})}{(\boldsymbol{\delta}\mathbf{x}^{(k)})^T\boldsymbol{\delta}\mathbf{x}^{(k)}}$$

Therefore, from (8.11) we have that

$$\tilde{\mathbf{J}}^{(k)} = \tilde{\mathbf{J}}^{(k-1)} + \frac{(\boldsymbol{\delta}\mathbf{F}^{(k)} - \tilde{\mathbf{J}}^{(k-1)}\boldsymbol{\delta}\mathbf{x}^{(k)})}{(\boldsymbol{\delta}\mathbf{x}^{(k)})^T\boldsymbol{\delta}\mathbf{x}^{(k)}}(\boldsymbol{\delta}\mathbf{x}^{(k)})^T \tag{8.13}$$

This method, called *Broyden's method*, requires only n evaluations of $\mathbf{F}^{(k)}$ at each step to update the slope rather than n^2 evaluations that are necessary to recompute the Jacobian matrix.

The outline of the algorithm is as follows:

1. Guess $\mathbf{x}^{(0)}$ and determine $\mathbf{J}^{(0)}$
2. Take a Newton step to get $\mathbf{x}^{(1)}$

3. Using $\mathbf{x}^{(k)}$, $\mathbf{x}^{(k-1)}$, $\mathbf{F}^{(k)}$, $\mathbf{F}^{(k-1)}$ and $\tilde{\mathbf{J}}^{(k-1)}$:
   a) Compute $\tilde{\mathbf{J}}^{(k)}$ using (8.13)
   b) Complete a Newton step (8.9b) and (8.9c) using $\tilde{\mathbf{J}}^{(k)}$.

Rather than inverting $\tilde{\mathbf{J}}^{(k)}$ at each iteration, we can use the *Sherman–Morrison formula*

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

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

$$(\tilde{\mathbf{J}}^{(k)})^{-1} = (\tilde{\mathbf{J}}^{(k-1)})^{-1} + \frac{\boldsymbol{\delta}\mathbf{x}^{(k)} - (\tilde{\mathbf{J}}^{(k-1)})^{-1}\boldsymbol{\delta}\mathbf{F}^{(k)}}{\boldsymbol{\delta}(\mathbf{x}^{(k)})^T\tilde{\mathbf{J}}^{(k-1)}\boldsymbol{\delta}\mathbf{F}^{(k)}}\boldsymbol{\delta}(\mathbf{x}^{(k)})^T(\tilde{\mathbf{J}}^{(k-1)})^{-1}.$$

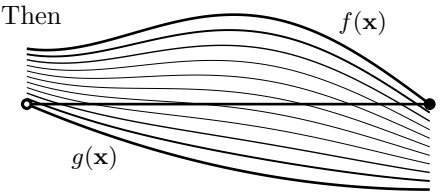
8.8 Homotopy continuation

Suppose that we want to find the zeros of $f(\mathbf{x})$. One might wonder if we could solve a simpler problem $g(\mathbf{x}) = 0$ as a means to solve $f(\mathbf{x}) = 0$. 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 n simpler system $g(\mathbf{x}) = 0$ for which the roots are known or can be found easily. One way to define a homotopy is

$$h(t, \mathbf{x}) = tf(\mathbf{x}) + (1-t)g(\mathbf{x}) \quad \text{for } t \in [0, 1].$$

Then



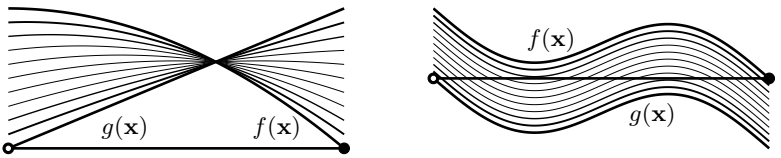
$$\begin{aligned} &\text{at } t = 0: \quad h(0, \mathbf{x}) = g(\mathbf{x}) = 0 \\ &\text{at } t = 1: \quad h(1, \mathbf{x}) = f(\mathbf{x}) = 0. \end{aligned}$$

Simply put, when the parameter $t = 0$, the solution $\mathbf{x}$ is a zero of g; and when $t = 1$, $\mathbf{x}$ 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 $t \in [0, 1]$. And to do this we may need to solve a nonlinear differential equation.

Take the homotopy

$$h(t, \mathbf{x}(t)) = 0 \tag{8.14}$$

where $\mathbf{x}(t)$ is a zero of the system h for the parameter t. We assume that $\mathbf{x}(t)$ exists and is unique for all $t \in [0, 1]$. Otherwise, the method is ill-posed. This is not a trivial assumption, because 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 (8.14) with respect to t gives us

$$\frac{\partial h}{\partial t} + \frac{\partial h}{\partial \mathbf{x}} \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = 0$$

where $\partial h/\partial \mathbf{x}$ is a Jacobian matrix and $\partial h/\partial t$ and $\mathrm{d}\mathbf{x}/\mathrm{d}t$ are both vectors. Therefore,

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = -\left[\frac{\partial h}{\partial \mathbf{x}}\right]^{-1} \frac{\partial h}{\partial t}.$$

We then solve this differential equation to get $\mathbf{x}(1)$.

A simple homotopy can be made by choosing

$$g(\mathbf{x}) = f(\mathbf{x}) - f(\mathbf{x}_0)$$

for some $\mathbf{x}_0$. Then $\mathbf{x}_0$ is a known zero of g. In this case

$$\begin{aligned} h(t, \mathbf{x}) &= tf(\mathbf{x}) + (t-1)[f(\mathbf{x}) - f(\mathbf{x}_0)] \\ &= f(\mathbf{x}) + (t-1)f(\mathbf{x}_0). \end{aligned} \tag{8.15}$$

The Jacobian matrix $\partial h/\partial \mathbf{x} = \partial f/\partial \mathbf{x}$ and $\partial h/\partial t = f(\mathbf{x}_0)$. In this case, we need to solve the ODE

$$\frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = -\left[\frac{\partial f}{\partial \mathbf{x}}\right]^{-1} f(\mathbf{x}_0).$$

at $t = 1$ given the initial conditions $\mathbf{x}_0$. To solve such an ODE is not typically trivial and often requires a numerical ODE solver.

Example. Suppose that we want to find the zeros of $f(\mathbf{x}) = 0$:

$$\begin{aligned} x^3 - 3xy^2 - 1 &= 0 \\ y^3 - 3x^2y &= 0. \end{aligned}$$

Define the homotopy as in (8.15) where we can take $\mathbf{x}_0$ to be what ever we want. Let's take $\mathbf{x}_0 = (1, 1)$. Then $f(\mathbf{x}_0) = (-3, -2)$. The Jacobian matrix is

$$\frac{\partial h}{\partial \mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}} = \partial_i f_j = \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ -6xy & 3y^2 - 3x^2 \end{bmatrix}$$

We need to solve the ODE

$$\frac{\mathrm{d}}{\mathrm{d}t} \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = - \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ -6xy & 3y^2 - 3x^2 \end{bmatrix}^{-1} \begin{bmatrix} -3 \\ -2 \end{bmatrix}$$

with initial conditions $(x(0), y(0)) = (1, 1)$. We can solve this problem in MATLAB using the following code

```
df = @(t,x) -inv([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])*[-3;-2];
[t,y] = ode45(df,[0;1],[1;1]);
```

The numerical solution given by `y(end,:)` is $(1.000, 0.000)$. Unfortunately, but understandably, the ODE solver has introduced numerical error of about 10^{-6}. To complete the problem, we apply Newton's method to $f(\mathbf{x})$ with the initial guess $\mathbf{x}^{(0)}$ given by `y(end,:)` to get the root $(-1, 0)$. See Figure 8.6 on page 189. ◀

Example. Define the homotopy

$$h(t, x) = f(x) - \mathrm{e}^{-t} f(x_0)$$

where $t \in [0, \infty)$. We want to find the path $x(t)$ which are roots of $h(t, x)$. That is, find $x(t)$ such that $h(t, x(t)) = 0$.

$$\begin{aligned} &\text{at } t = 0: && h(0, x) = f(x(0)) - f(x_0) \\ &\text{at } t = \infty: && h(\infty, x) = f(x(\infty)). \end{aligned}$$

Differentiating with respect to t gives us

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t} h(t, x) &= \frac{\partial f}{\partial x} \frac{\mathrm{d}x}{\mathrm{d}t} + \mathrm{e}^{-t} f(x_0) \\ &= \frac{\partial f}{\partial x} \frac{\mathrm{d}x}{\mathrm{d}t} + f(x(t)) = 0 \end{aligned}$$

where $\partial f/\partial x$ is the Jacobian matrix. Solving this equation for $\mathrm{d}x/\mathrm{d}t$ gives us

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\left[\frac{\partial f}{\partial x}\right]^{-1} f(x(t)).$$

Euler's method is perhaps the simplest way to solve this differential equation. We can approximate the derivative

$$\frac{\mathrm{d}x}{\mathrm{d}t} \approx \frac{x(t+h) - x(t)}{\Delta t}$$

for some step size Δt. Then

$$x^{(k+1)} = x^{(k)} - \Delta t \left[\left.\frac{\partial f}{\partial x}\right|_{x^{(k)}}\right]^{-1} f(x^{(k)}).$$

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

$$x^{(k+1)} = x^{(k)} - \left[\left.\frac{\partial f}{\partial x}\right|_{x^{(k)}}\right]^{-1} f(x^{(k)}).$$ ◀

Exercises

8.1. Steffensen's method

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{q(x^{(k)})} \quad \text{with} \quad q(x) = \frac{f(x + f(x)) - f(x)}{f(x)}$$

is a variation on Newton's method that uses $q(x)$ as approximation to the slope $f'(x)$. Show that this method is quadratically convergent under suitable conditions.

✍ 8.2. 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 an positive integer and $(1/f)^{(p)}$ denotes the pth 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 the Babylonian method for calculating a square root to get a method with cubic order convergence.

8.3. Use the Newton–Horner method to compute $x^{(1)}$ given $x^{(0)} = 4$ for the polynomial: $p(x) = 3x^5 - 7x^4 - 5x^3 + x^2 - 8x + 2$.

✍ 8.4. A matrix has the same spectrum as its transpose. So, we could also use the transpose of a companion matrix $\mathbf{A}$ to find the roots of a monic polynomial $p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1}x^{n-1} + x^n$. Suppose that $p(x)$ has n distinct roots $\{r_1, r_2, \ldots, r_n\}$. Find the eigenvectors of $\mathbf{A}^T$ (the left eigenvectors of the companion matrix).

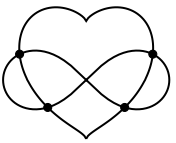
✍ 8.5. Suppose that we want to find the roots of $f(x) = 0$. Consider $x = x - \alpha f(x)$ for some α, to get the fixed-point method

$$x^{(k+1)} = \phi(x^{(k)})$$

where $\phi(x) = x - \alpha f(x)$. How does α affect stability and convergence of the method?

✍ 8.6. Use Newton's method, Broyden's method, and homotopy continuation to find the intersection of

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



CHAPTER 9

Interpolation

Often we want to

- Visualize computational results where $f(x) \approx \varphi(x)$.
- Compute an integral of f when computing the integral of φ is easier.
- Solve a differential equation.
- Fill in information missing in from data.

Often we take φ to be a polynomial. Other times φ may be a trigonometric polynomial, a rational function, or some other useful function. Let $\mathbb{P}_n$ be the space of polynomials of degree less than or equal to n. Take $p_n \in \mathbb{P}_n$. To approximate f by p_n we can use

1. Interpolation: Find the polynomial $p_n \in \mathbb{P}_n$ that coincides with the function f at some given points (or *nodes*) $x_0, x_1, \ldots, x_n$.
2. Best approximation: Find the polynomial p_n is the closest element in $\mathbb{P}_n$ to f with respect to some norm. That is,

$$\|f - p_n\| = \inf_{q \in \mathbb{P}_n} \|f - q\|.$$

In this chapter we look at interpolation, and in the next chapter we will examine best approximation.

9.1 Review of linear algebra terms

A *vector space* is a nonempty set V that is closed under linear combinations (addition and scalar multiplication over a field $\mathbb{F}$). Typically, the

field is the real numbers $\mathbb{R}$ or the complex numbers $\mathbb{C}$. A nonempty set $W \subset V$ is a *subspace* if and only if W is a vector space over $\mathbb{F}$. Consider a system of n vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\} \in V$. The set of all linear combinations of the system *generates* a subspace W of V. We call W the *span* of system and denote it by

$$W = \text{span}\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}.$$

The system $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$ is called the *spanning set* or the *generators* of W. The system of vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_m\}$ of a vector space V is said to be *linearly independent* if all linear combinations are unique. That is,

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

if and only if all of the scalars $a_1, a_2, \ldots, a_m$ are identically zero. Otherwise, we say that the system is *linearly dependent*. A *basis* of V is a system of linear independent generators of V. The number of elements in a basis of a vector space V is called the *dimension* of V. A vector space is called infinite dimensional, if for any n, there always exists linearly independent vectors of V. If $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\}$ is a basis of V, then any vector in $\mathbf{v} \in V$ has a unique decomposition with respect to the basis:

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

The strategy of function approximation is to choose a finite dimensional subspace to the space of functions, determine an appropriate basis to represent this subspace, and find the "closest" element (vector) in this subspace to the original function. Commonly used subspaces include polynomials, piecewise polynomials (splines), trigonometric functions (Fourier polynomials), exponential functions, and rational functions. In this chapter we will concentrate on polynomials and splines. We will save Fourier polynomials to the next chapter when we discuss orthogonal bases.

9.2 Polynomial interpolation

Consider the $n+1$-dimensional subspace of polynomials $\mathbb{P}_n$. Using the basis element $\phi_k(x) \in \mathbb{P}_n$, we can uniquely express

$$p_n(x) = \sum_{k=0}^{n} c_k\phi_k(x).$$

Our immediate goal is to determine the constants $c_0, c_1, \ldots, c_n$ such that $p_n(x_i) = y_i$ given $x_0, x_1, \ldots, x_n$ and values $y_0, y_1, \ldots, y_n$. That is, find

the coefficients c_k such that

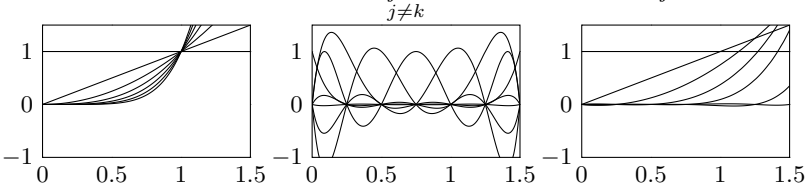
$$\sum_{k=0}^{n} c_k \phi_k(x_i) = y_i.$$

In matrix form, this says

$$\begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \phi_2(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_n(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \vdots & & \vdots \\ \phi_0(x_n) & \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}. \tag{9.1}$$

There are several important polynomial bases we can consider:

Canonical basis	Langrange basis	Newton basis
$\varphi_k(x) = x^k$	$\ell_k(x) = \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j}$	$\nu_k(x) = \prod_{j=0}^{k-1} (x - x_j)$



The plots above show polynomial bases for $j = 0, \ldots, 6$ with nodes at $x_j = j/4$. We can build any degree-n polynomial by taking a linear combination of using elements from any of these bases. Note in particular that the Newton basis can be defined recursively $\nu_k(x) = (x - x_k)\nu_{k-1}(x)$. We'll see this is a major advantage of the Newton polynomial basis. Before discussing the bases in depth, let's prove the uniqueness of a polynomial interpolating function. One way to prove the following theorem is to show that the matrix in (9.1) using the canonical basis (called a Vandermonde matrix) is nonsingular. This approach is left as an exercise. Here we will prove it using Newton bases.

Theorem 25. *Given $n+1$ distinct points $x_0, x_1, \ldots, x_n$ and arbitrary values $y_0, y_1, \ldots, y_n$, there exists a unique polynomial $p_n \in \mathbb{P}_n$ such that $p_n(x_i) = y_i$ for all $i = 0, \ldots, n$.*

Proof. We'll prove existence using induction. For $n = 0$, choose $p_0(x) = y_0$. Now, suppose that there is some p_{k-1} such that $p_{k-1}(x_i) = y_i$ for all

$i = 0, \ldots, k-1$. 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 $i = 0, \ldots, k-1$. To determine the value c for which $p_k(x_k) = y_k$, take c that solves

$$y_k = p_{k-1}(x_k) + c(x_k - x_0)(x_k - x_1)\cdots(x - x_{k-1}).$$

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 $i = 0, \ldots, n$. 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, $p_n \equiv q_n$. $\square$

Canonical basis

Take the canonical basis $\{1, x, \ldots, x^n\}$ and consider the problem of fitting an nth degree polynomial $y = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$ to data $(x_0, y_0), (x_2, y_2), \ldots, (x_n, y_n)$. Given $n+1$ distinct points, we can determine a nth-degree polynomial by solving the *Vandermonde* system $\mathbf{Vc} = \mathbf{y}$. From (9.1),

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Solving the system takes $O(n^3)$ operations, and numerically the Vandermonde matrix is increasingly ill-conditioned as the degree increases.

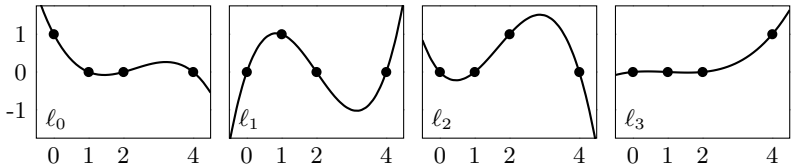
M The MATLAB function`vander` generates a Vandermonde matrix with rows $\begin{bmatrix} x_i^n & x_i^{n-1} & \cdots & 1 \end{bmatrix}$. Add `fliplr` to reverse them $\begin{bmatrix} 1 & \cdots & x_{i-1}^{n-1} & x_i^n \end{bmatrix}$.

Langrange polynomial basis

For a Lagrange polynomial

$$y_j = \sum_{i=0}^{n} y_i \ell_i(x_j) \quad \text{for} \quad \ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

That is, $\ell_i(x_j) = \delta_{ij}$. For a quick confirmation of this, note where $l_i(x)$ 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

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

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 Langrange polynomial requires $O(n^2)$ additions and multiplications.
- They are difficult to differentiate and integrate.
- If you add new data pair (x_n, y_n), you must start over from scratch.
- The computations can be numerically unstable.

Newton polynomial basis

The Newton polynomial basis fixes each of these limitations of the Lagrange form by recursively defining the basis elements $\nu_i(x)$:

$$1,\ (x - x_0),\ (x - x_0)(x - x_1),\ \cdots,\ \prod_{i=0}^{n-1}(x - x_i).$$

Defining $f(x_i) = y_i$ and using the Newton basis, (9.1) becomes

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & (x_1 - x_0) & 0 & \cdots & 0 \\ 1 & (x_2 - x_0) & \prod_{i=0}^{1}(x_2 - x_i) & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & (x_n - x_0) & \prod_{i=0}^{1}(x_n - x_i) & \cdots & \prod_{i=0}^{n-1}(x_n - x_i) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}.$$

The lower diagonal system can be solved using forward substitution. Note that the coefficient c_i is only a function of $x_0, \ldots, x_i$. To denote this dependency we will use the notation $c_i = f[x_0, x_1, \ldots, x_i]$. For example, using forward substitution to find $\{c_0, c_1, c_2\}$ for three nodes

$$\begin{aligned}
f[x_0] = c_0 &= f(x_0) \\
f[x_0, x_1] = c_1 &= \frac{f(x_1) - c_0}{x_1 - x_0} = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\
f[x_0, x_1, x_2] = c_2 &= \frac{f(x_2) - c_0 - c_1(x_2 - x_0)}{(x_2 - x_1)(x_2 - x_0)} \\
&= \frac{\dfrac{f(x_2) - f(x_0)}{x_2 - x_0} - \dfrac{f(x_0) - f(x_1)}{x_0 - x_1}}{x_2 - x_1}
\end{aligned}$$

Note that c_2 is the coefficient for x^2 term of $p_2(x)$. Because $p_2(x)$ is unique by Theorem 25, 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 that

$$f[x_0, x_1, x_2] = \frac{\dfrac{f(x_2) - f(x_1)}{x_2 - x_1} - \dfrac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0} = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}.$$

In general, we can compute the coefficients efficiently using *divided differences*. The divided difference $f[x_0, \cdots, x_{k-1}]$ can be computed using the recursion

$$\begin{aligned}
f[x_i] &= f(x_i) \\
f[x_j, \cdots, x_k] &= \frac{f[x_{j+1}, \cdots, x_k] - f[x_j, \cdots, x_{k-1}]}{x_k - x_j}.
\end{aligned}$$

It's convenient to arrange the divided differences in a tree for calculation:

$$\begin{array}{cc|ccc}
x_0 & f[x_0] & f[x_0, x_1] & f[x_0, x_1, x_2] & f[x_0, x_1, x_2, x_3] \\
x_1 & f[x_1] & f[x_1, x_2] & f[x_1, x_2, x_3] & \\
x_2 & f[x_2] & f[x_2, x_3] & & \\
x_3 & f[x_3] & & &
\end{array}$$

Then the coefficients c_i come from 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

$$p_n(x) = c_0 + (x - x_0)\left[c_1 + (x - x_1)\left[c_2 + (x - x_2)\left[c_3\right]\right]\right],$$

which takes $O(n)$ additions and multiplications.

Example. Let's find the interpolating polynomial for the function $f(x)$ where

$$\begin{array}{rcccc} x = & 1 & 3 & 5 & 6 \\ f(x) = & 0 & 4 & 5 & 7 \end{array}$$

We first build the divided difference table. From

$$\begin{array}{cc|ccc} 1 & 0 & \square & \square & \square \\ 3 & 4 & \square & \square & \\ 5 & 5 & \square & & \\ 6 & 7 & & & \end{array} \quad \text{we have that} \quad \begin{array}{cc|ccc} 1 & 0 & 2 & -3/8 & 7/40 \\ 3 & 4 & 1/2 & 1/2 & \\ 5 & 5 & 2 & & \\ 6 & 7 & & & \end{array}.$$

So, the interpolating polynomial is

$$p_n(x) = \mathbf{0} + \mathbf{2}(x-1) - 3/8(x-1)(x-3) + 7/40(x-1)(x-3)(x-5)$$

which can be evaluated using Horner's method

$$p_n(x) = \mathbf{0} + (x-1)\left[\mathbf{2} + (x-3)\left[-3/8 + (x-5)\left[7/40\right]\right]\right]. \qquad \blacktriangleleft$$

9.3 How good is polynomial interpolation?

Theorem 26 (Polynomial Interpolation Error)**.** *Let $f \in C^{n+1}$ and let $p_n \in \mathbb{P}_n$ interpolate f at $n+1$ points $x_0, \ldots, x_n$ on the interval $[a,b]$. Then for each $x \in [a,b]$ there is a $\xi \in [a,b]$ such that the pointwise error*

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^{n} (x - x_i).$$

Proof. If x is a node, then $f(x) - p_n(x) = 0$, and the theorem is clearly true. So, let x be any point other than a node. Since x is now fixed, we will need to choose another variable, say t, for the range to avoid confusion. Define

$$g(t) = f(t) - p_n(t) - (f(x) - p_n(x)) \left(\frac{\prod_{i=0}^{n} t - x_i}{\prod_{i=0}^{n} x - x_i} \right).$$

Note that x is now a parameter of g. Also, note that $g \in C^{n+1}([a,b])$ and

$$g(x) = g(x_0) = \cdots = g(x_n) = 0.$$

That is,

$$g \text{ has at least } n+2 \text{ roots on } [a,b].$$

By Rolle's theorem,

$$\begin{aligned} g' &\text{ has at least } n+1 \text{ roots on } [a,b]. \\ g'' &\text{ has at least } n \text{ roots on } [a,b]. \\ &\cdots\cdots\cdots\cdots\cdots\cdots \\ g^{(n+1)} &\text{ has at least one zero on } [a,b]. \text{ Let's call this zero } \xi. \end{aligned}$$

We can explicitly compute the $(n+1)$-derivative of g:

$$g^{(n+1)}(t) = f^{(n+1)}(t) - \underbrace{p^{(n+1)}(t)}_{0} - \frac{f(x) - p_n(x)}{\prod_{i=0}^{n} x - x_i}(n+1)!.$$

where $p_n^{(n+1)}(t) = 0$ because p_n has degree at most n. At $t = \xi$

$$0 = g^{(n+1)}(\xi) = f^{(n+1)}(\xi) - \frac{f(x) - p_n(x)}{\prod_{i=0}^{n} x - x_i}(n+1)!.$$

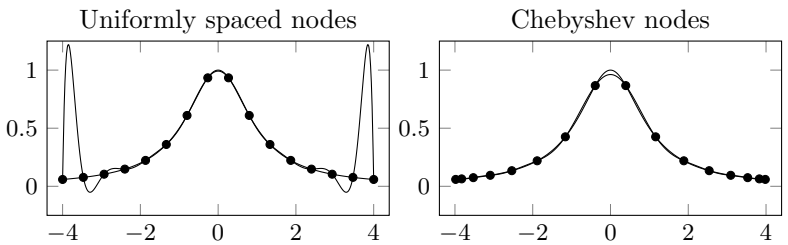
So it follows that

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^{n} (x - x_i). \qquad \square$$

Example. Suppose that we approximate $f(x) = \sin x$ on the interval $[0, \pi]$ using a degree-9 polynomial. Since $|f^{(\xi)}| \le 1$ and $\prod_{i=0}^{9} |x - x_i| \le 1$, the error

$$\|f(x) - p_9(x)\|_\infty \le \frac{1}{(n+1)!} = 2.7 \times 10^{-7}. \qquad \blacktriangleleft$$

Example. If use uniform nodes to construct a polynomial interpolant for the smooth function $f(x) = 1/(x^2+1)$ using a polynomial interpolant over the interval $[-4, 4]$ we get a large oscillations in near the end points.



This behavior is called *Runge's phenomenon* and shows us that simply adding increasing the order of a polynomial does not necessarily improve accuracy. By Theorem 26, for all $x \in [a, b]$ we have the upper bound

$$|e_n(x)| \leq \frac{1}{(n+1)!} \left\| \prod_{i=0}^{n} (x - x_i) \right\|_\infty \|f^{(n+1)}\|_\infty.$$

The term $\|f^{(n+1)}\|_\infty$ depends on the function, so we don't have control over it. But we can choose the interpolation points to minimize the term $\| \prod_{i=0}^{n} (x - x_i)\|_\infty$. This is exactly what we do when we choose Chebyshev nodes. ◀

Chebyshev polynomial

Consider the interval $[-1, 1]$. We can use the affine transformation $\frac{1}{2}(1 - x)a + \frac{1}{2}(1 + x)b$ to map $[-1, 1]$ to an interval $[a, b]$. The Chebyshev polynomials are defined using the three-term recurrence relationship

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \end{aligned}$$

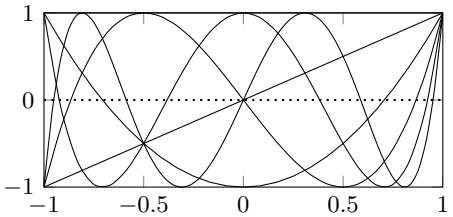
and can be expressed explicitly as

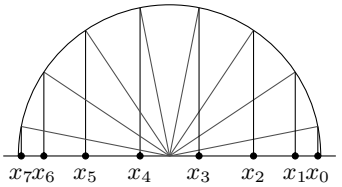
$$T_n(x) = \cos(n \arccos x) \quad \text{for } x \in [-1, 1].$$

Note that

1. The leading coefficient of T_n is $a_n = 2^{n-1}$.
2. $T_n(1) = 1$, $T_n(-1) = (-1)^n$
3. $|T_n(x)| \leq 1$ for $x \in [-1, 1]$
4. $|T_n(\tilde{x}_k)| = 1$ at the *Chebyshev abscissas* $\tilde{x}_k = \cos \frac{k\pi}{n}$
5. The roots of $T_n(x_k)$ are $x_k = \cos \frac{(k-1/2)\pi}{n}$ for $k = 1, \ldots, n$.

The Chebshev nodes are the abscissas of equispaced nodes on the unit-semicircle. On the left are Chebyshev polynomals $T_0, \ldots, T_5$ and on the right are the Chebyshev roots for T_8 are the abscissas of eight equispaced nodes on the unit-semicircle:





Theorem 27. *Every polynomial $p_n \in \mathbb{P}_n$ with leading coefficient $a_n \neq 0$ attains a maximum value of absolute value at least $|a_n|/2^{n-1}$ in the interval $[-1,1]$. Among the polynomials of degree n with leading coefficient 2^{n-1}, the Chebyshev polynomials $T_n(x)$ have the smallest maximum-norm $\|p_n\|_\infty = \max_{x\in[-1,1]} |p_n(x)|$.*

Proof. Let $p_n \in \mathbb{P}_n$ be a polynomial with leading coefficient 2^{n-1} and $|p_n(x)| < 1$ for $x \in [-1,1]$. Then $T_n - p_n$ is a polynomial of degree $n-1$ or less. Recall that $T_n(\tilde{x}_k) = (-1)^k$ at the Chebyshev abscissas $\tilde{x}_k = \cos\frac{k\pi}{n}$. For

$$\begin{aligned} k \text{ even:}\quad & T_n(\tilde{x}_k) = +1 \text{ and } p_n(\tilde{x}_k) < 1 \qquad \text{then } p_n(\tilde{x}_k) - T_n(\tilde{x}_k) < 0; \\ k \text{ odd:}\quad & T_n(\tilde{x}_k) = -1 \text{ and } p_n(\tilde{x}_k) > -1 \quad \text{then } p_n(\tilde{x}_k) - T_n(\tilde{x}_k) > 0. \end{aligned}$$

So, at the $n+1$ Chebyshev abscissas, the polynomial $p_n - T_n$ is alternating between positive and negative and therefore it has at least n roots in $[-1,1]$. But $p_n - T_n \in \mathbb{P}_{n-1}$ and can have at most $n-1$ roots, a contradiction. It follows that for each $p_n \in \mathbb{P}_n$ with leading coefficient 2^{n-1}, there must be some $x \in [-1,1]$ such that $|p_n(x)| \geq 1$. □

When trying to minimize the pointwise error for a function f

$$\begin{aligned} |f(x) - p_n(x)|_\infty &= \sup_{x\in[-1,1]} |f(x) - p_n(x)| \\ &\leq \left| \frac{1}{(n+1)!} f^{(n+1)}(\xi) \right| \sup_{x\in[-1,1]} \left| \prod_{i=0}^{n} (x - x_i) \right|, \end{aligned}$$

the best (only) thing to do is choose the nodes so that

$$\sup_{x\in[-1,1]} \left| \prod_{i=0}^{n} (x - x_i) \right|$$

is minimized. From Theorem 27, it follows that because the scaled Chebyshev polynomial has the smallest max norm on $[-1,1]$, we should choose x_i to coincide with the roots of $T_n(x)$.

In an interval $[a, b]$ we can use an affine transformation to reposition the Chebyshev nodes

$$x_i = \frac{a+b}{2} - \frac{b-a}{2}\cos\frac{\pi i}{n} \quad \text{for } i = 0, \ldots, n.$$

9.4 Hermite interpolation

We can extend our previous discussion to find interpolating polynomials that not only match the values of functions at nodes but also match their derivatives. That is, there is a unique polynomial $p_n \in \mathbb{P}_n$ such that for all $i = 0, \cdots, k$ and $j = 0, \cdots, \alpha_i$:

$$p_n^{(j)}(x_i) = f^{(j)}(x_i)$$

The polynomial p_n is called the *Hermite polynomial* interpolation of f at the points x_i.

Theorem 28. *Assume the function f is in $C^{n+1}([a, b])$. For all $x \in [a, b]$, there exists $\xi \in [\min_i x_i, \max_i x_i]$ such that*

$$e_n(x) = f(x) - p_n f(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^{k} (x - x_i)^{1+\alpha_i}.$$

Example. (Taylor polynomial) Find a polynomial that at x_0 has the value $f(x_0)$ and has the derivatives $f'(x_0)$, $f^{(2)}(x_0), \ldots, f^{(n)}(x_0)$. Let's write the polynomial as $p_n(x) = c_0 + c_1(x - x_0) + \cdots + c_n(x - x_0)^n$ By matching derivatives we have that

$$\begin{aligned}
p_n(x_0) &= c_0 = f(x_0) \\
p_n'(x_0) &= c_1 = f'(x_0) \\
p_n''(x_0) &= 2c_2 = f''(x_0) \\
&\vdots \\
p_n^{(n)}(x_0) &= (n!)c_n = f^{(n)}(x_0)
\end{aligned}$$

Hence, we have the Taylor polynomial

$$\begin{aligned}
p_n(x) =& f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2 + \cdots \\
& \cdots + \frac{1}{n!} f^{(n)}(x_0)(x - x_0)^n.
\end{aligned}$$

◀

In general, we can compute the Newton form of the Hermite interpolating polynomial by using a table of divided differences. We use the fact that by the mean value theorem

$$f[x_0, x_1, \cdots, x_n] = \frac{1}{n!} f^{(n)}(\xi).$$

Hence,

$$f[x_0, x_0, \cdots, x_0] = \lim_{x_1, \cdots, x_n \to x_0} f[x_0, x_1, \cdots, x_n] = \frac{1}{n!} f^{(n)}(x_0).$$

Example. Compute the polynomial interpolant p of minimal degree satisfying

$$p(1) = 2, p'(1) = 3, \quad p(2) = 6, \quad p'(2) = 7, \quad p''(2) = 8$$

for some m. To do this we compute the divided differences. First note that $p[1,1] = p'(1) = 3$, $p[2,2] = p'(2) = 7$ and $p[2,2,2] = \frac{1}{2!}p''(2) = \frac{8}{2} = 4$. Then

$$\begin{array}{cc|cccc} 1 & 2 & 3 & \square & \square & \square \\ 1 & 2 & \square & \square & \square & \\ 2 & 6 & 7 & 4 & & \\ 2 & 6 & \square & & & \\ 2 & 6 & & & & \end{array} \quad \Rightarrow \quad \begin{array}{cc|cccc} 1 & 2 & 3 & 1 & 2 & -1 \\ 1 & 2 & 4 & 3 & 1 & \\ 2 & 6 & 7 & 4 & & \\ 2 & 6 & \square & & & \\ 2 & 6 & & & & \end{array}$$

We get $p(x) = 2+3(x-1)+1(x-1)^2+2(x-1)^2(x-2)-1(x-1)^2(x-2)$. ◀

9.5 Splines

One of the problems with polynomial interpolation is that polynomials bases elements are global by design. Because of this, they are ill-suited to approximate discontinuous functions. An alternative is to use radial bases functions. See Exercise 9.3. Another approach is to use a piecewise polynomial functions called *splines*. In this section, we will concentrate on cubic splines. The ideas presented here can be extended to any degree spline.

Cubic splines

Let $x_0, \ldots, x_n$ be $n+1$ distinct nodes of $[a,b]$ with $a = x_0 < x_1 < \cdots < x_n = b$. Let $s(x) \in C^2[a,b]$ be a cubic spline relative to the points x_j (called *knots*). That is, for $x \in (x_j, x_{j+1})$ let $s_j(x)$ be a cubic

polynomial with $s_j(x_j) = y_j$. To determine $s(x)$ we must match the cubic polynomials $s_j(x)$ across the subintervals. A cubic polynomial has 4 degrees of freedom. Hence, we have 4 unknowns for each of the n intervals—a total of $4n$ unknowns. Now, lets look at the constraints. The spline is interpolating $s_j(x_j) = y_j$ providing $n+1$ constraints. Because $s(x) \in C^2$, it follows that

$$s_{j-1}(x_j) = s_j(x_j) \quad \text{and} \quad s'_{j-1}(x_j) = s'_j(x_j) \quad \text{and} \quad s''_{j-1}(x_j) = s''_j(x_j)$$

for $j = 1, \ldots, n-1$. We have a total of $n+1+3(n-1) = 4n-2$ constraints. This leaves us with 2 degrees of freedom remaining. Therefore, we need to enforce additional constraints to ensure a unique solution. The following constraints are called (1) complete, (2) natural, (3) periodic, and (4) "not-a-knot":

1. $s'(a) = f'(a)$ and $s'(b) = f'(b)$.
2. $s''(a) = s''(b) = 0$.
3. $s'(a) = s'(b)$ and $s''(a) = s''(b)$ if f is periodic.
4. Force two consecutive pieces to be the same, e.g., $s'''_1(x_1) = s'''_1(x_1)$ and $s'''_n(x_{n-1}) = s'''_n(x_{n-1})$.

Before computer-aided graphic design programs, engineers used thin strips of wood called splines to help draw curves. The thin beams were fixed to the knots x_i and either clamped at the boundary to prescribe the slope (complete splines) or unclamped so that the beam would be straight outside the interval $[a, b]$ (natural splines). If $y(x)$ describes the position of the thin wood beam, then the curvature is given by

$$K(x) = \frac{y''(x)}{(1 + y'(x)^2)^{3/2}}$$

and "deformation energy" of the beam is

$$E = \int_a^b K^2(x)\,\mathrm{d}x = \int_a^b \left(\frac{y''(x)}{(1 + y'(x)^2)^{3/2}} \right)^2 \mathrm{d}x$$

which for small deformations

$$E \approx \int_a^b \left[y''(x)\right]^2 \mathrm{d}x = \|y''\|_2^2.$$

Physically, the beam takes the shape so that the energy is minimized. This implies that the cubic spline is the smoothest interpolating polynomial.

Theorem 29. *Let s be an interpolating cubic spline of f at the nodes $a = x_0 < \cdots < x_{i+1} = b$ and let $y \in C^2[a,b]$ be an arbitrary interpolating function on f. Then*

$$\|s''\|_2 \le \|y''\|_2$$

if the spline has complete ($s'(a) = y'(a), s'(b) = y'(b)$), natural ($s''(a) = s''(b) = 0$) or periodic boundary conditions ($s'(a) = s'(b)$, $s''(a) = s''(b)$ and y is periodic).

Proof. Note that $y'' = s'' + (y'' - s'')$ and hence

$$\int_a^b (y'')^2 \,\mathrm{d}x = \int_a^b (s'')^2 \,\mathrm{d}x + \underbrace{2\int_a^b s''(y''-s'')\,\mathrm{d}x}_{(*)} + \underbrace{\int_a^b (y''-s'')^2\,\mathrm{d}x}_{\ge 0}.$$

If the term $(*)$ vanishes, then

$$\int_a^b (y'')^2 \,\mathrm{d}x \ge \int_a^b (s'')^2 \,\mathrm{d}x.$$

By integration by parts

$$\int_a^b s''(y''-s'')\,\mathrm{d}x = \underbrace{[s''(y'-s')]_a^b}_{=0 \text{ by assumption}} - \int_a^b s'''(y'-s')\,\mathrm{d}x.$$

The function s''' is (in general) discontinuous as the nodes $x_1, \ldots, x_{n-1}$. On the subinterval (x_i, x_{i+1}) the cubic spline s is a cubic polynomial s_i, and $s'''(x) = s_i'''(x) = d_i$ for some constant d_i.

$$\begin{aligned}
\int_a^b s''(y''-s'')\,\mathrm{d}x &= -\sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} d_i(y'-s_i')\,\mathrm{d}x \\
&= -\sum_{i=0}^{n-1} d_i \int_{x_i}^{x_{i+1}} y' - s_i'\,\mathrm{d}x \\
&= -\sum_{i=0}^{n-1} d_i \Big[\underbrace{(y(x_{i+1}) - s(x_{i+1}))}_{=0} - \underbrace{(y(x_i) - s(x_i))}_{=0}\Big] \\
&= 0.
\end{aligned}$$

□

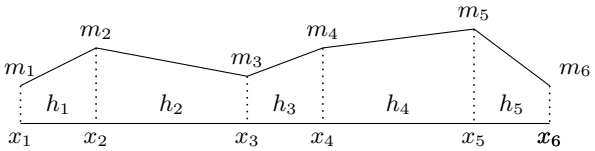
Computation of cubic splines

Let's find an efficient method to construct cubic spline interpolating values $f(x_i)$ for $a = x_0 < x_1 < \cdots < x_n = b$. Let $s(x)$ be a spline defined on

$[a, b]$ and take $s_i(x)$ to be the cubic polynomial on subinterval $[x_i, x_{i+1}]$. The second-order derivative on each subinterval is a straight line segment

$$s_i''(x) = m_{i+1}\frac{x - x_i}{h_i} + m_i\frac{x_{i+1} - x}{h_i}$$

where $h_i = x_{i+1} - x_i$ is the segment length and $m_i = s_3''(x_i)$:



We integrate twice to find $s_i(x)$

$$s_i(x) = \frac{1}{6}\frac{m_{i+1}}{h_i}(x - x_i)^3 + \frac{1}{6}\frac{m_i}{h_i}(x_{i+1} - x)^3 + A_i(x - x_i) + B_i$$

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 $s_i(x_{i+1}) = y_{i+1}$.

$$y_i = \frac{1}{6}\frac{m_i}{h_i}h^3 + B_i \quad \text{and} \quad y_{i+1} = \frac{1}{6}\frac{m_{i+1}}{h_i}h_i^3 + Ah_i + y_i - B_i$$

from which we have that

$$B_i = y_i - \frac{1}{6}m_i h_i^2 \quad \text{and} \quad A_i = \frac{y_{i+1} - y_i}{h_i} - \frac{1}{6}(m_{i+1} - m_i)h_i$$

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 $i = 1, \ldots, n - 1$. Substituting in for A_{i-1} and A_i, we have

$$s_{i-1}'(x_i) = \frac{1}{2}\frac{m_i}{h_{i-1}}(h_{i-1})^2 + \frac{y - y_{i-1}}{h_{i-1}} - \frac{1}{6}(m_i - m_{i-1})h_{i-1},$$
$$s_i'(x_i) = -\frac{1}{2}\frac{m_i}{h_i}(h_i)^2 + \frac{y_{i+1} - y_i}{h_i} - \frac{1}{6}(m_{i+1} - m_i)h_i.$$

Equating the two, we are left with

$$h_{i-1}m_{i-1} + 2(h_i + h_{i-1})m_i + h_i m_{i+1} = 6\left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right)$$

for $i = 1, \ldots, n-1$. 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

$$m_0 = 0 \quad \text{and} \quad m_n = 0.$$

Note that we can rewrite the $n+1$ equation system as an n equation system by eliminating the top and bottom equations from the system

$$\begin{bmatrix} 1 & & & & \\ a_1 & b_1 & c_1 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & 1 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ \vdots \\ m_{n-1} \\ m_n \end{bmatrix} = \begin{bmatrix} 0 \\ d_1 \\ \vdots \\ d_{n-1} \\ 0 \end{bmatrix}$$

we have

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-2} & b_{n-2} & b_{n-2} \\ & & & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_{n-2} \\ m_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{bmatrix}$$

where

$$a_i = h_{i-1},\ b_i = 2(h_i + h_{i-1}),\ c_i = h_i,\ d_i = 6\left(\frac{y_{i+1} - y_i}{h_i} - \frac{y_i - y_{i-1}}{h_{i-1}}\right).$$

We can write the following MATLAB function

```
function [X,Y] = splinex(x,y,N)
n = length(x);
h = diff(x);
C = diag(h(2:end-1),-1) + 2*diag(h(1:end-1)+h(2:end),0) ...
      + diag(h(2:end-1),1);
d = 6.*diff(diff(y)./h);
m = [0;C\d;0]
B = y(1:n-1) - m(1:n-1).*h.^2/6;
A = diff(y)./h-h./6.*[diff(m)];
X = linspace(min(x),max(x),N)';
i = sum(bsxfun(@lt,x,repmat(X,size(x))));
i(end) = n;
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);
```

The function takes a column vector input for x_i and y_i and outputs two column vectors `X` and `Y` containing N elements. Note:

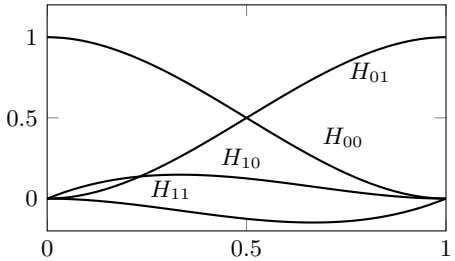


Figure 9.1: Bases of cubic Hermite spline on $[0, 1]$. Note $H_{11}(x) = -H_{10}(1 - x)$ and $H_{00}(x) = H_{01}(1 - x)$.

- The command `bsxfun` which allows one to perform binary operations on a matrix with a vector. An alternative to

```
i = sum(bsxfun(@lt,x,repmat(X,size(x))));
```

  is

```
[iX,ix] = meshgrid(X,x); i = sum(ix<=iX)';
```

- It is typically much faster to use MATLAB's sparse routines. To do this, replace

```
C = diag(h(2:end-1),-1) + 2*diag(h(1:end-1)+h(2:end),0) ...
      + diag(h(2:end-1),1);
```

  with

```
C = spdiags([h(2:end) 2*(h(1:end-1)+h(2:end)) ...
      h(1:end-1)],[-1 0 1],n-2,n-2);
```

Ⓜ The function `spline` returns a cubic spline with the not-a-knot condition.

Splines in parametric form

Consider a plane curve in parametric form $f(t) = (x(t), y(t))$ with $t \in [0, T]$. Take a set of points $f_i = (x_i, y_i)$ for $i = 0, \ldots, n$ and introduce the partition $0 = t_0 < t_1 < \cdots < t_n = T$. A natural way to parameterize the spline is to use the lengths of each segment

$$l_i = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}, \quad i = 1, \ldots, n$$

and set $t_0 = 0$ and $t_i = \sum_{k=1}^{i} l_k$. This function is called the *cumulative length spline* and is a good approximation as long as the curvature of $f(t)$ is small.

Cubic Hermite spline

Recall that a Hermite polynomial interpolant also uses the derivatives at the nodes to build polynomials. A cubic spline is uniquely determined by four bases elements on each subinterval. 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 $f'(x_{i+1})$. Let's 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 bases elements are given by (see Figure 9.1 on the previous page)

$$\begin{aligned}
H_{00}(x) &= 2x^3 - 3x^2 + 1 \\
H_{10}(x) &= x^3 - 2x^2 + x \\
H_{01}(x) &= -2x^3 + 3x^2 = H_{00}(1-x) \\
H_{11}(x) &= x^3 - x^2 = -H_{10}(1-x)
\end{aligned}$$

We can modify these bases to fit an arbitrary interval $[x_k, x_{k+1}]$ by taking

$$p(x) = p_0 H_{00}(t) + m_0 h_k H_{10}(t) + p_1 H_{01}(t) + m_1 h_k H_{11}(t)$$

where $h_k = x_{k+1} - x_k$; $t = (x - x_k)/h_k$; and $p_0 = y(x_k)$, $m_0 = y'(x_k)$, $p_1 = y(x_k)$, and $m_1 = y'(x_{k+1})$.

Ⓜ The MATLAB function `pchip` returns a cubic Hermite spline.

B-splines

An alternative approach to splines is building them using B-splines. A *B-spline* or *basis spline* has the properties:

1. each one has local support;
2. each one is nonnegative;
3. each one is piecewise polynomial of degree less than or equal to k with respect to interval (x_j, x_{j+1});
4. together they form a partition of unity.

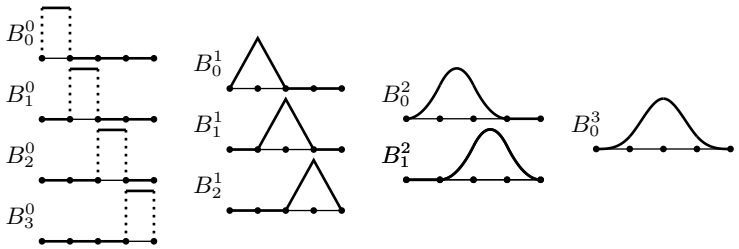


Figure 9.2: B-spline bases generated recursively by de Boor's algorithm.

Hence, degree-0 B-splines are piecewise constant, degree-1 B-splines are piecewise linear, degree-2 B-splines are piecewise quadratic, and so forth.

We can generate splines of any degree recursively starting with degree-0 splines using *de Boor's algorithm*. Let $a \leq x_0 < \ldots < x_n \leq b$. The degree-0 B-splines are *characteristic functions*

$$B_i^0(x) = \chi_{[x_i, x_{i+1})}(x) = \begin{cases} 1, & \text{if } x_i \leq x < x_{i+1} \\ 0, & \text{otherwise.} \end{cases}$$

Note that the set $B_i^0(x)$ forms a *partition of unity* of $[a, b]$, that is

$$\sum_{i=0}^{n} B_i^0(x) = 1 \quad \text{for } x \in [a, b].$$

We want to define higher degree B-splines that also form a partition of unity. To do this, we define $B_i^k(x)$ recursively

$$B_i^k(x) = \left(\frac{x - x_i}{x_{i+k} - x_i} \right) B_i^{k-1}(x) + \left(\frac{x_{i+k+1} - x}{x_{i+k+1} - x_{i+1}} \right) B_{i+1}^{k-1}(x)$$

which we can rewrite as

$$B_i^k = V_i^k B_i^{k-1} + (1 - V_{i+1}^k) B_{i+1}^{k-1}$$

with

$$V_i^k(x) = \frac{x - x_i}{x_{i+k} - x_i}.$$

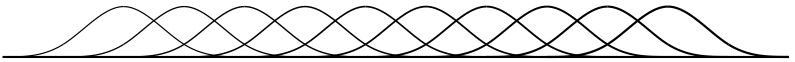
See Figure 9.2. The constant spline $B_i^0(x)$ has one subinterval for its nonzero support, the linear spline $B_i^1(x)$ has two subintervals for its nonzero support, the quadratic spline $B_i^2(x)$ has three subintervals for its nonzero support, and so forth. Note that the set $B_i^k(x)$ maintains the partition of unity.

Finally, suppose that the knots are equally spaced with separation h and a node at x_0. We can generate the cubic B-spline

$$B_i^3 = B\left(\frac{x - x_0}{h} - i\right)$$

by translating and stretching the B-spline

$$B(x) = \begin{cases} \frac{1}{6}(2+x)^3, & x \in [-2,1) \\ \frac{1}{6} + \frac{1}{2}(1+x) + \frac{1}{2}(1+x)^2 - \frac{1}{2}(1+x)^2, & x \in [-1,0) \\ \frac{1}{6} + \frac{1}{2}(1-x) + \frac{1}{2}(1-x)^2 - \frac{1}{2}(1-x)^2, & x \in [0,1) \\ \frac{1}{6}(2-x)^3, & x \in [1,2) \\ 0, & \text{otherwise.} \end{cases}$$



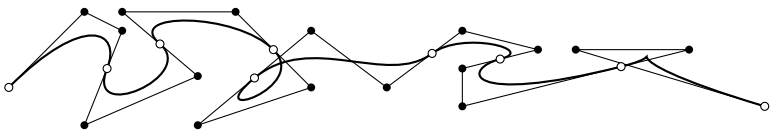
To evaluate a spline curve through the points $y_0, y_1, \ldots, y_n$:

$$s(x) = \sum_{i=0}^{n} y_i B_i(x).$$

Ⓜ The MATLAB 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 $k+1$ and `C` is an $k \times n$ array of the polynomial coefficients in each segment.

Bézier curves

Another way to create splines is by using composite Bézier curves. While splines are typically constructed by fitting polynomials through nodes, Bézier curves are constructed by using a set of control points to form a convex hull. Other than at the end points, a Bézier curve doesn't go through control points. A Bézier spline is built by matching position and slopes at the terminal control points (∘) of several of Bézier curves:



Bézier curves are all around us—just pick up a newspaper and you are bound to see Bézier curves. That's because composite Bézier curves are the mathematical encodings for glyphs used in virually every modern font from quadratic ones in TrueType fonts to the cubic ones in the Computer Modern font used in each letter of text you are reading right now.

A linear Bézier curve is a straight line segment through two control points $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$:

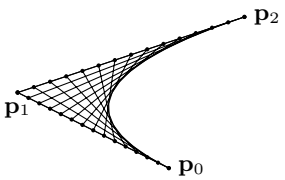
$$\mathbf{q}_{10}(t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1$$

where $t \in [0,1]$. Note that $\mathbf{q}_{10}(0) = \mathbf{p}_0$ and $\mathbf{q}_{10}(1) = \mathbf{p}_1$. By adding an additional control point $\mathbf{p}_2$, we can build a quadratic Bézier curve. First, define the linear Bézier curve

$$\mathbf{q}_{11}(t) = (1-t)\mathbf{p}_1 + t\mathbf{p}_2$$

where $t \in [0,1]$. Now, combine $\mathbf{q}_{10}$ and $\mathbf{q}_{11}$:

$$\mathbf{q}_{20}(t) = (1-t)\mathbf{q}_{10}(t) + t\mathbf{q}_{11}(t)$$



String art uses threads strung between nails hammered into a base board to create geometric patterns. You can think of the Bézier curve as the convex hull formed connecting threads from nails at $\mathbf{q}_{10}(t)$ to corresponding nails $\mathbf{q}_{11}(t)$. The threads or line segments are tangent the to the Bézier curve at $\mathbf{q}_{20}(t)$. If we expand the expression for $\mathbf{q}_{20}(t)$ we have

$$\mathbf{q}_{20}(t) = (1-t)^2\mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2\mathbf{p}_2.$$

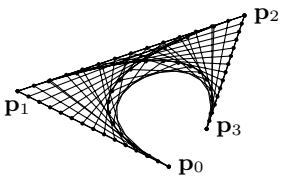
with a derivative

$$\begin{aligned}\mathbf{q}'_{20}(t) &= -2(1-t)\mathbf{p}_0 + 2(1-2t)\mathbf{p}_1 + 2t\mathbf{p}_2 \\ &= -2\left((1-t)\mathbf{p}_0 + t\mathbf{p}_1\right) + 2\left((1-t)\mathbf{p}_1 + t\mathbf{p}_2\right) \\ &= 2\left(\mathbf{q}_{11}(t) - \mathbf{q}_{10}(t)\right)\end{aligned}$$

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 $\mathbf{q}'_{20}(0) = 2(\mathbf{p}_2 - \mathbf{p}_1)$, which says that the Bézier curve is tangent to the line segment $\overline{\mathbf{p}_0\mathbf{p}_1}$ and $\overline{\mathbf{p}_1\mathbf{p}_2}$ at $\mathbf{p}_0$ and $\mathbf{p}_2$, respectively.

We can continue in the fashion by adding another control point $\mathbf{p}_3$ to build a cubic Bézier curve. Take

$$\begin{aligned}\mathbf{q}_{12}(t) &= (1-t)\mathbf{p}_2 + t\mathbf{p}_3 \\ \mathbf{q}_{21}(t) &= (1-t)\mathbf{q}_{11}(t) + t\mathbf{q}_{12}(t) \\ \mathbf{q}_{30}(t) &= (1-t)\mathbf{q}_{20}(t) + t\mathbf{q}_{21}(t)\end{aligned}$$



This time the 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.

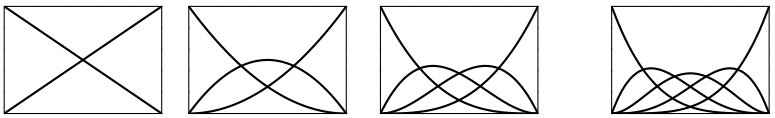
This recursive method of constructing the Bézier starting with linear Bézier curves then quadratic Bézier curves then cubic (and higher order) Bézier curves is called *de Casteljau's algorithm*. Expanding $\mathbf{q}_{30}$ we have the cubic *Bernstein polynomial*:

$$\mathbf{q}_{30}(t) = (1-t)^3\mathbf{p}_0 + 3t(1-t)^2\mathbf{p}_1 + 3t^2(1-t)\mathbf{p}_2 + t^3\mathbf{p}_3$$

In general, the n-degree Bernstein polynomial is given by

$$\mathbf{q}_{n0}(t) = \sum_{k=0}^{n} \binom{n}{k}(1-t)^{n-k}t^k\mathbf{p}_k = \sum_{k=0}^{n} B_{nk}(t)\mathbf{p}_k.$$

The bases $\{B_{n0}(t), B_{n1}(t), \ldots, B_{nn}(t)\}$ form a partition of unity:



B_{10}, B_{11} B_{20}, B_{21}, B_{22} $B_{30}, B_{31}, B_{32}, B_{33}$ $B_{40}, B_{41}.B_{42}, B_{43}, B_{44}$

Exercises

✍ 9.1. Prove Theorem 25 by using the canonical basis $\{1, x, \ldots, x^n\}$ to construct a Vandermonde matrix. Show that if $\{x_0, \ldots, x_n\}$ are distinct, then the Vandermonde matrix $\mathbf{V}$ is nonsingular. Hence the Vandermonde system has a unique solution. Hint: Show that $\det(\mathbf{V}) = \prod_{j=0}^{n}\prod_{i\neq j}(x_i - x_j)$ by induction on n.

✍ 9.2. Write a program that constructs a parametric cubic cumulative length spline $(x(t), y(t))$ with *periodic* boundary conditions. Plot the smooth curve through several knots picked at random. Alternatively, in MATLAB you may use the code

```
clf; x = []; y = []; but = 1;
axis([0 10 0 10]);
while but == 1
    [xi,yi,but] = ginput(1);
    x = [x xi]; y = [y yi];
    plot(x,y,'.'); axis([0 10 0 10]);
end
```

to interactively pick the knots.

✍ 9.3. A radial basis function (RBF) is any function ϕ whose value depends only on its distance from the origin. We can use them as interpolating functions:

$$y(x) = \sum_{i=0}^{n} c_i \phi(x - x_i).$$

Compare interpolation using a polynomial basis with the Gaussian and cubic RBFs: $\phi(x) = \exp(-x^2/2\sigma^2)$ and $\phi(x) = |x|^3$. For the Heaviside function

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

using 20 equally spaced nodes in $[-1, 1]$.

✍ 9.4. Collocation is a method of solving a differential equation by approximating the solution by an element in a finite dimensional space. Suppose that we have the linear differential equation

$$\mathrm{L}\, u(x) = w(x) \quad \text{with boundary conditions} \quad u(0) = a \quad \text{and} \quad u(1) = b$$

where L is a linear operator such as $\frac{\mathrm{d}^2}{\mathrm{d}x^2} + (\sin x)\frac{\mathrm{d}}{\mathrm{d}x} + (x^2+1)$. Approximate the solution by

$$u(x) = \sum_{j=0}^{n} c_j v_j(x)$$

for some basis v_j. Take the collocation points $x_i \in [0, 1]$ with $i = 0, \ldots, n$. Then solve the linear system of equations

$$\begin{aligned} &\sum_{j=0}^{n} c_j v_j(x_0) = a \\ &\sum_{j=0}^{n} c_j\, \mathrm{L}\, v_j(x_i) = w(x_i), \qquad i = 1, \ldots, n-1 \\ &\sum_{j=0}^{n} c_j v_j(x_n) = b \end{aligned}$$

Solve the two-point boundary-value problem

$$u'' + (20x + 2)u' + (96x^2 + 24x + 2)u = -12x^4 \mathrm{e}^{-3x^2}$$

with $u(0) = 0$ and $u(1) = 0$ using cubic B-splines with 10 or 20 equally spaced knots over the unit interval $[0, 1]$. Check the solution by computing

the residual $\mathrm{L}\,u(x) - w(x)$. Also, compare the numerical solution with the exact solution

$$u(x) = (x - x^2)\mathrm{e}^{-3x^2}.$$

Note: Because B-splines are locally supported, the system of equations will be sparse and banded.

CHAPTER 10

Approximating Functions

In the previous chapter we examined polynomial interpolation. For a function f, we found a polynomial that coincides with the function f at some given nodes $\{x_1, x_2, \ldots, x_n\}$ in $[a, b]$. In this chapter we consider *best approximation*. An important application of best approximation is for data compression. For a function f in some vector space F, we will find a function u in a subspace U that is closest to f with respect to some norm. That is,

$$\|f - u\| = \inf_{g \in U} \|f - g\|.$$

If the norm is the L^2-norm, e.g., $\|\varphi\|_2 = \int_a^b |\varphi(x)|^2 \, \mathrm{d}x$, then the approximation is called the least-squares approximation, the L^2-approximation, or the *Hilbert approximation*. If the norm is the L^∞-norm, e.g., $|\varphi|_\infty = \sup_{x \in [a,b]} |\varphi(x)|$, the approximation called the uniform approximation or the *Chebyshev approximation*. The least-squares approximation is popular primarily because it is robust and it is typically the easiest to implement. So, we will concentrate on the least-squares approximation.

10.1 Least-squares approximation

Inner product spaces

An *inner-product space* F is a vector space for which an inner product $(\cdot, \cdot)$ has been specified. An inner product is a mapping $F \times F \to \mathbb{C}$ with the properties

1. $(f, f) > 0$ if $f \neq 0$ (positivity)
2. $(f, f) = 0$ iff $f \equiv 0$ (definiteness)
3. $(f, \alpha g + \beta h) = \alpha (f, g) + \beta (f, h)$ (linearity)

4. $(f, g) = \overline{(g, f)}$ (Hermiticity)

We define the induced norm as $\|f\| = \sqrt{(f, f)}$.

Sometimes (as in the case of Fourier polynomials) we generalize the notion of an inner-product to a *pseudo-inner product* by removing the requirement of definiteness. In this case, the norm is replaced by a *semi-norm*.

Example. Important examples of inner product spaces include $\mathbb{R}^n$ with the inner product

$$(x, y) = \sum_{i=1}^{n} x_i y_i$$

and the space of continuous function on an interval $[a, b]$ with the inner product

$$(f, g)_w = \int_a^b f(x)g(x)w(x)\,\mathrm{d}x$$

for some given weighting function $w(x) > 0$. ◀

Orthogonality

We say that f is *orthogonal* to g if $(f, g) = 0$. We 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 $f \perp G$. Sometimes, we use the term *w-orthogonal* to explicitly indicate that a weighting function is included in the definition of the inner product $(f, g)_w$.

Example. Consider the inner product space of L^2-integrable functions with $(f, g) = \int_{-1}^{1} f(x)g(x)\,\mathrm{d}x$. Even functions are orthogonal to odd functions in this inner-product space. Notably, $x^2 - 1 \perp x$ and $\sin x \perp \mathrm{e}^{-x^2}$. ◀

Theorem 30. *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 $u - f \perp U$.*

Proof. ($\Leftarrow$) If $f - u \perp U$ then for any $g \in U$

$$\begin{aligned}\|f - g\|^2 &= \|(f - u) + (u - g)\|^2 \\ &= \|f - u\|^2 + \underbrace{2\,(f - u, f - g)}_{0} + \|u - g\|^2 \geq \|f - u\|^2.\end{aligned}$$

($\Rightarrow$) Suppose that u is a best approximation to f. Let $g \in U$ and $\lambda > 0$. Then

$$\|f - u + \lambda g\|^2 \geq \|f - u\|^2.$$

So,

$$\begin{aligned} 0 &\leq \|f - u + \lambda g\|^2 - \|f - u\|^2 \\ &= \|f - u\|^2 + 2\,(f - u, g) + \lambda^2 \|g\|^2 - \|f - u\|^2 \\ &= \lambda \left[2\,(f - u, g) + \lambda \|g\|^2\right]. \end{aligned}$$

Because $\lambda > 0$,

$$2\,(f - u, g) + \lambda \|g\|^2 \geq 0.$$

Letting λ shrink to zero, we get $(f - u, g) \geq 0$. Similarly, by replacing g with $-g$, we get $(f - u, -g) \geq 0$. So, $(f - u, g) \leq 0$. It follows that $(f - u, g) = 0$. Since g is arbitrary, $f - u \perp U$. $\square$

Normal equations

In general, if $\{u_1, u_2, \ldots, u_n\}$ is a basis for a subspace $U \subset F$ and we want to find the best approximation to $f \in F$ in U, we must find the $u \in U$ such that $u - f \perp U$. That is, we must solve

$$(u - f, u_i) = 0 \text{ for } i = 1, \ldots, n.$$

Take $u = \sum_{j=1}^{n} c_j u_j$. Then

$$\left(\sum_{j=1}^{n} c_j u_j - f, u_i\right) = 0 \text{ for } i = 1, \ldots, n$$

from which it follows that

$$\sum_{j=1}^{n} c_j\,(u_j, u_i) = (f, u_i) \text{ for } i = 1, \ldots, n.$$

These are known as the *normal equations*. In matrix form, the normal equations are

$$\begin{bmatrix} (u_1, u_1) & (u_2, u_1) & \ldots & (u_n, u_1) \\ (u_1, u_2) & (u_2, u_2) & \ldots & (u_n, u_2) \\ \vdots & \vdots & & \vdots \\ (u_1, u_n) & (u_2, u_n) & \ldots & (u_n, u_n) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (f, u_1) \\ (f, u_2) \\ \vdots \\ (f, u_n) \end{bmatrix}.$$

The matrix on the left is called the *Gram matrix*.

Example. Suppose that we want to find the best polynomial approximation $p_n(x) \in \mathbb{P}^n$ to $f(x)$ in the L^2-norm over the interval $[0, 1]$. Suppose that we use the canonical basis $\{\varphi_i\} = \{1, x, x^2, \ldots, x^n\}$, so that

$$p_n(x) = c_0 + c_1 x + c_2 x + \cdots + c_n x^n = \sum_{i=0}^{n} c_i \varphi_i(x).$$

To determine $\{c_i\}$ we solve $(p_n, \varphi_i) = (f, \varphi_i)$ for $i = 1, \ldots, n$:

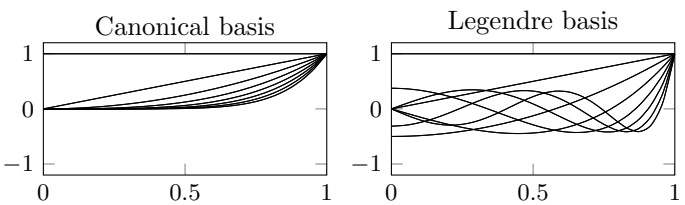
$$(\varphi_i, \varphi_j) = \left(x^i, x^j\right) = \int_0^1 x^{i+j}\,\mathrm{d}x = 2\frac{1}{i+j-1}.$$

We now must solve the normal equations

$$\begin{bmatrix} 1 & 1/2 & 1/3 & \cdots \\ 1/2 & 1/3 & 1/4 & \cdots \\ 1/3 & 1/4 & 1/5 & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (f, 1) \\ (f, x) \\ \vdots \\ (f, x^n) \end{bmatrix}$$

The matrix on the left, called the *Hilbert matrix*, is very ill-conditioned. The condition number for an 7×7 Hilbert matrix is roughly 10^8. Even for relatively small systems, the computations are unstable. We can expect that any error in our calculations may be magnified by as much as 10^8. And the condition number of a 13×13 Hilbert matrix exceeds machine epsilon.

We can get a better idea of why the canonical bases fails by looking at them. Here are the first eight basis elements for canonical bases and Legendre bases:



For the canonical bases, as the order p of the monomial x^p increases, there is smaller and smaller difference between them. On the other hand, the basis polynomials of the Legendre bases are all largely separated. So, it seems that a better solution is to use orthogonal polynomials. ◀

10.2 Orthonormal systems

We say that a set of vectors $\{u_1, u_2, \cdots\}$ is orthonormal in a inner product space if $(u_i, u_j) = \delta_{ij}$ for all i and j. Let's look at the normal equations when we use an orthonormal basis. Because $(u_i, u_j) = \delta_{ij}$ now,

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (f, u_1) \\ (f, u_2) \\ \vdots \\ (f, u_n) \end{bmatrix}.$$

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 is simply

$$p_n = \sum_{i=1}^{n} c_i u_i \quad \text{where} \quad c_i = (f, u_i)\,.$$

That is,

$$p_n = \sum_{i=1}^{n} (f, u_i)\, u_i. \tag{10.1}$$

We call this best approximation the orthogonal projection of f and (f, u_i) is simply the component of f in the u_i direction. We'll denote the orthogonal projection operator by P_n:

$$\mathrm{P}_n\, f = \sum_{i=1}^{n} (f, u_i)\, u_i.$$

Note that

1. P_n maps F onto the subspace U_n spanned by $\{u_1, u_2, \dots, u_n\}$.
2. $\mathrm{P}_n^2 = P_n$, i.e., P_n is a projection.
3. $(\mathrm{P}_n\, f, g) = (f, \mathrm{P}_n\, g)$. i.e., P_n is self-adjoint (Hermitian/symmetric).
4. $\mathrm{P}_n\, f$ is the best approximation of f in U_n.

We can summarize everything into the following theorem, which will be the main tool of the rest of the chapter.

Theorem 31. *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 $\mathrm{P}_n\, f$ where the orthogonal projection operator $\mathrm{P}_n = \sum_{i=1}^{n} (\cdot, u_i)\, u_i$.*

10.3 Complete orthonormal systems

So far we've only used a finite basis. Let's look at an infinite-dimensional vector space. An orthonormal system $\{u_1, u_2, \dots\}$ in inner product space F is a *complete orthogonal system* if

$$\lim_{n\to\infty} \|f - \mathrm{P}_n f\| \to 0$$

for every $f \in F$.

Theorem 32 (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 $\mathrm{P}_n = \sum_{i=1}^{n} (\cdot, u_i)\, u_i$. In the induced norm $\|\cdot\|^2 = (\cdot, \cdot)$, we simply have

$$\|\,\mathrm{P}_n f\|^2 = \left\| \sum_{i=1}^{n} (f, u_i)\, u_i \right\|^2 = \left(\sum_{i=1}^{n} (f, u_i)\, u_i, \sum_{j=1}^{n} (f, u_j)\, u_j \right).$$

We expand the inner product to get

$$\begin{aligned} \|\,\mathrm{P}_n f\|^2 &= \sum_{i=1}^{n} (f, u_i) \left(u_i, \sum_{i=j}^{n} (f, u_j)\, u_j \right) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} (f, u_i)\,(f, u_j)\,(u_i, u_j) \\ &= \sum_{i=1}^{n} |(f, u_i)|^2 \end{aligned}$$

because $(u_i, u_j) = \delta_{ij}$. Now, because the system is complete

$$\|f\|^2 = \lim_{n\to\infty} \|\,\mathrm{P}_n f\|^2 = \sum_{i=1}^{\infty} |\,(f, u_i)\,|^2. \qquad \square$$

Gram–Schmidt process

We can generate an orthonormal basis for an inner product space by using the Gram–Schmidt process. Let $\{v_1, v_2, \dots, v_n\}$ be a basis for some

subspace U_n in an inner-product space. Define

$$\begin{aligned}
\tilde{u}_1 &= v_1 \\
\tilde{u}_2 &= v_2 - \mathrm{P}_1\, v_2 \\
\tilde{u}_3 &= v_3 - \mathrm{P}_2\, v_3 \\
\vdots \quad & \quad \vdots \\
\tilde{u}_k &= v_k - \mathrm{P}_{k-1}\, v_k
\end{aligned}$$

where the projection operator $\mathrm{P}_k = \sum_{i=1}^{k} (\cdot, u_i)\, u_i$ and $u_i = \tilde{u}_i / \|\tilde{u}_i\|$. Then $\{u_i\}$ is an orthonormal basis for U_n.

Three-term recurrence

Applying the Gram–Schmidt process directly can be a lot of work. For generating orthogonal polynomials, there's a short cut. We can use a three-term recurrence relationship.

Theorem 33. *For each weighted inner product $(\cdot,\cdot)_w$, there are uniquely determined orthogonal polynomials $p_k \in \mathbb{P}_k$ with leading coefficient one satisfying the three-term recurrence relation*

$$\begin{cases} p_k(x) = (x + a_k)p_{k-1}(x) + b_k p_{k-2}(x) \\ p_0(x) = 1, \quad p_1(x) = x + a_1 \end{cases}$$

where

$$a_k = -\frac{(xp_{k-1}, p_{k-1})_w}{(p_{k-1}, p_{k-1})_w} \quad \textit{and} \quad b_k = -\frac{(p_{k-1}, p_{k-1})_w}{(p_{k-2}, p_{k-2})_w}.$$

Proof. We'll prove this by induction. First, the only polynomial of degree zero with leading coefficient one is $p_0(x) \equiv 1$. Suppose that the theorem holds for the orthogonal polynomials $p_0, p_1, \ldots, p_{k-1}$. Let p_k be an arbitrary polynomial of degree k and leading coefficient one. Then $p_k - xp_{k-1}$ is a polynomial with degree less than or equal to $k-1$. Because $p_0, p_1, \ldots, p_{k-1}$ form an orthonormal basis of $\mathbb{P}_{k-1}$

$$p_k - xp_{k-1} = \sum_{j=0}^{k-1} c_j p_j \quad \text{with} \quad c_j = \frac{(p_k - xp_{k-1}, p_j)}{(p_j, p_j)}.$$

Furthermore, if p_k is orthogonal to $p_0, p_1, \ldots, p_k$:

$$c_j = \frac{(p_k - xp_{k-1}, p_j)}{(p_j, p_j)} = \frac{(p_k, p_j)}{(p_j, p_j)} - \frac{(xp_{k-1}, p_j)}{(p_j, p_j)} = 0 - \frac{(xp_{k-1}, p_j)}{(p_j, p_j)}$$

The numerator is

$$(xp_{k-1}, p_j) = (p_{k-1}, xp_j) = (p_{k-1}, p_{j+1} - a_{j+1}p_j - b_j p_{j-1})$$

which equals zero for $j = 0, 1, \ldots, k-3$ and equals (p_{k-1}, p_{k-1}) for $j = k-2$. So, $c_0 = \cdots = c_{k-3} = 0$ and

$$a_k = c_{k-1} = -\frac{(xp_{k-1}, p_{k-1})}{(p_{k-1}, p_{k-1})}, \quad b_k = c_{k-2} = -\frac{(xp_{k-1}, p_{k-1})}{(p_{k-2}, p_{k-2})}. \qquad \square$$

Example. Use the L^2-inner product $(f, g) = \int_{-1}^{1} f(x)g(x)\,\mathrm{d}x$ to derive the first four *Legendre polynomials*. We start with $p_0(x) = 1$. Now, we compute $p_1(x)$.

$$a_1 = (x, 1) \,/\, (1, 1) = 0$$

So, $p_1(x) = x$. Note that p_1 is an odd function and $xp_1 = x^2$ is an even function, so $a_2 = 0$. Hence, $p_2(x) = (x + 0)p_1(x) + b_2 p_0(x)$ is an even function. It follows that $a_3 = (\text{odd}, \text{even}) = 0$ and it's not difficult to see from the three-term recurrence relation that p_k is an even function if k is even, p_k is an odd function if k is odd, and $a_k = 0$ for all k.

Let's compute p_2 and p_3.

$$b_2 = -\,(p_1, p_1) \,/\, (p_0, p_0) = (x, x) \,/\, (1, 1) = -\tfrac{1}{3}.$$

So,

$$\begin{aligned} p_2(x) &= x^2 - \tfrac{1}{3} \\ b_3 &= -\,(p_2, p_2) \,/\, (p_1, p_1) \\ &= -\left(\int_{-1}^{1} x^4 - \tfrac{2}{3}x^2 + \tfrac{1}{9}\,\mathrm{d}x\right) / \tfrac{2}{3} = -(\tfrac{2}{5} - \tfrac{4}{9} + \tfrac{2}{9}) / \tfrac{2}{3} = -\tfrac{4}{15}. \end{aligned}$$

So,

$$p_3(x) = x(x^2 - \tfrac{1}{3}) - \tfrac{4}{15}x = x^3 - \tfrac{3}{5}x. \qquad \blacktriangleleft$$

We can generate other w-orthogonal polynomials by using different inner product spaces:

Legendre Polynomials:	$(f,g)_w = \int_{-1}^{1} f(x)g(x)\,\mathrm{d}x$
Chebyshev Polynomials:	$(f,g)_w = \int_{-1}^{1} f(x)g(x)\frac{\mathrm{d}}{\sqrt{1-x^2}}$
Hermite Polynomials:	$(f,g)_w = \int_{-\infty}^{\infty} f(x)g(x)\mathrm{e}^{-x^2}\,\mathrm{d}x$
Laguerre Polynomials	$(f,g)_w = \int_{0}^{\infty} f(x)g(x)\mathrm{e}^{-x}\,\mathrm{d}x$

10.4 Fourier polynomials

So far we've concentrated on subspaces constructed using polynomial bases elements. If the function that we are interpolating or finding the best approximation to is periodic, we may want to use trigonometric polynomial (sine and cosine) basis elements.

We define the Fourier polynomials on $(0, 2\pi)$ using the exponential basis elements

$$\phi_\xi(x) = \mathrm{e}^{\mathrm{i}\xi x} \quad \text{for} \quad \xi = 0, \pm 1, \pm 2, \ldots$$

where $\mathrm{i} = \sqrt{-1}$. Recall that $\sin(\xi x) = \mathrm{Im}(\mathrm{e}^{\mathrm{i}\xi x})$ and $\cos(\xi x) = \mathrm{Re}(\mathrm{e}^{\mathrm{i}\xi x})$.

Theorem 34. *The exponential basis elements $\phi_\xi(x) = \mathrm{e}^{\mathrm{i}\xi x}$ form an orthogonal system $L^2(0, 2\pi)$ with the pseudo-inner product*

$$(f,g) = \frac{1}{2\pi}\int_0^{2\pi} f(x)\overline{g(x)}\,\mathrm{d}x.$$

Proof. We'll show that $(\phi_k, \phi_m) = \delta_{km}$.

$$\left(\mathrm{e}^{\mathrm{i}kx}, \mathrm{e}^{\mathrm{i}mx}\right) = \frac{1}{2\pi}\int_0^{2\pi} \mathrm{e}^{\mathrm{i}kx}\mathrm{e}^{-\mathrm{i}mx}\,\mathrm{d}x = \frac{1}{2\pi}\int_0^{2\pi} \mathrm{e}^{\mathrm{i}(k-m)x}\,\mathrm{d}x$$

$$= \begin{cases} \dfrac{1}{2\pi}\displaystyle\int_0^{2\pi} 1\,\mathrm{d}x = 1, & \text{if } k = m \\ \dfrac{1}{2\pi}\left.\dfrac{\mathrm{e}^{\mathrm{i}(k-m)x}}{i(k-m)}\right|_0^{2\pi} = \dfrac{1}{2\pi}\dfrac{1}{i(k-m)}[1-1] = 0, & \text{if } k \neq m \end{cases}$$

So, $\left(\mathrm{e}^{\mathrm{i}km}, \mathrm{e}^{\mathrm{i}mx}\right) = \delta_{km}$. □

Because the Fourier polynomials are orthogonal we can define a projection operator P_n such that

$$\mathrm{P}_n f = \sum_{\xi=0}^{n-1} \left(f, \mathrm{e}^{\mathrm{i}\xi x}\right)\mathrm{e}^{\mathrm{i}\xi x} = \sum_{\xi=0}^{n-1} c_\xi \mathrm{e}^{\mathrm{i}\xi x} \tag{10.2a}$$

where

$$c_\xi = \frac{1}{2\pi}\int_0^{2\pi} f(x)\mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x. \tag{10.2b}$$

The set of coefficients $\{c_\xi\}$ are called the Fourier coefficients and the inner product (f, ϕ_ξ) for arbitrary ξ is the called the *continuous Fourier transform*.

We often want to evaluate the inner-product numerically using a discrete set of points. In this case, we can similarly define a discrete inner product over $(0, 2\pi)$ by restricting ourselves to a mesh $x_j = 2\pi j/n$ for $j = 0, 1, \ldots, n-1$ and making a trapezoidal approximation to the integral in used in Theorem 34.

Theorem 35. *The exponential bases elements $\phi_\xi(x) = \mathrm{e}^{\mathrm{i}\xi x}$ are orthonormal in the discrete inner-product space $(f, g)_n = \frac{1}{n}\sum_{j=0}^{n-1} f(x_j)\overline{g(x_j)}$.*

Proof. The proof is a simple corollary to Theorem 34 by taking the piecewise-constant approximation to f and g through x_j. □

A projection operator in the discrete inner product space can be defined analogously to the continuous Fourier transform (10.2):

$$\mathrm{P}_n f(x) = \sum_{k=0}^{n-1} c_k \mathrm{e}^{\mathrm{i}kx} \quad \text{where} \quad c_k = \frac{1}{n}\sum_{j=0}^{n-1} f(x_j)\mathrm{e}^{-\mathrm{i}2\pi jk/n}.$$

The inner product $c_k = \left(f, \mathrm{e}^{\mathrm{i}kx}\right)_n$ for arbitrary integer k is called the *discrete Fourier transform* at the meshpoints x_j and

$$\mathrm{P}_n f(x_j) = \sum_{k=0}^{n-1} c_k \mathrm{e}^{\mathrm{i}2\pi jk/n}. \tag{10.3}$$

We call this the *inverse discrete Fourier transform*.

Interpolation with Fourier polynomials

Let's return to the idea of interpolation from the previous chapter. Suppose that we have equally spaced nodes $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ with $x_j = 2\pi j/n$. Using the exponential basis functions $\phi_k(x) = \exp(\mathrm{i}\xi x)$, the interpolating Fourier polynomial satisfies the system of equations

$$y_j = \sum_{j=0}^{n-1} c_k \phi_k(x_j) = \sum_{j=0}^{n-1} c_k \mathrm{e}^{\mathrm{i}2\pi jk/n} = \sum_{j=0}^{n-1} c_k \omega^{jk} \tag{10.4}$$

for some coefficients c_k. The value $\omega = \mathrm{e}^{\mathrm{i}2\pi/n}$ is an *nth root of unity* and we recognize from (10.3) that this transformation as 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 $c_k = (f, \phi_k)_n$. 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 $(\cdot,\cdot)_n$.

Example. Let's consider a connection between Fourier polynomials and algebraic polynomials. Suppose that we want to find the algebraic polynomial interpolant

$$p(z) = c_0 + c_1 z + c_2 z^2 + \cdots + c_{n-1} z^{n-1}, \qquad p(z) \in \mathbb{C}$$

passing though the nodes $\{(z_0, y_0), (z_1, y_1), \ldots, (z_{n-1}, y_{n-1})\}$ in $\mathbb{C} \times \mathbb{C}$.

If we use the canonical basis $\{1, z, \ldots, z^{n-1}\}$, then we must solve the $\mathbf{V}\mathbf{c} = \mathbf{y}$ where $\mathbf{V}$ is the Vandermonde matrix to get the coefficients c_j of the polynomial:

$$\begin{bmatrix} 1 & z_0 & z_0^2 & \cdots & z_0^n \\ 1 & z_1 & z_1^2 & \cdots & z_1^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & z_{n-1} & z_{n-1}^2 & \cdots & z_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}. \tag{10.5}$$

Now, suppose that we restrict $z_k = \omega^k = \mathrm{e}^{\mathrm{i}2\pi k/n}$ to equally spaced points along the unit circle $|z| = 1$. These points run counterclockwise around the unit circle staring with $z_0 = 1$. In this case, (10.5) is simply (10.4). This says that an algebraic polynomial in the complex plane is a Fourier polynomial on the unit circle. ◀

M The MATLAB function `fft` implements the Fastest Fourier Transform in the West (FFTW) library to evaluate the discrete Fourier transform. The inverse discrete Fourier transform can be computed using `ifft` and multidimensional FFTs can be computed using `fftn` and `ifftn`.

Smoothness and accuracy

Let's examine the behavior of the Fourier coefficients for the continuous inner product

$$c_\xi = \frac{1}{2\pi} \int_0^{2\pi} f(x) \mathrm{e}^{-\mathrm{i}\xi x} \, \mathrm{d}x.$$

The discrete inner product follows by analogy. Note that c_0 is simply the mean value of $f(x)$ over $(0, 2\pi)$. Also, note that if $f(x)$ is differentiable, then by integration by parts

$$\left(f', \mathrm{e}^{\mathrm{i}\xi x}\right) = \frac{1}{2\pi}\int_0^{2\pi} f'(x)\mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x = \frac{\mathrm{i}\xi}{2\pi}\int_0^{2\pi} f(x)\mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x = \mathrm{i}\xi c_\xi$$

because $f(x)$ is periodic. In general,

$$\left(f^{(p)}, \mathrm{e}^{\mathrm{i}\xi x}\right) = \frac{1}{2\pi}\int_0^{2\pi} f^{(p)}(x)\mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x = (\mathrm{i}\xi)^p c_\xi. \tag{10.6}$$

Theorem 36. *If $f \in H^p(0, 2\pi)$ then $|c_\xi| = o(\xi^{-p-1/2})$ and the L^2-error in the truncated Fourier series is $o(n^{-p})$.*

Proof.

$$\int_0^{2\pi} |f^{(p)}(x)|^2\,\mathrm{d}x \le \int_0^{2\pi} \sum_{j=1}^{p} |f^{(j)}(x)|^2\,\mathrm{d}x < \infty.$$

By Parseval's Theorem ($\|f\|^2 = \sum_{\xi=1}^{\infty} |\,(f, u_\xi)\,|^2$):

$$\frac{1}{2\pi}\int_0^{2\pi} |f^{(p)}(x)|^2\,\mathrm{d}x = \sum_{\xi=1}^{\infty} |\xi^p c_\xi|^2$$

In order for the series on the right to converge, it must follow that $|\xi^p c_\xi|^2 = o(k^{-1})$ as $\xi \to \infty$. In other words, $|c_\xi| = o(\xi^{-p-1/2})$. If f has infinitely many continuous derivatives in $L^2(\mathbb{R})$, then $|c_\xi| = o(|\xi|^{-m})$ for all $m \ge 0$. That is, the Fourier transform will have faster than polynomial decay, e.g., exponential decay.

The error in the truncated Fourier series $\mathrm{P}_n f$ is

$$\begin{aligned}\|f - \mathrm{P}_n f\|^2 &= \|\,\mathrm{P}_\infty f - \mathrm{P}_n f\|^2 \\ &= \sum_{\xi > |n|} |c_\xi|^2 = \sum_{\xi > |n|} o(\xi^{-2p-1}) = o(n^{-2p})\end{aligned}$$

So, the error is $o(n^{-p})$. If f has infinitely many continuous derivatives in $L^2(\mathbb{R})$, the error will have faster than polynomial decay as $n \to \infty$. □

10.5 Wavelets

Wavelets provide yet another orthogonal system for approximating functions. In this section, we examine multiresolution approximation using orthonormal wavelet basis functions.

Example. Multiresolution analysis provides a means of breaking up a function into the sum of background and fluctuations. 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:

$$\begin{array}{cccccccc} 1,3 & -3,-1 & 9,11 & 5,7 & 23,25 & 21,19 & 16,16 & 19,21 \\ 2 \mp 1 & -2 \mp 1 & 10 \mp 1 & 6 \mp 1 & 24 \mp 1 & 20 \mp -1 & 16 \mp 0 & 20 \mp 1 \end{array}$$

Compute the pairwise averages and differences again:

$$\begin{array}{cccc} 2,-2 & 10,6 & 24,20 & 16,20 \\ 0 \mp -2 & 8 \mp -2 & 22 \mp -2 & 18 \mp 2 \end{array}$$

Again:

$$\begin{array}{cc} 0,8 & 22,18 \\ 4 \mp 4 & 20 \mp 2 \end{array}$$

And one more time:

$$\begin{array}{c} 4,20 \\ 12 \mp 8 \end{array}$$

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 $12 - 8 - 4 - (-2) + 1 = 3$ gives the first two numbers in the original sequence. ◀

Take the space of square-integrable functions. A *multiresolution approximation* is a sequence of subspaces $\emptyset \subset \cdots \subset V_{-1} \subset V_0 \subset V_1 \subset \cdots \subset L^2$ such that the functions $f \in V_n$ are self-similar in scale and self-similar in time and are complete:

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

There exists a basis function $\phi(x) \in V_0$ called the *scaling function* or *father wavelet*. We will only consider normalized scaling functions that have compact support over the unit interval so that they are shift-orthogonal:

$\int_{-\infty}^{\infty} \phi(x)\phi(x-k) = \delta_{0k}$ for all integers k. By using dilation and translation, we can generate orthonormal basis functions in V_n:

$$\phi_{nk}(x) = 2^{n/2}\phi(2^n x - k).$$

Furthermore, we can define an orthogonal projection of a function $f \in L^2$:

$$\mathrm{P}_n f = \sum_{k=-\infty}^{\infty} (f, \phi_{nk})\, \phi_{nk} = \sum_{k=-\infty}^{\infty} c_{nk}\phi_{nk}.$$

The projection into V_n acts to smooth the data by filtering out smaller fluctuations. What we want now is something to record the fluctuations.

Because the scaling function has compact support, we have

$$\phi(x) = \sum_{k=-N}^{N} a_k \phi(2x-k) \quad \text{where} \quad a_k = \int_{-\infty}^{\infty} \phi(x)\phi(2x-k)\,\mathrm{d}x$$

for some integer N. Now, we can define a function $\psi(x)$, called the *wavelet function* or *mother wavelet*:

$$\psi(x) = \sum_{k=-N}^{N} (-1)^k a_{1-k}\phi(2x-k).$$

Then, $\psi(x)$ is an element of a subspace $W_0 \subset V_1$. Furthermore, by dilation and translation, we can build a set of basis functions $\psi_{nk}(x)$ in W_n, respectively, with the orthogonal projection operator

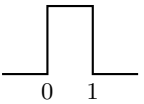
$$\mathrm{Q}_n f = \sum_{k=-\infty}^{\infty} (f, \psi_{nk})\, \psi_{nk} = \sum_{k=-\infty}^{\infty} d_{nk}\psi_{nk}.$$

Let W_n be the range of Q_n. Then $W_n \perp V_n$ and $V_n \oplus W_n = V_{n+1}$. We say that V_{n+1} is the *orthogonal sum* or *direct sum* of subspaces V_n and W_n, if $V_{n+1} = V_n + W_n$ and $V_n \cap W_n = \emptyset$. That is, for every $v \in V_{n+1}$ there is a $f \in V_n$ and $g \in W_n$ such that $v = f + g$. Furthermore, $\bigoplus_{m=-\infty}^{n-1} W_m = V_n$. In general, the subspaces W_n have the properties

1. $\bigoplus_n W_n$ is dense in L^2
2. $W_m \perp W_n$ for all $m \neq n$
3. $f(x) \in W_n$ if and only if $f(2x) \in W_{n+1}$ (dilation)
4. $f(x) \in W_n$ if and only if $f(x - 2^{-n}k) \in W_{n+1}$ (translation)

Example. The *Haar scaling function* is simplest orthogonal scaling function

$$\varphi(x) = \begin{cases} 1, & x \in [0,1) \\ 0, & \text{otherwise} \end{cases}$$



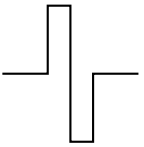
For the Haar scaling function

$$c_{nk} = (f, \varphi_{nk}) = \int_{-\infty}^{\infty} 2^{n/2} f(x)\varphi(2^n x - k)\,\mathrm{d}x = 2^{n/2} \int_{2^{-n}k}^{2^{-n}(k+1)} f(x)\,\mathrm{d}x$$

which is simply the average of $f(x)$ over $[2^{-n}k, 2^{-n}(k+1)]$.

The Haar mother wavelet

$$\psi(x) = \begin{cases} +1, & x \in [0,1/2) \\ -1, & x \in [1/2,1) \\ 0, & \text{otherwise} \end{cases}$$



So, $\psi_{nk}(x) = 2^{n/2}\psi(2^n x - k)$. Furthermore, we have the decomposition

$$\begin{cases} c_{jk} = (c_{j,2k} + c_{j+1,2k+1})/\sqrt{2} \\ d_{jk} = (c_{j+1,2k} - c_{j+1,2k+1})/\sqrt{2} \end{cases} \quad \text{for } k = 0, 1, \ldots, 2^j - 1$$

Inverting this system we also get reconstruction

$$\begin{cases} c_{j+1,2k} = (c_{jk} + d_{jk})/\sqrt{2} \\ c_{j+1,2k+1} = (c_{jk} + d_{jk})/\sqrt{2} \end{cases} \quad \text{for } k = 0, 1, \ldots, 2^j - 1$$

In practice, the algorithm can be computed recursively much like a fast Fourier transform:

```
for k = 0,1,...,2^n
  [ c_jk = f_k
for j = n-1,...,1,0
  [ for k = 0,1,...,2^j - 1
      [ c_jk = (c_{j,2k} + c_{j+1,2k+1})/√2
      [ d_jk = (c_{j+1,2k} - c_{j+1,2k+1})/√2
```

The discrete wavelet transform separates the fluctuations at multiple resolutions. Because of this, wavelets can be used for lossy image compression by zeroing out the elements in the transform corresponding to small fluctuations. Take the sequence of 256×256-pixel images below:

The first represents the uncompressed image with 60% nonzero elements in the Haar transform. Each subsequent image shows the impact of zeroing out the Haar transform leaving 5%, 2%, and 0.5% non-zero elements, respectively. ◀

10.6 Data Fitting

In the real world, we often want to find the "best" function that matches a data set. We may have a model function $y = f(x; c_1, c_2, \ldots, c_n)$ with output y, input x, and several unknown parameters $c_1, c_2, \ldots, c_n$. With a set of inputs and corresponding outputs, we can try to determine the parameters c_i that fit the model, but typically input and output data coming from observation and experiment has random error. So, the problem is inconsistent unless we (gasp!) overfit the data by changing the model. Luckily, the problem can nicely be handled using the least-squares best approximation.

Linear least squares approximation

Suppose that we have n parameters $\{c_1, c_2, \ldots, c_n\}$ in our model and that we have $m \geq n$ input and output values $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$. Also, suppose that our model is linear with respect to the parameters $c_1, c_2, \ldots, c_n$. That is,

$$u(x) = c_1\varphi_1(x) + c_2\varphi_2(x) + \cdots + c_n\varphi_n(x)$$

where $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ are basis elements for a subspace $U \subset F$. To find the best approximation to $f \in F$ in U, we must find the $u \in U$ such that $u - f \perp U$. In this case, we have the *normal equations*

$$(u, \varphi_i) = \sum_{j=1}^{n} c_j\,(\varphi_j, \varphi_i) = (f, \varphi_i) \text{ for } i = 1, \ldots, n.$$

where we define the inner product

$$(f, g) = \sum_{k=1}^{m} f(x_k)g(x_k).$$

In matrix form of the normal equations is

$$\begin{bmatrix} (\varphi_1, \varphi_1) & (\varphi_2, \varphi_1) & \ldots & (\varphi_n, \varphi_1) \\ (\varphi_1, \varphi_2) & (\varphi_2, \varphi_2) & \ldots & (\varphi_n, \varphi_2) \\ \vdots & \vdots & & \vdots \\ (\varphi_1, \varphi_n) & (\varphi_2, \varphi_n) & \ldots & (\varphi_n, \varphi_n) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} (f, \varphi_1) \\ (f, \varphi_2) \\ \vdots \\ (f, \varphi_n) \end{bmatrix}.$$

which in vector notation is simply

$$\mathbf{A}^T\mathbf{A}\mathbf{c} = \mathbf{A}^T\mathbf{f}$$

where $\mathbf{A}$ is an $m \times n$ matrix with elements given by $A_{ij} = \varphi_j(x_i)$. In the solution $\mathbf{c} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{f}$, the term $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ is called the *pseudo-inverse* of $\mathbf{A}$ and denoted $\mathbf{A}^+$. Formally, we have replaced the $\mathbf{c} = \mathbf{A}^{-1}\mathbf{f}$ with $\mathbf{c} = \mathbf{A}^+\mathbf{f}$. For more discussion of linear least squares and the pseudo-inverse 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 bases elements of an inner product space? For example, we may want to determine the Gaussian distribution

$$y = c_1 \mathrm{e}^{-(x-c_2)^2/2c_3^2}.$$

which best matches the some given data by determining the amplitude c_1, the mean c_2, and the standard deviation c_3. In general, for the given function f, we want to find the parameters $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ such that $y_i = f(x_i; c_1, c_2, \ldots, c_n)$ at given the points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$.

Let's switch the roles of $\mathbf{c}$ and $\mathbf{x}$ as the parameter and define $f_i(c) = f(c_1, c_2, \ldots, c_n; x_i)$. This allows us to write the system of residuals as

$$\begin{aligned} r_1 &= y_1 - f_1(c_1, c_2, \ldots, c_n) \\ r_2 &= y_2 - f_2(c_1, c_2, \ldots, c_n) \\ \vdots &= \qquad\qquad \vdots \\ r_m &= y_m - f_m(c_1, c_2, \ldots, c_n). \end{aligned}$$

In vector notation, this system is simply $\mathbf{r}(\mathbf{c}) = \mathbf{y} - \mathbf{f}(\mathbf{c})$. If this system were consistent and $m = n$, we could use Newton's method that we developed in Section 8.7 to estimate a $\mathbf{c}$ such that $\mathbf{r}(\mathbf{c}) = 0$. In such a case, we would solve

$$\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} - \left(\mathbf{J}_\mathbf{r}(\mathbf{c}^{(k)})\right)^{-1} \mathbf{r}(\mathbf{c}^{(k)})$$

where i, j component of the Jacobian $\mathbf{J}_\mathbf{r}(\mathbf{c})$ is $\partial r_i / \partial c_j$. The Jacobian $\mathbf{J}_\mathbf{r}(\mathbf{c}) = -\mathbf{J}_\mathbf{f}(\mathbf{c})$, so we equivalently have

$$\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + \left(\mathbf{J}_\mathbf{f}(\mathbf{c}^{(k)})\right)^{-1} \mathbf{r}(\mathbf{c}^{(k)})$$

where the $m \times n$ Jacobian $\mathbf{J}_\mathbf{f}(\mathbf{c})$ is $\partial f_i / \partial c_j$. Formally, because the system is overdetermined with $m > n$, we can replace the $\mathbf{J}_\mathbf{f}(\mathbf{c})^{-1}$ with the

pseudo-inverse $\mathbf{J}_\mathbf{f}^+ = (\mathbf{J}_\mathbf{f}^T\mathbf{J}_\mathbf{f})^{-1}\mathbf{J}_\mathbf{f}^T$ to get the *Gauss–Newton method*:

$$\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + \mathbf{J}_\mathbf{f}^+(\mathbf{c}^{(k)})\mathbf{r}(\mathbf{c}^{(k)}). \tag{10.7}$$

A rigorous derivation of the Gauss–Newton method is left as an exercise.

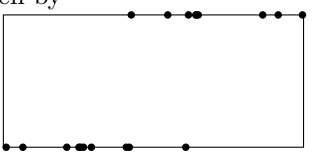
Because Newton's method can be unstable, one often regularizes the Jacobian matrix. One way to do this, is to make the matrix more diagonally dominant This has the effect of decoupling the system of equations. The *Levenberg–Marquardt method* is given by

$$\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} + (\mathbf{J}^T\mathbf{J} + \lambda\,\mathrm{diag}(\mathbf{J}^T\mathbf{J}))^{-1}\mathbf{J}^T\mathbf{r}^{(k)} \tag{10.8}$$

where the damping parameter $\lambda > 0$. The damping parameter is typically changed at each iteration, increasing it if the residual increases and decreasing it is the residual decreases. The Levenberg–Marquardt method is analogous to *Tikhonov regularization* used to solve underdetermined, inconsistent linear systems.

Example. The logistic function $f(x) = (\exp(-c_1x + c_2) + 1)^{-1}$ is often used to model binary classification where $x \in \mathbb{R}$ and $y \in \{0, 1\}$. Let's apply the Gauss–Newton method to data given by

```
randn("seed",88);
x = [randn([10 1]);randn([10 1])+2];
y = [zeros([10 1]);ones([10 1])];
```

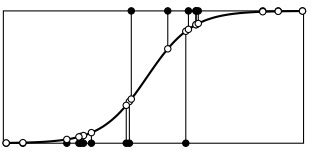


We'll define the annonymous function

```
f = @(c,x) 1./(exp(-c(1)*x+c(2))+1);
```

Take an initial guess `c = [1;1]`. Then,

```
function c = gauss_newton(x,y,c,f)
for j = 1:80
  G = gradient(f,c,x);
  c0 = c + (G'*G)\G'*(y - f(c,x));
  if norm(c0 - c)< 0.0001, break;
  else c = c0; end
end

function g = gradient(f,c,x)
for i = 1:length(c)
  d = zeros(size(c));
  d(i) = eps(c(i)).^(1/3);
  g(:,i) = (f(c+d,x)-f(c-d,x))/(2*d(i));
end
```



returns `[2.5379;2.0020]` after 9 iterations. ◀

Exercises

✍ 10.1. We can think about orthogonality in geometric terms by using the 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

$$(f,g) = \int_0^1 f(x)g(x)\,\mathrm{d}x.$$

(b) Use the definition of the angle between two subspaces

$$\cos\theta = \sup_{f\in F, g\in G} \frac{(f,g)}{\|f\|\|g\|}$$

to show that when n is large x^{n+1} is close to the subspace spanned by $\{1, x, x^2, \cdots, x^n\}$.

10.2. Show that the coefficients b_n in the three-term recursion relation of the Legendre polynomials are given by $b_{n+1} = n^2/(4n^2-1)$.

✍ 10.3. When we learn arithmetic, we are first taught integer values. Only later are we introduced to fractions (and irrational numbers) which complete the real line. Learning calculus is similar. We start by computing with whole derivatives. And later we are sometimes introduced to fractional derivatives. Just as $\frac{\mathrm{d}}{\mathrm{d}x}\frac{\mathrm{d}}{\mathrm{d}x}f(x) = \frac{\mathrm{d}^2}{\mathrm{d}x^2}f(x)$, we also have $\frac{\mathrm{d}^{1/2}}{\mathrm{d}x^{1/2}}\frac{\mathrm{d}^{1/2}}{\mathrm{d}x^{1/2}}f(x) = \frac{\mathrm{d}}{\mathrm{d}x}f(x)$. And in general, $\frac{\mathrm{d}^p}{\mathrm{d}x^p}\frac{\mathrm{d}^q}{\mathrm{d}x^q}f(x) = \frac{\mathrm{d}}{\mathrm{d}x}f(x)$ for $p+q=1$. Of course, this extension also provides a convenient means to introduce the antiderivative as $\frac{\mathrm{d}^{-1}}{\mathrm{d}x^{-1}}f(x)$

In this exercise, you will use the Fourier transform to explore fractional derivatives. Recall that from (10.6), that if $\hat{f}(\xi)$ is the Fourier transform of $f(x)$, then $(\mathrm{i}\xi)^p\hat{f}(\xi)$ is Fourier transform of the pth derivative of $f(x)$. Use MATLAB's `fft` and `ifft` commands to compute the fractional derivatives of the following functions:

- Sine: $\sin x$ over $[-\pi, \pi]$
- Piecewise-quadratic: $f = -\operatorname{sign}(x) \cdot \left(\left(|x| - \frac{1}{2} \right)^2 - \frac{1}{4} \right)$ over $[-1, 1]$
- Piecewise-linear: $f = -\operatorname{sign}(x) \cdot \left(\left(|x| - \frac{1}{2} \right)^2 - \frac{1}{4} \right)$ over $[-1, 1]$
- Gaussian: $\exp(-6x^2)$ over $[-2, 2]$

You may want to loop over several values of p and plot $f^{(p)}(x)$ at each iteration using the command `drawnow` to watch the evolution. Hint: Define the vector ξ as `k = [0:(n/2) (-n/2+1):-1]*(2*pi/L)` when n is even and `L` is the length of the domain.

✍ 10.4. Suppose that you have been handed some noisy data collected from an experiment, which from theory, you expect to be modeled by two Gaussian distributions

$$f(x; \mathbf{c}) = c_1 \mathrm{e}^{-c_2 (x - c_3)^2} + c_4 \mathrm{e}^{-c_5 (x - c_6)^2}$$

for some unknown parameters $\mathbf{c} = \{c_1, c_2, \ldots, c_6\}$. For this exercise, take the measured data to be

```
x = 8*rand([100 1]);
y = f(x,[1 3 3 2 3 6]) + 0.1*randn([100 1]);
```

with the function $f(x; \mathbf{c})$

```
f = @(c,x) c(1)*exp(-c(2).*(x-c(3)).^2)+ ...
           c(4)*exp(-c(5).*(x-c(6)).^2);
```

Use Newton's method or another method to recover the parameters $\mathbf{c}$. Take $\{2, 0.3, 2, 1, 0.3, 7\}$ as the initial guess for $\mathbf{c}$.

CHAPTER 11

Differentiation and Integration

In this chapter we'll apply the ideas of approximation and interpolation that we developed in earlier chapters to the problems of numerical differentiation and integration.

11.1 Differentiation

Numerical differentiation can be unstable due to subtractive cancellation leading to a high condition number. Therefore, high order methods are important. Recall from Chapter 9 that we can interpolate a differentiable function with a polynomial. Differentiation of this polynomial will give us one approximation of the derivative. In this section, we will examine the problem using a slightly different approach. The idea is covered more fully in Chapter 11 as it pertains to approximating differential equations.

Method of undetermined coefficients

Suppose that we want to find the derivative at x of some function f. Consider equally spaced nodes. From Taylor series expansion we have that

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + h^2 f''(x) + O(h^3) \\ f(x) &= f(x) \end{aligned}$$

Subtracting these equations gives us

$$f(x+h) - f(x) = hf'(x) + h^2 f''(x) + O(h^3)$$

From which we have that

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

The error is $hf''(x) + O(h^2)$ which tells us that we have a first-order method. To get a higher order methods, we can extend this idea using more nodes, but this time it is not completely obvious how to best combine the Taylor expansions. So, we will use instead use the *method of undetermined coefficients*. Suppose that we have three nodes:

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \tfrac{1}{2}h^2 f''(x) + O(h^3) \\ f(x) &= f(x) \\ f(x-h) &= f(x) - hf'(x) + \tfrac{1}{2}h^2 f''(x) + O(h^3). \end{aligned}$$

We want to eliminate as many terms that could contribute to error. Consider the unknowns a, b, c

$$\begin{aligned} af(x+h) &= af(x) + ahf'(x) + a\tfrac{1}{2}h^2 f''(x) + a\tfrac{1}{6}h^3 f''(x) + O(h^4) \\ bf(x) &= bf(x) \\ cf(x-h) &= cf(x) - chf'(x) + c\tfrac{1}{2}h^2 f''(x) - c\tfrac{1}{6}h^3 f''(x) + O(h^4). \end{aligned}$$

In order to ensure consistency, we must eliminate the $f(x)$ on the left and we must keep the $f'(x)$ term. Finally, we eliminate the $h^2 f''(x)$ term to get a second-order method.

$$\begin{aligned} a + b + c &= 0 \\ a - c &= 1 \\ \tfrac{1}{2}a + \tfrac{1}{2}c &= 0 \end{aligned}$$

Solving this equations gives us $a = \frac{1}{2}$, $b = 0$ and $c = -\frac{1}{2}$ which gives us

$$\frac{1}{2}f(x+h) - \frac{1}{2}f(x-h) = hf'(x) + \tfrac{1}{6}h^3 f'''(x) + O(h^4)$$

from which we have that

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

with an error of $\frac{1}{6}h^2 f'''(x) + O(h^3)$. Note that by symmetry we didn't even need to use the middle node.

Of course, we can extend this idea to four nodes to get a third-order method, five nodes to get a fourth-order-method, etc. And we could use a non-equally spaced nodes. To make life easier, we can use the Vandermonde matrix to implement the method of undetermined coefficients to find the approximations

$$f(x + \delta_i h) = \sum_{j=0}^{n} c_{ij} \frac{1}{j!} (\delta_i h)^j f^{(j)}(x) + O(h^{n+1})$$

where δ_i scale the step size h. The Vandermonde matrix is given by

$$\mathbf{V} = \begin{bmatrix} 1 & \delta_0 & \delta_0^2 & \cdots & \delta_0^n \\ 1 & \delta_1 & \delta_1^2 & \cdots & \delta_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \delta_n & \delta_n^2 & \cdots & \delta_n^n \end{bmatrix}.$$

Define the diagonal matrices $\mathbf{S} = \mathrm{diag}(1/0!, 1/1!, 1/2!, \cdots, 1/n!)$ and $\mathbf{H} = \mathrm{diag}(1, h, h^2, \cdots, h^n)$ and the coefficient matrix $\mathbf{C} = \left[c_{ij}\right]$. Then

$$\left[f(x+\delta_i h)\right] = \mathbf{VSHC}\left[f^{(j)}(x)\right] + O(h^{n+1}).$$

Then to get the coefficient matrix $\mathbf{HC}$, we simply need to calculate $(\mathbf{VS})^{-1}$. The first row of $\mathbf{C}$ is the coefficients c_{0i} for a $O(h^{n+1})$ approximation of $f(x)$, the second row is the coefficients c_{1i} for an $O(h^n)$ approximation of $f'(x)$, and so on. The coefficients of the truncation error is simply

$$\mathbf{HC}\left[\delta_j^{n+1}\right]/(n+1)!.$$

Example. We can third-order approximation to $f'(x)$ using nodes at $x-h$, x, $x+h$ and $x+2h$ in MATLAB:

```
d = [-1;0;1;2];
V = fliplr(vander(d));
S = diag(1./factorial([0:3]));
HC = inv(V*S);
```

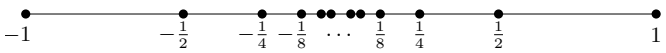
Coefficients are given by `rats(HC)` and the coefficients of the truncation error is given by `rats(HC*d.^4/factorial(4))`. From the second row,

$$f'(x) = \frac{-\frac{1}{3}f(x-h) - \frac{1}{3}f(x) + f(x+h) - \frac{1}{6}f(x+2h)}{h} + O(\tfrac{1}{12}h^3). \quad \blacktriangleleft$$

Ⓜ The command `rats` returns the rational approximation of number (or an array) approximation as a string. For example `rats(0.75)` returns `3/4`.

Richardson extrapolation

It would seem reasonable that we could improve accuracy by putting the nodes in strategic positions, rather than using the equally spaced nodes discussed above. Specifically, consider the following arrangement



Start with the center-difference formula computed earlier (keeping more terms this time). Also, define

$$\phi(h) = \frac{f(x+h) - f(x-h)}{2h}$$

to be the center-difference approximation. Then

$$\begin{aligned} f'(x) &= \frac{f(x+h) - f(x-h)}{2h} + \frac{1}{6}h^2 f'''(x) + \frac{1}{5!}h^4 f^{(5)}(x) + O(h^6) \\ &= \phi(h) + \frac{1}{6}h^2 f'''(x) + \frac{1}{5!}h^4 f^{(5)}(x) + O(h^6) \end{aligned}$$

and the center-difference formula for nodes at $\pm h/2$ is

$$\begin{aligned} f'(x) &= \frac{f(x+\frac{1}{2}h) - f(x-\frac{1}{2}h)}{h} + \frac{h^2}{3!2^2} f'''(x) + \frac{h^4}{5!2^4} f^{(5)}(x) + O(h^6) \\ &= \phi\left(\frac{h}{2}\right) + \frac{1}{24}h^2 f'''(x) + \left(5!\frac{1}{16}\right) h^4 f^{(5)}(x) + O(h^6) \end{aligned}$$

We can cancel the $O(h^2)$ term by taking the following combination

$$f'(x) = -\tfrac{1}{3}\phi(h) + \tfrac{4}{3}\phi\left(\frac{h}{2}\right) - \frac{1}{4 \cdot 5!}h^4 f^{(5)}(x) + O(h^6)$$

So, now we have a fourth-order method (with four terms). We could continue this idea to get a sixth-order method by also using $\phi(h/4)$.

This method of getting higher and higher accuracy is called *Richardson extrapolation*. Richardson extrapolation can be applied to any quantity that has an expansion

$$\mathrm{D}(h) = a_0 + a_1 h + a_2 h^2 + a_3 h^3 + \cdots$$

to successively eliminate the $O(h)$, $O(h^2)$, and higher orders tems. Take $0 < \delta < 1$. Typically, $\delta = \frac{1}{2}$. Then

$$\mathrm{D}(\delta h) = a_0 + a_1 \delta h + a_2 \delta^2 h^2 + a_3 \delta^3 h^3 + \cdots .$$

And so,

$$\mathrm{D}(\delta h) - \delta D(h) = a_0(1-\delta) + 0 + a_2(\delta^3 - \delta)h^2 + a_3(\delta^4 - \delta)h^3 + \cdots .$$

From which we have that

$$\frac{\mathrm{D}(\delta h) - \delta\,\mathrm{D}(h)}{1-\delta} = a_0 + \tilde{a}_2 h^2 + \tilde{a}_3 h^3 + \cdots .$$

In general,

$$
\begin{array}{ccccccc}
\mathrm{D}_{0,0} & & & & & & \\
 & \searrow & & & & & \\
\mathrm{D}_{1,0} & \rightarrow & \mathrm{D}_{1,1} & & & & \\
 & \searrow & & \searrow & & & \\
\mathrm{D}_{2,0} & \rightarrow & \mathrm{D}_{2,1} & \rightarrow & \mathrm{D}_{2,2} & & \\
 & \searrow & & \searrow & & \searrow & \\
\mathrm{D}_{3,0} & \rightarrow & \mathrm{D}_{3,1} & \rightarrow & \mathrm{D}_{3,2} & \rightarrow & \mathrm{D}_{3,3}
\end{array}
\qquad
\begin{aligned}
\mathrm{D}_{m,0} &= \phi(\delta^q h) \\
\mathrm{D}_{m,q+1} &= \frac{\mathrm{D}_{m,q} - \delta^{q+1}\,\mathrm{D}_{m-1,q}}{1-\delta^{q+1}} \quad \text{where } m \geq q.
\end{aligned}
$$

So,

$$\mathrm{D}_{m,n} = a_0 + O\left((\delta^m h)^{n+1}\right).$$

That is, if $\mathrm{D}_{0,0}$ gave us first order method, then $\mathrm{D}_{3,3}$ gives us a fourth order method. Furthermore, for a center-difference scheme if $\mathrm{D}_{0,0}$ gave us second-order method, then $\mathrm{D}_{3,3}$ gives us a eighth-order method using eight terms.

11.2 Numerical integration

A typical method for integrating a function is to first approximate the function by a polynomial interpolate and then to integrate the polynomial exactly. Recall from Chapter 9, that given a set of $n+1$ nodes we can determine the Lagrange basis for an nth-degree 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

$$
\begin{aligned}
\int p_n(x)\,\mathrm{d}x &= \int \left(\sum_{i=0}^{n} f(x_i)\ell_i(x)\right)\mathrm{d}x \\
&= \sum_{i=0}^{n} f(x_i) \int \ell_i(x)\,\mathrm{d}x \\
&= \sum_{i=0}^{n} A_i f(x_i)\,\mathrm{d}x \quad \text{where} \quad A_i = \int \ell_i(x)
\end{aligned}
$$

For uniformly spaced nodes, this method is called *Newton–Cotes*.

Suppose that we take two nodes at $x = \{0, 1\}$. Then

$$A_0 = \int_0^1 \ell_0(x)\,\mathrm{d}x = \int_0^1 1 - x\,\mathrm{d}x = \frac{1}{2}$$
$$A_1 = \int_0^1 \ell_1(x)\,\mathrm{d}x = \int_0^1 x\,\mathrm{d}x = \frac{1}{2}.$$

We have that $\int_0^1 f(x)\,\mathrm{d}x \approx \frac{1}{2}(f(0) + f(1))$. We call this the *trapezoidal rule*. Over an arbitrary interval $[a, b]$,

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{2}(f(a) + f(b)).$$

Now, suppose that we take three nodes at $x = \{0, \frac{1}{2}, 1\}$. Then

$$A_0 = \int_0^1 \ell_0(x)\,\mathrm{d}x = \int_0^1 2(1-x)(\tfrac{1}{2} - x))\,\mathrm{d}x = \frac{1}{6},$$
$$A_1 = \int_0^1 \ell_1(x)\,\mathrm{d}x = \int_0^1 4x(1-x)\,\mathrm{d}x = \frac{2}{3},$$
$$A_2 = \int_0^1 \ell_2(x)\,\mathrm{d}x = \int_0^1 2x(x - \tfrac{1}{2})\,\mathrm{d}x = \frac{1}{6}.$$

We have that $\int_0^1 f(x)\,\mathrm{d}x \approx \frac{1}{2}(f(0) + f(1))$. We call this *Simpson's rule*. Over an arbitrary interval $[a, b]$,

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right].$$

Error

Recall the Polynomials Interpolation Error Theorem 26 on page 203 that says that if $p_n \in \mathbb{P}_n$ interpolates $f \in C^{n+1}([a, b])$ at $n + 1$ points $x_0, \ldots, x_n$, then for each $x \in [a, b]$ there is a $\xi \in [a, b]$ such that the pointwise error

$$f(x) - p_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^{n} (x - x_i).$$

Therefore, the error for the trapezoidal method over an interval of length h is

$$\begin{aligned}\text{error} &= \int_0^h f(x) - p_1(x) \\ &= \frac{1}{2} f^{(2)}(\xi) \int_0^h (x)(x-h)\,\mathrm{d}x = \frac{1}{2} f^{(2)}(\xi)\left(-\frac{h^3}{6}\right) = -\frac{h^3}{12} f^{(2)}(\xi)\end{aligned}$$

for some $\xi \in [0, h]$. Now let's determine the error for the Simpson's method over $[-h, h]$, *i.e.,* over two subintervals each of length h. Recall that Simpson's method is derived using a quadratic interpolating polynomial and is therefore exact if $f(x)$ is a quadratic polynomial. Simpson's method is, in fact, also exact if $f(x)$ is cubic. This is because x^3 is an odd-function and its contribution integrates out to zero. Therefore, we really have a cubic polynomial. Take the interval $[-h, h]$ with nodes at $-h$, 0, and $+h$ and a fourth node at an arbitrary point c. Then

$$\begin{aligned}\text{error} = \int_{-h}^{h} f(x) - p_3(x)\,\mathrm{d}x &= \frac{1}{4!} f^{(4)}(\xi) \int_{-h}^{h} x(x-c)(x^2-h^2)\,\mathrm{d}x \\ &= \frac{1}{4!} f^{(4)}(\xi) \left(-\frac{4h^5}{15}\right) = -\frac{h^5}{90} f^{(4)}(\xi)\end{aligned}$$

for some $\xi \in [-h, h]$. We can get a more accurate method by adding nodes and using a higher order polynomial. But such an approach is not a good idea, principally because of the Runge phenomenon.

Instead, we can either use a piecewise polynomial approximation or we choose the nodes so that the uniform error is minimized (such as using Chebyshev points). We'll look at both of these options in turn.

Composite methods

A composite method uses a spline, a piecewise polynomial interpolant, to approximate an integral. Riemann sums use piecewise constant splines. A more accurate method uses linear splines. In this case, using the trapezoid method on each interval, gives us the *composite trapezoid method*. For n equally-space nodes,

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(a+ih) + f(b)\right] \quad \text{where} \quad h = \frac{|b-a|}{n}.$$

As we showed earlier, the error is for the trapezoid method is $\frac{1}{12}h^3 f^{(2)}(\xi)$ for some ξ. If we take $h = |b-a|/n$, then the error for the composite trapezoid method is bounded by $|b-a| \cdot |\frac{1}{12}h^2 \sup \left|f^2(x)\right|$, and we have a second order method. If we use twice as many intervals, we should expect the error to go down by about a factor of four. We can improve the composite trapezoid method by using Richardson extrapolation. Such an approach is called *Romberg integration*.

Using a quadratic spline, Simpson's method becomes the *composite Simpson's method*. For n equally spaced nodes, Simpson's method says

$$\int_{x_{i-1}}^{x_{i+1}} f(x)\,\mathrm{d}x \approx \frac{h}{6}\left(f(x_{i-1}) + 4f(x_i) + f(x_{i+1})\right).$$

Therefore,

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\big(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots \\ \cdots + 4f(x_{n-1}) + f(x_n)\big)$$

$$= \frac{h}{3}\left(f(x_0) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=2}^{n/2} f(x_{2i-2}) + f(x_n)\right).$$

Because the error of the Simpson's method is $h^5 f^{(4)}(\xi)/90$, the error of the composite Simpson's method over $[a, b]$ is $h^4|b - a| \sup f^{(4)}(\xi)/90$. By using twice as many intervals, we should expect the error to gown down by a factor of 16. In practice, one often uses an *adaptive* composite trapezoid or Simpson's method.

Ⓜ The MATLAB command `quad` evaluates an integral using adaptive Simpson quadrature.

11.3 Gaussian quadrature

Another way to build a high order method and minimize instability due to the Runge phenomenon is by choosing the points strategically. In Chapter 9, we used Chebyshev nodes (the zeros of the Chebyshev polynomial) to minimize the instability. In general, we can get a high order method by using the roots of an orthogonal polynomial as the nodes. Such an approach is called *Gaussian quadrature* or *Gauss–Christoffel quadrature*.

Theorem 37 (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 w-orthogonal to $\mathbb{P}_n$. That is,*

$$(p, q)_w = \int_a^b q(x)p(x)w(x)\,\mathrm{d}x = 0 \qquad \textit{for all } p \in \mathbb{P}_n$$

Let $x_0, x_1, \ldots, x_n$ be the zeros of q. Then

$$\int_a^b f(x)w(x)\,\mathrm{d}x = \sum_{i=0}^{n} A_i f(x_i)$$

for any $f \in \mathbb{P}_{2n+1}$ where

$$A_i = \int_a^b \ell_i(x)w(x)\,\mathrm{d}x$$

with Lagrange basis $\ell_i(x)$ with nodes at the zeros of q.

Proof. Suppose that $f \in \mathbb{P}_{2n+1}$ and $f \in \mathbb{P}_{n+1}$. Then by dividing f by q

$$f(x) = q(x)p(x) + r(x)$$

for some $p, r \in \mathbb{P}_n$. At the zeros of q,

$$f(x_i) = q(x_i)p(x_i) + r(x_i) = r(x_i).$$

Therefore,

$$\begin{aligned}\int_a^b f(x)w(x)\,\mathrm{d}x &= \underbrace{\int_a^b q(x)p(x)w(x)\,\mathrm{d}x}_{0} + \int_a^b r(x)w(x)\,\mathrm{d}x \\ &= \sum_{i=0}^{n} A_i r(x_i) \qquad (\text{because } r \in \mathbb{P}_n) \\ &= \sum_{i=0}^{n} A_i f(x_i).\end{aligned}$$

□

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.

Recall from Chapter 10, we can generate w-orthogonal polynomials by using different inner product spaces:

Polynomial	Interval	Weight	$\|1\|$
Legendre Polynomials	$[-1,1]$	1	2
Chebyshev Polynomials	$[-1,1]$	$1/\sqrt{1-x^2}$	$\pi/2$
Hermite Polynomials	$(-\infty,\infty)$	$\exp(-x^2)$	$\sqrt{\pi}$
Laguerre Polynomials	$(0,\infty)$	$\exp(-x)$	–

Determining the nodes and the weights

We still need to determine the nodes x_i and the weights A_i. Let's first find the nodes for an arbitrary family of orthogonal polynomials.

Recall Theorem 33 which says that for each weighted inner product $(\cdot,\cdot)$ there exist uniquely determined orthogonal polynomials $p_k \in \mathbb{P}_k$ with leading coefficient one satisfying the three-term recurrence relation

$$\begin{cases} p_k(x) = (x-a_k)p_{k-1}(x) - b_k p_{k-2}(x) \\ p_0(x) = 1, \quad p_1(x) = x - a_1 \end{cases}$$

where

$$a_k = \frac{(xp_{k-1}, p_{k-1})}{(p_{k-1}, p_{k-1})} \quad \text{and} \quad b_k = \frac{(p_{k-1}, p_{k-1})}{(p_{k-2}, p_{k-2})}.$$

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 relationship as

$$\|p_k\|\hat{p}_k(x) = (x - a_k)\|p_{k-1}\|\hat{p}_{k-1}(x) - b_k\|p_{k-2}\|\hat{p}_{k-2}(x).$$

Dividing through by $\|p_{k-1}\|$ gives us

$$\frac{\|p_k\|}{\|p_{k-1}\|}\hat{p}_k(x) = (x - a_k)\hat{p}_{k-1}(x) - b_k\frac{\|p_{k-2}\|}{\|p_{k-1}\|}\hat{p}_{k-2}(x)$$

and noting that $b_k = \|p_{k-1}\|^2/\|p_{k-2}\|^2$

$$\sqrt{b_{k+1}}\hat{p}_k(x) = (x - a_k)\hat{p}_{k-1}(x) - \sqrt{b_k}\hat{p}_{k-2}(x),$$

We can rearrange the terms

$$\sqrt{b_k}\hat{p}_{k-2}(x) + a_k\hat{p}_{k-1}(x) + \sqrt{b_{k+1}}\hat{p}_k(x) = x\hat{p}_{k-1}(x)$$

which is simply the linear system $\mathbf{J}_n\mathbf{p} = x\mathbf{p} + \mathbf{r}$:

$$\begin{bmatrix} a_1 & \sqrt{b_2} & & & \\ \sqrt{b_2} & a_2 & \sqrt{b_3} & & \\ & \ddots & \ddots & \ddots & \\ & & \sqrt{b_n} & a_{n-1} & \sqrt{b_{n+1}} \\ & & & \sqrt{b_{n+1}} & a_n \end{bmatrix} \begin{bmatrix} \hat{p}_0(x) \\ \hat{p}_1(x) \\ \vdots \\ \hat{p}_{n-1}(x) \\ \hat{p}_n(x) \end{bmatrix} =$$

$$x \begin{bmatrix} \hat{p}_0(x) \\ \hat{p}_1(x) \\ \vdots \\ \hat{p}_{n-1}(x) \\ \hat{p}_n(x) \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \sqrt{b_{n+2}}\hat{p}_{n+1}(x) \end{bmatrix}$$

Now, $p_{n+1}(x) = 0$ if and only if $\mathbf{r} = 0$ if and only if $\mathbf{J}\mathbf{p} = x\mathbf{p}$. That is, the roots of p_{n+1} are simply the eigenvalues of $\mathbf{J}_n$.

Now, let's get the weights. Note that by the Fundamental Theorem of Gaussian Quadrature

$$\sum_{k=0}^{n} A_k\hat{p}_i(x_k)\hat{p}_j(x_k) = \int \hat{p}_i(x)\hat{p}_j(x)w(x)\,\mathrm{d}x = (\hat{p}_i, \hat{p}_j) = \delta_{ij}$$

because the degree of $\hat{p}_i(x)\hat{p}_j(x) < 2n$. But this simply says that

$$\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$$

where $Q_{ij} = \sqrt{A_j}\hat{p}_i(x_j)$. Hence, $\mathbf{Q}$ is orthogonal. So, $\mathbf{Q}\mathbf{Q}^T = \mathbf{I}$ also. Therefore,

$$\sum_{k=0}^{n} \sqrt{A_i A_j}\hat{p}_k(x_i)\hat{p}_k(x_j) = \delta_{ij}$$

which says (for $i = j$) that

$$A_j \sum_{k=0}^{n} \hat{p}_k^2(x_j) = 1.$$

That is,

$$\sqrt{A_j}[\hat{p}_0(x_j), \hat{p}_1(x_j), \ldots, \hat{p}_n(x_j)]^T$$

is a unit eigenvector of $\mathbf{J}$. Let's examine the first component of this unit eigenvector $\sqrt{A_j}\hat{p}_0(x_j)$. Because $p_0(x) \equiv 1$, we have that $\hat{p}_0(x) \equiv 1/\|1\|$. Therefore, $\sqrt{A_j}$ simply equals $\|1\|$ times the first component of the unit eigenvector of $\mathbf{J}$. We have the following theorem.

Theorem 38. *The nodes x_i for Gaussian quadrature are given by the eigenvalues of the Jacobi matrix*

$$\begin{bmatrix} a_1 & \sqrt{b_2} & & & \\ \sqrt{b_2} & a_2 & \sqrt{b_3} & & \\ & \ddots & \ddots & \ddots & \\ & & \sqrt{b_n} & a_{n-1} & \sqrt{b_{n+1}} \\ & & & \sqrt{b_{n+1}} & a_n \end{bmatrix}.$$

The weights are are given by $\|1\|$ times the square of the first element of the normalized eigenvectors. For Legendre polynomials, $\|1\| = 2$.

Theorem 39. *For any function $f \in C^{2n+2}$, the error from Gaussian quadrature is*

$$\frac{f^{(2n+2)}(\xi)}{(2n+2)!}\,(p_{n+1}, p_{n+1})$$

where p_{n+1} is the $n+1$-orthogonal polynomial and ξ is some point in the interval of integration.

Gauss–Legendre quadrature

Recall that Legendre polynomials take the weight $w = 1$ over the domain $[-1, 1]$. Then

$$\int_{-1}^{1} f(x)\,\mathrm{d}x = \sum_{i=0}^{n} A_i f(x_i)$$

with $A_i = \int_{-1}^{1} \ell_i(x)\,\mathrm{d}x$ where $\ell_i(x)$ are the Lagrange basis polynomials and the nodes x_i are the zeros of the Legendre polynomials. First note that we can extend Gauss–Legendre quadrature to any bounded domain by using an affine transformation

$$x = \varphi(\xi) = \frac{b-a}{2}\xi + \frac{a+b}{2}$$

which maps $[-1,1] \to [a,b]$. Then

$$\int_a^b f(x)\,\mathrm{d}x = \frac{b-a}{2}\int_{-1}^{1} f(\varphi(\xi))\,\mathrm{d}\xi$$

So,

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{i=0}^{n} \tilde{A}_i f(x_i)$$

with $x_i = \varphi(\xi_i)$ and $\tilde{A}_i = \frac{1}{2}(b-a)A_i$.

Now let's suppose that you want to integrate

$$\int_{-\infty}^{\infty} f(x)\,\mathrm{d}x.$$

We can make a change of variable using a transformation which maps $(-\infty,\infty) \to (-1,1)$. Take $x = \tan(\pi\xi/2)$. Then

$$\mathrm{d}x = \frac{\pi}{2}\sec^2\frac{\pi\xi}{2}\,\mathrm{d}\xi$$

Then

$$\int_{-\infty}^{\infty} f(x)\,\mathrm{d}x = \frac{\pi}{2}\int_{-1}^{1} f\left(\tan\frac{\pi\xi}{2}\right)\sec^2\frac{\pi\xi}{2}\,\mathrm{d}\xi.$$

11.4 Monte Carlo integration

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 (position and velocity). For six particles (each with three position and velocity coordinates), we would need to integrate over a 36-dimensional space. Using a uniform mesh with only two grid-points in each direction, we would still need compute 2^{36} nodes. With a petaflop supercomputer (one that does 10^{15} floating-point operations per second), we could expect a solution in under a second. But with only two nodes in each dimension, we are likely to get a poor approximation. To reduce the error in the approximation by half, we could

use twice as many points in each dimension. Now, we would need 2^{72} nodes—in other words, more than two months using the same computer. The problem is plagued by the so-called *curse of dimensionality*.

A different approach is using Monte Carlo integration in which the quadrature nodes are chosen randomly (really, deterministically with a pseudo-random number generator) according to some probability distribution. Let $\mu(f)$ be the mean value of the function f in a bounded region Ω:

$$\mu(f) = \left(\int_\Omega \mathrm{d}\mathbf{x}\right)^{-1} \int_\Omega f(\mathbf{x})\,\mathrm{d}\mathbf{x}.$$

That is, the integral of $f(\mathbf{x})$ is the scaled mean value of $f(\mathbf{x})$:

$$\int_\Omega f(\mathbf{x})\,\mathrm{d}\mathbf{x} = \mu(f)\int_\Omega \mathrm{d}\mathbf{x}.$$

Suppose that we pick n points in Ω randomly and independently with identical distributions. In this case, we have the average

$$\mathrm{m}_n(f) = \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{x}_i).$$

By the law of large numbers

$$\lim_{n\to\infty} \mathrm{m}_n(f) \to \mu(f)$$

in distribution. The quadrature error $e_n(f) = \mu(f) - \mathrm{m}_n(f)$ can be characterized by the variance

$$\mu\left(e_n(f)\right) = \mu\left(\mathrm{m}_n(f) - \mu(f)\right) = \sigma\left(\mathrm{m}_n(f)\right).$$

By the central limit theorem,

$$\sigma\left(\mathrm{m}_n(f)\right) = \frac{\sigma(f)}{\sqrt{n}}.$$

So, as $n \to \infty$, the convergence rate is $O(n^{-1/2})$. Such a convergence rate is rather poor in comparison with even a simple uniform sum which has a convergence rate of $O(n^{-1})$ for one-dimensional problems. But, the convergence rate does not depend on the dimension of the domain.

Using n nodes to integrate a function over a one-dimensional domain using a simple Riemann sum approximation $\int_0^1 f(x)\,\mathrm{d}x \approx n^{-1}\sum_{i=1}^{n} f(x_i)$ 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 $O(N^{-1/d})$. If the dimension $d > 2$, then the Monte Carlo method wins over simple Riemann sums.

Example. John von Neumann once quipped "anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin," but he conceded that it was far faster than feeding a computer random numbers from punched cards. A sequence of pseudo-random numbers is 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 later half of the twentieth century linear congruence generators were commonly used pseudo-random number generators until they were for the most part replaced by better methods like the Mersenne twister developed in 1997 by Matsumoto and Nishimura. A linear congruence generator is given by

$$x_{i+1} = (ax_i + c) \mod m$$

for some integers a, c, m, and seed x_0. The Lehmer formulation takes $c = 0$ and m to be a prime number. By Theorem 19 on page 19, by choosing a to be primative root of m, the sequence has maximal period $m - 1$. For example, by taking $m = 13$ and $a = 2$ with the seed $x_0 = 5$, we get the sequence

$$5 \quad 10 \quad 7 \quad 1 \quad 2 \quad 4 \quad 8 \quad 3 \quad 6 \quad 12 \quad 11 \quad 9,$$

which then repeats. The minimal standard generator developed by Park and Miller in 1988 takes m as the Mersenne prime $M_{31} = 2^{31} - 1$ and $a = 7^5$ as a primative root of M_{31}. ◀

Sampling

Often in statistical mechanics and molecular dynamics, we want to sample from a probability distribution that is not simply the uniform distribution $\mathrm{unif}(0, 1)$. Let's examine two sampling methods—inversion sampling and rejection sampling.

Inversion sampling A function $p(x)$ is a probability density function if $p(x) \geq 0$ and $\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1$. The cumulative distribution function $P(u) = \int_{-\infty}^{u} p(x)\,\mathrm{d}x$ is the probability that a sampled value will be less than u. The inverse distribution function is $P^{-1}(x)$ allows us to sample from $p(x)$ by sampling x from a uniform distribution. For example, the standard normal distribution $p(x) = \exp(-x^2/2)/\sqrt{2/\pi}$ has the the error function $u = P(x) = \operatorname{erf} x$ as its cumulative distribution. To sample a from a normal distribution, we would compute $P^{-1}(x) = \operatorname{erf}^{-1} x$ where x is sampled from a uniform distribution.

Example. We don't need to directly computing the inverse error function to sample from the standard normal distribution. We could instead use the Box–Muller transform. Consider

$$p(x)p(y) = \frac{1}{2\pi}\mathrm{e}^{-(x^2+y^2)/2}.$$

By making the change of variable $x = r\cos\theta$ and $y = r\sin\theta$, we have the differential element

$$p(x)p(y)\,\mathrm{d}x\,\mathrm{d}y = \frac{1}{2\pi}\mathrm{e}^{-r^2/2} r\,\mathrm{d}r\,\mathrm{d}\theta = \left(r\mathrm{e}^{-r^2/2}\,\mathrm{d}r\right)\left(\frac{1}{2\pi}\,\mathrm{d}\theta\right).$$

We want to sample r from the density $r\exp(-r^2/2)$ and θ from $1/2\pi$. To do this we will use inversion sampling:

$$\begin{aligned} u &= \int_0^r r\mathrm{e}^{-r^2/2}\,\mathrm{d}r = 1 - \mathrm{e}^{-r^2/2}, \\ v &= \int_0^\theta \frac{1}{2\pi}\,\mathrm{d}\theta = \frac{\theta}{2\pi}. \end{aligned}$$

Solving for r and θ gives us $r = \sqrt{-2\log(1-u)}$ and $\theta = 2\pi v$ where u and v are independently sampled from the uniform distribution $\mathrm{unif}(0,1)$. Because u is taken uniformly from $(0,1)$, we could also sample $1-u$ and instead take $r = \sqrt{-2\log u}$. Therefore,

$$\begin{aligned} x &= \sqrt{-2\log u}\cos(2\pi v) \\ y &= \sqrt{-2\log u}\sin(2\pi v) \end{aligned}$$

gives two independent normally distributed samples when u and v are independently uniformly distributed samples. ◀

Rejection sampling Computing the inverse of a cumulative density function is not always easy. Instead, another way of sampling from a distribution f is to sample from a different, but close, distribution g and use the acceptance-rejection algorithm (developed by von Neumann in 1951). Take M such that $f(x) \le Mg(x)$ for all x in our domain.

1. Sample x from distribution with density g.

2. Sample u from $\mathrm{unif}(0,1)$. If $u < f(x)/Mg(x)$, then accept the value x as a sample drawn from f. Otherwise, reject it and start again with the first step, repeating until you have an x that is accepted.

Metropolis algorithm The Metropolis algorithm provides a sequence of random samples from a probability distribution $p(x)$. Consider a symmetric distribution $g(x|y)$, where $g(x|y) = g(y|x)$ such as a normal distribution $g(x|y) = \exp(-(x-y)^2/2)/\sqrt{2/\pi}$. The extension of the Metropolis algorithm to nonsymmetric test distributions is called the Metropolis–Hastings algorithm. Take $f(x)$ proportional to $p(x)$. Choose an arbitrary x_0, and for each iteration i:

1. Sample x' by from the distribution with density $g(x'|x_i)$.
2. Sample u from unif$(0,1)$. If $u \leq f(x')/f(x_i)$ then accept the candidate 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 accepted values.

Exercises

✍ 11.1. In applications (such as applying boundary conditions, modeling a [discontinuous] shock, or solving an ordinary differential equation), one often wants to compute a one-side derivative.

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 $f''(x)$ with nodes at x, $x+h$,$x+2h$ and $x+3h$.

✍ 11.2. The three term-recurrence for the Legendre polynomials is well-known to be

$$p_{n+1} = xp_n(x) - \frac{n^2}{4n^2-1}p_{n-1}(x).$$

So, $a_n = 0$ and $b_{n+1} = n^2/(4n^2-1)$.

1. Use Gauss–Legendre quadrature to numerically compute

$$\int_{-1}^{1} \mathrm{e}^{-16x^2}\,\mathrm{d}x = \frac{\sqrt{\pi}}{4}\,\mathrm{erf}(4)$$

with an error of less than 10^{-14}. (Be careful: Make sure your weights are positive.)

2. Repeat the exercise using composite Simpson's quadrature. (Be careful: watch your indices and make sure you are using an even number of subintervals.)

3. Repeat each part to observe the convergence rate of the numerical solution to the exact solution. Loop over several different numbers of nodes and plot the numerical error as a function of the number of nodes. Use an appropriate plot (such as a loglog plot or a semilog plot). Determine the convergence rate. Does it agree with theory?

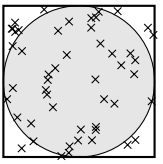
11.3. Now consider the integral

$$\int_0^1 x^{-1/2} \mathrm{e}^{-x} \, \mathrm{d}x = \sqrt{\pi} \, \mathrm{erf}(1).$$

1. Use Gauss–Legendre quadrature and composite Simpson's method to approximate the integral. Use the transformation $x = \frac{1}{2}(\xi + 1)$ with $dx = \frac{1}{2} d\xi$ to change the limits of integration from $[0, 1]$ to $[-1, 1]$. (A smarter transformation would be $x = \xi^2$, but this trivializes the problem.) Don't try too hard to get high accuracy.

2. Gauss–Legendre quadrature is lousy because of the singularity at 0. Compute a three-point Gaussian quadrature using the weight $w(x) = x^{-1/2}$.

✍ 11.4. One simple use of Monte Carlo integration is to compute π by noting that the area of a circle of unit radius is π. Take n samples $(x, y) \in [0, 1] \times [0, 1]$ and count the number of samples where (x, y) is in the unit circle, i.e., when $x^2 + y^2 < 1$. Then $\pi \approx m/4n$.



1. Use Monte Carlo integration to compute an approximation for π.

2. Use log-log plot of the error to confirm the convergence rate

3. Modify your solution in part 1 to use Monte Carlo integration to compute the volume of an 9-dimensional unit hypersphere.

Numerical Methods for Partial Differential Equations

CHAPTER 12

Ordinary Differential Equations

We begin the study of numerical methods for partial differential equations (PDEs) by examining numerical methods to solve ordinary differential equations (ODEs). One reason is that numerical techniques used to solve time-dependent partial differential equations often employ the *numerical method of lines* in which the space is discretized and the semi-discrete problem is solved using standard ODE solvers. For example, consider the heat equation $u_t = \Delta u$ where the subscript t denotes $\partial/\partial t$ and $\Delta = \sum_i \partial^2/\partial x_i^2$ is the Laplacian operator. The heat equation can be formally solved as a first-order linear ordinary differential equation to get the solution

$$u(x,t) = \mathrm{e}^{t\Delta} u(x,0)$$

where $\mathrm{e}^{t\Delta}$ is a linear operator acting on the initial conditions $u(x,0)$. A typical numerical solution involves discretizing the Laplacian operator Δ in space and employing an ODE solver in time. It is sufficient to consider first-order ODEs, because second-order and higher-order ODEs can be expressed as systems of first-order ODEs. Understanding stability, consistency and convergence of the numerical schemes for ordinary differential equations will help us to understand stability, consistency and convergence of the numerical schemes for partial differential equations.

12.1 Brief theory of ODEs

Consider the initial value problem

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f(t,u) \text{ with } u(0) = u_0.$$

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 $u, u^* \in \mathbb{R}$. For example,

$f(u) = \lvert u\rvert$	is Lipschitz on $\mathbb{R}$
$f(u) = u^2$	is not Lipschitz on $\mathbb{R}$ (it is locally Lipschitz)
$f(u) = \sqrt{\lvert u\rvert}$	is not Lipschitz on $\mathbb{R}$ (notably at $u = 0$)

Theorem 40. *If $f(t,u)$ is continuous in the region $0 \leq t \leq T$ and it is Lipschitz continuous in u, then there exists a unique continuously differentiable function $u(t)$ such that $u'(t) = f(t,u(t))$ and $u(0) = u_0$.*

Example. Let's see what happens when $f(t,u)$ is not Lipschitz. Take $f(t,u) = u^2$ with $u(0) = 1$ over the interval $t \in [0,2]$. The problem $u' = u^2$ has the solution

$$\frac{\mathrm{d}u}{u^2} = \mathrm{d}t \quad \Rightarrow \quad -\frac{1}{u} = t + C \quad \Rightarrow \quad u(t) = \frac{1}{1-t}.$$

Because, the solution blows up at $t = 1$, the solution does not exist on the interval $[1,2]$.

Now, consider $f(t,u) = \sqrt{u}$ with $u(0) = 0$ over the interval $t \in [0,2]$. The problem $u' = \sqrt{u}$ has the solution $u(t) = \frac{1}{4}t^2$. It also has the solution $u(t) = 0$. In fact, there are infinitely many solutions such as

$$u(t) = \begin{cases} 0 & t < t_c \\ \frac{1}{4}(t - t_c)^2 & t \geq t_c \end{cases}$$

where $t_c \in [0,2]$ is an arbitrary constant.

When $f(t,u)$ is not Lipschitz continuous is u, the solution may fail to exist; or if it exists, it may fail to be unique. ◀

We say that an initial value problem is *stable*, if a small perturbation in the initial data or in $f(t,u)$ produces a bounded change in the solution. Let's clarify what we mean by this.

Theorem 41. *An initial value problem*

$$\frac{\mathrm{d}u}{\mathrm{d}t} = f(t,u) \text{ with } u(0) = u_0 \tag{12.1}$$

is stable if there exists $K > 0$ such that for ε sufficiently small, the solution $v(t)$ to the perturbed problem

$$v'(t) = f(t, v) + \delta(t) \text{ with } v(0) = u_0 + \varepsilon_0 \tag{12.2}$$

satisfies $|v(t) - u(t)| \leq K\varepsilon$ whenever $|\varepsilon_0|, |\delta(t)| < \varepsilon$ for all $t \in [0, T]$.

Proof. The error is given by the difference between the solution $u(t)$ to the original problem (12.1) and the solution $v(t)$ to the perturbed problem (12.2): $e(t) = v(t) - u(t)$. Then

$$\begin{aligned} e'(t) &= v'(t) - u'(t) = f(t, v) - f(t, u) - \delta(t) \\ e(0) &= v(0) - u(0) = \varepsilon_0 \end{aligned}$$

If $f(t, u)$ is Lipschitz continuous is u, then

$$\begin{aligned} |e'(t)| &= |f(t, v) - f(t, u) - \delta(t)| \\ &\leq |f(t, v) - f(t, u)| + |\delta(t)| \\ &\leq L|v(t) - u(t)| + |\delta(t)| \\ &\leq L|e(t)| + |\delta(t)| \end{aligned}$$

Let ε be the maximum of ε_0 and $\max_{0 \leq t \leq T} |\delta(t)|$. Then

$$|e'(t)| \leq L|e(t)| + \varepsilon \qquad \text{and} \qquad |e(0)| \leq \varepsilon.$$

It follows that

$$|e(t)| \leq \frac{\varepsilon}{L}[(L+1)\mathrm{e}^{Lt} - 1] \leq \frac{\varepsilon}{L}[(L+1)\mathrm{e}^{LT} - 1] = \varepsilon K$$

for $0 \leq t \leq T$ which says that the error is bounded for $0 \leq t \leq T$. So, if $f(t, u)$ is Lipschitz continuous in u, then the initial value problem $u' = f(t, u)$ is stable. □

Here are the two takeaways from this section to keep in mind during not only the rest of this chapter but also the remainder of this course:

1. Well-posedness requires existence, uniqueness, and stability.

2. If $f(t, u)$ is continuous in t and Lipschitz continuous in u, then the problem $u' = f(t, u)$ is well-posed.

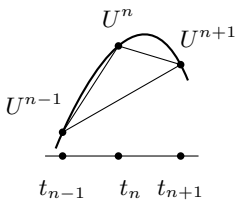
12.2 Numerical methods and stability

Let $u \equiv u(t)$. Suppose we are given with the initial value problem

$$u' = f(u) \qquad \text{where } u(0) = u_0. \tag{12.3}$$

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 use a fairly standard notation $k = \Delta t$ and $h = \Delta x$ to help tidy some otherwise messier expressions. The derivative u' can be approximated by difference operators:

Forward Difference	$\delta_+ U^n = \dfrac{U^{n+1} - U^n}{k}$
Backward Difference	$\delta_- U^n = \dfrac{U^n - U^{n-1}}{k}$
Centered Difference	$\delta_0 U^n = \dfrac{U^{n+1} - U^{n-1}}{2k}$



where U^n is the numerical approximation to $u(t_n)$. Using these operators, we can formulate several finite difference schemes to numerically solve the initial value problem (12.3):

Forward Euler (Explicit) $O(k)$

$$\frac{U^{n+1} - U^n}{k} = f(U^n) \tag{12.4}$$

Backward Euler (Implicit) $O(k)$

$$\frac{U^{n+1} - U^n}{k} = f(U^{n+1}) \tag{12.5}$$

Leapfrog (Explicit) $O(k^2)$

$$\frac{U^{n+1} - U^{n-1}}{2k} = f(U^n) \tag{12.6}$$

Trapezoidal (Implicit) $O(k^2)$

$$\frac{U^{n+1} - U^n}{k} = \frac{f(U^{n+1}) + f(U^n)}{2} \tag{12.7}$$

A finite difference scheme can be graphically represented using a *stencil*, which 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 t_{n+1} terms at the top. When we move to PDEs, we'll add a horizontal spatial component to the stencil.

In each of the above 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*.

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 $u(t+k)$:

$$u(t+k) = u(t) + ku'(t) + \tfrac{1}{2}k^2u''(t) + O(k^3)$$

Using the relation $u'(t) = f(u(t))$ gives

$$u(t+k) = u(t) + kf(u(t)) + k\tau_k$$

where the truncation $\tau_k = \frac{1}{2}ku'(t) + O(k^2)$. For $k\tau_k \approx 0$ we have

$$u(t+k) \approx u(t) + kf(u(t))$$

from which we have the forward Euler method

$$U^{n+1} = U^n + kf(U^n)$$

or equivalently

$$\frac{U^{n+1} - U^n}{k} = f(U^n).$$

The error for each step, given by the local truncation error $k\tau_k$, is $O(k^2)$. The error for the method, given by τ_k, is $O(k)$. We say that a numerical scheme is *consistent* if $\lim_{k\to 0} \tau_k = 0$, that is, we can make the numerical scheme approximate original problem to arbitrary accuracy.

We can improve the method by using a better approximation to the derivative. Taylor series

$$u(t_{n+1}) = u(t_n) + ku'(t_n) + \tfrac{1}{2}k^2u''(t_n) + \tfrac{1}{6}k^3u'''(t_n) + O(k^4)$$
$$u(t_{n-1}) = u(t_n) - ku'(t_n) + \tfrac{1}{2}k^2u''(t_n) - \tfrac{1}{6}k^3u'''(t_n) + O(k^4)$$

Subtracting the two gives us:

$$u(t_{n+1}) - u(t_{n-1}) = 2ku'(t_n) + \tfrac{1}{3}k^3u'''(t_n) + O(k^4)$$

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

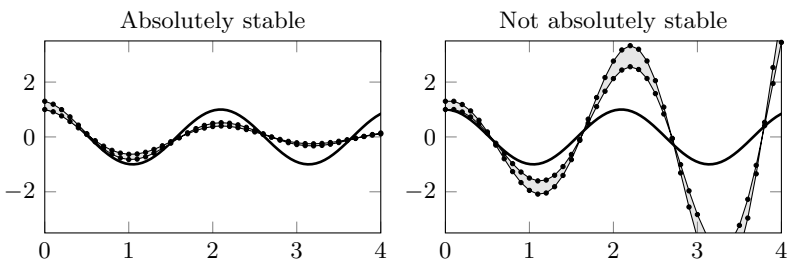
$$\frac{U^{n+1} - U^{n+1}}{2k} = f(U^n).$$

with truncation error $O(k^2)$.

We'll examine consistency in more depth when we look at multistep methods. For now, the forward Euler method is first order and the leapfrog method is second order. Similarly, the backward Euler method is first order and the trapezoidal method is second order.

Stability

A numerical method is *stable* if there exists a $K > 0$ such that a change in the starting values by a fixed amount produces a bounded change in the numerical solution for all $0 \le k \le K$ and $0 \le nk \le T$. Furthermore, a numerical method is *absolutely stable* if for a given step size k and a given differential equation, the change due to a perturbation of size δ in one of the time steps t_n is not larger than δ in any of the subsequent time steps. In other words, a method is stable if perturbations are bounded in finite time and absolutely stable if perturbations shrink over time.



Clearly, absolute stability is desirable because numerical errors will not only not grow but will diminish over time. Let's determine under what conditions a numerical method is absolutely stable. Let $U^0 = u(0)$ be the initial condition and V^0 be a perturbation of the 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

$$|e^n| = \left|\frac{e^n}{e^{n-1}}\right| \cdot \left|\frac{e^{n-1}}{e^{n-2}}\right| \cdots \left|\frac{e^1}{e^0}\right| \cdot |e^0|,$$

a sufficient condition for the error to be nonincreasing is for $|e^{n+1}/e^n| \le 1$ for all n. For $u'(t) = f(u,t)$, take the solution $v(t)$ of the initial conditions. Then for $e(t) = v(t) - u(t)$,

$$\begin{aligned} e' = v' - u' &= f(u+e) - f(u) \\ &= f(u) + ef'(u) + O(e^2) - f(u) = f'(u)e + O(e^2). \end{aligned}$$

For a system of ODEs, $f'(u)$ is the Jacobian matrix. We can diagonalize the Jacobian matrix and use its eigenvalues. This says that the error has the same growth rate as the solution $u(t)$, and we can study the error using the solution to the problem. Therefore, to determine the region of absolute stability, we will determine under what conditions $|r| = |U^{n+1}/U^n| \le 1$.

Regions of absolute stability

Consider the simple test problem $u'(t) = \gamma u(t)$ for an arbitrary complex value γ. For what time step k is a numerical scheme absolutely stable? To answer this question, we'll determine for what values $z = \gamma k$ the growth factor $|r| = |U^{n+1}/U^n| \le 1$. The region in the γk-plane where $|r| \le 1$ is called the *region of absolute stability* and it can be used to choose an appropriate numerical method. Let's find the regions of absolute stability for the forward Euler scheme, the backward Euler scheme, the leapfrog scheme and the trapezoidal scheme.

Forward Euler scheme. The forward Euler method for $u' = \gamma u$ is

$$\frac{U^{n+1} - U^n}{k} = \gamma U^n$$

from which we have that

$$U^{n+1} = (1 + \gamma k)U^n$$

and therefore

$$|U^{n+1}| \le |1 + \gamma k||U^n|.$$

Absolute stability requires $|1 + \gamma k| \le 1$, that is, the region inside the unit circle centered at $\gamma k = -1$. See Figure 12.1 on the following page. For a given value γ we may need to limit the size of our step k to ensure that

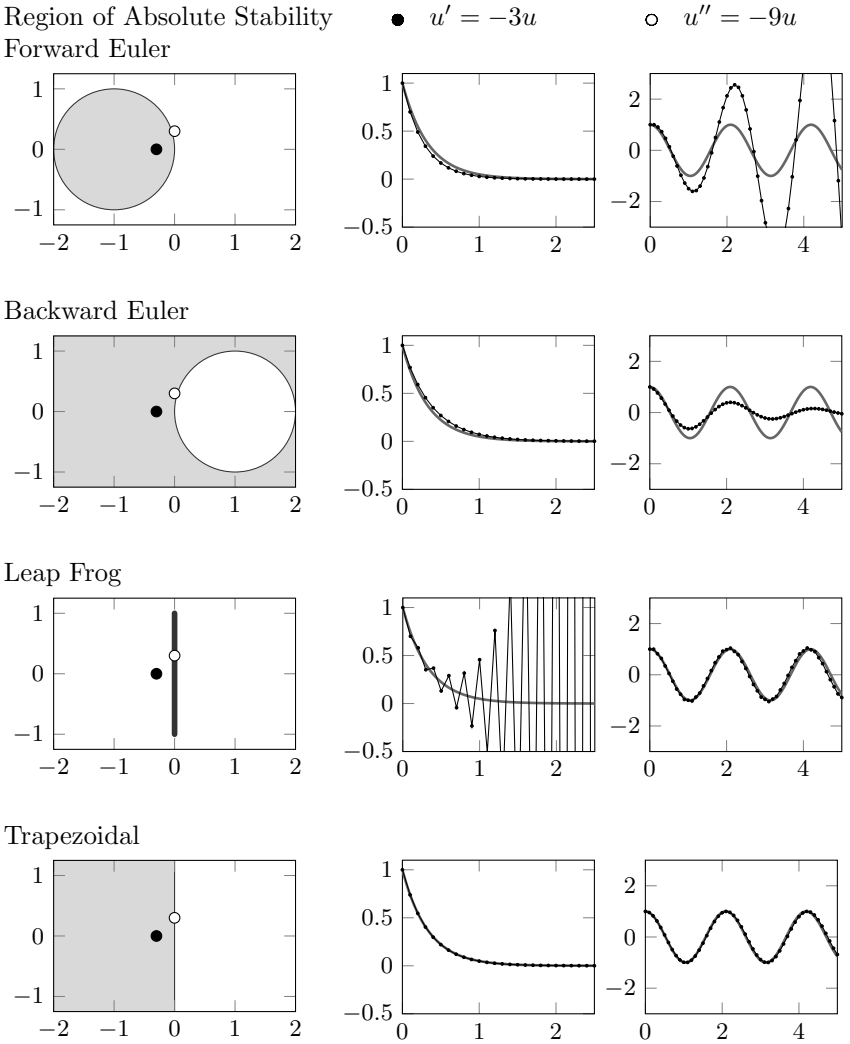


Figure 12.1: Regions of absolute stability and solutions to $u' = -3u$ and $u'' = -9u$ with stepsize $k = 0.1$ for forward Euler, backward Euler, leap frog and trapezoidal methods. The eigenvalues for these ODEs are -3 and $-3\mathrm{i}$ respectively. The value $\gamma k = -0.3$ (●) falls outside of the region of absolute stability for the leapfrog method and the value $\gamma k = -0.3\mathrm{i}$ (○) falls outside of the region of absolute stability for the forward Euler method.

γk 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 γ.

Backward Euler scheme. For the backward Euler method

$$\frac{U^{n+1} - U^n}{k} = \gamma U^{n+1}$$

from which we have that

$$(1 - \gamma k)U^{n+1} = U^n$$

and therefore

$$|U^{n+1}| \leq \frac{1}{|1 - \gamma k|}|U^n|$$

Absolute stability requires $|1 - \gamma k| \geq 1$. This means that we can take any size step size as long as we are outside of the unit circle centered at $\gamma k = 1$. A numerical method is said to be *A-stable*, if the region of absolute stability includes the entire left half γk-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 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

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -9 & 0 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} \quad \text{with} \quad \begin{bmatrix} u(0) \\ v(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

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. See Figure 12.1 on the preceding page.

Leapfrog scheme. We can rewrite

$$\frac{U^{n+1} - U^{n-1}}{k} = 2\gamma U^n,$$

as

$$U^{n+1} - 2\gamma k U^n - U^{n-1} = 0.$$

Let

$$r = \frac{U^{n+1}}{U^n} = \frac{U^n}{U^{n-1}}$$

then $r^2 = U^{n+1}/U^{n-1}$ and it follows that

$$r^2 - 2\gamma k r - 1 = 0.$$

We want to find when $|r| \le 1$ in the complex plane, so let's let $r = \mathrm{e}^{\mathrm{i}\theta}$ to use as the boundary. From this we have that

$$\mathrm{e}^{2\mathrm{i}\theta} - 2\gamma k \mathrm{e}^{\mathrm{i}\theta} - 1 = 0$$

and so

$$\mathrm{e}^{\mathrm{i}\theta} - \mathrm{e}^{-\mathrm{i}\theta} = 2\gamma k.$$

Therefore,

$$\mathrm{i}\sin\theta = \gamma k.$$

The leapfrog scheme is absolute stability only on the imaginary axis $\gamma k \in [-\mathrm{i}, +\mathrm{i}]$. See Figure 12.1 on page 268.

Trapezoidal scheme. Letting letting $r = U^{n+1}/U^n$ in the trapezoidal scheme

$$\frac{U^{n+1} - U^{n-1}}{k} = \gamma \frac{U^{n+1} + U^{n-1}}{2}$$

gives us

$$r^2 - 1 = \frac{1}{2}\gamma k(r^2 + 1)$$

or equivalently

$$\gamma k = 2\frac{r^2 - 1}{r^2 + 1}.$$

As before, take $r = \mathrm{e}^{\mathrm{i}\theta}$ to identify the boundary of the region of absolute stability. Then

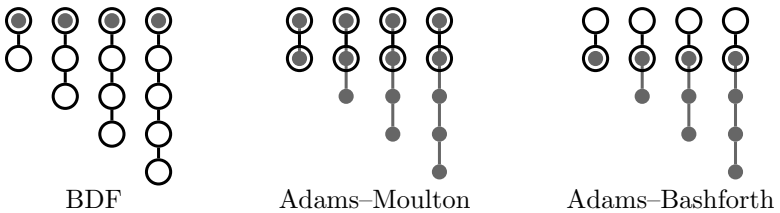
$$\gamma k = 2\frac{\mathrm{e}^{2\mathrm{i}\theta} - 1}{\mathrm{e}^{2\mathrm{i}\theta} + 1} = 2\mathrm{i}\tan\theta.$$

That is $\gamma k \in (-\mathrm{i}\infty, +\mathrm{i}\infty)$. So $|r| = 1$ along the entire imaginary axis (the domain boundary) and the region of absolute stability includes the entire left half γk-plane. Therefore, the trapezoidal scheme is A-stable. See Figure 12.1 on page 268.

A method is consistent if and only if the region of stability contains the origin $\gamma k = 0$. We call such a condition *zero-stability*.

12.3 Multistep methods

The forward and backward Euler methods both give first-order approximations to the solution by using piecewise linear functions. We can get a more accurate approximation to the derivative by using higher order piecewise polynomials—piecewise quadratic, piecewise cubic, and so forth. For an nth-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) and Adams methods are given below. Note that the first-order BDF is the same as the backward Euler, the second-order Adams–Moulton is the same as the trapezoidal, and the first-order Adams–Bashforth is the same as the forward Euler methods.



Backward differentiation formulas (BDF)

We can get a better approximation to the derivative of $u(t)$ by canceling out more terms of the Taylor series. We've already derived a first-order *backward differentiation formula*, the backward Euler method. Let's derive a second-order implicit method by using a piecewise quadratic approximation for u. Consider the nodes $u(t_n)$, $u(t_{n-1})$, and $u(t_{n-2})$:

$$\begin{aligned} u(t) &= u(t) \\ u(t-k) &= u(t) - ku'(t) + \tfrac{1}{2}k^2u''(t) + O(k^3) \\ u(t-2k) &= u(t) - 2ku'(t) + \tfrac{4}{2}k^2u''(t) + O(k^3) \end{aligned}$$

We want to cancel as many terms as possible to be left $O(k^3)$ by combining the three equations in some fashion. Let a_0, a_1 and a_2 be unknowns, then

$$\begin{aligned} a_0u(t) &= a_0u(t) \\ a_1u(t-k) &= a_1u(t) - a_1ku'(t) + \tfrac{1}{2}a_1k^2u''(t) + O(k^3) \\ a_2u(t-2k) &= a_2u(t) - 2a_2ku'(t) + \tfrac{4}{2}a_2k^2u''(t) + O(k^3). \end{aligned}$$

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

$$\begin{aligned} a_0 + a_1 + a_2 &= 0 \\ 0 - a_1 - 2a_2 &= 1/k \\ 0 + \tfrac{1}{2}a_1 + \tfrac{4}{2}a_2 &= 0. \end{aligned}$$

We solve this linear system for (a_0, a_1, a_2) and we get

$$a_0 = \tfrac{3}{2}\frac{1}{k}, \qquad a_2 = -2\frac{1}{k}, \qquad a_3 = \tfrac{1}{2}\frac{1}{k}.$$

So,

$$\frac{1}{k}\left(\tfrac{3}{2}u(t_n) - 2u(t_{n-1}) + \tfrac{1}{2}u(t_{n-2})\right) = u'(t_n) + O(k^2).$$

The method obtained is the second-order backward differentiation formula (BDF2)

$$\tfrac{3}{2}U^n - 2U^{n-1} + \tfrac{1}{2}U^{n-2} = kf(U^n).$$

An r-step (and rth order) backward differentiation formula takes the form

$$a_0U^n + a_1U^{n-1} + \cdots + a_{n-r}U^{n-r} = kf(U^n)$$

where the coefficients $a_0, \ldots, a_{n-r}$ are chosen appropriately.

Now, let's determine the region of absolute stability for BDF2. We take $f(u) = \gamma u$. Then letting $r = U^n/U^{n-1}$ in

$$\tfrac{3}{2}U^n - 2U^{n-1} + \tfrac{1}{2}U^{n-2} = kf(U^n)$$

we have

$$\tfrac{3}{2}r^2 - 2r + \tfrac{1}{2} = k\gamma r^2.$$

For which γk is $|r| \leq 1$? While we can easily compute this answer analytically using a 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 $r = \mathrm{e}^{\mathrm{i}\theta}$. Then the variable γk, can expressed as the rational function

$$\gamma k = \frac{\frac{3}{2}r^2 - 2r + \frac{1}{2}}{r^2}$$

We can plot this function using MATLAB:

```
r = exp(2i*pi*(0:.01:1));
plot((1.5*r.^2 - 2*r + 0.5)./r.^2);
```

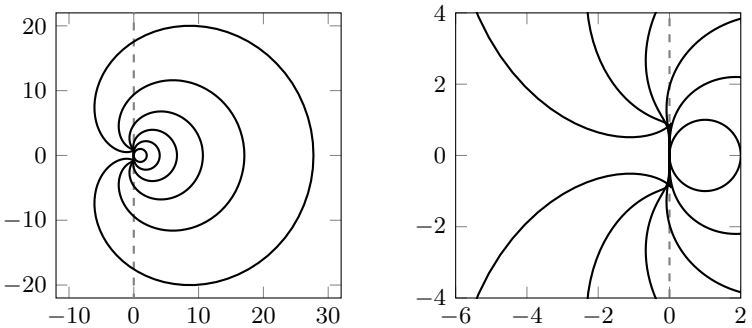


Figure 12.2: Left: internal contours of the regions of absolute stability for BDF1–BDF6. The regions of absolute stability are the areas outside these contours. Right: a close-up of the same contours.

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 12.2. This stability feature makes them particularly nice methods for stiff problems we'll encounter in Section 12.6. Note that BDF3 and higher are not A-stable, because their regions of absolute stability do not contain the entire left half-plane. BDF7 and higher are no longer zero-stable. The regions of absolute stability for the BDF methods get progressively smaller with increased accuracy. This tradeoff between stability and accuracy is typical but not necessary—higher order Runge–Kutta methods are often more stable than their lower order counterparts.

General multistep methods

Really, we are interested in getting a more accurate approximation to the whole equation $u' = f(u)$ and not simply the left hand side u'. A general s-step multistep method may be written as

$$\sum_{j=0}^{s} a_j U^{n-j} = k \sum_{j=0}^{s} b_j f(U^{n-j})$$

where the coefficients a_j and b_j are often zero. When $b_0 = 0$, the method is explicit; when $b_0 \neq 0$, the method is implicit.

For the initial value equation $u' = f(u)$ clearly $u' - f(u) = 0$. The

local truncation error of the finite difference approximation is defined as

$$\tau_{\text{local}} = \sum_{j=0}^{s} a_j u(t_{n-j}) - k \sum_{j=0}^{s} b_j f(u(t_{n-j}))$$

and it used to quantify the error for one time step. Error accumulates at each iteration. Because $t = nk$, it follows that $n = O(1/k)$. The *global truncation error* of a finite difference approximation is defined as

$$\tau_{\text{global}} = \frac{1}{k}\tau_{\text{local}}$$

and quantifies the error over several times steps. For example, the local truncation error of the leapfrog scheme is $O(k^3)$, but the global truncation error is $O(k^2)$.

From Taylor series expansion with $t_{n-j} = t_n - jk$, we have that

$$u(t_{n-j}) = u(t_n) - jku'(t_n) + \tfrac{1}{2}(jk)^2 u(t_n) + \cdots$$

$$f(u(t_{n-j})) = \frac{d}{dt}u(t_{n-j}) = u'(t_n) - jku''(t_n) + \tfrac{1}{2}(jk)^2 u'''(t_n) + \cdots$$

Then $\tau(t_n)$ equals

$$\frac{1}{k}\sum_{j=0}^{s} a_j u(t_n) - \sum_{j=0}^{s}(ja_j + b_j)u'(t_n) + k\sum_{j=0}^{s}(\tfrac{1}{2}j^2 a_j + b_j j)u''(t_n) - \cdots$$

$$\cdots - \frac{(-1)^p}{p!}k^{p-1}\sum_{j=0}^{s}\left(j^p a_j + pj^{p-1}b_j\right)u^{(p)}(t_n) + \cdots$$

From this equation, for consistency ($\lim_{k\to 0}\tau(t_n) = 0$) we need

$$\sum_{j=0}^{s} a_j = 0 \quad \text{and} \quad \sum_{j=0}^{s} ja_j + b_j = 0. \tag{12.8a}$$

To get an $O(k^p)$ method, we need to set the first $p+1$ coefficients to 0:

$$\sum_{j=0}^{s}\left(j^i a_j + ij^{i-1}b_j\right) = 0 \quad \text{for } i = 1, 2, \ldots, p. \tag{12.8b}$$

We can write this system in matrix form $\mathbf{Va} + \mathbf{V'b} = 0$ where $\mathbf{V}$ is the transpose of a Vandermonde matrix and $\mathbf{V'}$ is the tranpose of a confluent Vandermode matrix:

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & s \\ 1 & 2^2 & 3^2 & \cdots & s^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2^s & 3^s & \cdots & s^s \end{bmatrix}\begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_s \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 1 & \cdots & 1 \\ 2 & 4 & 6 & \cdots & 2s \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s & s2^{s-1} & s3^{s-1} & \cdots & s^s \end{bmatrix}\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_s \end{bmatrix} = 0.$$

Suppose that we want to maximize the order of an multistep method. The system (12.8) has $s+1$ equations with have $s+1$ unknown and $s+1$ prescribed variables. We might be tempted to try something like this

$$\begin{bmatrix} \mathbf{a}^T \\ \mathbf{b}^T \end{bmatrix} = \begin{bmatrix} 1 & a_1 & a_2 & \dots & a_{s/2} & 0 & \dots & 0 \\ b_0 & b_1 & b_2 & \dots & a_{s/2} & 0 & \dots & 0 \end{bmatrix}.$$

Afterall, 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 stability of the scheme. The order of a multistep method is limited by the first Dahlquist barrier.

Theorem 42 (First Dahlquist Stability Barrier)**.** *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.*

Proof. See Lambert [1991]. □

Multistep methods may require several starting values, i.e., to compute U^r for an r-step method, we first need $U^0, \dots, U^{r-1}$. For example, to implement the leapfrog method, first use 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 global truncation error from using one step of the 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 of the time derivative on the left-hand side, Adams methods have simple left-hand sides and complicated right-hand sides. A BDF method evaluates the time derivative at $n+1$ to match the right-hand side. Adams methods rely on the Mean Value Theorem to find an intermediate time somewhere in between t_n and $t_n + 1$. To do this, they work by adjusting the right hand side.

The Adams methods have the form $a_0 = 1$ and $a_1 = -1$ and $a_j = 0$ for $2 \le j \le s$. That is,

$$U^n = U^{n-1} + k \sum_{j=0}^{s} b_j f(U^{n-j}).$$

The explicit *Adams–Bashforth method* takes $b_0 = 0$ and the implicit *Adams–Moulton method* takes $b_0 \neq 0$. Note that the forward Euler is equivalent to the first-order Adams–Bashforth method (AB1) and that the trapezoidal method is equivalent to the second-order Adams–Moulton method (AM1).

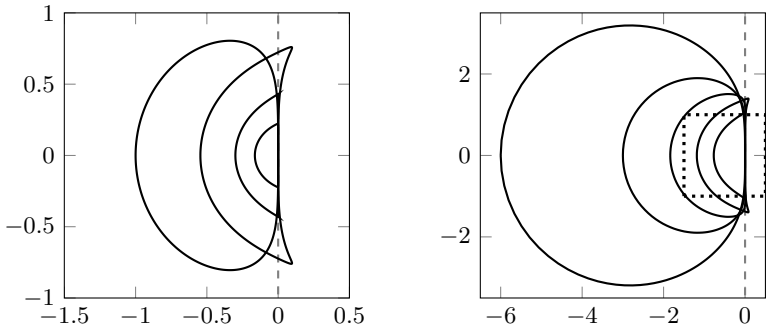


Figure 12.3: External contours of the regions of absolute stability for the explicit Adams–Bashforth methods AB2–AB5 (left) and implicit Adams–Moulton methods AM2–AM6 (right). The plots are at different scales. The plot region for the Adams–Bashforth is indicated with the dotted rectangle overlaying the plot region for the Adams–Moulton.

Predictor-corrector methods

Nonlinear implicit methods can be difficult to implement because they require an iterative nonlinear solver. One method is to use an explicit method for one time step to "predict" the implicit term. Then use an implicit method to "correct" the solution of the explicit method. Adams methods are particularly well-suited for combining as predictor-corrector methods:

$$\text{Predictor:} \qquad \tilde{U}^n = U^{n-1} + k\sum_{j=1}^{s} b_j f(U^{n-j}) \tag{12.9a}$$

$$\text{Corrector:} \qquad U^n = U^{n-1} + kb_0^* f(\tilde{U}^n) + k\sum_{j=1}^{s} b_j^* f(U^{n-j}). \tag{12.9b}$$

The two steps are often expressed as Predict–Evaluate–Correct–Evaluate (PECE) where "evaluate" refers to computing $f(U^n)$ with update U^n. 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.

Ⓜ The MATLAB function `ode113` uses an Adams–Bashforth–Moulton PECE solver with variable stepsize and variable orders 1 to 12.

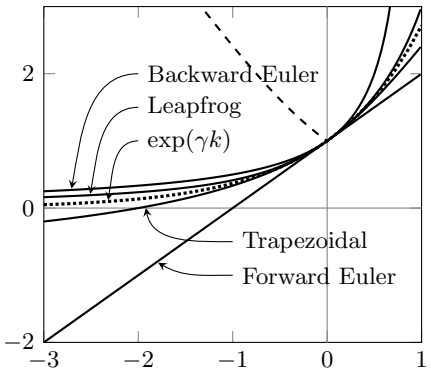


Figure 12.4: Plot of scaling factor r along the real axis. The backward Euler method is L-stable and r limits 0 as γk limits ∞. The trapezoidal method is A-stable but not L-stable and r limits -1. The leapfrog scaling is dominated by $|\gamma k| + \sqrt{|\gamma k|^2 + 1}$ depicted with a dashed curve. The ideal curve $\exp(\gamma k)$ depicted as a dotted curve.

Padé approximation

A *Padé approximation*

$$R_{m,n} = \frac{P_m(r)}{Q_n(r)} = \frac{\sum_{i=0}^{m} p_i r^i}{1 + \sum_{i=1}^{n} q_i r^i}$$

is a rational-function extension to the Taylor polynomial approximation to an analytic function. When $n = 0$, $Q_n(r) = 1$ and the Padé approximation is identical to a Taylor polynomial.

Recall that any linear differential equation $u' = \mathrm{L}\, u$ where L is an operator has the solution $u(t) = \exp(t\,\mathrm{L})u(0)$ where $\exp(t\,\mathrm{L})$ is an operator. The solution can be found using the numerical method $U^{n+1} = \exp(k\,\mathrm{L})U^n$. Defining $r = U^{n+1}/U^n$ as before we have $r = \exp(k\,\mathrm{L})$. Multistep methods can be viewed as a Padé approximations to exponential $r = \exp(\gamma k)$.

If $r(\gamma k)$ is an approximant of the exponential function $\exp(\gamma k)$, it follows that inverse γk is approximant of the logarithm evaluated at $(\gamma k, r) = (0, 1)$. That is to say, $\gamma k = P(r)/Q(r)$ is the Padé approximation to $\log r$ evaluated at the point $r = 1$ (at $\gamma k = 0$). For the Padé approximation $R_{m,n}(r)$ to $\log r$ and the Padé approximation $C_{m,n}(r)$ to $(\log r)/r$:

$R_{0,1} \leftrightarrow$ Forward Euler	$R_{p-1,0} \leftrightarrow$ BDF (order p)
$R_{1,0} \leftrightarrow$ Backward Euler	$R_{1,p-1} \leftrightarrow$ Adams–Moulton (order p)
$R_{1,1} \leftrightarrow$ Trapezoidal	$rC_{1,p-1} \leftrightarrow$ Adams–Bashforth (order p)
$rC_{2,0} \leftrightarrow$ Leapfrog	

12.4 Multistage: Runge–Kutta method

By directly integrating $u'(t) = f(u,t)$ we can change the differential equation into an integral equation

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(t, u(t))\,\mathrm{d}t.$$

The challenge now is to numerically evaluate the integral. One way of doing this is to use the Runge–Kutta method. Unlike multistep methods, Runge–Kutta methods can get high-order results without saving the solutions of previous time steps, making them good choices for adaptive step size. Because of this, Runge–Kutta methods are often methods of choice.

We can approximate the integral using quadrature

$$\int_{t_n}^{t_{n+1}} f(t, u(t))\,\mathrm{d}t \approx k \sum_{i=1}^{s} b_i f(t_i^*, u(t_i^*))$$

with appropriate weights b_i and nodes $t_i^* \in [t_n, t_{n+1}]$. In fact, by choosing the s nodes and weights so that we have a Gaussian quadrature, we can get a method that has order $2s$. But to apply this method we must already know the values $u(t_i^*)$. The method 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:

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)f(a).$$

Let's use this to compute U^{n+1} given U^n.

$$u(t_{n+1}) = u(t_n) + \int_{t_n}^{t_{n+1}} f(t, u(t))\,\mathrm{d}t \quad \Rightarrow \quad U^{n+1} = U^n + kf(t_n, U^n),$$

which is, of course, the forward Euler method. ◀

Example. The midpoint rule is a more accurate one-point quadrature rule

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)f\left(\frac{b+a}{2}\right).$$

This time we'll need to approximate $f(t_{n+1/2}, U^{n+1/2})$—we can use the forward Euler method. Take $K_1 = f(t_n, U^n)$ and

$$K_2 = f(t_n + \tfrac{1}{2}k, U^n + \tfrac{1}{2}kf(t_n, U^n)) = f(t_n + \tfrac{1}{2}k, U^n + \tfrac{1}{2}kK_1).$$

Then the midpoint rule says

$$u(t_{n+1}) = u(t_n) + \int_{t_n}^{t_{n+1}} f(t, u(t))\,\mathrm{d}t \quad \Rightarrow \quad U^{n+1} = U^n + kK_2.$$

This gives us the second-order Heun method:

$$\begin{aligned} K_1 &= f(t_n, U^n) \\ K_2 &= f(t_n + \tfrac{1}{2}k, U^n + \tfrac{1}{2}kK_1) \\ U^{n+1} &= U^n + kK_2 \end{aligned}$$

◀

Example. The trapezoidal rule is another one-point quadrature rule

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\left[\tfrac{1}{2}f(a) + \tfrac{1}{2}f(b)\right].$$

To compute U^{n+1} given U^n, we'll need to approximate $f(t_{n+1}, U^{n+1})$ using the forward Euler method. Take $K_1 = f(t_n, U^n)$ and

$$K_2 = f(t_n + k, U^n + kf(t_n, U^n)) = f(t_n + k, U^n + kK_1).$$

Combining these we have the second-order rule

$$U^{n+1} = U^n + k\left[\tfrac{1}{2}K_1 + \tfrac{1}{2}K_2\right]$$

◀

Example. To get high-order methods we add quadrature points. The next step up is Simpson's method, which says

$$\int_a^b f(x)\,\mathrm{d}x \approx (b-a)\left[\frac{1}{6}f(a) + \frac{2}{3}f\left(\frac{b+a}{2}\right) + \frac{1}{6}f(b)\right].$$

We'll need to approximate $f(t_{n+1/2}, U^{n+1/2})$ and $f(t_{n+1}, U^{n+1})$. ◀

The general s-stage Runge–Kutta method is given by

$$\begin{aligned} U^{n+1} &= U^n + k\sum_{i=1}^{n} b_i K_i \\ \text{with } K_i &= f\left(t_n + c_i k, U^n + k\sum_{j=1}^{s} a_{ij} K_j\right) \end{aligned} \tag{12.10}$$

In general, computing the coefficients a_{ij} is not a trivial exercise. For consistency we need to take $c_i = \sum_{j=1}^{s} a_{ij}$ and $\sum_{i=1}^{s} b_i = 1$. The coefficients are often conveniently given as a *Butcher tableau*

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

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, then the method is a diagonally implicit Runge–Kutta Method (DIRK).

The second-order Heun method (RK2) is given by

$$\begin{aligned} K_1 &= f(t_n, U^n) \\ K_2 &= f(t_n + \tfrac{1}{2}k, U^n + \tfrac{1}{2}kK_1) \\ U^{n+1} &= U^n + kK_2 \end{aligned} \qquad \begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array}$$

The fourth-order Runge–Kutta (RK4) is given by

$$\begin{aligned} K_1 &= f(t_n, U^n) \\ K_2 &= f(t_n + \tfrac{1}{2}k, U^n + \tfrac{1}{2}kK_1) \\ K_3 &= f(t_n + \tfrac{1}{2}k, U^n + \tfrac{1}{2}kK_2) \\ K_4 &= f(t_n + k, U^n + kK_3) \\ U^{n+1} &= U^n + \tfrac{1}{6}k(K_1 + 2K_2 + 2K_3 + K_4). \end{aligned} \qquad \begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

Example. Plot the regions of stability for the Runge–Kutta methods given by the following tableaus:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array} \qquad \begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ 1 & -1 & 2 & \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array} \qquad \begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

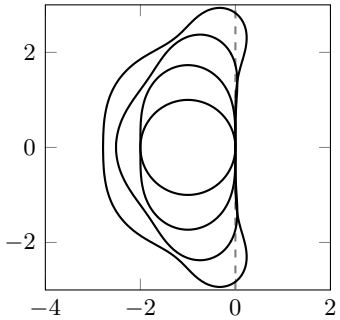


Figure 12.5: External contours of the regions of absolute stability for the explicit Runge–Kutta methods RK1–RK4. The regions of stability get progressively *larger* with increasing order.

The region of absolute stability is bounded by the $|r| = 1$ contour where $r = U^{n+1}/U^n$ where U^{n+1} is the solution to $u' = \gamma u$. For a Runge–Kutta method we can get an explicit expression for γk in terms of r. For $f(t, u) = \gamma u$, a general Runge–Kutta method is by (12.10)

$$U^{n+1} = U^n + k\mathbf{b}^T\mathbf{K}$$

with $\mathbf{K}$ given implicitly by

$$\mathbf{K} = \gamma(U^n\mathbf{E} + k\mathbf{A}\mathbf{K}) \tag{12.11}$$

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 (12.11) for $\mathbf{K}$ to get

$$\mathbf{K} = \gamma U^n(\mathbf{I} - \gamma k\mathbf{A})^{-1}\mathbf{E}$$

where $\mathbf{I}$ is an $s \times s$ identity matrix. Then

$$r = 1 + \gamma k\mathbf{b}^T(\mathbf{I} - \gamma k\mathbf{A})^{-1}\mathbf{E}. \tag{12.12}$$

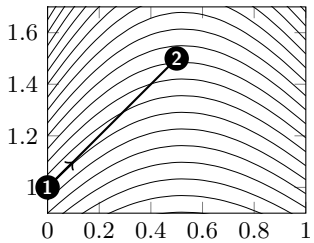
We can use MATLAB to compute the $|r| = 1$ contours of (12.12) for each tableau. The regions of absolute stability are plotted in Figure 12.5. ◀

Adaptive step size

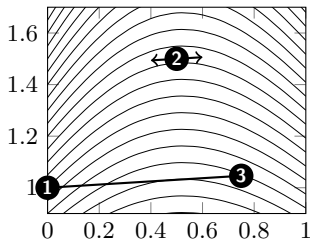
One advantage Runge–Kutta methods have over multistep methods is that it is easy to vary the step size k, taking smaller steps when we needed to minimize error and maintain stability and larger steps otherwise. For example, the Bogacki–Shampine method (implemented in MATLAB using the function `ode23`) combines a second-order and third-

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{3}{4} & 0 & \frac{3}{4} & & \\
1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\
\hline
 & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8}
\end{array}
$$

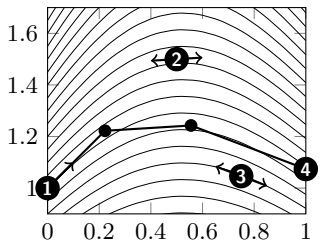
The Bogacki-Shampine scheme given by the Butcher tableau on the left is implemented in MATLAB using `ode23`. It can be implemented as an adaptive step size routine.



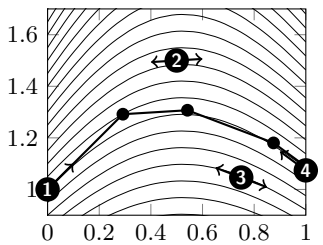
From the second row of the Butcher tableau. Starting with the slope $f(0, u(0))$ at ❶ we find the approximate solution at $t = \frac{1}{2}$ by using a simple forward Euler approximation to arrive at ❷.



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 ❷ and nothing from ❶. Again, we use a simple linear extrapolation.



The fourth row says we now try to find the solution at $t = 1$ by first traveling 2/9 with a slope given by ❶, then a distance 1/3 with a slope given by ❷, and then a distance 4/9 with slope from ❸



The final row tells us to first travel 7/24 with a slope given by ❶, then 1/4 with a slope given by ❷, then 1/3 with a slope given by ❸, and finally 1/8 at the slope from ❹. The solution provides a third-order correction over the solution from ❹.

Figure 12.6: 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 $u(0) = 1$.

order Runge–Kutta method using the same quadrature points

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{3}{4} & 0 & \frac{3}{4} & & \\
1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\
\hline
 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\
 & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8}
\end{array}
$$

For the Bogacki-Shampine, K_i are the same for the second-order and the third-order methods. By subtracting the solutions we are able to approximate the truncation error. If the truncation error is above a given tolerance, we set k to $k/2$. If the truncation error is below another given tolerance, we set k to $2k$.

12.5 Nonlinear ODEs

So far, we've focused on linear ODEs, which can be usually be solved analytically. Let's briefly discuss nonlinear ODEs starting with the stability condition. For linear ODEs, we used the eigenvalue γ 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 ODE $u' = f(u)$, we have

$$f(u) \approx f(u_0) + (u - u_0)f'(u_0) = f'(u_0)u + [f(u_0) - u_0 f'(u_0)]\,.$$

This says that we should consider $\gamma = \max_{u_0} f'(u_0)$ for our eigenvalue to determine stability and step size. For a system, we take the spectral radius (the magnitude of the largest eigenvalue) of the Jacobian matrix.

Implicit methods are more difficult to implement but they are more stable. To solve a nonlinear equation $\Phi(x) = 0$, one often uses Newton's method

$$X^{j+1} = X^j - [\Phi'(X^j)]^{-1}\Phi(X^j) \tag{12.13}$$

where $\Phi'(X^j)$ is the Jacobian matrix of Φ. Fixed-point iteration can also work:

$$X^{j+1} = X^j - \Phi(X^j). \tag{12.14}$$

Example. Solve the nonlinear ODE $u' = f(u)$ using the backward Euler method. We can't solve for U^n explicitly in $U^n = U^{n-1} + kf(U^n)$, so we use Newton's method. Let the $X^0 = U^{n-1}$. Then define

$$\Phi(X^j) = X^j - X^0 - kf(X^j)$$

which we use in (12.13). ◀

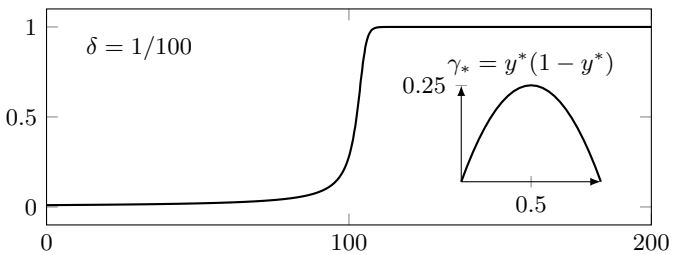
Both Newton's method and fixed point iteration may not converge (to the correct solution) if the initial guess is not sufficiently close to the target. Therefore, it is often useful to use an explicit method to get a sufficiently close solution and then use an implicit method to get even closer for each time step. Such a method is called a predictor-corrector method. Predictor-corrector methods often uses a combination of Adams–Bashforth and Adams–Moulton because the Adams–Moulton has significantly larger stability regions.

12.6 Stiff equations

Example. Consider the simple model for flame propagation

$$y' = y^2(1-y) \quad \text{with} \quad y(0) = \delta$$

where $y(t)$ is the radius of a ball of flame. See Moler [2004].



The solution slowly evolves when $y(t)$ is near zero until $t \approx 1/\delta$. At this point the solution rapidly changes to $y(t) \approx 1$. After this rapid change, the solution again slowly evolves asymptotically approaching $y = 1$. To get a better idea of what's going on in the nonlinear problem, consider the quasilinear form $y' = \gamma_* y$ with $\gamma_* = y_*(1-y_*)$ with a fixed value $y_* \approx y$. When y is close to 0 or 1, γ is close to 0, but when y increases γ increases to $1/4$ at $y = 1/2$. ◀

Example. Consider the behavior of the solution of the ODE

$$u'(t) = \gamma(\cos t - u) - \sin t \qquad u(0) = u_0 \quad \text{with} \quad \gamma > 0 \quad \text{and} \quad |\gamma| \gg 1.$$

The solution is

$$u(t) = (u_0 - 1)\mathrm{e}^{-\gamma t} + \cos t.$$

There are two scales—a slow time scale and a fast time scale—over which the solution evolves. Initially, the solution behaves like $u(t) = u_0 \mathrm{e}^{-\gamma t}$ which quickly decays. After a long time, the solution behaves like $u(t) = \cos t$. ◀

An initial value problem is said to be *stiff* if the dynamics react on at least two largely different time scales. As we shall see in the next chapter, diffusion equations are stiff. They change quickly when the spatial gradient is large and slowly when the spatial gradient is small, often evolving quickly until the solution smooths out and then evolving more slowly. 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.

To better quantify an initial value problem as stiff, we have the following equivalent definiton: $u' = f(u)$ is called stiff if the Jacobian matrix $f'(u)$ has a wide range of eigenvalues

$$\frac{\max|\gamma|}{\min|\gamma|} = \frac{\max|f'(u)|}{\min|f'(u)|} \gg 1,$$

meaning that there are at least two scales of different orders of magnitude.

Because a stiff problem has at least one relatively large eigenvalue, it is natural to use an implicit numerical method (especially an A-stable method), such as the backward Euler or trapezoidal method, to ensure stability even when the solution dynamics are evolving slowly. The trapezoidal method is A-stable, but it has transient (long-term) behavior when the eigenvalue is large and negative. For stiff problems, A-stability is usually not enough. For example, consider the initial value problem $u' = -\gamma u$ with $u(0) = 1$ where $\gamma > 0$. The trapezoidal method is

$$U^{n+1} - U^n = -\tfrac{1}{2}\gamma k \left(U^{n+1} + U^n\right)$$

or

$$U^{n+1} = r(\gamma k)U^n \quad \text{where} \quad r(\gamma k) = \frac{1 - \frac{1}{2}\gamma k}{1 + \frac{1}{2}\gamma k}$$

Because the trapezoidal method is A-stable, we can take a large time step size k and still ensure stability. When $\gamma k \gg 1$, $r(\gamma k) \approx 1$. And so,

$$U^n \approx (-1)^n U^0.$$

So, the numerical solution will oscillate for a long time. On the other hand, using the backward Euler method

$$U^n - U^{n-1} = -\gamma k U^n.$$

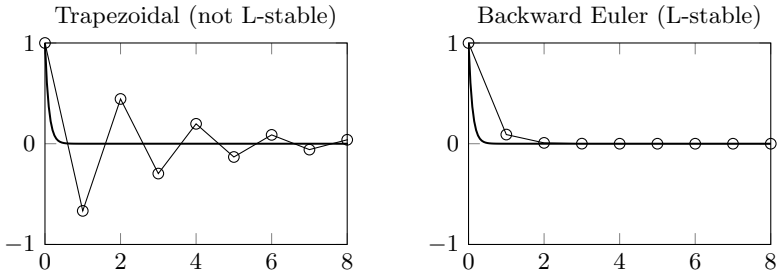


Figure 12.7: The numerical solutions to $u' = -10u$ using the second-order trapezoidal method and first-order backward Euler method with step size $k = 1$. The trapezoidal method is A-stable, but because it is not L-stable, it exhibits oscillations and slow decay.

Then

$$U^n = \frac{1}{1+\gamma k} U^{n-1} = \left(\frac{1}{1+\gamma k}\right)^n U^0 = (r(\gamma k))^n U^0 \to 0 \quad \text{as } |\gamma k| \to \infty.$$

A method $U^{n+1} = r(\gamma k)U^n$ is *L-stable* if $\lim_{|\gamma k|\to\infty} |r(\gamma k)| < 1$. In other words, a method is L-stable if the interior of its region of absolute stability contains $-\infty$. An L-stable scheme has more weight in the implicit term than the explicit terms. All backward differentiation forumulas are L-stable. See Figure 12.7.

12.7 Splitting methods

Often PDEs have stiff, linear terms coupled with non-stiff, nonlinear terms. Examples include

Navier–Stokes equation	$\rho \dfrac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} = -\nabla p + \mu \Delta \mathbf{v} + f$
reaction-diffusion equation	$\dfrac{\partial u}{\partial t} = \Delta u + u(1-u^2)$
nonlinear Schrödinger equation	$\mathrm{i}\dfrac{\partial \psi}{\partial t} = -\frac{1}{2}\Delta\psi + \lvert\psi\rvert^2\psi$

The stiff linear term is best treated implicitly, while the nonlinear term is best treated explicitly to simplify implementation. Such an approach is called *splitting*.

Operator splitting

Let's examine the stiff nonlinear ODE

$$u' = \cos^2 u - 10u \quad \text{with} \quad u(0) = u_0. \tag{12.15}$$

We can write the problem as $u' = \mathrm{N}\,u + \mathrm{L}\,u$, where $\mathrm{N}\,u = \cos^2 u$ and $\mathrm{L}\,u = -10u$. In this problem, the solution u is being forced by both $\mathrm{N}\,u$ and $\mathrm{L}\,u$ *concurrently*. We can't solve this problem analytically, but we can approximate the solution for small times by considering $\mathrm{N}\,u$ and $\mathrm{L}\,u$ as acting *successively*. We can do this in two ways:

1. First solve $u' = \mathrm{N}\,u$. Then, use the solution as the initial conditions for $u' = \mathrm{L}\,u$.

2. First solve $u' = \mathrm{L}\,u$. Then, use the solution as the initial conditions for $u' = \mathrm{N}\,u$.

Let's evaluate each of these approximate solutions after a time k.

1. The solution to $u' = \cos^2 u$ with initial condition $u(0) = u_0$ is $u(t) = \tan^{-1}(t + u_0)$. The solution to $u' = -10u$ with initial condition $\tan^{-1}(k + u_0)$ is

$$u(k) = \mathrm{e}^{-10k} \tan^{-1}\left(k + \tan u_0\right).$$

2. The solution to $u' = -10u$ with initial condition $u(0) = u_0$ is $u(t) = \mathrm{e}^{-10t} u_0$. The solution to $u' = \cos^2 u$ with initial condition $\mathrm{e}^{-10k} u_0$ is

$$u(k) = \tan^{-1}\left(k + \tan\left(\mathrm{e}^{-10k} u_0\right)\right).$$

The natural question to ask is "how close are these approximations to the exact solution?" Both of these solutions are plotted in Figure 12.8 on page 289.

Suppose we are given the problem $u' = \mathrm{A}\,u + \mathrm{B}\,u$ where A and B are arbitrary operators. Consider a simple splitting where we alternate by solving $u' = \mathrm{A}\,u$ and then solving $u' = \mathrm{B}\,u$ for the one time step. We can write (and implement) this procedure as

$$\begin{aligned} U^* &= \mathrm{S}_1(k) U^n \\ U^{n+1} &= \mathrm{S}_2(k) U^* \end{aligned}$$

or equivalently as

$$U^{n+1} = \mathrm{S}_2(k)\,\mathrm{S}_1(k) U^n$$

where $\mathrm{S}_1(k)$ and $\mathrm{S}_2(k)$ are solution operators.

What is the splitting error of applying the operators A and B successively rather than concurrently? Suppose that $\mathrm{S}_1(k)$ and $\mathrm{S}_2(k)$ are exact solution operators

$$\mathrm{S}_1(k)U^n = \mathrm{e}^{k\,\mathrm{A}}U^n \quad \text{and} \quad \mathrm{S}_2(k)U^n = \mathrm{e}^{k\,\mathrm{B}}U^n$$

Then

$$\begin{aligned} \text{Concurrently:} \qquad & U^{n+1} = \mathrm{e}^{k(\mathrm{A}+\mathrm{B})}U^n \\ \text{Successively:} \qquad & \tilde{U}^{n+1} = \mathrm{e}^{k\,\mathrm{B}}\mathrm{e}^{k\,\mathrm{A}}U^n \end{aligned}$$

The splitting error is $|U^{n+1} - \tilde{U}^{n+1}|$. By Taylor series expansion, we have that

$$\begin{aligned} \mathrm{e}^{k\,\mathrm{A}} &= \mathrm{I} + k\,\mathrm{A} + \tfrac{1}{2}k^2\,\mathrm{A}^2 + O(k^3) \\ \mathrm{e}^{k\,\mathrm{B}} &= \mathrm{I} + k\,\mathrm{B} + \tfrac{1}{2}k^2\,\mathrm{B}^2 + O(k^3) \end{aligned}$$

and hence

$$\begin{aligned} \mathrm{e}^{k\,\mathrm{B}}e^{k\,\mathrm{A}} &= \mathrm{I} + k(\mathrm{A}+\mathrm{B}) + \tfrac{1}{2}k^2(\mathrm{A}^2 + \mathrm{B}^2 + 2\,\mathrm{B}\,\mathrm{A}) + O(k^3) \\ \mathrm{e}^{k(\mathrm{A}+\mathrm{B})} &= \mathrm{I} + k(\mathrm{A}+\mathrm{B}) + \tfrac{1}{2}k^2(\mathrm{A}^2 + \mathrm{B}^2 + \mathrm{A}\,\mathrm{B} + \mathrm{B}\,\mathrm{A}) + O(k^3). \end{aligned}$$

Subtracting these two operators gives us

$$\mathrm{e}^{k(\mathrm{A}+\mathrm{B})} - \mathrm{e}^{k\,\mathrm{B}}\mathrm{e}^{k\,\mathrm{A}} = \tfrac{1}{2}k^2(\mathrm{A}\,\mathrm{B} + \mathrm{B}\,\mathrm{A} - 2\,\mathrm{B}\,\mathrm{A}) + O(k^3)$$

There is no splitting error if $\mathrm{A}\,\mathrm{B} = \mathrm{B}\,\mathrm{A}$. 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 splitting error in operator splitting by using *Strang splitting*. For each time step, we solve $u' = \mathrm{A}\,u$ for a half time step; then solve $u' = \mathrm{B}\,u$ for the a full time step; and finally, solve $u' = \mathrm{A}\,u$ for a half time step. Because

$$\mathrm{e}^{k\,\mathrm{A}/2}\mathrm{e}^{k\,\mathrm{B}}\mathrm{e}^{k\,\mathrm{A}/2} - \mathrm{e}^{k(\mathrm{A}+\mathrm{B})} = O(k^3),$$

Strang splitting is second order. (See Exercise 12.7.) Note that

$$\begin{aligned} U^{n+1} &= \mathrm{S}_1(\tfrac{1}{2}k)\,\mathrm{S}_2(k)\,\mathrm{S}_1(\tfrac{1}{2}k)U^n \\ &= \mathrm{S}_1(\tfrac{1}{2}k)\,\mathrm{S}_2(k)\,\mathrm{S}_1(\tfrac{1}{2}k)\,\mathrm{S}_1(\tfrac{1}{2}k)\,\mathrm{S}_2(k)\,\mathrm{S}_1(\tfrac{1}{2}k)U^{n-1} \end{aligned}$$

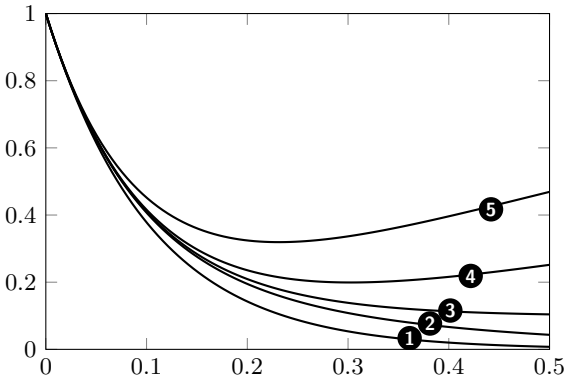


Figure 12.8: Solution to (12.15) using general splitting in ❶ and ❺, Strang splitting in ❷ and ❹, and exact solution in ❸. Each operator is inverted exactly without numerical error over a time step $k = 0.5$.

But since $\mathrm{S}_1(\frac{1}{2}k)\,\mathrm{S}_1(\frac{1}{2}k) = \mathrm{S}_1(k)$, this becomes

$$= \mathrm{S}_1(\tfrac{1}{2}k)\,\mathrm{S}_2(k)\,\mathrm{S}_1(k)\,\mathrm{S}_2(k)\,\mathrm{S}_1(\tfrac{1}{2}k)U^{n-1}$$

Continuing like this we get

$$= \mathrm{S}_1(\tfrac{1}{2}k)\big[\mathrm{S}_2(k)\,\mathrm{S}_1(k)\big]^{n-1}\,\mathrm{S}_2(k)\,\mathrm{S}_1(\tfrac{1}{2}k)U^0$$

So, to implement Strang splitting, we only need to solve $u' = \mathrm{A}\,u$ for a half time step on the initial and final time steps. In between, we use simple splitting. In this manner, we get $O(k^2)$ global splitting error without much more computation. It is impossible to have a splitting method with splitting error $O(k^3)$ or smaller.

IMEX methods

Another way to approach stiff problems is to use an implicit-explicit (IMEX) method. This method keeps the same discretization for the time derivative but uses an implicit discretization for the stiff linear term and an explicit discretization for the nonlinear term.

Example. A typical second-order IMEX method combines the second-order Adams–Bashforth and the second-order Adams–Moulton methods. Recall that the second-order Adams–Moulton method is really just the trapezoidal method. For both of these methods, the time derivatives are approximated to second-order at $t_{n+1/2}$. For $u' = \mathrm{L}\,u + \mathrm{N}\,u$, we have

$$\begin{aligned}\frac{U^{n+1} - U^n}{k} &= \mathrm{L}\,U^{n+1/2} + \mathrm{N}\,U^{n+1/2} \\ &\approx \tfrac{1}{2}\,\mathrm{L}\,U^{n+1} + \tfrac{1}{2}\,\mathrm{L}\,U^n + \tfrac{3}{2}\,\mathrm{N}\,U^n - \tfrac{1}{2}\,\mathrm{N}\,U^{n-1}\end{aligned}$$

◀

Example. We can also build a second-order IMEX scheme using BDF2 for the implicit part:

$$u' = \mathrm{L}\,u \qquad \Rightarrow \qquad \frac{1}{k}\left(\tfrac{3}{2}U^{n+1} - 2U^n + \tfrac{1}{2}U^{n-1}\right) = \mathrm{L}\,U^{n+1}$$

Backward differentiation formulas give us an approximation of the derivative at t_{n+1}. So, to avoid splitting error we should also evaluate the $\mathrm{N}\,U$ at U^{n+1} by extrapolating using the terms U^n and U^{n-1} to approximate U^{n+1}. We use Taylor series to find the approximation:

$$\begin{aligned} u(t_n) &= u(t_{n+1}) - ku'(t_{n+1}) + \frac{k^2}{2}u''(t_{n+1}) + O(k^3) \\ u(t_{n-1}) &= u(t_{n+1}) - 2ku'(t_{n+1}) + \frac{4k^2}{2}u''(t_{n+1}) + O(k^3). \end{aligned}$$

Combining these equations we have that

$$2u(t_n) - 1u(t_{n-1}) = u(t_{n+1}) - k^2u''(t_{n+1}) + O(k^3).$$

So, the IMEX method for $\mathrm{D}\,U^{n+1} = \mathrm{L}\,U^{n+1} + \mathrm{N}\,U^{n+1}$ is

$$\frac{\frac{3}{2}U^{n+1} - 2U^n + \frac{1}{2}U^{n-1}}{k} = \mathrm{L}\,U^{n+1} + \mathrm{N}\left[2U^n - U^{n-1}\right].$$

We can get higher order by using higher-order extrapolation. ◀

Other ways of implementing IMEX methods include IMEX Adams Methods which combine Adams–Bashforth and Adams–Moulton methods and IMEX Runge–Kutta methods. See Ascher et al. [1997].

Integrating factors

Integrating factors give us another way to deal with stiff problems. Consider the problem

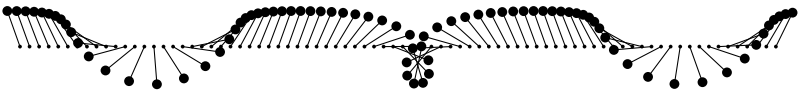
$$u' = \mathrm{L}\,u + \mathrm{N}\,u \quad \text{with} \quad u(0) = u_0$$

where L is a linear operator and N is a nonlinear operator. If we set $v = \mathrm{e}^{-t\,\mathrm{L}}\,u$, then

$$v' = \mathrm{e}^{-t\,\mathrm{L}}\,\mathrm{N}\left(\mathrm{e}^{t\,\mathrm{L}}\,v\right) \quad \text{with} \quad v(0) = \mathrm{e}^{-t\,\mathrm{L}}\,u_0.$$

There is no splitting error, but such a method of solution may only mollify the stiffness.

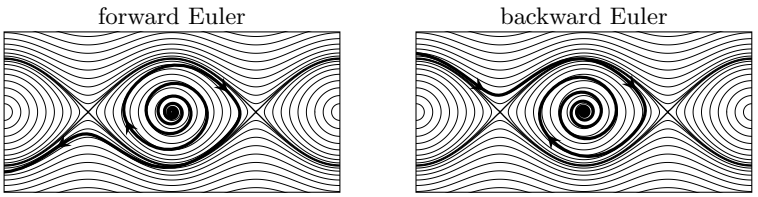
12.8 Symplectic integrators



The equation of motion of a simple pendulum is $\theta'' + \kappa \sin\theta = 0$ where $\theta(t)$ is the angle and the coefficient $\kappa = g/\ell$ is the acceleration due to gravity divided by the length of the pendulum. We'll take the mass, length, and gravitational acceleration to be one to simplify the discussion. In this nondimensionalized system with $\kappa = 1$ and with the position $q(t)$ and the angular velocity $p(t)$, we have the system of equations

$$\frac{\mathrm{d}q}{\mathrm{d}t} = p \quad \text{and} \quad \frac{\mathrm{d}p}{\mathrm{d}t} = \sin q.$$

This problem can be solved analytically in terms of elliptic functions. See Ochs [2011]. 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 $H = T + V = \frac{1}{2}p^2 - \cos q$ 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 ($q_0 = 0$ and $p_0 = 0.05$) and backward Euler methods ($q_0 = 0$ and $p_0 = 2.2$) with timestep $k = \frac{1}{5}$. In both cases, the pendulum starts at the bottom of its swing. When the pendulum has sufficient momentum $|p_0| > 2$, there is enough energy to for it to swing over the top. Otherwise, the pendulum swings back and forth with $|q(t)| < \pi$.



The forward Euler method is unstable and the total energy grows exponentially in time. The pendulum starts off with very little movement. Over time, the pendulum continues to speed up, 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. Now, the pendulum starts with enough energy to swing over the top, but over time energy dissipates until the pendulum all but stops at the bottom of its

swing. In both methods, the energy is not conserved giving an unphysical solution. In many dynamical problems, such as planetary motion and molecular dynamics, using a scheme that conserves the energy over a long period of time is important to get a physically correct solution. Using smaller timesteps and higher order methods can help, but over time energy may still not be conserved. So, let's look at a class of solvers that is aimed at conserving the Hamiltonian.

The pendulum belongs to the general class of Hamiltonian systems

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -\nabla_q H \quad \text{and} \quad \frac{\mathrm{d}q}{\mathrm{d}t} = +\nabla_p H.$$

By defining $r \equiv (p, q)$:

$$\frac{\mathrm{d}r}{\mathrm{d}t} = \mathrm{D}_H\, r \equiv J\nabla H(r) \quad \text{where} \quad J = \begin{bmatrix} 0 & -\mathrm{I} \\ +\mathrm{I} & 0 \end{bmatrix}.$$

For a time-independent Hamiltonian $H(p, q)$

$$\frac{\mathrm{d}H}{\mathrm{d}t} = \nabla_p H \frac{\mathrm{d}p}{\mathrm{d}t} + \nabla_q H \frac{\mathrm{d}q}{\mathrm{d}t} = 0,$$

and we see that the Hamiltonian is a conserved quantity.

Symplectic methods are numerical methods that seek to preserve the volume of a differential two-form $|dp \wedge dq|$ in phase space and conserve the Hamiltonian. Let's start with the semi-implicit or symplectic Euler method, a first-order scheme that combines the purely implicit forward Euler method and the purely explicit backward Euler method:

$$P^{n+1} = P^n - k\nabla_q H(P^{n+1}, Q^n) \tag{12.16a}$$

$$Q^{n+1} = Q^n + k\nabla_p H(P^{n+1}, Q^n) \tag{12.16b}$$

or alternatively

$$P^{n+1} = P^n - k\nabla_q H(P^n, Q^{n+1}) \tag{12.17a}$$

$$Q^{n+1} = Q^n + k\nabla_p H(P^n, Q^{n+1}). \tag{12.17b}$$

Let's restrict ourselves 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 by implementing (12.16) for a half-timestep followed by (12.17) for a half-timestep:

$$\begin{aligned} P^{n+1/2} &= P^n - \tfrac{1}{2}k\nabla_q V(Q^n) \\ Q^{n+1} &= Q^n + k\nabla_p T(P^{n+1/2}) \\ P^{n+1} &= P^{n+1/2} - \tfrac{1}{2}k\nabla_q V(Q^{n+1}) \end{aligned}$$

This second-order approach, called the Verlet method, was reintroduced by Loup Verlet in 1967 (Verlet [1967]) and can be viewed as an application of Strang splitting.

We can continue to build a higher order method by composing Verlet methods together. For a fourth-order method, take a Verlet with a timestep of αk followed by a Verlet with timestep $(1-2\alpha)k$ followed by a Verlet with timestep αk where $\alpha = (2-2^{-1/3})^{-1}$. See Forest and Ruth [1990], Yoshida [1990], Hairer et al. [2006].

12.9 Theory: consistency, stability, convergence

We'll wrap up this chapter with the Lax Equivalence Theorem, which says 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 ODEs and PDEs to give us the right results.

Consider the general initial value problem $u_t = \mathrm{L}\, u$ where L is a linear operator. In the next chapter, we will consider L to be the Laplacian operator and we will get the heat equation $u_t = \Delta u$. In Chapter 14, we will consider $\mathrm{L} = c\cdot\nabla$ and we will get the advection equation $u_t = c\cdot\nabla u$.

Suppose that $u(t,\cdot)$ is the analytic solution to $u_t = \mathrm{L}\, u$. Let $t_n = nk$ be the uniform discretization of time and U^n denote the finite difference approximation of $u(t_n,\cdot)$. Define a finite difference operator H_k such that $U^{n+1} = \mathrm{H}_k U^n$. The subscript k in H_k denotes the dependence on the step size k. Of course, $U^{n+1} = \mathrm{H}_k^{n+1} U^0$. For a first-order ODE, U^n and H_k are scalars. For a system of ODEs or a PDE, U^n is a vector and H_k is a matrix. Define the error $E_k(t_n,x) = U^n - u(t_n,x)$. A method is *convergent* in some particular norm $\|\cdot\|$ if the error $\|E_k(t,\cdot)\| \to 0$ as $k \to 0$ for any fixed $t \ge 0$ and for any initial data u_0. Define the *truncation error* to be

$$\tau_k(t_n,\cdot) = \frac{1}{k}\big[u(t+k,\cdot) - \mathrm{H}_k u(t,\cdot)\big].$$

A method is *consistent* if the truncation error $\|\tau_k(t,\cdot,\| \to 0$ as $k \to 0$. A method is of *order* p if for all sufficiently smooth initial data with compact support, there exists a constant C_τ such that $\|\tau_k(\cdot,t)\| \le C_\tau k^p$ for all $k < k_0$ and $0 \le t < T$.

Define an operator norm $\|\cdot\|$ as

$$\|A\| = \max_{x\in\mathbb{R}^n} \frac{\|Ax\|}{\|x\|} = \max_{\|x\|=1} \|Ax\|$$

A method is *stable* if for each time T, there exists a C_s and $k_0 > 0$ such that $\|\mathrm{H}_k^n\| \le C_s$ for all $nk \le T$ and $k \le k_0$. If $\|\mathrm{H}_k\| \le 1$, then $\|\mathrm{H}_k^n\| \le \|\mathrm{H}_k\|^n \le 1$ which implies stability. We can even relax conditions to $\|\mathrm{H}_k\| \le 1 + \alpha k$ for some constant α, since in this case

$$\|\mathrm{H}_k^n\| \le \|\mathrm{H}_k\|^n \le (1+\alpha k)^n \le \mathrm{e}^{\alpha k n} \le \mathrm{e}^{\alpha T}.$$

Theorem 43 (Lax Equivalence Theorem)**.** *A consistent, stable method is convergent.*

Proof. For a finite difference operator H_k, the truncation error at t_{n-1} is

$$\tau_k(t_{n-1}, x) = \frac{1}{k}(u(t_n,\cdot) - \mathrm{H}_k(u(t_{n-1},\cdot)).$$

We can write the exact solution as

$$u(t_n,\cdot) = \mathrm{H}_k u(t_{n-1},\cdot) + k\tau_k(t_{n-1},\cdot).$$

Furthermore, the error at t_n is

$$\begin{aligned}
E_k(t_n,\cdot) &= U^n - u(t_n,\cdot) \\
&= \mathrm{H}_k U^{n-1} - \mathrm{H}_k u(t_{n-1},\cdot) - k\tau_k(t_{n-1}) \\
&= \mathrm{H}_k E(t_{n-1},\cdot) - k\tau_k(t_{n-1},\cdot) \\
&= \mathrm{H}_k\left[\mathrm{H}_k(E(t_{n-2},\cdot)) - k\tau_k(t_{n-2},\cdot)\right] - k\tau_k(t_{n-1},\cdot) \\
&= \quad\vdots \\
&= \mathrm{H}_k^n E(0,\cdot) - k\sum_{j=1}^{n} \mathrm{H}_k^{j-1}\tau_k(t_{j-1},\cdot).
\end{aligned}$$

By the triangle inequality

$$\|E_k(t_n,\cdot)\| \le \|\mathrm{H}_k^n\|\|E(0,\cdot)\| + k\sum_{j=1}^{n} \|\mathrm{H}_k^{j-1}\|\|\tau_k(t_{j-1},\cdot)\|$$

From the stability condition, we have that $\|\mathrm{H}_k^j\| \le C_s$ for all $1 \le j \le n$ and all $0 \le nk \le T$.

$$\|E_k(t_n,\cdot)\| \le C_S\left(\|E(0,\cdot)\| + k\sum_{j=1}^{n}\|\tau_k(t_{j-1},\cdot)\|\right)$$

From the consistency condition, we have that $\|\tau_k(t_{j-1},\cdot)\| \to 0$ as $k \to 0$. If the method is pth order consistent, then $\|\tau_k(\cdot,t)\| \le C_\tau k^p$

$$\begin{aligned}\|E_k(t_n,\cdot)\| &\le C_S(\|E_k(0,\cdot)\| + C_\tau k^p kn)\\ &\le C_1(\|E_k(0,\cdot)\| + Tk^p).\end{aligned}$$

If the initial error $\|E_k(0,\cdot)\| \le C_2 k^p$ then $\|E_k(t_n,\cdot)\| \le Ck^p$ for some C for all $0 \le nk \le T$. □

Exercises

✍ 12.1. A θ-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 θ-scheme. Hint: Express the complex variable γk as $x+iy$ and find the locus of points for $r^2 \le 1$.

✍ 12.2. Plot contours in the γk-plane using several values of $|r|$ for the forward Euler, backward Euler, trapezoidal, and leapfrog methods.

12.3. Use the system of equation (12.8) to compute the 3-step Adams–Bashforth, Adams–Moulton and BDF methods. Plot the regions of absolute stability for these methods.

12.4. Show that for a linear problem, one step of the Crank–Nicolson method is the same as applying a forward Euler method for one half time step followed by the backward Euler methods for one half time step.

✍ 12.5. Plot the region of stability for the Merson method, whose Butcher tableau is

$$\begin{array}{c|ccccc}
0 & & & & & \\
\frac{1}{3} & \frac{1}{3} & & & & \\
\frac{1}{3} & \frac{1}{6} & \frac{1}{6} & & & \\
\frac{1}{2} & \frac{1}{8} & 0 & \frac{3}{8} & & \\
1 & \frac{1}{2} & 0 & -\frac{3}{2} & 2 & \\
\hline
 & \frac{1}{6} & 0 & 0 & \frac{2}{3} & \frac{1}{6}
\end{array}$$

.

12.6. Let $\varepsilon \ll 1$ and δ
$ll1$ and consider the ODE $u'(t) = -\varepsilon^{-1}u(t) + f(t)$. Suppose that you want to compute the solution $u(1)$ to within δ. In terms of ε and δ,

what step size should you take if using a second-order Adams–Moulton method? What about a third-order Adams–Moulton method?

12.7. Let A and B be two arbitrary operators. Show that

$$\mathrm{e}^{k\,\mathrm{A}/2}\mathrm{e}^{k\,\mathrm{B}}\mathrm{e}^{k\,\mathrm{A}/2} - \mathrm{e}^{k(\mathrm{A}+\mathrm{B})} = O(k^3)$$

from which it follows that splitting error using Strang splitting is $O(k^2)$.

12.8. Sometimes it is easier to implement

$$\mathrm{N}\,U^{n+1/2} \approx \mathrm{N}\left(\tfrac{3}{2}U^n - \tfrac{1}{2}U^{n-1}\right)$$

instead of

$$\mathrm{N}\,U^{n+1/2} \approx \tfrac{3}{2}\,\mathrm{N}\,U^n - \tfrac{1}{2}\,\mathrm{N}\,U^{n-1}$$

in the second-order Adams–Bashforth method. Show that these two approximation are the same up to $O(k^2)$.

✍ 12.9. Develop a third-order L-stable IMEX scheme. Study the stability of the method by writing the characteristic polynomial and plotting the regions of absolute stability in the complex plane for the implicit part and the explicit part of the IMEX scheme.

12.10. MATLAB has several built-in "black box" routines for solving ODEs numerically: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`, and `ode15i`. For each function, give a short description of the method it uses and what the advantages/disadvantages of using that method.

✍ 12.11. Plot the regions of absolute stability for the Adams–Bashforth–Moulton predictor-corrector methods: AB1–AM2, AB2–AM3, AB3–AM4, and AB4–AM5 with $PE(CE)^m$ for $s = 0, 1, \ldots, 4$.

✍ 12.12. Find the numerical solution to the initial value problem

$$u'(t) = -100(u - \cos t) - \sin t \qquad \text{with} \qquad u(0) = 1$$

using the forward Euler, the trapezoidal and a fourth-order Runge–Kutta methods using appropriate step sizes. Plot the error between the numerical solution and the analytic solution at time $t = 2$ as a function of the time stepsize k using a log-log plot. Compute the slope of the plot to confirm that you are getting the correct rate of convergence.

12.13. Show that the Verlet method for the system $u' = p$ and $p' = f(u)$ is the same as the centered-difference scheme for $u'' = f(u)$:

$$U^n - 2U^{n-1} + U^{n-2} = kf(U^n).$$

12.14. A coupled pendulum is created by connecting two pendula together with a spring. The Hamiltonian is

$$H = \tfrac{1}{2}(p_1^2 + p_2)^2 - \cos q_1 - \cos q_2 - \varepsilon \cos(q_1 - q - 2)$$

where ε is a coupling parameter. Use a symplectic integrator to solve the system with suitable initial conditions and coupling parameter.

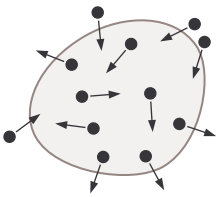
12.15. The Lorenz equation is a simple (and extensively studied) model motivated by atmospheric dynamics:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x), \qquad \frac{\mathrm{d}y}{\mathrm{d}t} = \rho x - y - xz, \qquad \frac{\mathrm{d}z}{\mathrm{d}t} = -\beta z + xy \tag{12.18}$$

with $\sigma = 10$, $\beta = 8/3$, $\rho = 28$. Plot the solution $(x(t), z(t))$

CHAPTER 13

Parabolic Equations



A group of molecules, bacteria, insects, or even people will often move in a random way. As a result these "particles" tend to spread out. On the large scale, this evolution can be modeled by the diffusion process. Let $u(t, \mathbf{x})$ be the concentration of particles in $\mathbb{R}^n$ and consider an *arbitrarily shaped (but fixed)* domain $\Omega \subset \mathbb{R}^n$. Then the mass of particles at some time t in Ω is $\int_\Omega u(t, \mathbf{x})\,\mathrm{d}V$. Since the particles are moving, the mass of particles in Ω changes as some particles enter the domain and other particles leave it. Flux is the number of particles passing a point per unit time. The change in the number of particles in Ω is equal to the integral of the flux of particles $J(t, \mathbf{x})$ through the boundary

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_\Omega u(t, \mathbf{x})\,\mathrm{d}V = \int_{\partial\Omega} J(t, \mathbf{x}) \cdot \mathbf{n}\,\mathrm{d}A$$

By invoking the divergence theorem, we have that

$$\int_\Omega \frac{\partial}{\partial t} u(t, \mathbf{x})\,\mathrm{d}V = \int_\Omega \nabla \cdot J(t, \mathbf{x})\,\mathrm{d}V$$

Since the domain Ω was arbitrarily chosen, it follows that

$$\frac{\partial}{\partial t} u(t, \mathbf{x}) = \nabla \cdot J(t, \mathbf{x})$$

Fickian diffusion says that the flux $J(t, \mathbf{x})$ is proportional the gradient of the concentration $u(t, \mathbf{x})$. This makes sense if we consider a random walk process. In one-dimension, consider adjacent cells at $x - \frac{1}{2}h$ and $x + \frac{1}{2}h$ separated by an interface at x. If a particle moves to the right or the left with constant velocity $m(x)$, then the flux across the interface can be approximated by

$$J(t, x) \approx \frac{m(x)(u(t, x + h/2) - u(t, x - h/2))}{h}.$$

In the limit as $h \to 0$, we have that

$$J = m(x)\frac{\partial}{\partial x}u.$$

So, in general,

$$\frac{\partial u}{\partial t} = \nabla \cdot (m(\mathbf{x})\nabla u).$$

Example. Consider a heat conducting bar. If $u(t, x)$ is temperature, then $u_t + J_x = 0$ where $J(x)$ is the heat flux. Fourier's law states that $J(x) = -m(x)u_x$ where $m(x) > 0$ is the heat conductivity. It follows that $u_t = (m(x)u_x)_x$. Suppose that the bar has a uniform heat conductivity $m(x) = a$ and the ends of the bar at $x = 0$ and $x = 1$ are in contact with heat sinks of constant temperature $u = 0$. Furthermore, suppose that the bar has an initial temperature distribution $u_0(x)$. Then the heat equation is given by the initial-boundary value problem

$$u_t = au_{xx}, \quad u(0, x) = u_0(x), \quad u(t, 0) = 0 \quad u(t, 1) = 0. \tag{13.1}$$

This initial value boundary value problem is well-posed, and it can be solved analytically by separation of variables or using a Fourier transform. In the next section, we will use it as a starting place to examine numerical methods for parabolic PDEs. ◀

Parabolic partial differential equations are first order in time and second order in space. They include the heat or diffusion equations, reaction-diffusion problems, and viscous fluid dynamics. Important examples of diffusion also occur frequently in biology. One such example is chemotaxis. Animals and bacteria often communicate by releasing chemicals. Not only do the bacteria spread out due to normal diffusion mechanisms, the bacteria may move also in a direction up a concentration gradient, following the strongest attractant, which itself is subject to diffusion.

Linear parabolic systems have several properties. One of the properties is the *maximum principle* which says that the maximum value of a

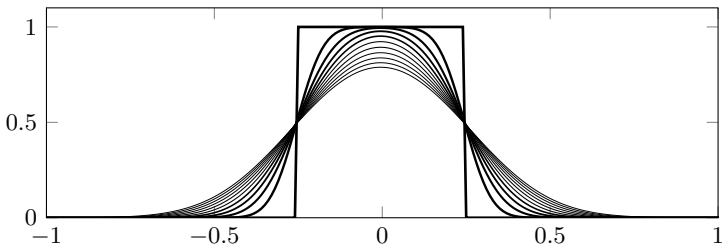


Figure 13.1: Solution to the heat equation $u_t = u_{xx}$. Several snapshots in time are superimposed with the thicker lines occurring earlier.

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. Another property is that the system is dissipative. That is, the L^2-norm of the solution decreases in time. A third property is that the solution for $t > 0$ is smooth and grows smoother in time. For a complete review see Evans [2010].

13.1 Method of lines

A typical means of numerically solving a time-dependent PDE is using the *method of lines*. We discretize space and keep time continuous to derive a semidiscrete method. 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 $0 = x_0 < x_1 < \cdots < x_N = 1$. For simplicity, define the a uniform mesh $x_j = jh$ where h is the grid spacing. Let $U_j(t) \approx u(t, x_j)$. To get a second-order approximation of the second derivative, we can use a centered-difference approximation

$$\mathrm{D}^2 U_j = \mathrm{D}\left(\frac{U_{j+1/2} - U_{j-1/2}}{h}\right) = \frac{U_{j+1} - 2U_j + U_{j+1}}{h^2}$$

Hence, $u_t = au_{xx}$ becomes

$$\frac{\partial}{\partial t} U_j = a\frac{U_{j+1} - 2U_j + U_{j+1}}{h^2} \tag{13.2}$$

with $0 \le j \le N$. We now have a system of $N + 1$ ordinary differential equations, and we can use numerical methods for ordinary differential equations to solve the system of equations. Note that the system is not closed. In addition to the interior mesh points ($0 \le j \le N$), we also have mesh points outside of the domain ($j = -1$ and $j = N + 1$). We will use

the boundary conditions to explicitly remove these "ghost points" and close the system.

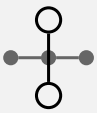
This approach of employing the numerical method of lines to solve a partial differential equation is often called *time-marching*. We can combine any consistent ODE solver with any consistent discretization of the Laplacian. In the previous chapter, we discussed the forward Euler, the backward Euler, leap frog and the trapezoidal methods to solve ODEs. Let's look at applying these methods to solve PDEs.

Forward (Explicit) Euler Method $O(k+h^2)$

$$\frac{U_j^{n+1}-U_j^n}{k} = a\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \tag{13.3}$$

Backward (Implicit) Euler Method $O(k+h^2)$

$$\frac{U_j^{n+1}-U_j^n}{k} = a\frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{h^2} \tag{13.4}$$



Richard Method (Unstable!) $O(k^2+h^2)$

$$\frac{U_j^{n+1}-U_j^{n-1}}{2k} = a\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \tag{13.5}$$

Crank–Nicolson (Trapezoidal) Method $O(k^2+h^2)$

$$\frac{U_j^{n+1}-U_j^n}{k} = \tfrac{1}{2}a\frac{U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}}{h^2} + \tfrac{1}{2}a\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \tag{13.6}$$

Boundary conditions

The method of lines converted the PDE into a system of ODEs with $N+3$ unknowns and $N+1$ equations. In addition to the interior mesh points at $x_0, \ldots, x_N$, we also have *ghost points* outside the domain. We use the boundary constraints to explicitly remove the ghost points and close the system by eliminate the two unknowns U_{-1} and U_{N+1}.

A *Dirichlet boundary condition* specifies the value of the solution on the boundary. For example, consider a heat conducting bar in contact with heat sinks of temperatures c_0 and c_1 at at the ends of the bar $x = 0$

and $x = 1$. We would use the boundary conditions $u(t,0) = c_0$ and $u(t,1) = c_1$. Consider the uniformly spaced meshpoints $\{x_0, x_1, \ldots, x_N\}$. We can eliminate the left ghost point U_{-1} by using a second-order approximation $\frac{1}{2}(U_{-1} + U_1) \approx U_0 = u(0) = c_0$. In this case substituting, $U_{-1} = 2c_0 - U_1$ in (13.2) gives us

$$\frac{\partial}{\partial t} U_0 = a\frac{2U_1}{h^2} + 2a\frac{c_0}{h^2}. \tag{13.7}$$

The right ghost point can similarly be eliminated.

Example. Let's implement a backward Euler scheme for (13.1) that uses Dirichlet boundary conditions. Let $\lambda = ak/h^2$. From (13.4) and (13.7) we have

$$\begin{aligned}
U_0^{n+1} - U_0^n &= -2\lambda U_0^{n+1} + 2\lambda c_0 \\
U_j^{n+1} - U_j^n &= \lambda\left(U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}\right) \\
U_N^{n+1} - U_N^n &= -2\lambda U_N^{n+1} + 2\lambda c_0.
\end{aligned}$$

We move the U^{n+1} terms to the left-hand side and the U^n terms to the right-hand side:

$$\begin{bmatrix} 1+2\lambda & & & & \\ -\lambda & 1+2\lambda & -\lambda & & \\ & \ddots & \ddots & \ddots & \\ & & -\lambda & 1+2\lambda & -\lambda \\ & & & & 1+2\lambda \end{bmatrix} \begin{bmatrix} U_0^{n+1} \\ U_1^{n+1} \\ \vdots \\ U_{N-1}^{n+1} \\ U_N^{n+1} \end{bmatrix} = \begin{bmatrix} U_0^n \\ U_1^n \\ \vdots \\ U_{N-1}^n \\ U_N^n \end{bmatrix} + \begin{bmatrix} 2\lambda c_0 \\ 0 \\ \vdots \\ 0 \\ 2\lambda c_1 \end{bmatrix}.$$

At each time step we need to invert a tridiagonal matrix by using a tridiagonal solver. A tridiagonal solver efficiently uses Gaussian elimination by only operating on the diagonal and two off-diagonals. By doing so, a tridiagonal solver requires only $O(3N)$ operations to solve a linear system whereas general Gaussian elimination requires $O(\frac{2}{3}N^3)$ operations. For $N = 400$, using a solver designed sparse matrices is about twenty-five times faster than a solver designed for full matrices. ◀

M In MATLAB, the command `mldivide` (\) will automatically implement a tridiagonal solver on a tridiagonal matrix if the matrix is formatted as a sparse matrix. The MATLAB command `help sparfun` returns a list of matrix functions for working with sparse matrices.

A *Neumann boundary condition* specifies the value of the first derivative of the solution (or the gradient) on the boundary. For example, to

model a bar that is insulated at both ends, we set the heat flux $q(x)$ to be zero on the boundary giving $u_x(0) = u_x(1) = 0$. Zero Neumann boundary conditions are also called reflecting boundary conditions.

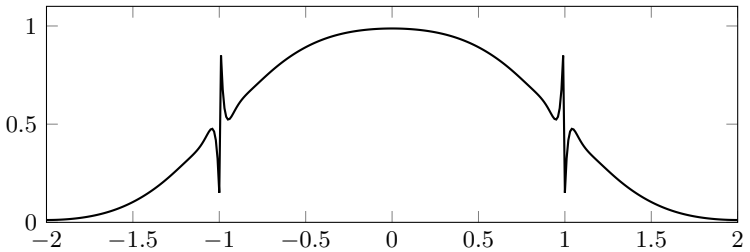
Example. Let's implement a Crank–Nicolson scheme with reflecting boundary conditions. The Crank–Nicolson scheme is second-order in space, so we should also approximate the boundary conditions with the same order. The second-order approximations for the derivatives $u_x(t, x_1)$ and $u_x(t, x_N)$ are $(U_1 - U_{-1})/h$ and $(U_{N+1} - U_{N-1})/h$. For reflecting boundary conditions $u_x(t, x_1) = 0$ and $u_x(t, x_N) = 0$, so we can eliminate the left ghost points by setting $U_{-1} = U_1$ and $U_{N+1} = U_{N-1}$. Let $\lambda = ak/h^2$. Then the Crank–Nicolson method (13.6) is

$$\begin{aligned}
(2+2\lambda)U_0^{n+1} - 2\lambda U_1^{n+1} &= (2-2\lambda)U_0^n + 2\lambda U_1^n \\
-\lambda U_{j-1}^{n+1} + (2+2\lambda)U_j^{n+1} - \lambda U_{j+1}^{n+1} &= \lambda U_{j-1}^n + (2-2\lambda)U_j^n + \lambda U_{j+1}^n \\
2\lambda U_{N-1}^{n+1} - (2+2\lambda)U_N^{n+1} &= 2\lambda U_{N-1}^n + (2-2\lambda)U_N^n
\end{aligned}$$

The following MATLAB 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.

```
function heat
dx = .01; dt = .01; L = 2;
x = (-L:dx:L)'; n = length(x);
lambda = dt/dx^2;
Q = spdiags(repmat([1 -2 1],[n 1]),-1:1,n,n)*lambda;
Q(1,2) = 2*lambda; Q(n,n-1)=2*lambda;
A = (2*speye(n) + Q);
B = (2*speye(n) - Q);
u = (abs(x)<1);
for i = 1:1000
  u = B\(A*u);
  plot(x,u); ylim([0 1]);drawnow;
end
```

Running this code, we observe transient spikes at $x = -1$ and $x = 1$, where the initial condition had discontinuities.



These are clearly numerical artifacts. What causes them and how can we get a better solution? We can start to answer these questions by remembering that the Crank–Nicolson method is A-stable but not L-stable and exhibits transient solutions for high eigenvalues. We'll reexamine numerical stability in the next section and in the next chapter when we discuss dispersive and dissipative schemes. ◀

A third type of boundary condition, the *Robin boundary condition* (or the mixed boundary condition), is a combination of Dirichlet and Neumann boundary conditions

$$\begin{aligned} u(0) + b_0 u_x(0) &= c_0, \\ u(0) + b_1 u_x(1) &= c_1 \end{aligned}$$

for some constants b_0, b_1, c_0 and c_1.

If the problem has *periodic boundary conditions* $u(0) = u(1)$, the system can no longer be made to be tridiagonal. Instead, we are left with a circulant matrix which can be solved more simply with a Fourier spectral method as we will see in Chapter 16.

A problem could also have *open boundary conditions*. Although we may only be interested in the dynamics in a finite region $x \in [-1, 1]$, diffusion happens over an unbounded domain $(-\infty, \infty)$. One solution is to solve the problem over the unbounded domain by using a nonuniform mesh. An equivalent solution is to map the original problem to a finite domain by making a transform like $\tanh \alpha x \mapsto \tilde{x}$.

13.2 Stability: von Neumann analysis

In Chapter 12, we found region of absolute stability for a numerical method for a linear ODE $u' = \gamma u$ by determining the values γk for which a perturbation of the numerical solution will decrease over time. We now want to determine under what conditions a numerical method for a linear PDE is stable. We can use a Fourier transform to decouple a PDE into a system of ODEs. Because eigenvalues are unaffected by change in basis, we can use the linear stability analysis we developed for ODEs in the previous chapter. Because numerical stability is a local behavior, assuming periodic boundary conditions will not greatly change the outcome of the analysis.

Take the domain $(0, 2\pi)$ and uniform discretization in space $x_j = jh$ with $h = \frac{2\pi}{2N+1}$. The discrete Fourier transform is defined as

$$\hat{u}(t, \xi) = \frac{1}{2N+1} \sum_{j=0}^{2N} u(t, x_j) \mathrm{e}^{-\mathrm{i}\xi jh}$$

and the discrete inverse Fourier transform is defined as

$$u_j(t, x_j) = \sum_{\xi=-N}^{N} \hat{u}(t, \xi) \mathrm{e}^{\mathrm{i}\xi j h}$$

where the wave number ξ is an integer (because the domain is bounded).

We can use the method of lines to spatially discretize the heat equation

$$u_t = a u_{xx} \qquad \text{with} \qquad 0 < x < 2\pi$$

to get

$$\frac{\partial}{\partial t} u(t, x_j) = a \frac{u(t, x_{j+1}) - 2u(t, x_j) + u(t, x_{j-1})}{h^2}. \tag{13.8}$$

Substituting the definition of the discrete inverse Fourier transform:

$$\sum_{\xi=-N}^{N} \frac{\partial}{\partial t} \hat{u}(t, \xi) \mathrm{e}^{\mathrm{i}\xi j h} = \sum_{\xi=-N}^{N} a \frac{\mathrm{e}^{i\xi(j+1)h} - 2\mathrm{e}^{i\xi j h} + \mathrm{e}^{\mathrm{i}\xi(j-1)h}}{h^2} \hat{u}(t, \xi).$$

Equivalently,

$$\sum_{\xi=-N}^{N} \frac{\partial}{\partial t} \hat{u}(t, \xi) \mathrm{e}^{\mathrm{i}\xi j h} = \sum_{\xi=-N}^{N} a \frac{\mathrm{e}^{\mathrm{i}\xi h} - 2 + \mathrm{e}^{-\mathrm{i}\xi h}}{h^2} \hat{u}(t, \xi) \mathrm{e}^{\mathrm{i}\xi j h}.$$

This equality is true for each integer j, so

$$\frac{\partial \hat{u}}{\partial t} = a \frac{\mathrm{e}^{\mathrm{i}\xi h} - 2 + \mathrm{e}^{-\mathrm{i}\xi h}}{h^2} \hat{u}.$$

for each $\xi = -N, \cdots, N$. By simplifying the expression

$$a \frac{\mathrm{e}^{\mathrm{i}\xi h} - 2 + \mathrm{e}^{-\mathrm{i}\xi h}}{h^2} = \frac{2a}{h^2}(\cos \xi h - 1) = -4 \frac{a}{h^2} \sin^2 \frac{\xi h}{2}$$

we have that

$$\frac{\partial \hat{u}}{\partial t} = -4 \frac{a}{h^2} \sin^2 \frac{\xi h}{2} \hat{u}. \tag{13.9}$$

Note that (13.9) is now a linear ordinary differential equation $\hat{u}_t = \mu_\xi \hat{u}$ where the eigenvalues $\mu_\xi = -4a/h^2 \sin^2(\xi h/2)$ are all non-positive real numbers.

Recall the regions of stablity for the forward Euler, backward Euler, leapfrog, and trapezoidal methods. See Figure 13.2 on the facing page. The method is stable as long as all $\mu_\xi k$ lie in the region of stability. The eigenvalue μ_ξ can be as big as $|\mu_\xi| = -4a/h^2$. Using this bound along with associated the region of stability we can determine the stability conditions for the various methods.

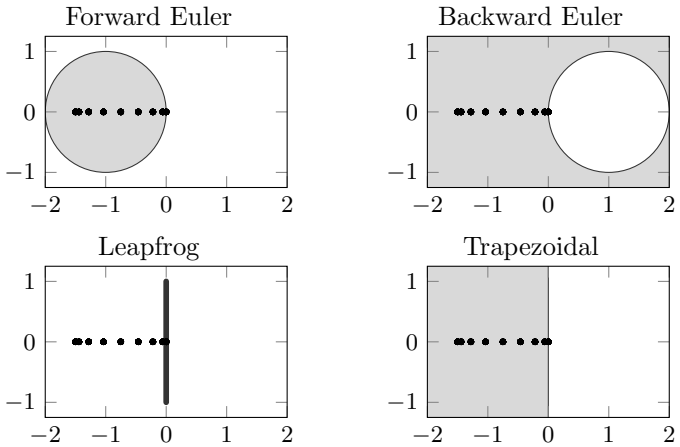


Figure 13.2: Regions of absolute stability (shaded) in the $\gamma\xi$-plane for the forward Euler, backward Euler, leapfrog, and trapezoidal methods and eigenvalues μ_ξ of the centered-difference approximation with $ak/h^2 = \frac{3}{2}$.

Example. Determine the conditions for stability if we implement (13.8) using the explicit Euler method. In Chapter 12, we found that the region of absolute stability for the forward Euler method is the unit circle centered at -1.The forward Euler method is stable if $\mu_\xi h$ is in the region of stability for all ξ. See Figure 13.2. Because $\mu_\xi = -4a/h^2 \sin^2(\xi h/2)$, $|\mu_\xi|$ can be as large as $4a/h^2$. Furthermore, the region of absolute stability requires that $|\mu_\xi k| \leq 2$, so it follows that $k \leq h^2/2a$ for absolute stability. Such a stability condition is called the *Courant–Friedrichs–Lewy condition* or *CFL condition*. ◀

Example. Determine the conditions for stability if we implement (13.8) using the implicit Euler method. In Chapter 12, we determined that for the backward Euler method, there was no constraint on k when the eigenvalues are on the negative real axis. Therefore, the backward Euler method is *unconditionally stable*. The trade-off to get unconditional stability is that we need to implement a tridiagonal solver at each time step. When h is small, this extra work is worth it. ◀

In the above analysis, we used that property that $\{e^{i\xi jh}\}$ forms a linearly independent basis, and therefore, we only need to compare the Fourier coefficients. We can abbreviate the stability analysis a bit by

making the formal substitution $U_j^n \mapsto A\mathrm{e}^{\mathrm{i}\xi jh}$. A method is stable if and only if $|A| \leq 1$. We call this *von Neumann stability analysis*.

Example. Determine the stability condition of the Richard method

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = a\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

The Richard method combines the second-order leapfrog scheme in time with second-order centered-difference approximation in space. We set $U_j^n \mapsto A^n\mathrm{e}^{\mathrm{i}\xi jh}$:

$$\frac{A^{n+1}\mathrm{e}^{\mathrm{i}\xi jh} - A^{n-1}\mathrm{e}^{\mathrm{i}\xi jh}}{2k} = aA^n\frac{\mathrm{e}^{\mathrm{i}\xi h} - 2 + \mathrm{e}^{-\mathrm{i}\xi h}}{k^2}\mathrm{e}^{\mathrm{i}\xi jh}.$$

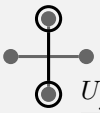
Then

$$\frac{A^2 - 1}{2k} = -a\frac{4}{h^2}\sin^2\frac{\xi h}{2}A$$

from which we have that

$$A^2 + 8a\frac{k}{h^2}\sin^2\frac{\xi h}{2}A - 1 = 0$$

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 Richard method is unconditionally unstable! This result should also be clear from the region of absolute stability of the Leap frog method $(-\mathrm{i}, \mathrm{i})$ and the fact that μ is a negative real number. See Figure 13.2 on the preceding page. A slight modification to the Richard method is the implicit Dufort–Frankel method. ◀



Dufort–Frankel Method $O(k^2 + h^2 + \frac{k^2}{h^2})$

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = a\frac{U_{j+1}^n - (U_j^{n+1} + U_j^{n-1}) + U_{j-1}^n}{h^2} \tag{13.10}$$

Example. Let's determine the stability condition of the Dufort–Frankel method by von Neumann analysis. Setting $U_j^n \mapsto A^n\mathrm{e}^{\mathrm{i}j\xi h}$ and dividing by $A^{n-1}\mathrm{e}^{\mathrm{i}j\xi h}$ in the Dufort–Frankel method (13.10) gives us

$$\begin{aligned}\frac{A^2 - 1}{2k} &= a\frac{A\mathrm{e}^{\mathrm{i}\xi h} - (A^2 + 1) + A\mathrm{e}^{-\mathrm{i}\xi h}}{h^2} \\ &= a\frac{2A\cos\xi h - (A^2 + 1)}{h^2}.\end{aligned}$$

Let $\lambda = ak/h^2$, then

$$(1+2\lambda)A^2 - (4\lambda\cos\xi h)A - (1-2\lambda) = 0.$$

The roots to this quadratic are

$$\begin{aligned} A_\pm &= \frac{1}{2(1+2\lambda)}\left[4\lambda\cos\xi h \pm \sqrt{16\lambda^2\cos^2\xi h + 4(1-4\lambda^2)}\right] \\ &= \frac{1}{2(1+2\lambda)}\left[4\lambda\cos\xi h \pm \sqrt{4-16\lambda^2\sin^2\xi h}\right] \\ &= \frac{1}{1+2\lambda}\left[2\lambda\cos\xi h \pm \sqrt{1-4\lambda^2\sin^2\xi h}\right]. \end{aligned}$$

We consider two cases.
Case 1: $(1-4\lambda^2\sin^2\xi h \geq 0)$

$$|A\pm| \leq \frac{2\lambda+1}{1+2\lambda} = 1$$

Case 2: $(1-4\lambda^2\sin^2\xi h \leq 0)$

$$\begin{aligned} |A\pm| &= \frac{2\lambda\cos\xi h \pm i\sqrt{4\lambda^2\sin^2\xi h - 1}}{1+2\lambda} \\ |A\pm|^2 &= \frac{(2\lambda\cos\xi h)^2 + 4\lambda^2\sin^2\xi h - 1}{(1+2\lambda)^2} = \frac{4\lambda^2-1}{4\lambda^2+4\lambda+1} \leq 1. \end{aligned}$$

So, the Dufort–Frankel scheme is unconditionally stable.

But we're not finished yet. Let's compute the truncation error of the Dufort–Frankel method by substituting for each term the Taylor expansion about (x_j, t_n). First note that we can rewrite

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = a\frac{U_{j+1}^n - (U_j^{n+1} + U_j^{n-1}) + U_{j-1}^n}{h^2}$$

as

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = a\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} - a\frac{U_j^{n+1} - 2U_j^n + U_j^{n-1}}{h^2}$$

Then Taylor expansion simply gives us

$$u_t + \frac{k^2}{6}u_{ttt} + \cdots = a\left(u_{xx} + \frac{h^2}{12}u_{xxxx} + \cdots\right) - a\left(\frac{k^2}{h^2}u_{tt} + \cdots\right).$$

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, and we are actually finding the solution to the equation

$$u_t = au_{xx} + au_{tt}.$$

This says that although the Dufort–Frankel scheme is absolutely stable, it is only consistent when $k \ll h$. Furthermore, we don't get second-order accuracy unless $k = O(h^2)$. ◀

13.3 Higher-dimensional methods

In two dimensions, the heat equation is

$$u_t = u_{xx} + u_{yy}.$$

We make the approximation

$$U_{ij}^n = u(x_i, y_j, t_n)$$

and define the discrete operators in x and y

$$\delta_x^2 U_{ij} = (U_{i+1,j} - 2U_{i,j} + U_{i-1,j})/h^2 \tag{13.11a}$$

$$\delta_y^2 U_{ij} = (U_{i,j+1} - 2U_{i,j} + U_{i,j-1})/h^2 \tag{13.11b}$$

where $\Delta x = \Delta y = h$. The Crank–Nicolson method is

$$\frac{U_{ij}^{n+1} - U_{ij}^n}{k} = \tfrac{1}{2}\left(\delta_x^2 U_{ij}^{n+1} + \delta_x^2 U_{ij}^n + \delta_y^2 U^{n+1} + \delta_y^2 U_{ij}^n\right). \tag{13.12}$$

The method is second order in time and space and unconditionally stable. For von Neumann analysis in two dimensions we substitute $U_{ij}^n = A^n \mathrm{e}^{\mathrm{i}(i\xi h + j\eta h)}$ and stability requires that $|A| \le 1$.

But we no longer have a tridiagonal system. In fact, if we have 100 grid points in both the x- and y-directions, then we would need to invert a $10^4 \times 10^4$ sparse matrix. In three-dimensions, we would need to invert a $10^6 \times 10^6$ matrix. Instead, we will use operator splitting. Let's examine two ways of splitting the right-hand operator of (13.12) while still keeping the method implicit: the fractional step method

$$\frac{U_{ij}^{n+1} - U_{ij}^n}{k} = \tfrac{1}{2}\left(\overline{\delta_x^2 U_{ij}^{n+1}} + \overline{\delta_x^2 U_{ij}^n} + \underline{\delta_y^2 U_{ij}^{n+1}} + \underline{\delta_y^2 U_{ij}^n}\right)$$

and the alternate direction implicit (ADI)

$$\frac{U_{ij}^{n+1} - U_{jk}^n}{k} = \tfrac{1}{2}\left(\overline{\delta_x^2 U_{ij}^{n+1}} + \underline{\delta_x^2 U_{ij}^n} + \underline{\delta_y^2 U_{ij}^{n+1}} + \overline{\delta_y^2 U_{ij}^n}\right)$$

where the $\overline{\text{overline}}$ and the $\underline{\text{underline}}$ notation is used to group terms for the operator splitting. For example, one iteration of ADI is

$$\begin{aligned}\frac{U_{ij}^{*}-U_{ij}^{n}}{k} &= \tfrac{1}{2}\left(\delta_x^2 U_{ij}^{*}+\delta_y^2 U_{ij}^{n}\right)\\ \frac{U_{ij}^{n+1}-U_{ij}^{n}}{k} &= \tfrac{1}{2}\left(\delta_x^2 U_{ij}^{*}+\delta_y^2 U_{ij}^{n+1}\right).\end{aligned}$$

Fractional step method

Consider the splitting method

$$u_t = \mathrm{D}\,u \quad \text{where} \quad \mathrm{D}=\mathrm{D}_1+\mathrm{D}_2+\cdots+\mathrm{D}_p \quad \text{and} \quad \mathrm{D}_i=\frac{\partial^2}{\partial^2 x_i}$$

We solve $u_t = \mathrm{D}_i\,u$ successively for $1 \le i \le p$. In this way, a multi-dimensional problem becomes a succession of one-dimensional problems. Discretize $\mathrm{D}_i \mapsto \delta_{x_i}^2$ and use the Crank–Nicolson method are each stage

$$\frac{U^{n+i/p}-U^{n+(i-1)/p}}{k} = \mathrm{D}_i\,\frac{U^{n+i/p}+U^{n+(i-1)/p}}{2} \qquad i=1,\ldots,p$$

This method is unconditionally stable. What about the accuracy? At each fractional step

$$(\mathrm{I}-\tfrac{1}{2}k\,\mathrm{D}_i)U^{n+i/p} = (\mathrm{I}+\tfrac{1}{2}k\,\mathrm{D}_i)U^{n+(i-1)/p}$$

where I is the identity operator. If $\|\tfrac{1}{2}k\,\mathrm{D}\| < 1$,

$$(\mathrm{I}-\tfrac{1}{2}k\,\mathrm{D})^{-1} = \mathrm{I}+\tfrac{1}{2}k\,\mathrm{D}+\tfrac{1}{4}k^2\,\mathrm{D}^2+O(k^3)$$

and so

$$\begin{aligned}U^{n+i/p} &= \left(\mathrm{I}+\tfrac{1}{2}k\,\mathrm{D}_i+\tfrac{1}{4}k^2\,\mathrm{D}_i^2+O(k^3)\right)\left(\mathrm{I}+\tfrac{1}{2}k\,\mathrm{D}_i\right)U^{n+(i-1)/p}\\ &= \left(\mathrm{I}+k\,\mathrm{D}_i+\tfrac{1}{2}k^2\,\mathrm{D}_i^2+O(k^3)\right)U^{n+(i-1)/p}.\end{aligned}$$

Continuing,

$$U^{n+1} = \prod_{i=1}^{p}\left(\mathrm{I}+k\,\mathrm{D}_i+\tfrac{1}{2}k^2\,\mathrm{D}_i^2+O(k^3)\right)U^n$$

and hence

$$U^{n+1}-U^n = \underbrace{\tfrac{1}{2}k\sum_i \mathrm{D}_i(U^{n+1}+U^n)}_{\text{Crank–Nicolson}} + \underbrace{\tfrac{1}{4}k^2\sum_{i,j}\mathrm{D}_i\,\mathrm{D}_j(U^{n+1}+U^n)}_{O(k^2)} + O(k^3).$$

So, the fractional step method is second order.

Alternating direction implicit method (ADI)

In the two-dimensional ADI method, with $\lambda = k/h^2$, we take

$$U^{n+1/2} - U^n = \frac{1}{2}\lambda\left(\delta_x^2 U^{n+1/2} + \delta_y^2 U^n\right), \tag{13.13a}$$

$$U^{n+1} - U^{n+1/2} = \frac{1}{2}\lambda\left(\delta_x^2 U^{n+1/2} + \delta_y^2 U^{n+1}\right). \tag{13.13b}$$

We need two tridiagonal solvers for this two-stage method.

To examine stability take the Fourier transforms ($x \mapsto \xi$ and $y \mapsto \eta$)

$$\hat{U}^{n+1/2} = \hat{U}^n + \tfrac{1}{2}\lambda\left(-4\sin^2\frac{\xi h}{2}\right)\hat{U}^{n+1/2} + \tfrac{1}{2}\lambda\left(-4\sin^2\frac{\eta h}{2}\right)\hat{U}^n$$

$$\hat{U}^{n+1} = \hat{U}^n + \tfrac{1}{2}\lambda\left(-4\sin^2\frac{\xi h}{2}\right)\hat{U}^{n+1/2} + \tfrac{1}{2}\lambda\left(-4\sin^2\frac{\eta h}{2}\right)\hat{U}^{n+1}$$

from which

$$\hat{U}^{n+1} = \frac{1 - 2\lambda\sin^2\dfrac{\eta h}{2}}{1 + 2\lambda\sin^2\dfrac{\xi h}{2}}\hat{U}^{n+1/2} \quad \text{and} \quad \hat{U}^{n+1/2} = \frac{1 - 2\lambda\sin^2\dfrac{\eta h}{2}}{1 + 2\lambda\sin^2\dfrac{\xi h}{2}}\hat{U}^n.$$

So,

$$\hat{U}^{n+1} = \underbrace{\frac{\left(1 - 2\lambda\sin^2\dfrac{\xi h}{2}\right)\left(1 - 2\lambda\sin^2\dfrac{\eta h}{2}\right)}{\left(1 + 2\lambda\sin^2\dfrac{\eta h}{2}\right)\left(1 + 2\lambda\sin^2\dfrac{\xi h}{2}\right)}}_{\mu}\hat{U}^n.$$

For absolute stability $|\mu| \le 1$ for all λ. Because the denominator is always larger than the numerator, $\mu \le 1$ so the method is unconditionally stable.

Now let's determine the order of the ADI method. Differencing and summing (13.13)

$$(a) - (b): \qquad 2U^{n+1/2} = U^{n+1} + \tfrac{1}{2}\lambda\delta_y^2(U^n - U^{n+1}) \tag{13.14a}$$

$$(a) + (b): \quad U^{n+1} - U^n = \lambda\delta_x^2 U^{n+1/2} + \tfrac{1}{2}\lambda\delta_y^2(U^n + U^{n+1}) \tag{13.14b}$$

Substituting (13.14a) into (13.14b)

$$\begin{aligned} U^{n+1} - U^n &= \frac{\lambda\delta_x^2}{2}\left(U^n + U^{n+1} + \frac{\lambda\delta_y^2}{2}(U^n - U^{n+1})\right) + \frac{\lambda\delta_y^2}{2}(U^n + U^{n+1}) \\ &= \underbrace{\lambda\left(\delta_x^2\frac{U^n + U^{n+1}}{2} + \delta_y^2\frac{U^n + U^{n+1}}{2}\right)}_{\text{Crank–Nicolson}} + \underbrace{\frac{\lambda^2}{4}\delta_x^2\delta_y^2\left(U^n - U^{n+1}\right)}_{O(\lambda^2 h^4)}. \end{aligned}$$

Since $\lambda = k/h^2$, $O(\lambda^2 h^4) = O(h^2)$. So, the method is the same order as the Crank–Nicolson scheme $O(h^2 + k^2)$. The fractional step method is easier to implement than ADI method, but in general the ADI method is more accurate.

A three-dimensional problem

$$u_t = u_{xx} + u_{yy} + u_{zz}$$

is treated similarly. Take $\Delta x = \Delta y = \Delta z = h$ and $\lambda = k/h^2$. Then the ADI method is

$$\begin{aligned}
U^* - U^n &= \tfrac{1}{6}\lambda\left[\delta_x^2(U^* + U^n) + \delta_y^2(2U^n) + \delta_z^2(2U^n)\right] \\
U^{**} - U^n &= \tfrac{1}{6}\lambda\left[\delta_x^2(U^* + U^n) + \delta_y^2(U^{**} + U^n) + \delta_z^2(2U^n)\right] \\
U^{n+1} - U^{**} &= \tfrac{1}{6}\lambda\left[\delta_x^2(U^* + U^n) + \delta_y^2(U^{**} + U^n) + \delta_z^2(U^{n+1} + U^n)\right]
\end{aligned}$$

13.4 Nonlinear diffusion equation

Consider the heat equation on a rod for which the heat conductivity $m(u) > 0$ changes as a function of the temperature. In this case,

$$\frac{\partial}{\partial t}u = \frac{\partial}{\partial x}\left(m(u)\frac{\partial}{\partial x}u\right) \tag{13.15a}$$

$$u(0,x) = u_0(x) \tag{13.15b}$$

$$u(t,0) = u(t,1) = 0. \tag{13.15c}$$

In this case, we can solve the problem by using the method of lines

$$\begin{aligned}
\frac{\partial}{\partial t}u &= \frac{\partial}{\partial x}\left(m(u)\frac{\partial}{\partial x}u\right) \\
&\approx \frac{m(U_{j+1/2})\frac{\partial}{\partial x}U_{j+1/2} - m(U_{j-1/2})\frac{\partial}{\partial x}U_{j-1/2}}{h} \\
&\approx m(U_{j+1/2})\frac{U_{j+1} - U_j}{h^2} - m(U_{j-1/2})\frac{U_j - U_{j-1}}{h^2} \\
&\approx m\left(\frac{U_j + U_{j+1}}{2}\right)\frac{U_{j+1} - U_j}{h^2} - m\left(\frac{U_{j-1} + U_j}{2}\right)\frac{U_j - U_{j-1}}{h^2}
\end{aligned} \tag{13.16}$$

Because the diffusion equation is stiff, the ODE solver should be a stiff (implicit) solver. Otherwise, we will be forced to take several small timesteps to ensure stability. See Table 13.1. Because the problem is nonlinear, we will need to solve a system of nonlinear equations. Typically, this is done using Newton's method. MATLAB has several stiff ODE

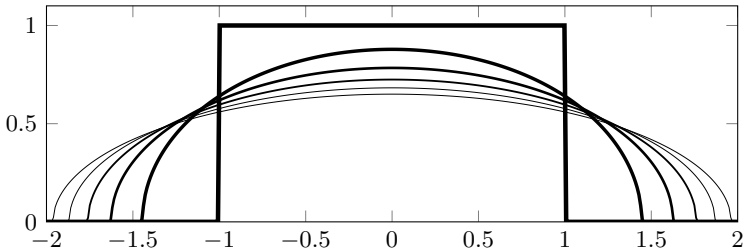


Figure 13.3: Solution to $u_t = (m(u)u_x)_x$ with $m(u) = u^2$. Several snapshots in time are superimposed with the thicker lines occurring earlier.

solver	**time steps**	**run time (sec)**
`ode15s`	754	1.5
`ode23t`	1573	3.4
`ode23s`	1130	5.7
`ode15s`*	754	15.1
`ode23t`*	1573	26.9
`ode23`	11817	29.1
`ode113`	18784	69.6
`ode23s`*	1130	219.2
`ode45`	35777	220.8

Table 13.1: Run times of the MATLAB program `porousmedium` with 400 meshpoints in space. Stiff solvers are significantly faster (up to 150 times faster) than nonstiff solvers. Asterisks indicate run times of solvers when the sparsity pattern was not provided.

solvers. To implement Newton's method in the implicit ODE solvers, MATLAB numerically computes and inverts the Jacobian matrix. Luckily, our system of nonlinear equations is sparse, so the Jacobian matrix is also sparse—a tridiagonal matrix. We must explicitly tell MATLAB that the Jacobian is sparse by passing it the sparsity pattern for the Jacobian using the `odeset` command. In this case, the sparsity pattern is a tridiagonal matrix of ones.

Example. The MATLAB code below implements (13.16) with $m(u) = u^2$ using `ode15s`, which uses a variation of the adaptive-step BDF methods. The nonlinear diffusion equation $u_t = (u^p u_x)_x$ for some p is known as the porous medium equation and is used to model the dispersion of grasshoppers and the flow of groundwater. The solution is plotted in Figure 13.3.

```
function [t,x,U] = porousmedium
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];
u = double(abs(x)<1);
options = odeset('JPattern',spdiags(ones([n 3]),-1:1,n,n));
[t,U] = ode15s(Du,[0 2],u,options);
```

The solution `U` can be interpolated in time and animated using the following code:

```
U = interp1(t,U,linspace(0,2,200)');
for i=1:size(U,1), plot(x,U(i,:)); ylim([0 1]); drawnow; end
```

Note that $u^2 u_x = (\frac{1}{3}u^3)_x$. In this case, it may be simpler to implement $u_t = (\frac{1}{3}u^3)_{xx}$ instead, and we could simply replace the appropriate lines of the function `porousmedium` with the following:

```
m = @(u) u.^3/3;
Du = @(t,u) [0;diff(m(u),2)/dx^2;0];
```

◀

Stability

Linear von Neumann stability analysis cannot be applied to problems with *variable coefficients* and *nonlinear problems*. In these cases, the *energy method* is an important tool of checking stability. We define the "energy" of a system as the L^2-norm of the variable u. The system is stable if the energy is non-increasing. Multiplying equation (13.15a) by u and integrating over the domain, we have that

$$\int_0^1 u \frac{\partial}{\partial t} u \, \mathrm{d}x = \int_0^1 u \frac{\partial}{\partial x} \left(m(u) \frac{\partial}{\partial x} u \right) \mathrm{d}x.$$

By using integration by parts on the right-hand side of the equation, we have

$$\frac{1}{2} \frac{\partial}{\partial t} \int_0^1 u^2 \, \mathrm{d}x = - \int_0^1 m(u) \left(\frac{\partial}{\partial x} u \right)^2 \mathrm{d}x$$

This expression says that $\frac{\partial}{\partial t} \|u(t,\cdot)\|_2 \leq 0$.

We can apply the same idea to the numerical scheme (13.16) by using

the discrete l^2-norm.

$$\sum_j U_j \frac{\partial}{\partial t} U_j = \sum_j \frac{1}{h^2} \left[m_{j+1/2}(U_{j+1} - U_j)U_j - m_{j-1/2}(U_j - U_{j-1})U_j \right]$$

$$\begin{aligned} \frac{1}{2}\frac{\partial}{\partial t} \sum_j U_j^2 &= \frac{1}{h^2} \sum_j m_{j+1/2}(U_{j+1} - U_j)U_j - m_{j-1/2}(U_j - U_{j-1})U_j \\ &= -\frac{1}{h^2} \sum_j m_{j-1/2}(U_j - U_{j-1})^2 \le 0 \end{aligned}$$

So, the l^2-norm is nonincresing: $\|U^n\|_2 \le \|U^0\|_2$.

Example. Discrete norms are equivalent. So, we can use other norms. For example, let's use the l^∞-norm to confirm the stability condition of the forward Euler discretization of the linear heat equation (13.3)

$$\frac{U_j^{n+1} - U_j^n}{k} = a \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

Letting $\lambda = ak/h^2$, then

$$U_j^{n+1} = \lambda U_{j-1}^n + (1 - 2\lambda)U_j^n + \lambda U_{j+1}^n.$$

If we take $0 \le \lambda \le \frac{1}{2}$, then

$$\begin{aligned} |U_j^{n+1}| &= \lambda |U_{j-1}^n| + (1 - 2\lambda)|U_j^n| + \lambda |U_{j+1}^n| \\ &\le (\lambda + 1 - 2\lambda + \lambda)\|U^n\|_\infty = \|U^n\|_\infty \end{aligned}$$

for all j. So,

$$\|U^{n+1}\|_\infty \le \|U^n\|_\infty \le \cdots \le \|U^0\|_\infty$$

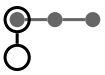
for all n when the CFL condition $k \le h^2/2a$ is met. ◀

Exercises

13.1. Show that the heat equation is conservative (no heat is lost or gained) if the boundaries are insulating (flux through the boundary is zero).

✍ 13.2. By Taylor series expansion, determine the truncation error of the Crank–Nicolson scheme for $u_t = u_{xx}$.

✍ 13.3. Suppose that we want to solve the heat equation $u_t = u_{xx}$ numerically. Instead of using $\{U_{j-1}, U_j, U_{j+1}\}$ to approximate the u_{xx} at x_j, one could instead use $\{U_j, U_{j+1}, U_{j+2}\}$ to approximate the u_{xx} at x_j. For example, the stencil for the backward Euler method is



This alternative approach is bad for a number of reasons. First, it provides only a first-order approximation whereas the centered-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. 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 $N = 20$ 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 $t = 1$ (with the analytic solution) to determine the order of convergence. To do this, you should use several values for the grid spacing h while keeping the time step $k \ll h$ constant. Then use several values for the time step k keeping the mesh $h \ll k$.

✍ 13.5. The Schrödinger equation

$$\mathrm{i}\varepsilon \frac{\partial \psi}{\partial t} = -\tfrac{1}{2}\varepsilon^2 \frac{\partial^2 \psi}{\partial x^2} + V(x)\psi$$

models the quantum mechanical behavior of a particle in a potential field $V(x)$. The parameter ε is the scaled Plank constant, which gives the relative length and time scales, and $\mathrm{i} = \sqrt{-1}$. The probability of finding a particle at a position x is given by $\rho(t, x) = |\psi(t, x)|^2$. (Note that the Schrödinger equation is not a parabolic PDE. The L^2-norm of the solution is constant in time and has no maximum principal. The equation has attributes more closely related to hyperbolic PDEs.)

1. For constant potential $V(x) \equiv V$, find the stability conditions (CFL conditions) using a second-order centered-space discretization along with forward-Euler, leapfrog, Crank–Nicolson (trapezoidal), and Runge–Kutta (RK4) time stepping.

2. Use Crank–Nicolson (trapezoidal) time stepping to solve the initial value problem for the harmonic oscillator with $V(x) = \frac{1}{2}x^2$, $\psi(0,x) = \mathrm{e}^{-(x-1)^2/2\varepsilon}$, and $\varepsilon = 0.3$ over a domain $[-3,3]$. The analytic solution is given by

$$\rho(t,x) = \mathrm{e}^{-(x-\cos t)^2/\varepsilon}.$$

Use the slope of the log-log plot of the error with the analytic solution at $t = 2\pi$ to determine the order of convergence. To do this, you should use several values for the grid spacing h while keeping the time step $k \ll h$ constant. Then use several values for the time step k keeping the mesh $dx \ll k$.

3. What happens when the mesh spacing h or the time step k are about equal to or larger than ε?

✍ 13.6. In polar coordinates, the heat equation $u_t = \Delta u$ 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 problem, the heat 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 Crank–Nicolson method for this problem and implement it for the initial conditions $u(0,r) = \tanh(4-8r)$. Use insulating boundary conditions at $r = 1$: $\left.\frac{\partial u}{\partial r}\right|_{r=1} = 0$. Use symmetry to determine an appropriate boundary condition at $r = 0$. Implement the method and plot the solution at several steps in time.

✍ 13.7. 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 pool of water with $u = 1$ representing the ice and $u = -1$ representing the water.

Determine the numerical solution to the two-dimensional Allen–Cahn equation using a numerical method that is second order in time and space with stability condition independent of ε. Consider the domain

$[-8, 8] \times [-8, 8]$ with reflecting boundary conditions. Take $\varepsilon = 1/50$ and take the initial conditions

$$u(0, x, y) = \begin{cases} 1, & \text{if } x^4 < 32x^2 - 16y^2 \\ -1, & \text{otherwise} \end{cases}$$

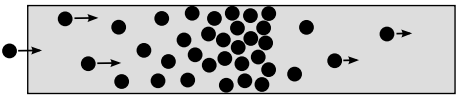
Also, try random initial conditions: `u = randn(N)`. Observe the behavior of the solution over time. You can do this by plotting the solution $u(x, y, t)$ as a density map at several points in time

```
colormap(1-gray(200));
image((u+1)*100);
```

Hint: Use Strang splitting—the resulting ODE can be solved analytically. You can easily implement a two-dimensional fractional step method in MATLAB using `DX*U*DY'` where `DX` and `DY` are the operators in x and y and `U` is the dependent variable.

CHAPTER 14

Hyperbolic Equations



Nonlinear hyperbolic equations are synonymous with shock wave formation from the hydraulic lift of a tsunami as it approaches the shore to the sonic boom of a jet aircraft to the pressure wave of an atomic bomb. Hyperbolic equations also model magnetohydrodynamics of plasmas, crowd dynamics, combustion, visco-elasticity of earthquakes, and traffic flow.

A principle difference between hyperbolic and parabolic equations is that while solutions to linear parabolic equations are always smooth (and become increasingly smoother over time). Solutions to hyperbolic equations often are not. In fact, the solutions of nonlinear hyperbolic equations with smooth initial conditions may become discontinuous in finite time resulting in shock waves. Such a discontinuity introduces difficulties when we try to approximate it using a high order polynomial interpolant.

Another difference is that the stability condition for explicit schemes for hyperbolic PDEs are typically $k = O(h)$ rather than the $k = O(h^2)$ that we had for parabolic PDEs. While there was a significant advantage to using a stiff (implicit) solver for parabolic equations, there is no significant advantage for hyperbolic PDEs. Therefore, we only consider explicit schemes.

The subject of numerical methods for hyperbolic equations has had considerable development in the last fifty years. Much of the interest arising out of a desire to efficiently simulate bomb blasts, supersonic flow, and semiconductors. This chapter is an introduction—see LeVeque [2002] for a complete coverage.

14.1 Linear hyperbolic equations

Let $u(x,t)$ be some physical quantity such as density, pressure, or velocity. Recall from the previous chapter that 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 Chapter 13, the flux was given by Fick's Law $f(x) = m(x)\frac{\partial}{\partial x}u$. For now, let's simply take $f(u) = cu$. We'll look at the more general nonlinear case later. Information propagating with velocity c is described by the linear advection equation

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{14.1a}$$

$$u(x,0) = u_0(x) \tag{14.1b}$$

where $u_0(x) \in \mathbb{R}$. This problem can be solved by the *method of characteristics*. Compare the total derivative of $u(t,x)$ and the advection equation:

$$\frac{\mathrm{d}}{\mathrm{d}t}u(t,x(t)) = \frac{\partial u}{\partial t} + \frac{\mathrm{d}x}{\mathrm{d}t}\frac{\partial u}{\partial x} \quad \text{and} \quad \frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0.$$

By taking $\mathrm{d}x/\mathrm{d}t = c$, the total derivative

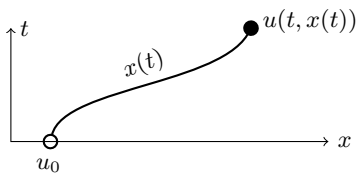
$$\frac{\mathrm{d}}{\mathrm{d}t}u(t,x(t)) = 0,$$

which says that the solution $u(t,x(t))$ is constant along the *characteristic curves* $x(t)$ for which $\mathrm{d}x/\mathrm{d}t = c$. The characteristics curves are exactly those for which $x(t) = ct + x_0$ for some x_0. The solution is then given by

$$u(x(t),t) = u(x(0),0) = u(x_0,0) = u_0(x_0) = u_0(x - ct).$$

The value c is called the *characteristic speed*. We call u the *Reimann invariant* because it is constant along the characteristic curves. It is not necessary that c be a constant—the method of characteristics also works for general $f(u)$. We will return to the nonlinear case later in the chapter.

The solution $u(t,x)$ to the advection equation is constant $u_0 = u(0,x(0))$ along a characteristic $x(t)$.



14.2 Methods for linear hyperbolic equations

We first will develop numerical schemes to solve for the simple linear problem $u_t + cu_x = 0$. The schemes developed for this linear problem will serve as prototypic methods that we'll extend to solve nonlinear hyperbolic problems later on.

Upwind

The simplest scheme for $u_t + cu_x = 0$ would be to take the forward Euler approximation in time and a first-order approximation in space. Such a scheme is called an *upwind method*.

Upwind Method ($c > 0$) $O(k+h)$

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_j^n - U_{j-1}^n}{h} = 0 \tag{14.2}$$

Upwind Method ($c < 0$) $O(k+h)$

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_{j+1}^n - U_j^n}{h} = 0 \tag{14.3}$$

Let's look at the stability conditions of the upwind method. Consider the semidiscrete equation

$$\frac{\partial}{\partial t}u(t, x_j) = c\frac{u(t, x_j) - u(t, x_{j-1})}{h}.$$

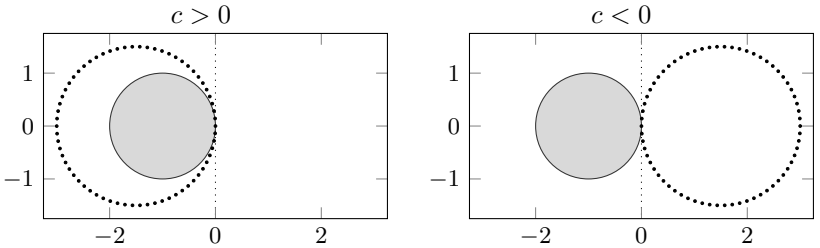
In the Fourier domain (replacing $u(t, x_j)$ with $\hat{u}(t, \xi)\mathrm{e}^{\mathrm{i}\xi jh}$), we have the decoupled system

$$\frac{\partial}{\partial t}\hat{u}(t, \xi) = c\frac{\mathrm{e}^{\mathrm{i}\xi h} - 1}{h}\hat{u}(t, \xi)$$

First, assume that $c \geq 0$. The locus of $\mathrm{e}^{\mathrm{i}\xi h} - 1$ is a circle of unit radius centered at -1. So $\frac{c}{h}(\mathrm{e}^{\mathrm{i}\xi h} - 1)$ is a circle centered at $-c/h$ with radius c/h. Recall that the region of stability for the forward Euler method is the unit circle centered at -1. Therefore, the upwind method (14.2) is stable if $k < h/c$. Recall from the previous chapter that this is called the *CFL condition* for stability and $|c|k/h$ is called the *CFL number* for the upwind scheme.

We can better see the stability conditions for upwind method by overlaying the eigenvalues of $\frac{c}{h}(U_j^n - U_{j-1}^n)$ on the region of absolute stability

for forward Euler $\frac{1}{k}(U_j^{n+1} - U_j^n)$. For $c > 0$, the upwind method is made stable by scaling k so that $k < h/c$. When $c < 0$, all of the eigenvalues of $\frac{c}{h}(U_j^n - U_{j-1}^n)$are in the right half-plane. So, the upwind method (14.2) is unconditionally unstable.



We can modify (14.2) when $c < 0$ with $u_x(x_j, t) \approx (U_{j+1}^n - U_j^n)/h$ giving us method (14.3). Similarly, the upwind scheme (14.3) is unconditionally unstable for $c > 0$. This should make intuitive sense—to predict the weather we need to look in the direction from which it is coming (upwind) not the direction in which it is going (downwind). To implement the schemes, we must determine the sign of c. We can combine the two schemes (14.2) and (14.3) to get a scheme that does this automatically

$$\frac{U_j^{n+1} - U_j^n}{k} + \frac{c + |c|}{2}\frac{U_j^n - U_{j-1}^n}{h} + \frac{c - |c|}{2}\frac{U_{j+1}^n - U_j^n}{h} = 0.$$

How can we interpret the CFL condition physically? Information propagates at a finite speed along any characteristic of the advection equation (14.1) with $c > 0$. So, the solution at any specific point in space can only be influenced by the initial conditions in some bounded region of space. This region is called the *domain of dependence*. The analytic solution to the advection equation (14.1) is $u(t_n, x_j) = u_0(x_j - ct_n)$. Take uniform stepsize k in time and a uniform mesh h in space. The domain of dependence for the analytic solution is $[x_j, x_j + cnk]$. The domain of dependence for the numerical solution is $[x_j, x_j + nh]$. By keeping $ck/h < 1$, we ensure that the domain of dependence of the exact solution is contained in the domain of dependence of the numerical solution.

Centered-Difference Method (Unstable!) $O(k + h^2)$

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_{j+1}^n + U_{j-1}^n}{2h} = 0 \tag{14.4}$$

The upwind scheme gives a method that is $O(h + k)$. It seems reasonable that we could improve the order by using a centered-difference in space approximation (14.4). Such a scheme is $O(k + h^2)$. What about

stability? By formally replacing $u(t, x_j)$ with $\hat{u}(t,\xi)\mathrm{e}^{\mathrm{i}\xi jh}$, we have that the decoupled system

$$\frac{\partial}{\partial t}\hat{u}(t,\xi) = c\frac{\mathrm{e}^{\mathrm{i}\xi h} - \mathrm{e}^{\mathrm{i}\xi h}}{2h}\hat{u}(t,\xi) = -\frac{c}{h}\mathrm{i}\sin\xi h\hat{u}(t,\xi)$$

in the Fourier domain. The eigenvalues are purely imaginary, but the region of stability for the forward Euler scheme is a unit circle in the left half plane, so the the method is unconditionally unstable.

Lax–Friedrichs

Let's modify the centered-difference scheme to make it more stable. One way to do this is to make it centered-difference in space and centered-difference in time (leapfrog). See Problem 14.3. Another way is to approximate the U_j^n term in the time derivative as $\frac{1}{2}(U_{j+1}^n + U_{j-1}^n)$. We call this method the Lax–Friedrichs Method (14.5).

Lax–Friedrichs Method $\qquad O(k + h^2/k)$

$$\frac{U_j^{n+1} - \frac{1}{2}(U_{j+1}^n + U_{j-1}^n)}{k} + c\frac{U_{j+1}^n - U_{j-1}^n}{2h} = 0 \tag{14.5}$$

The method is order $O(k + h^2/k)$ and conditionally stable. Von Neumann analysis says that

$$A - \frac{1}{2}\left(\mathrm{e}^{\mathrm{i}\xi h} + \mathrm{e}^{-\mathrm{i}\xi h}\right) = -c\frac{k}{h}\mathrm{i}\sin\xi h.$$

We solve for A

$$A = \cos\xi h - \mathrm{i}c\frac{k}{h}\sin\xi h$$

from which we see that

$$|A|^2 = \cos^2\xi h + \left(c\frac{k}{h}\right)^2\sin^2\xi h$$

The Lax–Friedrichs scheme is stable if $|c|k/h \le 1$. Therefore, the CFL number for the Lax–Friedrichs scheme is $|c|k/h$.

Lax–Wendroff

Let's look at another way to improve stability of the unstable centered-difference-in-space, forward-Euler-in-time scheme (14.4). From the Taylor series expansion about $u \equiv u(t_n, x_j)$:

$$\begin{aligned}
U_j^n &= u \\
U_j^{n+1} &= u + hu_t + \tfrac{1}{2}h^2 u_{tt} + O(h^3) \\
U_{j+1}^n &= u + hu_x + \tfrac{1}{2}h^2 u_{xx} + \tfrac{1}{6}h^3 u_{xxx} + O(h^4) \\
U_{j-1}^n &= u - hu_x + \tfrac{1}{2}h^2 u_{xx} - \tfrac{1}{6}h^3 u_{xxx} + O(h^4)
\end{aligned}$$

The scheme

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_{j+1}^n - U_{j-1}^n}{2h} = 0$$

is consistent with

$$u_t + cu_x = -\tfrac{1}{2}ku_{tt} + O(k^2 + h^2) \tag{14.6}$$

Because $u_t + cu_x = 0$, we have that $u_t = -cu_x$ from which $u_{tt} = -cu_{tx} = c^2 u_{xx}$. Therefore, (14.6) becomes

$$u_t + cu_x = -\tfrac{1}{2}c^2 ku_{xx} + O(k^2 + h^3)$$

Note that $u_t = -\frac{1}{2}c^2 ku_{xx}$ is a backward heat equation and therefore it is unstable. We can fix the scheme by counteracting $-\frac{1}{2}c^2 ku_{xx}$ with $+\frac{1}{2}c^2 ku_{xx}$. Not only will this improve stability, but it also improves the accuracy to $O(k^2 + h^2)$ That is, instead we can use

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_{j+1}^n - U_{j-1}^n}{2h} = \frac{c^2}{2}k\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

The reason that the original centered-difference in space, forward-Euler scheme failed was because the eigenvalues were along the imaginary axis and fell entirely outside of the region of stability of the forward-Euler scheme no matter how small k was. By adding a viscosity term u_{xx} to the problem, we push the eigenvalues into the left-half plane and by taking k sufficiently small, we can ensure stability.

Lax–Wendroff Method $O(k^2 + h^2)$

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_{j+1}^n - U_{j-1}^n}{2dx} = \frac{c^2}{2}k\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2} \tag{14.7}$$

Note that by rearranging the terms of the Lax–Friedrichs scheme we get a form which looks very similar to the Lax–Wendroff scheme:

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_{j+1}^n - U_{j-1}^n}{2h} = \frac{h^2}{2k}\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}$$

Like the Lax–Wendroff scheme, the Lax–Friedrichs scheme achieves stability by adding a viscosity (diffusion) term onto the (unstable) centered-difference scheme. Whereas the Lax–Wendroff scheme adds just enough viscosity ($c^2k/2$) to counteract the unstable second-order term, the Lax–Friedrichs scheme goes overboard adds a bit more viscosity ($h^2/2k$). We can generalize the Lax–Wendroff and Lax–Friedrichs methods by using a diffusion coefficient Q:

$$\frac{\partial}{\partial t}U_j = -c\frac{U_{j+1} - U_{j-1}}{2h} + Q\frac{U_{j+1} - 2U_j + U_{j-1}}{h^2}. \tag{14.8}$$

From earlier in this chapter and the previous chapter, we know the that the Fourier transform of this equation is

$$\frac{\partial}{\partial t}\hat{u}(t,\xi) = -4\frac{Q}{h^2}\sin\frac{\xi h}{2}\,\hat{u}(t,\xi) - \mathrm{i}\frac{c}{h}\sin\xi h\,\hat{u}(t,\xi)$$

with eigenvalues

$$-4\frac{Q}{h^2}\sin^2\frac{\xi h}{2} - \mathrm{i}\frac{c}{h}\sin\xi h.$$

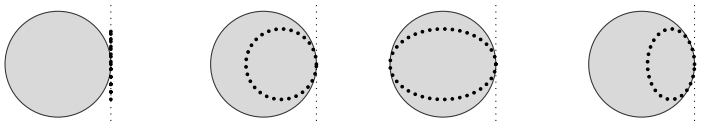
The eigenvalues lie along an ellipse in the negative half plane bounded by $-4Q/h^2$ along the real axis and $\pm c/h$ along the imaginary axis. For a forward Euler discretization this ellipse must be scaled by k to lie entirely within the unit circle centered at -1. For the Lax–Friedrichs method ($Q = h^2/2k$) and the k-scaled ellipse is

$$-2\sin^2\frac{\xi h}{2} - \mathrm{i}\frac{ck}{h}\sin\xi h.$$

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 is $k < h/|c|$ (from the imaginary direction). For the Lax–Wendroff method ($Q = c^2/2k$) and the k-scaled ellipse is

$$-2\left(\frac{ck}{h}\right)^2\sin^2\frac{\xi h}{2} - \mathrm{i}\frac{ck}{h}\sin\xi h.$$

When $k < h/|c|$ the k-scaled ellipse is completely inside the unit circle and when $k > h/|c|$ the k-scaled ellipse is completely outside the unit circle. The CFL numbers for the Lax–Wendroff method and the Lax–Friedrichs method are both $|c|k/h$.



Centered difference Upwind Lax–Friedrichs Lax–Wendroff

The Lax–Friedrichs and Lax–Wendroff can be expressed using a forward Euler time discretization (14.8). The eigenvalues of space discretization are overlaid on the region of absolute stability for forward Euler. The eigenvalues of the centered-difference scheme are outside of the region of absolute stability, so that scheme is unconditionally unstable. The eigenvalues for the Lax–Friedrichs scheme extend farther to the left than those of the Upwind scheme and consequently, the Lax–Friedrichs method is more dissipative. Similary, the eigenvalues of the Upwind scheme extend farther to the left than those of the Lax–Wendroff.

The remainder of this section looks at consistency and stability of the Lax–Wendroff method one more time as an alternative approach to what is discussed above. Let's rederive the Lax–Wendroff scheme from scratch. The Lax–Wendroff scheme achieves second-order accuracy by adding extra terms to cancel out the u_{tt} terms of the Taylor expansion of $u(t,x)$ in time. From the linear problem $u_t + cu_x = 0$ we have that $u_t = -cu_x$. Therefore, $u_{tt} = -cu_{tx} = c^2 u_{xx}$. Then the Taylor series expansion of $u(t_{n+1}, x_j)$ about $u \equiv u(t_n, x_j)$ is

$$\begin{aligned} u(t_{n+1}, x_j) &= u + ku_t + \frac{k^2}{2} u_{tt} + O(k^3) \\ &= u - cku_x + c^2 \frac{k^2}{2} u_{xx} + O(k^3). \end{aligned}$$

By using a second order discretization of ∂_x and ∂_{xx}, we get the scheme

$$\frac{U_j^{n+1} - U_j^n}{k} + c\frac{U_{j+1}^n - U_{j-1}^n}{2h} = \frac{c^2}{2} k \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}.$$

We can check the stability condition using von Neumann analysis

$$\frac{A-1}{k} + \mathrm{i}\frac{c}{h} \sin \xi h = -2c^2 \frac{k}{h^2} \sin^2 \frac{\xi h}{2}$$

from which we have that (again taking $\lambda = ck/h$)

$$A = 1 - 2\lambda^2 \sin^2 \frac{\xi h}{2} - \mathrm{i}\lambda \sin \xi h$$

Therefore, $|A| \le 1$ if and only if $|\lambda| \le 1$. So, the CFL number for the Lax–Wendroff scheme is $|c|k/h$.

14.3 Numerical diffusion and numerical dispersion

By examining the truncation error for these numerical schemes, we will get a more complete understanding of the nature of the numerical solution. Let's start by looking at the upwind scheme. From Taylor series expansion we find that the upwind scheme is a numerical approximation of

$$u_t + cu_x = \frac{1}{2}ch\left(1 - \frac{ck}{h}\right)u_{xx} + O(h^2). \tag{14.9}$$

That is, the upwind scheme is a first order approximation to

$$u_t + cu_x = 0, \tag{14.10}$$

but it is a second order approximation to

$$u_t + cu_x = \tfrac{1}{2}ch\left(1 - \frac{ck}{h}\right)u_{xx}. \tag{14.11}$$

The right-hand side of (14.11) 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 of approximating the original problem (14.10), it does a better job approximating the parabolic equation (14.11). Note what happens if we take smaller step sizes in time (keeping the mesh size in space fixed). As $ck/h \to 0$, we get $u_t + cu_x = \frac{1}{2}chu_{xx}$ and the solution is overly dissipative. On the other hand, note that if we take k equal to the CFL number ($k = h/c$) then not only does the u_{xx} term of (14.9) 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 (14.10), but it is a third-order approximation to

$$u_t + cu_x = -\frac{1}{6}ch^2\left(1 - \left(\frac{ck}{h}\right)^2\right)u_{xxx}. \tag{14.12}$$

The right-hand side of (14.12) contributes *numerical dispersion* to the computed solution.

There is an important distinction between dissipation and dispersion. If we take a Fourier component

$$u(t,x) = \mathrm{e}^{\mathrm{i}(\omega t - \xi x)}$$

as an *ansatz* to the modified upwind equation (14.11) $u_t + cu_x = \alpha u_{xx}$, we have the relationship

$$i\omega e^{i(\omega t - \xi x)} + c(-i\xi)e^{i(\omega t - \xi x)} = \alpha(-\xi^2)e^{i(\omega t - \xi x)}$$

from which we have that

$$i\omega + c(-i\xi) = \alpha(-\xi^2).$$

From this equation, we can determine the *dispersion relation* $\omega \equiv \omega(\xi)$

$$\omega = c\xi + i\alpha\xi^2.$$

Plugging this ω back into our ansatz gives us

$$\begin{aligned} u(t,x) &= e^{i(\omega t - \xi x)} \\ &= e^{ic\xi t}e^{-\alpha\xi^2 t}e^{-i\xi x} \\ &= \underbrace{e^{i\xi(ct-x)}}_{\text{advection}} \cdot \underbrace{e^{-\alpha\xi^2 t}}_{\text{dissipation}} \end{aligned}$$

Dissipation damps out terms with high wave numbers ξ. Using the same ansatz in the modified Lax–Wendroff equation (14.12) $u_t + cu_x = \beta u_{xxx}$ gives

$$i\omega e^{i(\omega t - \xi x)} + c(-i\xi)e^{i(\omega t - \xi x)} = \beta(-i\xi^3)e^{i(\omega t - \xi x)}$$

which we can simplify to derive the dispersion relation

$$\omega = c\xi + \beta\xi^3$$

For the Lax–Wendroff scheme we have

$$u = e^{i(c\xi + \beta\xi^3)t}e^{-i\xi x}.$$

The *group velocity* is given by the first derivative of the dispersion relationship

$$v_{\text{group}} = \omega'(\xi) = c + 3\beta\xi^2.$$

If the group velocity is a function of ξ, we say that the wave is *dispersive*. The *phase velocity* is given by the ratio of the dispersion relation and ξ:

$$v_{\text{phase}} = \frac{\omega}{\xi} = c + \beta\xi^2.$$

For a dispersive wave, waves of different wave numbers ξ travel with different velocities resulting in oscillations in the numerical solution. On the other hand, numerical viscosity damps out high wave numbers. High wave numbers are associated with large derivatives, for example, where there are discontinuities. A dispersive scheme oscillates around a discontinuity. A dissipative scheme smooths out the discontinuity. See Figure 14.1 on the facing page.

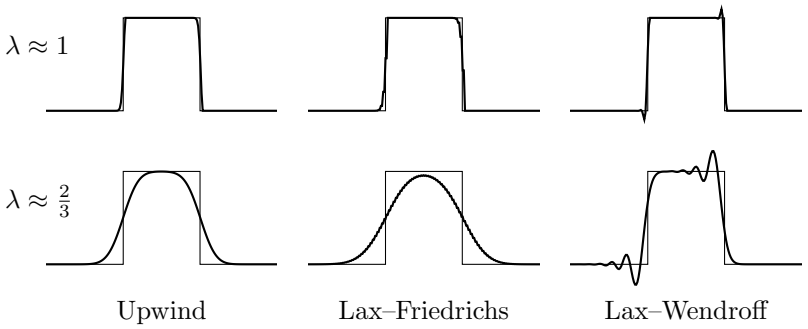


Figure 14.1: Solution to the advection equation $u_t + u_x = 0$ using the upwind, Lax–Friedrichs, and Lax–Wendroff methods for a rectangle function. Solutions are unstable for $\lambda > 1$. When λ is much smaller than one, the solutions exhibit significant diffusion (in the upwind and Lax–Friedrichs methods) and dispersion (Lax–Wendroff method). Solutions move from left to right.

14.4 Linear hyperbolic systems

Let's turn our attention to the one-dimensional linear wave equation

$$u_{tt} - c^2 u_{xx} = 0. \tag{14.13}$$

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 $\mathbf{u} \in \mathbb{R}^n$ 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:

$$\mathbf{T}^{-1}\mathbf{A}\mathbf{T} = \mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$$

where $\mathbf{T}$ is a matrix of eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$. That is, $\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{v}_i$. The eigenvalues λ_i are the *characteristic speeds*. For the wave equation (14.13)

$$\mathbf{A} = \begin{bmatrix} 0 & c^2 \\ -1 & 0 \end{bmatrix}$$

and the characteristic speeds are $\lambda_\pm = \pm c$.

Because we can diagonalize the matrix $\mathbf{A}$, the system can be completely decoupled into a system of independent advection equations. Let's look at this idea using the method of characteristics. Consider

$$\mathbf{u}_t + \mathbf{A}\mathbf{u}_x = \mathbf{0}.$$

Multiply the equation by $\mathbf{T}^{-1}$

$$\mathbf{T}^{-1}\mathbf{u}_t + \mathbf{T}^{-1}\mathbf{A}\mathbf{u}_x = \mathbf{0}$$

which is the same as

$$\mathbf{T}^{-1}\mathbf{u}_t + \mathbf{T}^{-1}\mathbf{A}\mathbf{T}\mathbf{T}^{-1}\mathbf{u}_x = \mathbf{0}$$

which is simply

$$\mathbf{T}^{-1}\mathbf{u}_t + \mathbf{\Lambda}\mathbf{T}^{-1}\mathbf{u}_x = \mathbf{0}$$

By letting $\mathbf{v} = \mathbf{T}^{-1}\mathbf{u}$ we have the equivalent system

$$\mathbf{v}_t + \mathbf{\Lambda}\mathbf{v}_x = \mathbf{0}.$$

in which each term uncouples

$$\partial_t v_i + \lambda_i \partial_x v_i = 0 \qquad i = 1, \ldots, n. \tag{14.14}$$

We solved the advection equation earlier. The solution is

$$v_i(t, x) = v_i(0, x - \lambda_i t)$$

The initial conditions are given by $\mathbf{v}_0 = \mathbf{T}^{-1}\mathbf{u}_i$. By setting $\mathbf{u} = \mathbf{T}\mathbf{v}$, the problem is solved.

Now, let's solve the problem numerically. We can discretize the uncoupled system (14.14) using, for example, the upwind scheme

$$\begin{aligned}\frac{(V_i)_j^{n+1} - (V_i)_j^{n+1}}{k} \\ + \frac{\lambda_i + |\lambda_i|}{2}\frac{(V_i)_j^n - (V_i)_{j-1}^n}{h} + \frac{\lambda_i - |\lambda_i|}{2}\frac{(V_i)_{j+1}^n - (V_i)_j^n}{h} = 0.\end{aligned} \tag{14.15}$$

If we let

$$\mathbf{\Lambda}^\pm = \begin{bmatrix} \ddots & & \\ & \frac{1}{2}(\lambda_i \pm |\lambda_i|) & \\ & & \ddots \end{bmatrix}$$

then scheme (14.15) is the same as

$$\frac{\mathbf{V}_j^{n+1} - \mathbf{V}_j^n}{k} + \mathbf{\Lambda}^+ \frac{\mathbf{V}_j^n - \mathbf{V}_{j-1}^n}{h} + \mathbf{\Lambda}^- \frac{\mathbf{V}_{j+1}^n - \mathbf{V}_j^n}{h} = 0.$$

We can rewrite this equation as

$$\mathbf{T}^{-1} \frac{\mathbf{U}_j^{n+1} - \mathbf{U}_j^n}{k} + \mathbf{\Lambda}^+ \mathbf{T}^{-1} \frac{\mathbf{U}_j^n - \mathbf{U}_{j-1}^n}{h} + \mathbf{\Lambda}^- \mathbf{T}^{-1} \frac{\mathbf{U}_{j+1}^n - \mathbf{U}_j^n}{h} = 0$$

which is equivalent to

$$\frac{\mathbf{U}_j^{n+1} - \mathbf{U}_j^n}{k} + \mathbf{T}\mathbf{\Lambda}^+ \mathbf{T}^{-1} \frac{\mathbf{U}_j^n - \mathbf{U}_{j-1}^n}{h} + \mathbf{T}\mathbf{\Lambda}^- \mathbf{T}^{-1} \frac{\mathbf{U}_{j+1}^n - \mathbf{U}_j^n}{h} = 0$$

which is equivalent to

$$\frac{\mathbf{U}_j^{n+1} - \mathbf{U}_j^n}{k} + \mathbf{A}^+ \frac{\mathbf{U}_j^n - \mathbf{U}_{j-1}^n}{h} + \mathbf{A}^- \frac{\mathbf{U}_{j+1}^n - \mathbf{U}_j^n}{h} = 0$$

where $\mathbf{A}^\pm = \mathbf{T}\mathbf{\Lambda}^\pm \mathbf{T}^{-1}$ is called the *characteristic decomposition*.

14.5 Nonlinear hyperbolic equations

We now extend the ideas previously discussed for the prototypic linear hyperbolic equation (14.1) to the nonlinear hyperbolic equation. We start with

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \tag{14.16}$$

where $u(t, x)$ is defined along the real line. This equation is also called a *conservation law* and the function $f(x)$ is called the *flux*. If we integrate (14.16) in x, we have that

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_\Omega u \,\mathrm{d}x + \int_\Omega \frac{\partial}{\partial x} f(u) \,\mathrm{d}x = 0.$$

If there is no flux through the boundary, then $\frac{\mathrm{d}}{\mathrm{d}t} \int_\Omega u \,\mathrm{d}x = 0$. It follows that $\int_\Omega u \,\mathrm{d}x$ is constant. So, u is a conserved quantity.

If the flux $f(u)$ is differentiable (f and u are Lipschitz continuous), we can apply the chain rule

$$\frac{\partial}{\partial x} f(u) = f'(u) \frac{\partial u}{\partial x}$$

and (14.16) becomes

$$\frac{\partial u}{\partial t} + f'(u)\frac{\partial u}{\partial x} = 0.$$

We can solve this problem using the method of characteristics (as we did in for the linear case)

$$\frac{\mathrm{d}}{\mathrm{d}t}u(t, x(t)) = \frac{\partial u}{\partial t} + \frac{\mathrm{d}x}{\mathrm{d}t}\frac{\partial u}{\partial x} = \frac{\partial u}{\partial t} + f'(u)\frac{\partial u}{\partial x} = 0.$$

This says that the solution $u(t, x)$ is constant (a Riemann invariant) along the characteristics given by

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f'(u).$$

Because u is invariant along $x(t)$, it follows that

$$\frac{\mathrm{d}x}{\mathrm{d}t} = f'(u(t, x(t)) = f'(u(0, x(0))) = f'(u_0(x_0)).$$

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 Lipschitz continuous at some point, the chain rule is no longer valid. Even if $u_0(x)$ is initially smooth, $u(t, x)$ is not necessarily continuous for all time if the flux is nonlinear. To see what happens, we consider the inviscid Burgers equation, a toy model for the Euler equations of gas dynamics.

Burgers' equation

Burgers' equation is given by

$$u_t + \left(\frac{u^2}{2}\right)_x = 0 \tag{14.17}$$

with initial conditions $u(0, x) = u_0(x)$. As long as $u(t, x)$ remains differentiable in x, we can use the chain rule to rewrite (14.17) as

$$u_t + uu_x = 0. \tag{14.18}$$

In this equation u now takes the place of the wave speed c that we had in the linear hyperbolic equation. The wave speed u is also the Riemann

invariant. That is, the characteristics $x_0(t)$ are straight line paths with slopes $\mathrm{d}x/\mathrm{d}t = u(0,x)$ along which $u(t,x(t)) = u(0,x(0))$ is constant. When u is larger, the solution $u(t,x)$ moves faster and it seems plausible that the higher part of the wave will move faster than the lower part of the wave and eventually catch up with it and possibly overtake it, say some critical time t_b. See Figure 14.2 on the next page. That is to say, if a characteristic curve of higher slope is to the left of one of lower slope, then the two curves will intersect at some point. The solution at this point would then be multivalued because we have differing information coming from two (or more) characteristic curves. A multivalued solution is both unphysical and mathematically ill-posed. So, something is wrong with either our reasoning or our formulation.

At the time the first derivative u_x becomes infinite, u develops a discontinuity or a *shock*. We can determine when this happens by finding the point when solution overturns, i.e., the derivative $\mathrm{d}u/\mathrm{d}xf$ is infinite.

$$\frac{\mathrm{d}u}{\mathrm{d}x} = \frac{\mathrm{d}u}{\mathrm{d}x_0}\frac{\mathrm{d}x_0}{\mathrm{d}x} = u_0'(x_0)\left(\frac{\mathrm{d}x}{\mathrm{d}x_0}\right)^{-1} = \frac{u_0'(x)}{u_0'(x)t+1}. \tag{14.19}$$

Since we want to find out when $\mathrm{d}x/\mathrm{d}x$ blows up, we set the right-hand side of (14.19) to ∞ and solve for t. In this case, we have that $t = -1/u_0'(x_0)$. The time that the first shock develops will be $t_b = \min_{x_0} -1/u_0'(x_0)$. For the Burgers equation, shocks develop only if at some point the gradient of the initial condition is negative, i.e., $u_0'(x) < 0$ for some x.

Can the faster part really overtake the slower moving part? No. We relied on the fact that $u(t,x)$ was smooth in order to apply the chain rule to take us from (14.17) to (14.18). When $(f(u))_x = \infty$, this argument is clearly invalidated. In the next section, we will look at how we can solve the ill-posed problem.

Weak solutions

As stated in the previous section, when the shock develops the differentiation that leads to $u_t + f'(u)u_x = 0$ is no longer valid and a unique solution no longer exists. Instead, we need to find a more *physical* equation to use. Mathematically, we will extend the solution by a "generalized weak solution." To construct the weak solution of $u_t + f(u)_x = 0$:

1. Multiply the equation by a sufficiently smooth *test function* φ that vanishes at the boundaries;

2. Integrate over space and time; and

3. Use integration by parts to pass the derivatives over to φ. Since φ vanishes at the boundaries, there will be no boundary terms.

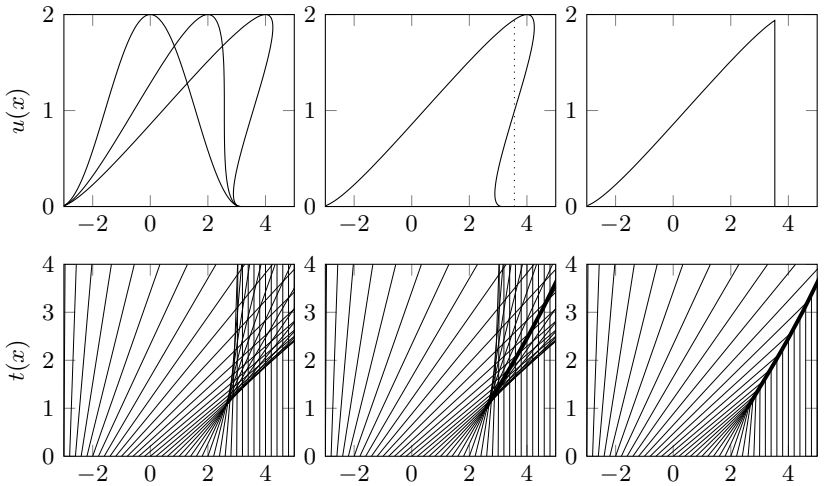


Figure 14.2: Top left: solutions u to Burgers' equation at $t = 0$, $t_b = 1$ (initial shock formation) and $t = 2$ (unphysical multivalued solution). Top right: weak solution at $t = 2$. Bottom: characteristic curves $x(t)$.

Essentially, we are taking using a well-behaved function φ to take the derivatives of $f(u)$, so that we can avoid differentiating $f(u)$ especially when $f(u)$ becomes non-differentiable. In this chapter, we'll use weak solutions as a means of determining an analytical solution. In the next chapter, we see that weak solutions can also be used numerically as the finite-element method.

Suppose that we have a test function φ, which is differentiable and has *compact support*. Compact support means that the closure of the set over which φ is nonzero is bounded. Notably, φ 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 Ω. In this case, $u_t + f(u)_x = 0$ implies

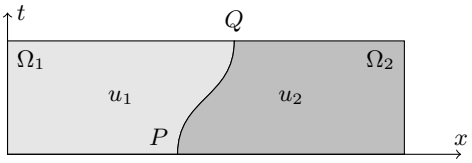
$$\int_0^\infty \int_\Omega (\varphi u_t + \varphi f(u)_x)\,\mathrm{d}x\,\mathrm{d}t = 0.$$

We integrate by parts to get

$$\int_0^\infty \int_{x_L}^{x_R} (\varphi_t u + \varphi_x f(u))\,\mathrm{d}x\,\mathrm{d}t = 0. \tag{14.20}$$

If (14.20) is satisfied 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 (strong) solution.

Consider a domain $\Omega \in \{(x,t)\} \subset \mathbb{R}^2$ and an arbitrary curve PQ dividing into Ω_1 and Ω_2 ($\Omega = \Omega_1 \cup \Omega_2$).



Suppose the $u(t,x)$ is a weak solution to $u_t + f(u)_x = 0$. Then

$$\begin{aligned} 0 &= -\iint_\Omega \varphi u_t + \varphi f(u)_x \,\mathrm{d}x\,\mathrm{d}t + \iint_\Omega \varphi u_t + \varphi f(u)_x \,\mathrm{d}x\,\mathrm{d}t \\ &= \iint_\Omega \varphi_t u + \varphi_x f(u) \,\mathrm{d}x\,\mathrm{d}t + \iint_\Omega \varphi u_t + \varphi f(u)_x \,\mathrm{d}x\,\mathrm{d}t \end{aligned}$$

Because φ vanishes on the boundary:

$$\begin{aligned} &= \iint_\Omega (\varphi u)_t + (\varphi f(u))_x \,\mathrm{d}x\,\mathrm{d}t \\ &= \iint_{\Omega_1} (\varphi u)_t + (\varphi f(u))_x \,\mathrm{d}x\,\mathrm{d}t + \iint_{\Omega_2} (\varphi u)_t + (\varphi f(u))_x \,\mathrm{d}x\,\mathrm{d}t \end{aligned}$$

By Green's Theorem:

$$= \int_{\partial\Omega_1} -\varphi u \,\mathrm{d}x + \varphi f(u) \,\mathrm{d}t + \int_{\partial\Omega_2} -\varphi u \,\mathrm{d}x + \varphi f(u) \,\mathrm{d}t$$

Because $\varphi(\partial\Omega) = 0$:

$$\begin{aligned} &= \int_P^Q -\varphi u_L \,\mathrm{d}x + \varphi f(u_L) \,\mathrm{d}t + \int_Q^P -\varphi u_R \,\mathrm{d}x + \varphi f(u_R) \,\mathrm{d}t \\ &= \int_P^Q \varphi(u_R - u_L) \,\mathrm{d}x - \varphi\left(f(u_R) - f(u_L)\right) \,\mathrm{d}t \end{aligned}$$

So, for all test functions $\varphi \in C_0^1(\Omega)$,

$$(u_R - u_L) \,\mathrm{d}x - \left(f(u_R) - f(u_L)\right) \,\mathrm{d}t = 0$$

along PQ. Therefore,

$$s = \frac{\mathrm{d}x}{\mathrm{d}t} = \frac{f(u_R) - f(u_L)}{u_R - u_L}.$$

This is called the *Rankine–Hugoniot jump condition* for shocks and s is called the shock speed. Note that in the limit as $u_L \to u_R$, $s \to f'(u)$. To simplify notation, one often uses brackets to indicate difference:

$$s = \frac{f(u_R) - f(u_L)}{u_R - u_L} \equiv \frac{[f]}{[s]}.$$

We can now continue to use the PDE with the weak solution.

Riemann problem

A *Riemann problem* is a PDE for which the initial condition is given by a two constant states. For the advection equation, it is

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{with} \quad u(x,0) = \begin{cases} u_L & x < 0 \\ u_R & x > 0 \end{cases} \tag{14.21}$$

Notice that if we scale t and x both by a positive constant c

$$\tilde{t} = ct, \qquad \tilde{x} = cx,$$

then the scaled problem is identical to the original problem

$$\frac{\partial u}{\partial \tilde{t}} + \frac{\partial}{\partial \tilde{x}} f(u) = 0 \quad \text{with} \quad u(\tilde{x},0) = \begin{cases} u_L & \tilde{x} < 0 \\ u_R & \tilde{x} > 0 \end{cases}$$

This suggests a solution (ansatz) for the problem. Let $\xi = x/t = \tilde{x}/\tilde{t}$. We will find a *self-similar solution* $u(x,t) = u(\xi)$ to Burgers' equation (where $f(x) = u^2/2$) to simpify the two-variable PDE into a one-variable ODE.

$$\begin{aligned} 0 &= u_t + \left(\tfrac{1}{2}u^2\right)_x \\ &= \xi_t u' + \xi_x u u' \\ &= \left(-\frac{x}{t^2}\right) u' + \left(\frac{1}{t}\right) u u' \end{aligned}$$

Multiplying by t and replacing x/t by ξ, we have

$$-\xi u' + u u' = u'(u - \xi) = 0$$

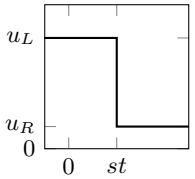
Either $u' = 0$, in which cases $u(x)$ is constant, $u = \xi$, in which case $u(x,t) = x/t$. The solution may also be a piecewise combination of these solutions. Let's consider the two cases when $u_L > u_R$ and when $u_L < u_R$.

1. If $u_L > u_R$, then the Rankine–Hugoniot condition gives

$$s = \frac{\frac{1}{2}u_R^2 - \frac{1}{2}u_L^2}{u_R - u_L} = \frac{1}{2}(u_R + u_L)$$

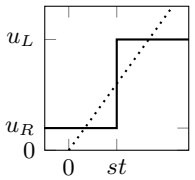
and the exact solution is

$$u(x,t) = \begin{cases} u_L, & \text{if } x/t < s \\ u_R, & \text{if } x/t > s \end{cases}$$



2. $u_L < u_R$, then one exact solution is

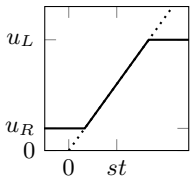
$$u(x,t) = \begin{cases} u_L, & \text{if } x/t < s \\ u_R, & \text{if } x/t > s \end{cases}$$



where $s = (u_R + u_L)/2$ is given by the Rankine–Hugoniot condition.

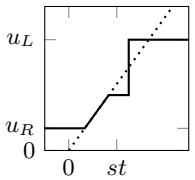
Another solution is

$$u(x,t) = \begin{cases} u_L, & \text{if } x/t < u_L \\ x/t, & \text{if } u_L < x/t < u_R \\ u_R, & \text{if } x/t > u_R \end{cases}$$



A third solution is

$$u(x,t) = \begin{cases} u_L, & \text{if } x/t < u_L \\ x/t, & \text{if } u_L < x/t < u_M \\ u_M, & \text{if } u_M < x/t < s \\ u_R & \text{if } x/t > s \end{cases}$$



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 physical entropy.) For the problem $u_t + f(u)_x = 0$ where $f(u)$ is the

convex flux, a solution satisfies the *Lax entropy condition*

$$f'(u_L) \;>\; s \equiv \frac{f(u_R) - f(u_L)}{u_R - u_L} \;>\; f'(u_R).$$

This says that the characteristics, which have speed $f'(u)$, must impinge from both sides on the shock, which has speed s.

For Burgers' equation $f(u) = u^2/2$, so $f'(u) = u$. When

1. $u_L > u_R$.

$$f'(u_L) = u_L > \tfrac{1}{2}(u_R + u_L) > u_R = f'(u_R)$$

   Therefore, the solution is an entropy shock.

2. $u_L < u_R$. The shock does not satisfy the entropy condition. So, the only possible weak solution is the (continuous) rarefaction wave.

Example. **Traffic flow equation.** 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 of sinks). Let $\rho(x,t)$ be the density of cars measured in cars per car length. So, $\rho = 0$ says that the road is completely empty and $\rho = 1$ says that there is bumper-to-bumper traffic. Then

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x} f(\rho) = 0$$

models the traffic flow where the flux $f(\rho)$ tells us the density of cars passing a given point in a given time interval (number of cars per second). The flux function $f(\rho) = U(\rho)\rho$ where $U(\rho)$ simply tells us the speed that each car is traveling. Let's make a couple reasonable assumptions to model $U(\rho)$:

1. Everyone travels as fast as they can while still obeying the posted speed limit $u_{\max}$.

2. Everyone keeps a safe distance behind the car ahead and adjusts their speed accordingly. If there are no other cars on the road, then drive as fast as possible and 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 the that flux

$$f(\rho) = U(\rho)\rho = u_{\max}(\rho - \rho^2)$$

is zero when the roads are completely empty (there are no cars to move) and when the roads are completely full (no one is moving). The flux $f(\rho)$ is maximum at

$$f'(\rho) = u_{\max}(1 - 2\rho) = 0$$

when $\rho = \frac{1}{2}$ (the roads are half full with the cars traveling at $\frac{1}{2}u_{\max}$, half the posted speed limit). So, our traffic equation is now

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(u_{\max}(1-\rho)\rho) = 0.$$

This equation is very similar in form to the Burgers equation.

Let's consider two simple examples: cars approaching an red stop light and cars leaving after the stop light turns green. Both of these examples are Riemann problems.

Red light We imagine several cars traveling at with ρ_L approaching a queue of stopped cars with $\rho_R = 1$. In this case we have

$$\rho(x,0) = \begin{cases} \rho_L, & x < 0 \\ \rho_R = 1, & x \geq 0 \end{cases}$$

The Rankine–Hugoniot relationship gives

$$s = \frac{f(\rho_L) - f(\rho_R)}{\rho_L - \rho_R} = \frac{u_{\max}\rho_L(1-\rho_L) - 0}{\rho_L - 1} = -u_{\max}\rho_L$$

There is a shock wave moving backwards as the cars queue up.

Green light We imagine several cars in queue with $\rho = 1$ and speed $u_L = 0$. This time the Lax entropy condition tells us that there is a rarefaction wave. The leading car moves forwards with a speed $u_{\max}$ and a rarefaction wave moves backwards with a speed $f'(\rho) = u_{\max}(1-2\rho)$ with $\rho = 1$. That is, speed $f'(1) = -u_{\max}$. ◀

14.6 Hyperbolic systems of conservation laws

Let's extend the discussion to a system of equations

$$\frac{\partial}{\partial t}\mathbf{u} + \frac{\partial}{\partial x}\mathbf{F}(\mathbf{u}) = 0 \qquad \text{with } \mathbf{u} \in \mathbb{R}^n. \tag{14.22}$$

If the Jacobian $\mathbf{F}'(\mathbf{u})$ has n real eigenvalues and hence a complete set of linearly independent eigenvectors, then the system (14.22) is called *hyperbolic*. In this case, there is an invertible map $\mathbf{T}(\mathbf{u})$ such that

$$\mathbf{T}^{-1}\mathbf{F}'\mathbf{T} = \mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \cdots, \lambda_n).$$

The eigenvalues λ_i are the characteristic speeds and $\mathbf{T} = (\mathbf{v}_1, \cdots, \mathbf{v}_n)$ are the eigenvectors. But nonlinear systems cannot be diagonalized globally.

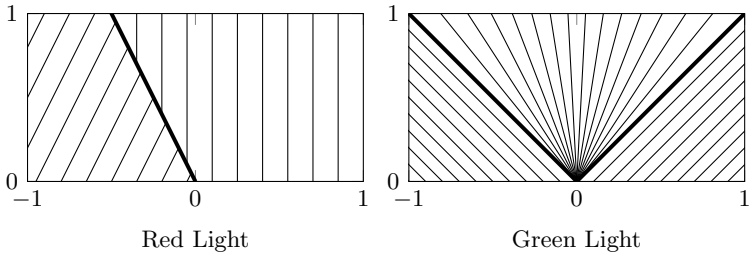


Figure 14.3: Characteristics for traffic flow with $\rho_L = 1/2$.

Example. Let's examine the one-dimensional shallow water equations and show that they form a hyperbolic system. The shallow water equations are given by

$$h_t + (hu)_x = 0 \tag{14.23a}$$

$$(hu)_t + (hu^2 + \tfrac{1}{2}gh^2)_x = 0 \tag{14.23b}$$

where g is the gravitational acceleration, h is the height of the water, and u is the velocity of the water. Let $m = hu$ denote the momentum. Then (14.23) becomes

$$\begin{aligned} h_t + m_x &= 0 \\ m_t + \frac{2m}{h}m_x - \frac{m^2}{h^2}h_x + ghh_x &= 0 \end{aligned}$$

which can be written as

$$\begin{bmatrix} h \\ m \end{bmatrix}_t + \underbrace{\begin{bmatrix} 0 & 1 \\ gh - u^2 & 2u \end{bmatrix}}_{\mathbf{F}'} \begin{bmatrix} h \\ m \end{bmatrix}_x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The eigenvalues of the Jacobian matrix $\mathbf{F}'$ are given by

$$\begin{vmatrix} -\lambda & 1 \\ gh - u^2 & 2u - \lambda \end{vmatrix} = \lambda^2 - 2u\lambda + u^2 - gh.$$

So $\lambda_\pm = \frac{1}{2}\left(2u \pm \sqrt{4u^2 - 4u^2 + 4gh}\right) = u \pm \sqrt{gh}$. If the height of the water h is positive, then there are two real eigenvalues and the system is strictly hyperbolic.

The two eigenvalues give us the characteristic speeds for the shallow water equation. Notably the speed of water waves are functions of the depth of the water h (and the velocity of the current u). Tsunamis,

for example, occur when a disturbance creates a wave in the ocean with wavelength of 100 km or more and a displacement of a meter or less. The wave travels quickly in the ocean where the depth may be several kilometers (speed $= \pm\sqrt{gh}$). Once it reaches the continental shelf, where the depth decreases dramatically, the wave slows considerably, and the height of the wave grows. A shock forms as the faster characteristics intersect with the slower characteristics.

Refraction of water waves is also caused by the $\sqrt{gh}$ speed dependence. Ocean waves turn to hit the beach perpendicularly even when the wind is not blowing straight into shore. Waves are produced by a Kelvin–Helmholtz instability resulting from the wind blowing over water. In the deeper water, waves are driven in the direction of the wind. When they reach shallow water, the wave moves at different speeds. The part of the wave in shallower water moves slower than the part in deeper, turning the wave into shore ◀

Riemann invariants

As we saw in Section 14.1, information is advected along characteristics. The quantity that is constant along a characteristic curve is the *Riemann invariant*. For the advection equation $u_t + cu_x = 0$, the characteristics are given by $\mathrm{d}x/\mathrm{d}t = c$ and the Riemann invariant is u. For a general scalar conservation law $u_t + f(u)_x = 0$, we have that $u_t + f'(u)u_x = 0$ when u is differentiable. The characteristics are given by $\mathrm{d}x/\mathrm{d}t = f'(u)$ and it follows that

$$\frac{\mathrm{d}}{\mathrm{d}t}u(x(t),t) = \frac{\partial u}{\partial t} + \frac{\mathrm{d}x}{\mathrm{d}t}\frac{\partial u}{\partial x} = \frac{\partial u}{\partial t} + f'(u)\frac{\partial u}{\partial x} = 0.$$

So, $u(x,t)$ is the Riemann invariant. For systems of conservation laws, it's a little more complicated. Let's once again consider the system

$$\frac{\partial}{\partial t}\mathbf{u} + \frac{\partial}{\partial x}F(\mathbf{u}) = 0 \qquad \text{with } \mathbf{u} \in \mathbb{R}^n.$$

If $\mathbf{u}$ is differentiable,

$$\frac{\partial}{\partial t}\mathbf{u} + \mathbf{F}'(\mathbf{u})\frac{\partial}{\partial x}\mathbf{u} = 0 \tag{14.25}$$

where $\mathbf{F}'$ is the Jacobian matrix of $\mathbf{F}$ with real eigenvalues $\{\lambda_1, \cdots, \lambda_n\}$. Let ∇w_i—for some $w \equiv w(\mathbf{u})$—be the *left eigenvector* of $\mathbf{F}'(\mathbf{u})$ corresponding to the eigenvalues λ_i. Then left multiplying (14.25) by ∇w_i

$$\nabla w_i \frac{\partial}{\partial t}\mathbf{u} + \nabla w_i \mathbf{F}'(\mathbf{u})\frac{\partial}{\partial x}\mathbf{u} = 0.$$

Because ∇w_i is the left eigenvector of $\mathbf{F}'(\mathbf{u})$, it follows that

$$\nabla w_i \frac{\partial}{\partial t}\mathbf{u} + \lambda_i \nabla w_i \frac{\partial}{\partial x}\mathbf{u} = 0.$$

And by the chain rule, we have

$$\frac{\partial}{\partial t} w_i + \lambda_i \frac{\partial}{\partial x} w_i = 0 \tag{14.26}$$

which is simply the scalar advection equation. So, w_i is a Riemann invariant, because it is constant along the characteristic curve given by $\mathrm{d}x/\mathrm{d}t = \lambda_i$. Notice that (14.26) is simply the diagonalization of the original system.

Example. Let's compute the Riemann invariants for the shallow water equations. Recall that

$$\mathbf{F}' = \begin{bmatrix} 0 & 1 \\ gh - u^2 & 2u \end{bmatrix}$$

with eigenvalues $\lambda\pm = u \pm \sqrt{gh}$. We compute the left eigenvectors to be $\left(u \mp \sqrt{gh}, 1\right)$. Recall that we defined $\mathbf{u} = (h, m)$ with $m = hu$. Putting the eigenvectors in terms of h and m, we have

$$\left(\frac{m}{h} \mp \sqrt{gh}, 1\right).$$

Finally, we need this vector to be an exact differential, so let's rescale this vector by $1/h$—it will still be an eigenvector. We now have

$$\nabla_{\mathbf{u}} w_\pm = \left(\frac{m}{h^2} \mp \sqrt{\frac{g}{h}}, \frac{1}{h}\right).$$

So,

$$w_\pm(h, m) = \frac{m}{h} \mp 2\sqrt{gh} = u \mp 2\sqrt{gh}.$$

The Riemann invariants $u \mp 2\sqrt{gh}$ are constant along the characteristics $\mathrm{d}x/\mathrm{d}t = u \pm \sqrt{gh}$. ◀

Open boundary conditions

The advection equation $u_t + cu_x = 0$ has one space derivative. So, it needs one influx boundary condition for uniqueness. Recall that the

heat equation which has two derivatives in space needed two boundary conditions. A hyperbolic system of conservation laws (14.22) with n equations needs n boundary conditions.

Often we want to solve the shallow water equations or the Euler equations of gas dynamics using open boundary conditions. For example, in the ocean we may want to prescribe inflow boundary conditions to model incoming water wavers. Or we may want to model airflow over an airfoil by injecting a flow upwind of the airfoil. Downwind of the airfoil, we want the gas to escape the domain without reflecting back into our simulation. Or maybe we may want to model a bomb blast in the semi-infinite domain above the ground. Without an open boundary, information will likely be reflected back into our domain of interest polluting our solution. It is typically a bad idea to force the boundary to be equal to the external data, because the external data does not perfectly match the information leaving the domain.

One simple fix is to use a larger domain and stop the simulation before the reflected information pollutes the solution. We could also use a thin sponge layer with added viscosity to absorb energy at the boundary. Other approaches include creating an artificial boundary that tries to match the reflected waves with waves of opposite amplitude. Such non-reflecting boundary conditions were developed by Enquist and Majda in 1977.

Let's look at a characteristic-based approach to implement open boundary conditions. Recall that the Riemann invariant is constant along a characteristic. For the shallow water equation we had a right-moving characteristic with slope $u + \sqrt{gh}$ and left-moving characteristic with slope $u - \sqrt{gh}$. At the boundary, we will prescribe the Riemann invariant along the characteristic flowing into the domain. Suppose that we want an influx boundary condition on the right and an open boundary on the left.

Take the boundaries at $x = x_L$ and $x = x_R$ and suppose that the external information is given by $h(x_L, t) = h_L$ and $u(x_L, t) = u_L$ on the left and $h(x_R, t) = h_R$ and $u(x_R, t) = u_R$ on the right. We will match the Riemann invariants on the left by taking $w_-(h, u)$ to match the information outside entering our domain and $w_+(h, u)$ to match the solution inside our domain. Conversely, we will match the Riemann invariants on the right by taking $w_+(h, u)$ to match the solution inside our domain and $w_-(h, u)$ to match the information outside our domain. The Riemann invariants are

$$w_+ = u - 2\sqrt{gh} \qquad \text{and} \qquad w_- = u + 2\sqrt{gh},$$

so we have

$$\text{Left side:} \begin{cases} w_- = \text{extrapolation of } u + 2\sqrt{gh} \\ w_+ = u_L - 2\sqrt{gh_L} \end{cases}$$

$$\text{Right side:} \begin{cases} w_- = u_R + 2\sqrt{gh_R} \\ w_+ = \text{extrapolation of } u - 2\sqrt{gh} \end{cases}$$

Solving this system of equations for u and h gives us the boundary conditions for our problem:

$$\begin{aligned} h &= (w_- - w_+)/16g \\ u &= (w_- + w_+)/2. \end{aligned}$$

For example, consider the left boundary at x_0. To determine w_- with second order accuracy we extrapolate

$$w_- = (2U_1 - U_2) + 2\sqrt{g(2H_1 - H_2)},$$

and we set w_+ according to the external information u_L and h_L

$$w_+ = u_L + 2\sqrt{gh_L}.$$

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 side.

The Riemann problem

First-order Godonov schemes approximate the problem by 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 CFL condition.

The Riemann problem for a system of hyperbolic equations

$$\mathbf{u}_t + F(\mathbf{u})_x = 0 \qquad \text{with} \qquad \mathbf{u}(x,0) = \begin{cases} \mathbf{u}_L, & x < 0 \\ \mathbf{u}_R, & x > 0 \end{cases}$$

can be solved analytically using a self-similar solution by taking $\mathbf{u}(x,t) = \mathbf{u}(x/t) = \mathbf{u}(\xi)$. Then

$$\left(-\frac{x}{t^2}\right)\mathbf{u}' + \frac{1}{t}F'(\mathbf{u})\mathbf{u}' = 0$$

from which we have that

$$(F'(\mathbf{u}) - \xi\mathbf{I})\,\mathbf{u}' = 0.$$

So either $\mathbf{u}' = 0$ in which case $\mathbf{u}$ is constant in ξ or $\det(F'(\mathbf{u}) - \xi\mathbf{I}) = 0$ in which case ξ is an eigenvalue of $F'(\mathbf{u})$. The solution is piecewise constant connected by shocks or rarefactions when $\mathbf{u}' = 0$. The solution associated with $\det(F'(\mathbf{u}) - \xi\mathbf{I}) = 0$ is a contact discontinuity that travels with characteristic speed ξ. Unlike shock discontinuities, across which there is mass flow, contact discontinuities are merely discontinuities that flow with the fluid. Think of the surface separating two different types of fluids that moving together.

14.7 Methods for nonlinear hyperbolic systems

Example. Consider the Riemann problem

$$u_t + \left(\frac{u^2}{2}\right)_x = 0 \qquad \text{with} \qquad u(x,0) = \begin{cases} 1, & x < 0 \\ 0, & x > 0 \end{cases}.$$

The shock speed is $s = [f(u)]/[u] = \frac{1}{2}$, so the exact solution is given by

$$u(x,0) = \begin{cases} 1 & x < t/2 \\ 0 & x > t/2 \end{cases}.$$

The upwind scheme for

$$u_t + uu_x = 0$$

is

$$\frac{U_j^{n+1} - U_j^n}{k} + U_j^n \frac{U_j^n - U_{j-1}^n}{h} = 0 \tag{14.27}$$

with $x_j = jh$. For $x_j \le 0$, $U_j^1 = U_j^0$ and for $j > 0$, $U_j^1 = U_j^0$. Therefore, $U_j^1 = U_j^0$ for all j, and hence $U_j^n = U_j^0$ for all n, This says that the numerical shock speed is identically zero. Although (14.27) appears to be a reasonable scheme, it gives the *wrong* solution. The error is that $u_t + uu_x = 0$ is not in conservation form. ◀

Recall that a discretization U_j is *conservative* if $\sum_j U_j$ is constant in time. Equivalently, a scheme is conservative if it can be written as

$$\partial_t U_j + \frac{F_{j+1/2} - F_{j-1/2}}{h} = 0$$

where the numerical flux $F_{j+1/2} = F(u_{j-p}, \ldots, u_{j+p})$. In this case,

$$\partial_t \sum_j U_j + \frac{1}{h} \sum_j (F_{j+1/2} - F_{j-1/2}) = 0$$

from which it follows that

$$\partial_t \sum_j U_j = 0$$

if there is no net flux through the boundaries.

Example. The Lax–Friedrichs scheme is conservative.

$$\frac{U_j^{n+1} - \frac{1}{2}(U_{j+1}^n - U_{j-1}^n)}{k} + \frac{f(U_{j+1}^n) - f(U_{j-1}^n)}{2h} = 0$$

can be rewritten as

$$\frac{U_j^{n+1} - U_j^n}{k} + \frac{U_j^n - \frac{1}{2}(U_{j+1}^n + U_{j-1}^n)}{k} + \frac{f(U_{j+1}^n) - f(U_{j-1}^n)}{2h} = 0.$$

So, we have

$$\frac{U_j^{n+1} - U_j^n}{k} + \frac{F_{j+1/2}^n - F_{j-1/2}^n}{h} = 0$$

where the numerical flux

$$F_{j+1/2}^n = \frac{f(U_{j+1}^n) + f(U_j^n)}{2} - \frac{h}{2k}(U_{j+1}^n - U_j^n). \qquad \blacktriangleleft$$

Theorem 44 (Lax–Wendroff Theorem)**.** *If a consistent, conservative discretization of $u_t + F(u)_x = 0$ converges, then it converges to a weak solution of $u_t + F(u)_x = 0$.*

Note that the Lax–Friedrichs scheme can also be written as

$$\underbrace{\frac{U_j^{n+1} - U_j^n}{k} + \frac{f(U_{j+1}^n) - f(U_{j-1}^n)}{2h}}_{\text{centered-difference}} = \underbrace{\frac{h^2}{2k}}_{O(h)} \underbrace{\frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}}_{\text{numerical viscosity}}$$

We can improve the scheme a little bit by changing the numerical flux. The *local Lax–Friedrichs* method defines the numerical flux by

$$F_{j+1/2} = \frac{f(U_{j+1}) + f(U_j)}{2} - \frac{\sqrt{a_j}}{2}(U_{j+1} - U_j)$$

where

$$\sqrt{a_j} = \sup_{U_j < u < U_{j+1}} |f'(u)|$$

and $k \le h/\sqrt{a_j}$ by the CFL condition.

Finite volume methods

Rather than solving the problem using Taylor series expansion in the form of a finite-difference scheme we can use a integral formulation in called a *finite volume method*. In such a method we consider breaking the space domain into a cells centered at $x_{j+1/2} = (x_{j+1} + x_j)/2$ 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

$$\frac{1}{kh}\int_{t_n}^{t_{n+1}}\int_{x_{j-1/2}}^{x_{j+1/2}} (u_t + f(u)_x)\,\mathrm{d}x\,\mathrm{d}t = 0$$

which in turn can be integrated to be

$$\frac{1}{kh}\int_{x_{j-1/2}}^{x_{j+1/2}} u\Big|_{t_n}^{t_{n+1}}\mathrm{d}x + \frac{1}{kh}\int_{t_n}^{t_{n+1}} f(u)\Big|_{x_{j-1/2}}^{x_{j+1/2}}\mathrm{d}t = 0$$

or simply

$$\int_{x_{j-1/2}}^{x_{j+1/2}} u(t_{n+1},x)\,\mathrm{d}x - \int_{x_{j-1/2}}^{x_{j+1/2}} u(t_n,x)\,\mathrm{d}x + \int_{t^n}^{t^{n+1}} f(u(t,x_{j+1/2}))\,\mathrm{d}t - \int_{t^n}^{t^{n+1}} f(u(t,x_{j-1/2}))\,\mathrm{d}t = 0 \quad (14.28)$$

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

$$U_j^n = \frac{1}{h}\int_{x_{j-1/2}}^{x_{j+1/2}} u(t_n,x)\,\mathrm{d}x$$

and the flux at x_j over the time interval $[t_n, t_{n+1}]$ to be

$$F_{j+1/2}^{n+1/2} = \frac{1}{k}\int_{t_n}^{t_{n+1}} f(u(t,x_{j+1/2}))\,\mathrm{d}t.$$

Think of u as density and U as mass, and think of f as flux density and F as flux. Then we have

$$\frac{U_j^{n+1} - U_j^n}{k} + \frac{F_{j+1/2}^{n+1/2} - F_{j-1/2}^{n+1/2}}{h} = 0. \quad (14.29)$$

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

$$U_j^n = \frac{1}{2}\left(U_{j-1/2}^n + U_{j+1/2}^n\right)$$

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

$$\frac{U_{j+1/2}^{n+1} - \frac{1}{2}(U_j^n + U_{j+1}^n)}{k} + \frac{f(U_{j+1}^n) - f(U_j^n)}{h} = 0$$

where the CFL condition is given by $k \le |\lambda_{\max}|h/2$ where $\lambda_{\max}$ is the largest eigenvalue of the Jacobian $f'(u)$. Recall that approximating $U_{j+1/2}^n$ with $\frac{1}{2}(U_j^n + U_{j+1}^n)$ introduces numerical viscosity into the solution.

To extend the method to *second order*, we need to use *piecewise linear* approximations (Nessyahu and Tadmor [1990]). Define

$$P_j(x) = U_j^n + \partial_x U_j^n \cdot (x - x_j), \qquad x_j < x < x_{j+1}.$$

Then

$$\begin{aligned}
U_{j+1/2}^n &= \frac{1}{h}\int_{x_j}^{x_{j+1}} u(t_n, x)\,\mathrm{d}x \\
&\approx \frac{1}{h}\int_{x_j}^{x_{j+1}} P_j(x)\,\mathrm{d}x \\
&= \frac{1}{h}\int_{x_j}^{x_{j+1/2}} P_j(x)\,\mathrm{d}x + \frac{1}{h}\int_{x_j}^{x_{j+1/2}} P_{j+1}(x)\,\mathrm{d}x \\
&= \frac{1}{2}(U_j^n + U_{j+1}^n) + \frac{1}{8}h(\partial_x U_j^n - \partial_x U_{j+1}^n).
\end{aligned}$$

From (14.29), we have

$$U_{j+1/2}^{n+1} = \frac{1}{2}(U_j^n + U_{j+1}^n) + \frac{1}{8}h(\partial_x U_j^n - \partial_x U_{j+1}^n) - \frac{k}{h}\left(F_{j+1}^{n+1/2} - F_j^{n+1/2}\right).$$

We still need a second-order approximation for the flux $F_j^{n+1/2}$. Using the midpoint rule:

$$F_j^{n+1/2} = \frac{1}{k}\int_{t_n}^{t_{n+1}} f(u(t,x_j))\,\mathrm{d}t \approx f(u(t_{n+1/2}, x_j))$$

with

$$u(t_{n+1/2}, x_j) \approx u(t_n, x_j) + \frac{k}{2}\partial_t u(t_n, x_j) = u(t_n, x_j) - \frac{k}{2}\partial_x f(u(t_n, x_j))$$

where we use $u_t + f(u)_x = 0$ for the equality above. This give us the staggered, two-step predictor-corrector method:

$$\begin{aligned} U_j^{n+1/2} &= U_j^n - \frac{k}{2}\partial_x f(U_j^n) \\ U_{j+1/2}^{n+1} &= \frac{1}{2}(U_j^n + U_{j+1}^n) \\ &\qquad + \frac{h}{8}(\partial_x U_j^n - \partial_x U_{j+1}^n) - \frac{k}{h}\left[f(U_{j+1}^{n+1/2}) - f(U_j^{n+1/2})\right] \end{aligned}$$

We need to use slope limiters to approximate the slopes $\partial_x U_j$ and $\partial_x f(U_j)$, otherwise we introduce oscillations. For $\partial_x U_j$, we define the slope σ_j to be

$$\sigma_j = \frac{U_{j+1} - U_j}{h}\phi(\theta_j) \quad \text{where} \quad \theta_j = \frac{U_j - U_{j-1}}{U_{j+1} - U_j}$$

where common limiting functions are the minmod and the van Leer.

$$\begin{aligned} &\text{minmod} && \phi(\theta) = \max(0, \min(1, \theta)) \\ &\text{van Leer} && \phi(\theta) = \frac{|\theta| + \theta}{1 + |\theta|} \end{aligned}$$

The van Leer slope limiter gives a sharper shock that 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. For example, whenever $U_j - U_{j-1}$ and $U_{j+1} - U_j$ have opposite signs. While the slope limiter removes oscillations and stabilizes the solution, it reduces the method to first-order approximation near an oscillation. Hence, near discontinuities we can get at best first-order and more typically half-order convergence. To get higher orders, one may use essentially non-oscillatory (ENO) interpolation schemes (Shu and Osher [1988]).

Notice that σ_j is a symmetric function of U_{j-1}, U_j and U_{j+1}. That is,

$$\frac{U_{j+1} - U_j}{h}\phi\left(\frac{U_j - U_{j-1}}{U_{j+1} - U_j}\right) = \frac{U_j - U_{j-1}}{h}\phi\left(\frac{U_{j+1} - U_j}{U_j - U_{j-1}}\right).$$

Also, note that the slope limiter as written above is undefined whenever $U_{j+1} = U_j$. But this happens precisely when we want the slope to be zero. We can fix the problem by adding a conditional to the denominator as in the following the MATLAB code

```
dU = [0;diff(U)];
s = dU.*phi( [dU(2:end);0]./(dU + (dU==0)) );
```

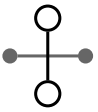
Notice that `dU` is padded with a zero on the left and then on the right, so that `s` has the same number of elements as `U`.

Exercises

✍ 14.1. Show that the order of convergence for the Lax–Friedrichs scheme is $O(k + h^2/k)$ and compute the dispersion relation. Taking the CFL condition and numerical viscosity into consideration, how should we best choose h and k to minimize error.

14.2. Solve the advection equation $u_t + u_x = 0$, with initial conditions $u(0, x) = 1$ if $|x - \frac{1}{2}| < \frac{1}{4}$ and $u(0, x) = 0$ otherwise, using the upwind scheme and the Lax–Wendroff scheme. Use the domain $x \in [0, 1]$ with periodic boundary conditions. Compare the solutions at $t = 1$.

✍ 14.3. Consider a finite-difference scheme for the linear hyperbolic equation $u_t + cu_x = 0$ that uses the following stencil (leap-frog in time and center-difference in space):



Determine the order of the scheme and the stability conditions for a consistent scheme. Discuss whether the scheme is dispersive or dissipative. Comment on how dispersion or dissipation affects the stability, i.e., is this a good scheme? Show that you can derive this scheme starting with (14.4) and counteracting the unstable term.

14.4. Consider a finite-difference scheme for $u_t + u_x = 0$ that uses a fourth-order centered-difference approximation for u_x and the fourth-order Runge–Kutta method in time. Determine the order of the scheme and discuss the the stability. Determine whether the scheme is dispersive or dissipative.

✍ 14.5. 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 ρ 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 that equations. To close the system we add the equation of state for an ideal polytropic gas:

$$E = \tfrac{1}{2}\rho u^2 + \frac{p}{\gamma - 1}$$

where $\gamma > 1$ is a constant. (For air, $\gamma \approx 1.4$) 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. Hint: Express the problem as a nonlinear hyperbolic system using the variables ρ, u and p.

✍ 14.6. 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 (using the method of characteristics or some other means) and numerically using the local Lax–Friedrichs method. Use the analytic solution to confirm that the numerical solution is correct by plotting together at several time steps.

✍ 14.7. The "dam break" problem. Consider the shallow water equations

$$h_t + (hu)_x = 0 \tag{14.31a}$$

$$(hu)_t + (hu^2 + \tfrac{1}{2}gh^2)_x = 0 \tag{14.31b}$$

where g is the gravitational acceleration, h is the height of the water, and u us the velocity of the water. Set $g = 1$. Solve the problem with initial conditions $h = 1$ if $x < 0$, $h = 0.2$ if $x > 0$ and $u = 0$ over the domain $[-1, 1]$ using the first and second order central schemes with constant boundary conditions. (For the second order, you will need to use a slope limiter.) Use 100 and 200 points for space discretization and the time step according to the CFL condition. Plot the numerical result at $t = 0.25$ against the "exact" solution (using a very fine mesh in space and step-size in time).

CHAPTER 15

Finite Element Method

In this chapter, we examine the Finite Element Method (FEM) and applications of it to elliptic equations. Typical elliptic equations are Laplace's equation and Poisson's equation, which model the steady-state distribution of electrical charge or temperature either with a source term (Poisson's equation) or without (Laplace's equation), and the steady-state Schrödinger equation. 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 an unknown temperature distribution $u(x)$, a known heat source $f(x)$ and a heat conductivity $\kappa > 0$ with constant temperature $u(0) = u(1) = 0$ at the ends of the bar. Fourier's law states that the heat flux $q(x) = -\kappa u'(x)$ and the conservation of energy says that $q'(x) = f(x)$. Take $\kappa = 1$. The steady-state solution can be found be formatting the problem in three different ways—as a *boundary value problem* Ⓓ, as a *minimization problem* Ⓜ, and as a *variational problem* Ⓥ (also called the weak formulation).

Ⓓ *Boundary value:* Find u such that $-u'' = f$ with $u(0) = u(1) = 0$.

The set V is a *vector space* if $\alpha u + v \in V$ for all real numbers α and $u, v \in V$. Let V be the vector space of piecewise-differentiable functions over the interval $[0, 1]$ that vanish on the endpoints. Define the *total*

potential energy

$$F(v) = \tfrac{1}{2}\langle v', v'\rangle - \langle f, v\rangle$$

where the inner product

$$\langle u, v\rangle = \int_0^1 u(x)v(x)\,\mathrm{d}x.$$

In the next section, we'll introduce the notation $a(\cdot, \cdot)$ and $b(\cdot)$ in the place of these inner products. For now, let's keep things simple.

Ⓜ *Minimization:* Find $u \in V$ such that $F(u) \le F(v)$ for all $v \in V$. In other words, find the solution that minimizes the energy.

Ⓥ *Variational:* Find $u \in V$ such that $\langle u', v'\rangle = \langle f, v\rangle$ for all $v \in V$.

Theorem 45. *Under suitable conditions the boundary value problem Ⓓ, minimization problem Ⓜ and variational problem Ⓥ are all equivalent.*

Proof. First, we show Ⓓ is equivalent to Ⓥ. We start by showing Ⓓ implies Ⓥ. If $-u'' = f$, then for all $v \in V$

$$\begin{aligned} -\int_0^1 u''v\,\mathrm{d}x &= \int_0^1 fv\,\mathrm{d}x \\ u'v'\big|_0^1 - \int_0^1 u'v'\,\mathrm{d}x &= \int_0^1 fv\,\mathrm{d}x \\ \langle u', v'\rangle &= \langle f, v\rangle \end{aligned}$$

Now, we show Ⓥ implies Ⓓ given that u' is differentiable.

$$\begin{aligned} \int_0^1 u'v'\,\mathrm{d}x &= \int_0^1 fv\,\mathrm{d}x \qquad \text{for all } v \in V \\ u'v'\big|_0^1 - \int_0^1 u''v\,\mathrm{d}x &= \int_0^1 fv\,\mathrm{d}x \\ \int_0^1 (u'' + f)v\,\mathrm{d}x &= 0 \end{aligned}$$

which is true for all $v \in V$. Therefore, $-u'' = f$.

Now, we show Ⓜ is equivalent to Ⓥ. First, we'll show that Ⓥ follows from Ⓜ. Define the perturbation $g(\varepsilon) = F(u + \varepsilon w)$ for an arbitrary function w with $\varepsilon > 0$. The *variational derivative* of F is defined as

$\delta F = \mathrm{d}g/\mathrm{d}\varepsilon|_{\varepsilon=0}$. The energy $F(u)$ is at a minimum u when its variation derivative equals zero. Let's find when this occurs.

$$\begin{aligned} g(\varepsilon) &= \tfrac{1}{2}\langle u' + \varepsilon w',\, u' + \varepsilon w'\rangle - \langle f,\, u + \varepsilon w\rangle \\ &= \tfrac{1}{2}\langle u',\, u'\rangle + \varepsilon\langle u',\, w'\rangle + \tfrac{1}{2}\varepsilon^2\langle w',\, w'\rangle - \langle f,\, u\rangle - \varepsilon\langle f,\, w\rangle \\ 0 = \left.\frac{\mathrm{d}g}{\mathrm{d}\varepsilon}\right|_{\varepsilon=0} &= \langle u',\, w'\rangle + \varepsilon\langle w',\, w'\rangle - \langle f,\, w\rangle\Big|_{\varepsilon=0} = \langle u',\, w'\rangle - \langle f,\, w\rangle\,. \end{aligned}$$

So $\langle u',\, w'\rangle = \langle f,\, w\rangle$ for all $w \in V$.
Finally, we show that Ⓜ follows from Ⓥ:

$$\begin{aligned} F(v) &= F(u + v - u) \\ &= F(u + w) \quad \text{where } w = v - u \in V \\ &= \tfrac{1}{2}\langle u' + w',\, u' + w'\rangle - \langle f,\, u + w\rangle \\ &= \tfrac{1}{2}\langle u',\, u'\rangle + \langle u',\, w'\rangle + \tfrac{1}{2}\langle w',\, w'\rangle - \langle f,\, u\rangle - \langle f,\, w\rangle \\ &= F(u) + \tfrac{1}{2}\langle w',\, w'\rangle \qquad \text{because } \langle u',\, w'\rangle = \langle f,\, w\rangle \\ &\geq F(u) \end{aligned}$$

□

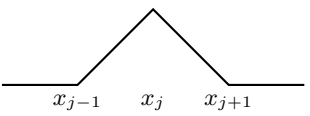
The basic idea of the finite element method is to find a solution to the variational problem Ⓥ or the minimization problem Ⓜ in a finite dimensional subspace V_h of V. The finite element solution u_h is a projection of u. The minimization problem is called the *Raleigh–Ritz* formulation and the variational problem is called the *Galerkin* formulation.

Ⓜ *Raleigh–Ritz*: Find $u_h \in V_h$ such that $F(u_h) \leq F(v_h)$ for all $v_h \in V_h$.

Ⓥ *Galerkin*: Find $u_h \in V_h$ such that $\langle u_h',\, v_h'\rangle = \langle f,\, v_h\rangle$ for all $v_h \in V_h$.

Let's apply the Galerkin formulation to the original boundary value problem. Consider V_h to be the subspace of continuous piecewise-linear functions with nodes at $\{x_i\}$. We may also consider piecewise-quadratic or piecewise-polynomial basis functions as basis elements. We'll restrict ourselves to a uniform partition with nodes at $x_j = jh$ and with $x_0 = 0$ 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

$$\varphi_j(x) = \begin{cases} 1 + t_j(x), & t_j(x) \in (-1, 0] \\ 1 - t_j(x), & t_j(x) \in (0, 1] \\ 0, & \text{otherwise.} \end{cases}$$



with $t_j(x) = (x - x_j)/h$. Note that $\{\varphi_j(x)\}$ forms a *partition of unity*, i.e., $\sum_j \varphi_j(x) = 1$ for all x. Then $v_h \in V_h$ is defined by

$$v_h(x) = \sum_{j=0}^{m+1} v_j \varphi_j(x)$$

with $v_j = v(x_j)$ and $v \in V$. The finite element solution is given by

$$u_h x) = \sum_{i=0}^{m} \xi_i \varphi_i(x)$$

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 ξ_i such that

$$\left\langle \sum_{0=1}^{m} \xi_i \varphi'(x), \sum_{j=0}^{m} v_j \varphi_j'(x) \right\rangle = \left\langle f, \sum_{j=1}^{m} v_j \varphi_j(x) \right\rangle.$$

By bilinearity of the inner product, this is the same as

$$\sum_{j=1}^{m} v_j \left\langle \sum_{i=1}^{m} \xi_i \varphi_i'(x), \varphi_j'(x) \right\rangle = \sum_{j=1}^{m} v_j \langle f, \varphi_j(x) \rangle.$$

In order that this equality holds for all v_j it must follow that

$$\left\langle \sum_{i=1}^{m} \xi_i \varphi_i'(x), \varphi_j'(x) \right\rangle = \langle f, \varphi_j(x) \rangle \quad \text{for } j = 1, \ldots, m.$$

Again, by bilinearity

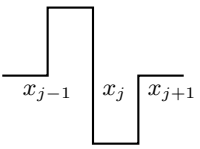
$$\sum_{i=1}^{m} \left\langle \varphi_i'(x), \varphi_j'(x) \right\rangle \xi_i = \langle f, \varphi_j(x) \rangle \quad \text{for } j = 1, \ldots, m.$$

So, we have the linear system of equation

$$\mathbf{A}\xi = \mathbf{b}$$

where $\mathbf{A}$ is a matrix with elements $a_{ij} = \left\langle \varphi_i', \varphi_j' \right\rangle$, $\mathbf{b} = (f_1, \ldots, f_m)^T$ where $f_j = \langle f, \varphi_j \rangle$, and $\xi = (\xi_1, \ldots, \xi_m)^T$. $\mathbf{A}$ is called the *stiffness matrix* and $\mathbf{b}$ is called the *load vector*. Computing the elements of $\mathbf{A}$:

$$a_{ij} = \left\langle \varphi_j', \varphi_i' \right\rangle = \begin{cases} \int_0^1 (\varphi_i')^2 \, \mathrm{d}x = \dfrac{2}{h}, & i = j \\ \int_0^1 \varphi_i' \varphi_j' \, \mathrm{d}x = -\dfrac{1}{h}, & i = j \pm 1 \\ 0, & \text{otherwise.} \end{cases}$$



This is just the finite-difference method:

$$-\frac{\xi_{j+1} - 2\xi_j + \xi_{j+1}}{h^2} = f_j.$$

with a stiffness matrix

$$\mathbf{A} = \frac{1}{h^2}\begin{bmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}.$$

15.2 A two-dimensional example

Let's extend what we developed in the previous section to the two-dimensional Poisson equation

$$-\Delta u = f \quad \text{where} \quad u\big|_{\partial\Omega} = 0 \tag{15.1}$$

for a region $\Omega \in \mathbb{R}^2$. Before we write the Poisson equation in variational form, we'll need some more theory and an additional identity.

A mapping $b : V \to \mathbb{R}$ is a *linear functional* if

$$b(\alpha u + \beta v) = \alpha b(u) + \beta b(v)$$

for all real numbers α and β and $u, v \in V$. The mapping $a : V \times V \to \mathbb{R}$ is a *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 α and β and $u, v, w \in V$.

The space of L^2-functions (square-integrable functions) is

$$L^2(\Omega) = \Big\{ v \;\Big|\; \int_\Omega |v|^2 \,\mathrm{d}x < \infty \Big\}.$$

In this case, we define the L^2-inner product as $\langle v,\, w\rangle_{L^2} = \int_\Omega vw \,\mathrm{d}x$ 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 kth derivatives are also L^2-functions. We define the Sobolev inner product as

$$\langle v,\, w\rangle_{H^1} = \int_\Omega vw + \nabla v \cdot \nabla w \,\mathrm{d}x$$

and the Sobolev norm as

$$\|v\|_{H^1} = \sqrt{\int_\Omega v^2 + |\nabla v|^2 \,\mathrm{d}x}.$$

Finally, define $H_0^1(\Omega)$ to be the set of all Sobolev functions which vanish on the the boundary of Ω. The Sobolev norms are extension the L^2-norms and give a measure of the smoothness (or regularity) of a function.

Theorem 46 (Green's formula)**.**

$$\iint_\Omega v\Delta u \,\mathrm{d}V = \int_{\partial\Omega} v\frac{\partial u}{\partial n}\,\mathrm{d}A - \iint_\Omega \nabla v \cdot \nabla u \,\mathrm{d}V.$$

where $\frac{\partial u}{\partial n} = \nabla u \cdot \hat{\mathbf{n}}$ *is a directional derivative with unit outer normal* $\hat{\mathbf{n}}$*.*

Proof. Note that

$$\sum_i \partial_j(v\partial_i u) = \sum_i (\partial_j v)(\partial_i u) + \sum_i v\partial_{ii}u.$$

In other words,

$$\nabla \cdot (v\nabla u) = \nabla v \cdot \nabla u + v\Delta u.$$

Integrating both sides over Ω gives us

$$\int_\Omega \nabla \cdot (v\nabla u)\,\mathrm{d}V = \int_\Omega \nabla v \cdot \nabla u + v\Delta u \,\mathrm{d}V.$$

From this it follows that

$$\begin{aligned}\int_\Omega v\Delta u \,\mathrm{d}V &= \int_\Omega \nabla \cdot (v\nabla u)\,\mathrm{d}V - \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}V \\ &= \int_{\partial\Omega} (v\nabla u)\cdot \hat{\mathbf{n}}\,\mathrm{d}A - \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}V \\ &= \int_{\partial\Omega} v\frac{\partial u}{\partial n}\,\mathrm{d}A - \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}V.\end{aligned}$$

□

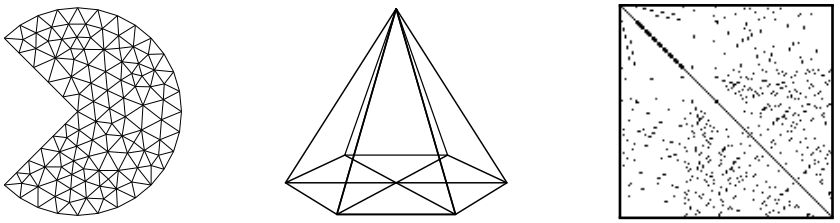


Figure 15.1: Triangulation of a domain Ω (left) using a simple pyramidal basis functions (middle) and the resultant sparse stiffness matrix (right).

Now, let's get back to the Poisson equation. Let $V = H_0^1(\Omega)$, the space of Sobolev functions over Ω that vanish on the boundary $\partial\Omega$. Then from (15.1), for all $v \in V$

$$-\iint_\Omega v\Delta u \,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega fv \,\mathrm{d}x\,\mathrm{d}y. \tag{15.2}$$

Using Green's formula and noting that v vanishes on $\partial\Omega$, we have

$$\iint_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega fv \,\mathrm{d}x\,\mathrm{d}y.$$

Define the bilinear functional $a(\cdot,\,\cdot)$ as

$$a(u,\,v) = \iint_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x\,\mathrm{d}y$$

and the linear functional $b(\cdot)$

$$b(v) = \iint_\Omega vf \,\mathrm{d}x\,\mathrm{d}y.$$

Then the variation formulation is

Ⓥ Find $u \in V$ such that $a(u,\,v) = b(v)$ for all $v \in V$.

Let's now look at the implementation using FEM. Consider some triangulation T_h of Ω. 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. The finite element space $V_h \subset H_0^1(T_h)$ is the space of functions that are linear over each element K_j. For all $v_h \in V_h$ we can express

$$v_h(x,y) = \sum_{i=1}^{m} v(n_i)\varphi_i(x,y)$$

where $\{\varphi_i\}$ is a set of pyramidal basis functions with $\varphi_i(n_j) = \delta_{ij}$. The finite element formulation is

❺ Find $u_h \in V_h$ such that $a(u_h,\, v) = b(v)$ 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 $j = 1, \ldots, m$

$$a\left(\sum_{i=1}^{m} \xi_i \varphi_i(x),\, \varphi_j\right) = b(\varphi_j)$$

from which it follows by linearity that

$$\sum_{i=1}^{m} \xi_i a(\varphi_i,\, \varphi_j) = b(\varphi_j)$$

which we can write as $\mathbf{A}\xi = \mathbf{b}$, where elements of the stiffness matrix are

$$a_{ij} = a(\varphi_i,\, \varphi_j) = \int_{T_h} \nabla\varphi_i \cdot \nabla\varphi_j \,\mathrm{d}x\,\mathrm{d}y$$

and elements of the load vector are

$$b_j = b(\varphi_j) = \int_{T_h} \varphi_j(x) f(x) \,\mathrm{d}x\,\mathrm{d}y.$$

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 a FEM problem:

$$-\Delta u + u = f \quad \text{with} \quad \left.\frac{\partial u}{\partial n}\right|_{\partial\Omega} = g.$$

Let $V = H^1(\Omega)$ and take $v \in V$. This time v is allowed to be non-zero on the boundary $\partial\Omega$. Then

$$\int_\Omega -v\Delta u + vu \,\mathrm{d}x\,\mathrm{d}y = \int_\Omega fv \,\mathrm{d}x\,\mathrm{d}y.$$

Using Green's identity, we have that

$$\int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x\,\mathrm{d}y - \int_{\partial\Omega} v\frac{\partial u}{\partial n} \,\mathrm{d}s + \int_\Omega vu \,\mathrm{d}x\,\mathrm{d}y = \int_\Omega fv \,\mathrm{d}x\,\mathrm{d}y.$$

On the boundary we have that $\frac{\partial u}{\partial n} = g$, so

$$\int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x\,\mathrm{d}y + \int_\Omega vu \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x\,\mathrm{d}y + \int_{\partial\Omega} gv \,\mathrm{d}s.$$

Let

$$a(u,\, v) = \int_\Omega \nabla u \cdot \nabla v + uv \,\mathrm{d}x\,\mathrm{d}y$$

and let

$$b(v) = \langle f,\, v\rangle + \langle\!\langle g,\, v\rangle\!\rangle \equiv \int_\Omega fv \,\mathrm{d}x\,\mathrm{d}y + \int_{\partial\Omega} gv \,\mathrm{d}s.$$

The variational form of the Neumann problem is

Ⓥ Find $u \in V$ such that $a(u,\, v) = b(v)$ for all $v \in V$.

Let V_h be the space of continuous, piecewise-linear functions on Ω with a triangulation T_h, and let $\{\varphi\}$ be the set of basis functions. Recasting the problem as a finite element approximation

❻ Find $u_h \in V_h$ such that $a(u_h,\, v) = b(v)$ for all $v \in V_h$ where

$$u_h(x, y) = \sum_{i=1}^{m} \xi_i \varphi_i(x, y).$$

Then for $j = 1 \ldots, m$

$$\sum_{i=1}^{m} \xi_i a(\varphi_i,\, \varphi_j) = b(\varphi_j)$$

So, $\mathbf{A}\xi = \mathbf{b}$ where

$$a_{ij} = a(\varphi_i,\, \varphi_j) = \int_{T_h} \nabla\varphi_i \cdot \nabla\varphi_j + \varphi_i\varphi_j \,\mathrm{d}x$$

and

$$b_j = \int_{T_h} f\varphi_j \,\mathrm{d}x + \int_{\partial T_h} g\varphi_j \,\mathrm{d}s.$$

Note that this time our stiffness matrix $\mathbf{A}$ and load vector $\mathbf{b}$ includes the boundary elements. For a one-dimensional problem with $x \in [0, 1]$, elements a_{ij} and b_j are simply

$$a_{ij} = \int_0^1 \varphi_i'\varphi_j' + \varphi_i\varphi_j \,\mathrm{d}x$$

and

$$b_j = \int_\Omega f(x)\varphi_j(x) \,\mathrm{d}x + \begin{cases} g(0), & \text{if } j = 0 \\ g(1), & \text{if } j = m \\ 0, & \text{otherwise} \end{cases}. \qquad \blacktriangleleft$$

15.3 Stability and convergence

This section discusses when a finite element method works and how well it works. To do this, we 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* iff there exists $\gamma > 0$ such that $|a(u, v)| \leq \gamma \|u\|_V \|v\|_V$ for all $u, v \in V$. "Continuous" is also called "*bounded*." Similarly, a linear function b is continuous if there exists $\Lambda > 0$ such that $|b(u)| \leq \Lambda \|u\|_V$ for all $u \in V$. A blinear form a is *coercive* if there exists $\alpha > 0$ such that $|a(u, u)| \geq \alpha \|u\|_V^2$ for all $u \in V$. "Coercive" is often called "*V-elliptic*."

Theorem 47 (Lax–Milgram Lemma)**.** *For the Hilbert space V, the variational problem*

Ⓥ *Find $u \in V$ such that $a(u, v) = b(v)$ for all $v \in V$.*

is well-posed, meaning that the solution exists, is unique, and is stable, if a is symmetric, a is coercive, and a and b are continuous.

Proof. Suppose that there are two solutions u_1 and u_2. So, $a(u_1, v) = b(v)$ and $a(u_2, v) = b(v)$ for all $v \in V$. Then $|a(u_1 - u_2, v)| = 0$. By V-ellipticity

$$\alpha \|u_1 - u_2\|_V \leq a(u_1 - u_2, u_1 - u_2) = 0$$

So $u_1 = u_2$. Also,

$$\alpha \|u\|_V^2 \leq a(u, u) = b(u) \leq \Lambda \|u\|_V$$

So, $\|u\|_V \leq \Lambda/\alpha$. So, the solution is stable. □

A couple of inequalities will be useful. So, let's look at them now.

Theorem 48 (Cauchy–Schwarz Inequality)**.** $|\langle v, w\rangle| \leq \|v\|\|w\|$

Proof. Note that for any α, $0 \leq \|v - \alpha w\|^2 = \|v\|^2 - 2\alpha \langle v, w\rangle + |\alpha|^2 \|w\|$. By taking $\alpha = |\langle v, w\rangle|/\|w\|^2$, we have that $0 \leq \|v\|^2 \|w\|^2 - |\langle v, w\rangle|^2$. □

Theorem 49 (Poincaré Inequality)**.** *For $u \in H_0^1$ and Ω bounded, then*

$$\int_\Omega u^2 \, \mathrm{d}x \leq \beta \int_\Omega (u')^2 \, \mathrm{d}x \quad \textit{for some } \beta.$$

Proof. We'll prove an easier one-dimensional case. Without loss of generality, take $u(0) = 0$ and $x \in [0, 1]$. By the Cauchy–Schwarz inequality

$$u(x) = \int_0^x u'(x)\,\mathrm{d}x \le \sqrt{\int_0^x 1\,\mathrm{d}x}\sqrt{\int_0^x (u')^2\,\mathrm{d}x} \le \sqrt{\int_0^1 (u')^2\,\mathrm{d}x}.$$

Squaring and integrating both sides give us

$$\int_0^1 u^2(x)\,\mathrm{d}x \le \int_0^1 \left(\int_0^1 (u')^2\,\mathrm{d}x\right)\,\mathrm{d}x = \int_0^1 (u')^2\,\mathrm{d}x. \qquad \square$$

Example. Show that the FEM formulation of the Poisson equation is well-posed (unique and stable): $-u'' = f$ with $u(0) = u(1) = 0$.

Define the finite element space as $V = H_0^1(\Omega)$.

$$\int_0^1 u'v'\,\mathrm{d}x = \int_0^1 fv\,\mathrm{d}x$$

Let

$$a(u,\, v) = \int_0^1 u'v'\,\mathrm{d}x \quad \text{and} \quad b(v) = \int_0^1 fv\,\mathrm{d}x.$$

Ⓥ Find $u \in H_0^1(\Omega)$ such that $a(u,\, v) = b(v)$ for all $v \in H_0^1(\Omega)$.

Let

$$\|u\|_{H_0^1} = \sqrt{\int_\Omega u^2 + |u'|^2\,\mathrm{d}x}.$$

We need to check that the assumptions hold. Then the results follow by the Lax–Milgram Lemma.

1. Show that a is continuous. By the Cauchy–Schwarz inequality

$$a(u,\, v) = \left|\int_0^1 u'v'\,\mathrm{d}x\right| \le \sqrt{\int_0^1 |u'|^2\,\mathrm{d}x}\sqrt{\int_0^1 |v'|^2\,\mathrm{d}x} \le \|u\|_{H^1}\|v\|_{H^1}$$

2. Show that a is coercive. Using the Poincaré inequality $a(u,\, u) = \int_0^1 (u')^2\,\mathrm{d}x \ge \alpha \int_0^1 u^2 + (u')^2\,\mathrm{d}x$. Now, the assumption holds.

3. Show that b is continuous

$$|b(v)\,| = \left|\int_0^1 fv\,\mathrm{d}x\right| \le \|f\|_{L^2}\|v\|_{L^2} \le \|f\|_{L^2}\|v\|_V \qquad \blacktriangleleft$$

Theorem 50 (Céa's Lemma). *If a is bounded and coercive, then*

$$\|u - u_h\|_V \le (\gamma/\alpha) \min_{w_h \in V} \|u - w_h\|_V.$$

Proof. For $w_h \in V_h \subset V$,

$$\begin{aligned} a(u_h, w_h) &= \langle f, w_h \rangle \quad \text{for all } w_h \in V_h \quad \text{and} \\ a(u, w_h) &= \langle f, w_h \rangle \quad \text{for all } w_h \in V. \end{aligned}$$

Therefore, $a(u - u_h, w_h) = 0$ for all $w_h \in V$. This says, that u_h is a projection of u in the inner-product $a(\cdot, \cdot)$. Let $e = u - u_h$. Then,

$$\begin{aligned} \alpha\|e\|_V^2 &\le a(e, e) && \text{(because } a \text{ is coercive)} \\ &= a(e, u - w_h) + a(e, w_h - u_h) && \text{for all } w_h \in V_h \\ &= a(e, u - w_h) \quad \text{for all } w_h \in V_h && \text{(because } w_h - u_h \in V_h) \\ &\le \gamma\|e\|_V \|u - w_h\|_V && \text{(because } a \text{ is bounded)} \\ \|e\|_V &\le (\gamma/\alpha)\|u - w_h\|_V && \text{for all } w_h \in V_h \end{aligned}$$

It follows that $\|u - u_h\|_V \le (\gamma/\alpha) \min_{w_h \in V} \|u - w_h\|_V$. □

Céa's Lemma says that the finite element solution u_h is the best solution up to the constant γ/α. One can use Céa's Lemma to estimate the Galerkin approximation error. Let $\Pi_h^1 u$ a piecewise linear polynomial interpolant of the solution $u \in H^2$. The

$$\begin{aligned} \|u - u_h\|_{H_0^1} &\le (\gamma/\alpha) \min_{w_h \in V_h} \|u - w_h\|_{H_0^1} \\ &\le (\gamma/\alpha)\|u - \Pi_h^1 u\|_{H_0^1} \\ &\le (\gamma/\alpha)Ch\|u''\|_L^2 \\ &\le (\gamma/\alpha)Ch\|u\|_{H_0^2} \end{aligned}$$

For the proof of the third inequality, see Quarteroni et al. [2007]. For k-order polynomial basis functions, the H_0^1-error is $O(h^k)$ if $u \in H^{k+1}$.

15.4 Time-dependent problems

The finite element method is often used to solve time-dependent partial differential equations. Consider the one-dimensional heat equation with a source term $f(x)$

$$\begin{cases} u_t - u_{xx} = f(x) \\ u(0,t) = u(1,t) = 0 \\ u(x,0) = u_0(x) \end{cases} \tag{15.3}$$

Let $V = H_0^1([0,1])$. Then for all $v \in V$,

$$\int_0^1 v u_t \,\mathrm{d}x - \int_0^1 v u_{xx} \,\mathrm{d}x = \int_0^1 v f \,\mathrm{d}x$$

from which we have that

$$\int_0^1 v u_t \,\mathrm{d}x + \int_0^1 v_x u_x \,\mathrm{d}x = \int_0^1 v f \,\mathrm{d}x$$

The Galerkin formulation is

Ⓥ 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 $\{\varphi_0, \ldots, \varphi_n\}$ be a basis of V_h. Let

$$u_h(x,t) = \sum_{i=0}^{n} \xi_i(t)\varphi_i(x)$$

and take $v = \varphi_j$. Then we have the system

$$\sum_{i=0}^{n} \xi_i(t) \int_0^1 \varphi_i \varphi_j \,\mathrm{d}x + \sum_{i=0}^{n} \xi_i(t) \int_0^1 \varphi_i' \varphi_j' \,\mathrm{d}x = \int_0^1 f \varphi_j \,\mathrm{d}x$$

for $j = 0, \ldots, n$. More concisely, we have the system of ODEs

$$\mathbf{A}\xi'(t) + \mathbf{B}\xi(t) = \mathbf{c} \tag{15.4}$$

where the elements of $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{c}$ are

$$a_{ij} = \int_0^1 \varphi_i \varphi_j \,\mathrm{d}x, \quad b_{ij} = \int_0^1 \varphi_i' \varphi_j' \,\mathrm{d}x \quad \text{and} \quad c = \int_0^1 f \varphi_j \,\mathrm{d}x$$

We can rewrite (15.4) as

$$\xi'(t) + \mathbf{A}^{-1}\mathbf{B}\xi(t) = \mathbf{A}^{-1}\mathbf{c}$$

which we can solve using an ODE method such as the backward Euler

$$\frac{\xi^{n+1} - \xi^n}{k} + \mathbf{A}^{-1}\mathbf{B}\xi^{n+1} = \mathbf{A}^{-1}\mathbf{c}$$

or the Crank–Nicolson

$$\frac{\xi^{n+1} - \xi^n}{k} + \tfrac{1}{2}\mathbf{A}^{-1}\mathbf{B}\xi^{n+1} + \tfrac{1}{2}\mathbf{A}^{-1}\mathbf{B}\xi^{n} = \mathbf{A}^{-1}\mathbf{c}.$$

Numerical stability can be determined by confirming that the energy decreases over time. Consider the heat equation (15.3) with zero source term $u_t = u_{xx}$. Multiplying both sides by u gives us $uu_t = uu_{xx}$ from which it follows that

$$\tfrac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\int_0^1 u^2\,\mathrm{d}x = -\int_0^1 (u_x)^2\,\mathrm{d}x \le 0$$

So, $\|u(\cdot,t)\|_{L^2} \le \|u(\cdot,0)\|_{L^2}$.

Example. Show that the FEM backward-Euler scheme for the heat equation is unconditionally stable. The FEM backward-Euler scheme is

$$\left\langle \frac{u_h^{n+1}-u_h^n}{k},\, v\right\rangle + a\big(u_h^{n+1},\, v\big) = 0 \quad \text{for all } v \in V_h$$

where the energy norm $a(u,\,v) = \int_0^1 u'v'\,\mathrm{d}x$. Take $v = u_h^{n+1}$

$$\big\langle u_h^{n+1},\, u_h^{n+1}\big\rangle - \big\langle u_h^n,\, u_h^{n+1}\big\rangle = -ka\big(u_h^{n+1},\, u_h^{n+1}\big) \le 0.$$

So,

$$\|u_h^{n+1}\|_{L^2}^2 \le \big\langle u_h^n,\, u_h^{n+1}\big\rangle \le \|u_h^n\|_{L^2}\|u_h^{n+1}\|_{L^2}$$

by the Cauchy–Schwarz inequality. Therefore, $\|u_h^{n+1}\|_{L^2} \le \|u_h^n\|_{L^2}$ for all n and all stepsizes k. Hence, the scheme is unconditionally stable. ◀

Exercises

✍ 15.1. Solve numerically the differential equation by the finite element method using piecewise linear elements:

$$-u_{xx} - u + 8x^2 = 0 \quad \text{for} \quad 0 < x < 1$$

for the following sets of boundary conditions:

1. Dirichlet $\quad u(0) = u(1) = 0$
2. Neumann $\quad u_x(0) = 0, \quad u_x(1) = 1$

✍ 15.2. Consider the boundary value problem

$$\begin{aligned} &\frac{\mathrm{d}^4 u}{\mathrm{d}x^4} = f, \qquad 0 < x < 1 \\ &u(0) = u'(0) = u(1) = u'(1) = 0. \end{aligned} \tag{15.5}$$

for the deflection of a uniform beam under the load $f(x)$.

1. Show that the problem (15.5) can be given the following variational formulation:

$$\text{Find } u \in V \text{ such that } a(u,\, v) = b(f) \qquad \text{for all} \quad v \in V$$

where $V = \{\, v \,:\, v$ and v' are continuous on [0,1], v'' is piecewise continuous, and $v(0) = v'(0) = v(1) = v'(1) = 0\,\}$.

2. Show that the variational formulation of the problem (15.5) is well-posed. You will need to take $u \in H_0^2([0,1])$.

3. Construct a finite-dimensional subspace V_h of $H_0^2(0,1)$. Formulate a finite element method based on the space V_h. Find the corresponding linear system of equations in the case of a uniform partition. Determine the solution for $f(x) = 384$. Compare it with the exact solution. Note: You will need to choose your solution in $H_0^2([0,1])$.

$$u(x) = \sum_{i=0}^{m+1} \xi_j \phi_j(x) + \eta_j \psi_j(x)$$

where the cubic Hermite elements are given by

$$\phi_j(t) = \begin{cases} -2t^3 - 3t^2 + 1, & t \in [-1,0] \\ +2t^3 - 3t^2 + 1, & t \in [0,1] \\ 0, & \text{otherwise} \end{cases}$$

and

$$\psi_j(t) = \begin{cases} +t^3 + 2t^2 + t, & t \in [-1,0], \\ +t^3 - 2t^2 + t, & t \in [0,1], \\ 0, & \text{otherwise} \end{cases}$$

with $t = (x - x_j)/h$.

15.3. Show that the Crank–Nicolson FEM scheme for the heat equation $u_t = u_{xx}$ is unconditionally stable.

CHAPTER 16

Fourier Spectral Methods

To improve the convergence of a finite-difference method we use more grid points in order to achieve a better approximation of the derivatives. With each additional gridpoint, we were able to increase the order of convergence by one. One might surmise that we could achieve the best accuracy by using all the gridpoints available. This is precisely what spectral methods do. In this chapter, we examine an important class of spectral methods—the Fourier spectral method—which uses trigonometric interpolation to approximate a solution. Rather than achieving mth order accuracy using an m-point finite-difference approximation, Fourier spectral methods achieve *spectral accuracy*—$O(h^m)$ for all $m \geq 1$—if the solution is smooth. If only the first p derivatives of the function exist, we have at most $O(h^{p+1})$.

Since we use trigonometric interpolation, we require that the solution be periodic. If we are not interested in modeling boundary interaction, this restriction is often acceptable. For non-periodic problems, absorbing "sponge" layers and reflecting boundaries can sometimes be added. One may also use other spectral method as the Chebyshev spectral method, which uses Chebyshev polynomials for non-periodic solutions.

16.1 Discrete Fourier transform

Trigonometric interpolation

Let the function $u(x)$ be periodic with period 2π, i.e, $u(x + 2\pi) = u(x)$. We can later modify the period of $u(x)$ by rescaling x. Consider a uniform grid $x_j = j\pi/n$ with $j = 0, \ldots, 2n - 1$. We can approximate $u(x)$ by a

Fourier polynomial with nodes at x_j

$$p(x) = \sum_{\xi=-n}^{n-1} c_\xi \mathrm{e}^{+\mathrm{i}\xi x} \quad \text{by choosing } c_\xi \text{ so that} \quad u(x_j) = \sum_{\xi=-n}^{n-1} c_\xi \mathrm{e}^{-\mathrm{i}\xi x_j}.$$

To find x_ξ, we'll need the help of an identity.

Theorem 51. *For all integers ξ*

$$\sum_{j=0}^{2n-1} \mathrm{e}^{\mathrm{i}\xi x_j} = \begin{cases} 2n, & \textit{if } \xi = 0, \\ 0, & \textit{otherwise.} \end{cases}$$

Proof.

$$\sum_{j=0}^{2n-1} \mathrm{e}^{\mathrm{i}\xi x_j} = \sum_{j=0}^{2n-1} \mathrm{e}^{\mathrm{i}\xi j\pi/n} = \sum_{j=0}^{2n-1} \omega^{\xi j} = \begin{cases} \dfrac{\omega^{2n}-1}{\omega-1}, & \text{if } \xi \neq 1 \\ 2n, & \text{if } \xi = 1. \end{cases}$$

where $\omega = \mathrm{e}^{\mathrm{i}\pi/n}$. Note that

$$\omega^{2n} = \left(\mathrm{e}^{\mathrm{i}\xi\pi/n}\right)^{2n} = \mathrm{e}^{\mathrm{i}\xi 2\pi} = 1. \qquad \square$$

Then

$$\sum_{j=0}^{2n-1} u(x_j)\mathrm{e}^{-\mathrm{i}\xi x_j} = \sum_{j=0}^{2n-1} \sum_{l=-n}^{n-1} c_\xi \mathrm{e}^{\mathrm{i}(l-\xi)x_j} = \sum_{l=-n}^{n-1} c_\xi \sum_{j=0}^{2n} \mathrm{e}^{\mathrm{i}(l-\xi)x_j} = 2nc_\xi.$$

So, we have that

Discrete Fourier Transform	$c_\xi = \dfrac{1}{2n}\displaystyle\sum_{j=0}^{2n-1} u(x_j)\mathrm{e}^{-\mathrm{i}\xi x_j}$	$\xi = -n\ldots, n-1$
Discrete Inverse Fourier Transform	$u(x_j) = \displaystyle\sum_{j=-n}^{n-1} c_\xi \mathrm{e}^{\mathrm{i}\xi x_j}$	$j = 0, \ldots, 2n-1$

We can think of the discrete Fourier transform as a Riemann sum and the Fourier transform as a Riemann integral:

Fourier Transform	$\hat{u}(\xi) = \dfrac{1}{2\pi}\displaystyle\int_{-\infty}^{\infty} u(x)\mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x$
Inverse Fourier Transform	$u(x) = \displaystyle\int_{-\infty}^{\infty} \hat{u}(\xi)\mathrm{e}^{\mathrm{i}\xi x}\,\mathrm{d}\xi$

This gives us the Fourier series

The 2π-periodization of the function $u(x)$ is the function $u_p(x) = \sum_{m=-\infty}^{\infty} u(x+2\pi m)$. Then $u_p(x) = u_p(x+2\pi m)$, if the series converges. 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

$$u_p(x) = \sum_{\xi=-\infty}^{\infty} c_\xi \mathrm{e}^{\mathrm{i}\xi x} \tag{16.1}$$

with

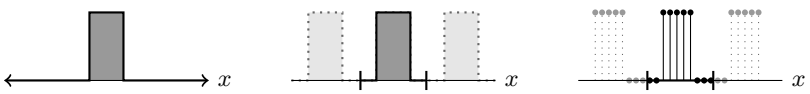
$$c_\xi = \frac{1}{2\pi}\int_0^{2\pi} u_p(x)\mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x = \frac{1}{2\pi}\int_{-\infty}^{\infty} u(x)\mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x$$

which is simply the Fourier transform $\hat{u}(\xi)$.This says that

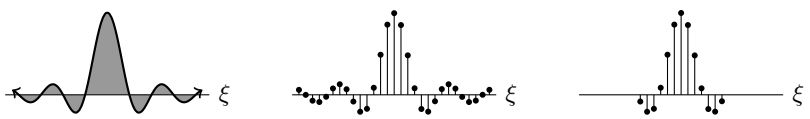
$$\sum_{m=-\infty}^{\infty} u(x+2\pi m) = \sum_{m=-\infty}^{\infty} \hat{u}(m)\mathrm{e}^{\mathrm{i}mx},$$

which is known as the Poisson summation formula. If $u(x)$ vanishes outside of $[0, 2\pi)$, then $u(x) = \sum_{m=-\infty}^{\infty} \hat{u}(m)\exp(\mathrm{i}mx)$. Furthermore, by truncating the Fourier series (16.1) $\sum_{m=-\infty}^{\infty} \to \sum_{m=-n}^{n-1}$, i.e., restricting the frequency domain, we have the discrete Fourier transform.

We can summarize the Fourier transform, the Fourier series, and the discrete Fourier transform by comparing them in the spatial domain x and the frequency domain ξ.



In the Fourier transform the spatial domain x extends over the entire real line $(-\infty, \infty)$. In the Fourier series, the spatial domain is periodic. In the discrete Fourier transform, the spatial domain is both periodic and discrete.



In the Fourier transform the frequency domain ξ also extends over the entire real line $(-\infty, \infty)$. In the Fourier series, the frequency domain also extends over the real line $(-\infty, \infty)$ but is discrete. In the discrete Fourier transform, the frequency domain is both discrete and bounded.

Spectral differentiation

Let's find the derivative of the Fourier polynomial approximation of $u(x)$. As before, we take $u(x)$ with period 2π. In this case,

$$\frac{\mathrm{d}}{\mathrm{d}x}u(x) = \sum_{\xi=-n}^{n-1} c_\xi \frac{\mathrm{d}}{\mathrm{d}x}\mathrm{e}^{\mathrm{i}\xi x} = \sum_{\xi=-n}^{n-1} (\mathrm{i}\xi c_\xi)\mathrm{e}^{\mathrm{i}\xi x.}$$

That is to say, the discrete Fourier transform of $\frac{d}{dx}u$ is $\mathrm{i}\xi\hat{u}$. When $u(x)$ has period L, we make the transform $x \mapsto (2\pi/L)x$ and $dx \mapsto (2\pi/L)dx$. So, the discrete Fourier transform of $\frac{d}{dx}u$ is $\mathrm{i}(2\pi/L)\xi\hat{u}$ for period L. An analogous statement for the Fourier transform holds by integration by parts. Of course, as long is $u(x)$ is smooth, we can differentiate any number of times. So, the Fourier transform

$$\mathrm{F}\left[\frac{\mathrm{d}^p}{\mathrm{d}x^p}u\right] \text{ is } \left(\mathrm{i}\xi\frac{2\pi}{L}\right)^p \mathrm{F}[u].$$

Example. Consider the heat equation $u_t = u_{xx}$ with initial conditions $u(0,x)$. The Fourier transform of the heat equation $\hat{u}_t = -\xi^2\hat{u}$ has the solution $\hat{u} = e^{-\xi^2 t}\hat{u}(0,\xi)$. So, $u(t,x) = \mathrm{F}^{-1}\left[e^{-\xi^2 t}\,\mathrm{F}\,u(0,x)\right]$. ◀

Smoothness and spectral accuracy

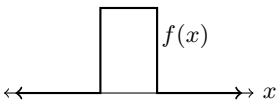
Smooth functions have rapidly decaying Fourier transforms. The Fourier transform of the smoothest functions like the constant function or sine function are Dirac-delta distributions. And Gaussian functions $\exp(-x^2)$ are transformed to Gaussian functions.

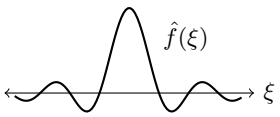
The rectangle function

$$f(x) = \begin{cases} 1, & x \in [-\frac{1}{2}, \frac{1}{2}] \\ 0, & \text{otherwise} \end{cases}$$

has the Fourier transform

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x = \int_{-\frac{1}{2}}^{\frac{1}{2}} \mathrm{e}^{-\mathrm{i}\xi x}\,\mathrm{d}x = \frac{\mathrm{e}^{\mathrm{i}\xi} - \mathrm{e}^{-\mathrm{i}\xi}}{\mathrm{i}\xi} = \frac{\sin\xi/2}{\xi/2} = \operatorname{sinc}\frac{\xi}{2}.$$





The rectangle function is not differentiable at $x = \pm\frac{1}{2}$. In fact, it is not even continuous at those points. Its Fourier transform decays as $O(|\xi|^{-1})$ as $|\xi| \to \infty$. The rectangle function can thought of as a piecewise constant B-spline $B_{\cdot,0}(x)$. We can use a linear combination of scaled and translated cardinal basis functions $B^0(x)$ to build piecewise constant functions, the Fourier transform of which are 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 bases $B^1(x)$. One way to do this is to take the self-convolution of $B^0(x)$. Another way is to take the antiderivative

$$g(x) = \int_{-\infty}^{x} f(x-1/2) - f(x+1/2)\,\mathrm{d}x = \begin{cases} 1+x, & x \in [-1,0] \\ 1-x, & x \in [0,1] \\ 0, & \text{otherwise} \end{cases}$$

Its Fourier transform is

$$\hat{g}(\xi) = \operatorname{sinc}^2 \frac{\xi}{2} = \frac{\sin^2 \xi/2}{(\xi/2)^2}.$$

The hat function is not differentiable at $x = -1, 0, 1$ but it is weakly differentiable everywhere. We can use a linear combination of scaled and translated cardinal basis functions $B^1(x)$ to build piecewise linear functions, whose Fourier transforms happen to be linear combination of sinc-squared functions that decay as $O(|\xi|^{-2})$ as $|\xi| \to \infty$.

In general, a piecewise polynomial function of degree p can be constructed using a B-spline of order p. The Fourier transform of a cardinal B-spline with unit-separated knots is given by $\hat{B^p}(x) = \operatorname{sinc}^{p+1}(\xi/2)$. With knots separated by distance h, the Fourier transform is given by $h \operatorname{sinc}^{p+1}(\xi/2h)$.

In practice, we can't take all frequencies and must be satisfied with a Fourier polynomial $\mathrm{P}_n f$ as our approximation to the Fourier series f:

$$f(x) = \sum_{\xi=-\infty}^{\infty} c_\xi \mathrm{e}^{-\mathrm{i}\xi x} = \sum_{\xi=-n}^{n-1} c_\xi \mathrm{e}^{-\mathrm{i}\xi x} + \tau_n = \mathrm{P}_n f(x) + \tau_n$$

Because the Fourier series is a complete, orthonormal system, we can use Parseval's Theorem on page 226 to compute the l^2-error of the truncation

error

$$\|\tau_n\|^2 = \|f - \mathrm{P}_n f\|^2 = \|\mathrm{P}_\infty f - \mathrm{P}_n f\|^2 = \sum_{\xi > |n|} |c_\xi|^2.$$

Let f be approximated as a B-spline with knots separated by distance h. The Fourier transform is given by linear combinations of $h\,\mathrm{sinc}^{p+1}(\xi/2h)$. Because we can approximate f using a linear combination of B-splines and the Fourier transform of $B_0^p(x)$ is $\mathrm{sinc}^p\,\xi$ it follows that if f has $p-1$ continuous derivatives in $L^2(\Omega)$ for some $p \geq 0$ and a pth derivative of bounded variation, then it follows that $c_\xi = O(\mathrm{sinc}^{p+1}\,\xi) = O(|\xi|^{-(p+1)})$. Furthermore,

$$\|\tau_n\|^2 = \sum_{\xi > |n|} |c_\xi|^2 = \sum_{\xi > |n|} O(|\xi|^{-2p-2}) = O(n^{-2p-1}).$$

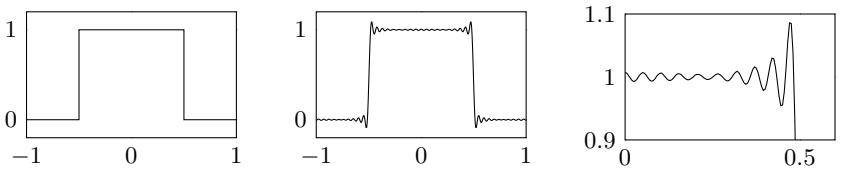
So, $\|\tau_n\| = O(n^{-p-1/2})$.

For a uniform grid over the interval $[0, L)$ with spacing $h = L/2n$, we have that $O(n^{-p})$ or $O(h^p)$.

Example. Numerically, we can't take all frequencies and must be satisfied working with a Fourier polynomial as our approximation:

$$f(x) = \sum_{\xi - \infty}^{\infty} c_\xi \mathrm{e}^{-\mathrm{i}\xi x} = \sum_{\xi = -n}^{n} c_\xi \mathrm{e}^{-\mathrm{i}\xi x} + \tau_n = P_n(x) + \tau_n$$

where the truncation $|\tau_n| = O(n^{-(p+1)})$ where p denotes the number of times that $f(x)$ is differentiable. For discontinuous functions $|\tau_n| = O(n^{-1})$ and we get slow convergence. The trigonometry polynomial approximation results in a Gibb's phenomenon. The band-limited discrete Fourier transform of a discontinuous function results in oscillations.



◀

16.2 Nonlinear stiff PDES

Time splitting

If a PDE combines a stiff linear term with a nonlinear term, we can consider time-splitting. This allows us to integrate the linear term exactly and avoid the stability issues. Consider the PDE $u_t = \mathrm{L}\,u + \mathrm{N}(u)$, where L is a stiff linear operator and N is a nonlinear operator. Take the simple splitting

1. $v_t = \mathrm{L}\,v$ where $v(0,x) = u(0,x)$
2. $w_t = \mathrm{N}\,w$ where $w(0,x) = v(t,x)$

then $u(t,x) \approx w(t,x)$ with $O(t^2)$ splitting error.

The Fourier Transform of the equation $v_t = \mathrm{L}\,v$ is $\hat{v}_t = \hat{\mathrm{L}}\hat{v}$, which has the solution

$$\hat{v} = \mathrm{e}^{t\hat{\mathrm{L}}}\hat{v}(0,\xi) = \mathrm{e}^{t\hat{\mathrm{L}}}\hat{u}(0,\xi).$$

So, the solution $u(t,x) \approx w(t,x)$ where $w(t,x)$ solves

$$w_t = \mathrm{N}(w) \text{ where } w(t,0) = v(t,x) \text{ with } v(t,x) = \mathrm{F}^{-1}\left[\mathrm{e}^{t\hat{\mathrm{L}}}\,\mathrm{F}\,u(0,x)\right].$$

Recall from Chapter 12 that we can increase order of the method by using Strang splitting implementation.

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

$$u_t = Lu + N(u)$$

where $Lu = (\mathrm{d}^2/\mathrm{d}x^2)u$ and $N(u) = \varepsilon^{-1}u(1-u^2)$. The Fourier transform of $\mathrm{d}^2/\mathrm{d}x^2$ is $-\xi^2$ and the ODE

$$u_t = \varepsilon^{-1}u(1-u^2)$$

has the analytical solution

$$u(\Delta t, x) = \frac{u_0}{\sqrt{u_0^2 - (u_0^2-1)\mathrm{e}^{-2\varepsilon\Delta t}}}$$

where the initial conditions $u_0(x) = u(0,x)$. Note that if the ODE did not have an analytic solution, we could have solved it numerically.

For each timestep t_n we have

1. $v(x) = \mathrm{F}^{-1}\left[\mathrm{e}^{-\xi^2 \Delta t}\,\mathrm{F}\left[u(t_n, x)\right]\right]$

2. $u(t_{n+1}, x) = u_0\left[u_0^2 - (u_0^2 - 1)\mathrm{e}^{-2\varepsilon\Delta t}\right]^{-1/2}$ where $u_0 = v(x)$.

Each time step has a $O((\Delta t)^2)$ splitting error, so we have $O(\Delta t)$ error after n iterations. By using Strang splitting we can achieve $O((\Delta t)^2)$ error. ◀

Example. A electron in an external potential $V(x)$ is modeled in quantum mechanics using the Schrödinger equation

$$\mathrm{i}\varepsilon u_t = -\tfrac{1}{2}\varepsilon^2 u_{xx} + V(x)u$$

with scaled Planck constant ε. The Schrödinger equation can be rewritten as

$$u_t = \tfrac{1}{2}\mathrm{i}\varepsilon u_{xx} - \mathrm{i}\varepsilon^{-1}V(x)u.$$

We can separate the spatial operator into the kinetic and potential terms

$$\tfrac{1}{2}\mathrm{i}\varepsilon u_{xx} \qquad \text{and} \qquad -\mathrm{i}\varepsilon^{-1}V(x)u.$$

Both of these terms are linear in u and we can solve the problem using Strang splitting

$$u(x, t_{n+1}) = \mathrm{e}^{-\mathrm{i}V(x)\Delta t/2\varepsilon}\,\mathrm{F}^{-1}\,\mathrm{e}^{-\mathrm{i}\varepsilon\xi^2\Delta t/2}\,\mathrm{F}\,\mathrm{e}^{-\mathrm{i}V(x)\Delta t/2\varepsilon}u(x, t_n)$$

for spectral-order-in-space and second-order-in-time method. ◀

Integrating factors

We are unable to get better than second-order accuracy by using time-splitting. We can achieve arbitrary-order accuracy if we don't split the operators. Of course, we must use an implicit A-stable or L-stable method to avoid stability issues caused by the stiffness. This can be difficult if we have nonlinear terms, but we can use integrating factors to mollify the stiffness.

Suppose that we want to solve $u_t = \mathrm{L}\,u + \mathrm{N}(u)$ pseudo-spectrally using integrating factors, 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

$$u_t - \mathrm{L}\,u = \mathrm{N}(u).$$

Then, taking the Fourier transform, we get

$$\hat{u}_t - \hat{\mathrm{L}}\hat{u} = \mathrm{F}[\mathrm{N}(u)].$$

Multiple both sides of the equation by $\mathrm{e}^{-t\hat{\mathrm{L}}}$ and simplify to get

$$\left(\mathrm{e}^{-t\hat{\mathrm{L}}}\hat{u}\right)_t = \mathrm{e}^{-t\hat{\mathrm{L}}}\,\mathrm{F}[\mathrm{N}(u)].$$

Let $\hat{w} = \mathrm{e}^{-t\hat{\mathrm{L}}}\hat{u}$. Then the solution

$$u(t,x) = \mathrm{F}^{-1}\left[\mathrm{e}^{t\hat{\mathrm{L}}}w(t,\xi)\right]$$

where

$$\frac{\partial}{\partial t}\hat{w}(t,\xi) = \mathrm{e}^{-t\hat{\mathrm{L}}}\,\mathrm{F}\left[\mathrm{N}\left(\mathrm{F}^{-1}\left[\mathrm{e}^{t\hat{\mathrm{L}}}\hat{w}(t,\xi)\right]\right)\right] \tag{16.2}$$

with initial conditions

$$\hat{w}(0,\xi) = \mathrm{e}^{-t\hat{\mathrm{L}}}\,\mathrm{F}[u(0,x)].$$

The benefit of such integrating factors is that there is no splitting error. But, integrating factors only mollify the impact of stiffness. For really stiff problems, a better approach is to use time-splitting.

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

$$\begin{aligned}
&\text{Set} && \hat{w}(t_n,\xi) = \mathrm{F}\,u(t_n,x)\\
&\text{Solve} && \frac{\partial}{\partial t_{n+1}}\hat{w}(t,\xi) = \mathrm{e}^{-\Delta t\hat{\mathrm{L}}}\,\mathrm{F}\left[\mathrm{N}\left(\mathrm{F}^{-1}\left[\mathrm{e}^{\Delta t\hat{\mathrm{L}}}\hat{w}(t_{n+1},\xi)\right]\right)\right]\\
&\text{Set} && u(t_{n+1},x) = \mathrm{F}^{-1}\left(\mathrm{e}^{\Delta t\hat{\mathrm{L}}}\hat{w}(t_{n+1},\xi)\right)
\end{aligned}$$

It's not necessary to switch back completely to the u variable. Instead we can solve the problem in the $\hat{w}$ variable and only switch back when we want to observe the solution.

1. Take $\hat{w} = \mathrm{F}[u(0,x)]$ as the initial conditions.
2. At each time step t_n to t_{n+1}, implement (16.2) with a time step Δt for initial conditions $\hat{w}$ using an ODE solver such as a fourth-order Runge–Kutta to get a high-order method.
3. At the end of each time interval, take $\hat{w} = \mathrm{e}^{\Delta t\hat{\mathrm{L}}}\hat{w}$ as the initial condition for the next time step.
4. Finally, take $u(t,x) = \mathrm{F}^{-1}\,\hat{w}$.

16.3 Incompressible Navier–Stokes equation

In this final section, we examine the two-dimensional incompressible Navier–Stokes equation. The numerical solution follows a similar treatment discussed in Deuflhard and Hohmann [2003]. One-dimensional Fourier spectral methods can be easily modified to handle two- and three-dimensional problems by replacing the vector ξ with the tensor $\xi_x + \xi_y$ or $\xi_x + \xi_y + \xi_z$.

Consider the two-dimensional incompressible Navier-stokes equation

$$\mathbf{u}_t + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) = -\nabla p + \varepsilon \Delta u \tag{16.3}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{16.4}$$

which models fluid flow. The vector $\mathbf{u} = (u, v)$ is the velocity field, $p(x, y)$ is the pressure and ε is the inverse of the Reynold's number. The symbol $\otimes$ denotes a tensor product—$\nabla \cdot (\mathbf{u} \otimes \mathbf{u})$ is $\sum_j \partial_j u_i u_j$ in index notation. Explicitly, the Navier–Stokes equation says

$$\begin{aligned} u_t + (u^2)_x + (uv)_y &= -p_x + \varepsilon(u_{xx} + u_{yy}) \\ v_t + (uv)_y + (v^2)_x &= -p_y + \varepsilon(v_{xx} + v_{yy}) \\ u_x + u_y &= 0. \end{aligned}$$

To solve the problem, we will find $\mathbf{u}$ using the conservation of momentum equation (16.3) and use the conservation of mass (16.4) as a constraint by forcing $\mathbf{u}$ to be divergence-free. Because (16.3) is has stiff linear terms and nonlinear terms, we will use a second-order IMEX Adams–Bashforth/Crank–Nicolson method

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = \underbrace{-\tfrac{1}{2}(\nabla p^{n+1} + \nabla p^n)}_{\text{Crank–Nicolson}} + \underbrace{\tfrac{3}{2}\mathbf{H}^n - \tfrac{1}{2}\mathbf{H}^{n-1}}_{\text{Adams–Bashforth}} + \underbrace{\tfrac{1}{2}\varepsilon(\Delta \mathbf{u}^{n+1} + \Delta \mathbf{u}^n)}_{\text{Crank–Nicolson}}$$

where $\mathbf{H}^n = \nabla \cdot (\mathbf{u}^n \otimes \mathbf{u}^n)$. There are a couple problems with this setup. First, we don't explicitly know p^{n+1}. Second, we haven't enforced the constraint (16.3). So, let's modify the approach by introducing an intermediate solution $\mathbf{u}^*$:

$$\frac{\mathbf{u}^* - \mathbf{u}^n}{\Delta t} = -\tfrac{1}{2}\nabla p^n + \tfrac{3}{2}\mathbf{H}^n - \tfrac{1}{2}\mathbf{H}^{n-1} + \tfrac{1}{2}\varepsilon(\Delta \mathbf{u}^* + \Delta \mathbf{u}^n) \tag{16.5a}$$

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^*}{\Delta t} = -\tfrac{1}{2}\nabla p^{n+1} + \tfrac{1}{2}\varepsilon(\Delta \mathbf{u}^{n+1} - \Delta \mathbf{u}^*) \tag{16.5b}$$

We'll be able to use the intermediate solution so that we don't explicitly need to compute the pressure p. First, solve (16.5a). Formally, we have

$$\mathrm{M}_- \, \mathbf{u}^* = -\tfrac{1}{2}\nabla p^n + \tfrac{3}{2}\mathbf{H}^n - \tfrac{1}{2}\mathbf{H}^{n-1} + \mathrm{M}_+ \, \mathbf{u}^n$$

where the operators

$$\mathrm{M}_- = (\tfrac{1}{\Delta t} - \tfrac{1}{2}\varepsilon\Delta) \quad \text{and} \quad \mathrm{M}_+ = (\tfrac{1}{\Delta t} - \tfrac{1}{2}\varepsilon\Delta).$$

From this we have that

$$\mathbf{u}^* = \mathrm{M}_- \left(-\tfrac{1}{2}\nabla p^n + \tfrac{3}{2}\mathbf{H}^n - \tfrac{1}{2}\mathbf{H}^{n-1} + \mathrm{M}_+ \mathbf{u}^n\right). \tag{16.6}$$

We need to determine ∇p^{n+1} in such a way as to implicitly enforce the IMEX scheme by using (16.5b) to get ∇p^{n+1}

$$-\tfrac{1}{2}\nabla p^{n+1} = \mathrm{M}_-(\mathbf{u}^{n+1} - \mathbf{u}^*).$$

Equivalently,

$$-\tfrac{1}{2}\nabla p^n = \mathrm{M}_-(\mathbf{u}^n - \mathbf{u}^{*-1}) \tag{16.7}$$

where $\mathbf{u}^{*-1}$ is the intermediate solution at the previous timestep.

Substituting $-\frac{1}{2}\nabla p^n$ back into the first equation we have that

$$\begin{aligned}\mathbf{u}^* &= \mathrm{M}_-^{-1}\left(\mathrm{M}_-(\mathbf{u}^n - \mathbf{u}^{*-1}) + \tfrac{3}{2}\mathbf{H}^n - \tfrac{1}{2}\mathbf{H}^{n-1} + \mathrm{M}_+ \mathbf{u}^n\right)\\ &= \mathbf{u}^n - \mathbf{u}^{*-1} + \mathrm{M}_-^{-1}\left(\tfrac{3}{2}\mathbf{H}^n - \tfrac{1}{2}\mathbf{H}^{n-1} + \mathrm{M}_+ \mathbf{u}^n\right).\end{aligned}$$

Finally, enforce (16.4) 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

$$\nabla\cdot\mathbf{u}^{n+1} = \nabla\cdot\mathbf{u}^* - \nabla\cdot\nabla\Delta^{-1}\nabla\cdot\mathbf{u}^* = \nabla\cdot\mathbf{u}^* - \Delta\Delta^{-1}\nabla\cdot\mathbf{u}^* = 0.$$

(Recognize the projection operators $\mathbf{P_A} = \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ and $\mathbf{I} - \mathbf{P_A}$?)

We now have

$$\begin{aligned}\mathbf{u}^* &= \mathbf{u}^n - \mathbf{u}^{*-1} + \mathrm{M}_-^{-1}\left(\tfrac{3}{2}\mathbf{H}^n - \tfrac{1}{2}\mathbf{H}^{n-1} + \mathrm{M}_+ \mathbf{u}^n\right)\\ \mathbf{u}^{n+1} &= \mathbf{u}^* - \nabla\Delta^{-1}\nabla\cdot\mathbf{u}^*.\end{aligned}$$

To initialize the multistep method, we'll take

$$\mathbf{u}^{-1} = \mathbf{u}^{*-1} = \mathbf{u}^0.$$

In the Fourier domain, we replace

$$\nabla \to i(\xi_x, \xi_y) \quad \text{and} \quad \Delta = \nabla\cdot\nabla \to -\xi_x^2 - \xi_y^2 = -|\xi|^2.$$

And, from (16.8) we get

$$\hat{\mathbf{u}}^* = \hat{\mathbf{u}}^n - \hat{\mathbf{u}}^{*-1} + \hat{\mathrm{M}}_+^{-1}\left(\tfrac{3}{2}\hat{\mathbf{H}}^n - \tfrac{1}{2}\hat{\mathbf{H}}^{n-1} + \hat{\mathrm{M}}_-\hat{\mathbf{u}}^n\right) \tag{16.9a}$$

$$\hat{\mathbf{u}}^{n+1} = \hat{\mathbf{u}}^* - \frac{(\xi\cdot\hat{\mathbf{u}}^*)}{|\xi|^2}\xi \tag{16.9b}$$

where

$$\hat{\mathrm{M}}_- = \tfrac{1}{\Delta t} - \tfrac{1}{2}\varepsilon|\xi|^2 \quad \text{and} \quad \hat{\mathrm{M}}_+ = \tfrac{1}{\Delta t} + \tfrac{1}{2}\varepsilon|\xi|^2.$$

We must take some care when we divide by $|\xi|^2$ in (16.9b) because one of the components is zero. In this case we modify the term by replacing the zeros by ones. This "correction" is fixed when we subsequently multiply by ξ.

We haven't discussed implementation of the nonlinear term $\hat{\mathbf{H}}$ yet. Note that for $\mathbf{u} = (u, v)$ we have that the x-component of $\mathbf{H}$ is

$$H(u, v) = (uv)_y + (u^2)_x$$

with a similar form for a y-component. The Fourier transform of a product of two functions uv is the convolution $\hat{u} * \hat{v}$—a very inefficient operator to implement numerically. Therefore, we'll evaluate $\hat{\mathbf{H}}$ in real space. Hence, we take

$$\hat{H}(\hat{u}, \hat{v}) = i\xi_y\,\mathrm{F}(\mathrm{F}^{-1}\,\hat{u}\,\mathrm{F}^{-1}\,\hat{v}) + i\xi_x\,\mathrm{F}\left((\mathrm{F}^{-1}\,\hat{u})^2\right).$$

To visualize the solution, we use a tracer. Imagine dropping ink (or smoke) into a fluid and then following the motion of the ink. Numerically, we can simulate a tracer by solving the advection equation

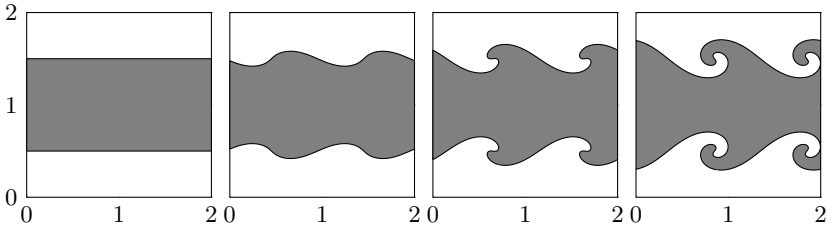
$$Q_t + \mathbf{u}\cdot\nabla Q = 0 \tag{16.10}$$

using, for example, the Lax–Wendroff method where $\mathbf{u} = (u, v)$ is the solution to the Navier–Stokes equation at each time step.

Consider the stratified flow with shear

$$(u, v) = \left(\tfrac{1}{2}(1 + \tanh(10 - 5|1 - y|)(1 + \tfrac{1}{2}\sin 2\pi x), 0\right)$$

The fluid is moving in the x-direction with two speeds separated by a narrow interface. Think of wind blowing over water. Such a situation leads to produces a Kelvin–Helmholtz instability, producing surface waves in water and leading to turbulance in other laminar fluids. Note the Kelvin–Helmholtz instability in solution to the Navier-Stokes equation at $t = 0$, 0.4, 0.8 and 1.2:



In MATLAB we have the following

```
H = @(u,v,ikx,iky) ikx.*fft2(ifft2(u).^2) + ...
    iky.*fft2(ifft2(u).*ifft2(v));
flux = @(Q,c) c.*diff([Q(end,:);Q]) + ...
    0.5*c.*(1-c).*diff([Q(end,:);Q;Q(1,:)],2);
L = 2; n = 128; e = 0.001; dt = .002;
k = [0:(n/2-1) 0 (-n/2+1):-1]*(2*pi/L);
[ikx,iky] = meshgrid(1i*k);
k2 = ikx.^2+iky.^2; k2i = k2; k2i(find(k2==0))=1;
x = (1:n)*(L/n); dx = x(2)-x(1); lambda = dt/dx;
[x,y] = meshgrid(x);
Q = 0.5*(1+tanh(10*(1-abs(L/2 -y)/(L/4))));
v = zeros(size(x)); u = Q.*(1+0.5*sin(L*pi*x));
u = fft2(u); v = fft2(v); us = u; vs = v;
Hx = H(u,v,ikx,iky); Hy = H(v,u,iky,ikx);
for i = 1:600
  Q = Q - flux(Q,lambda*real(ifft2(v))) - ...
      flux(Q',lambda*real(ifft2(u))')';
  image(U*100);colormap(jet(100));drawnow;
  Hxo = Hx; Hyo = Hy; Hx = H(u,v,ikx,iky); Hy = H(v,u,iky,ikx);
  M1 = (1/dt + (e/2)*k2); M2 = (1/dt - (e/2)*k2);
  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)./k2i;
  u = us - ikx.*phi;
  v = vs - iky.*phi;
end
```

The variable `e` is used for the scaling parameter ε, and `k` $= \xi$ is a temporary variable used to construct `ikx` $= \mathrm{i}\xi_x$, `iky` $= \mathrm{i}\xi_y$, and `k2` $= -|\xi|^2 = -\xi_x^2 + \xi_y^2$. The variable `Q` gives the density of the tracer, which evolves using the advection equation (16.10) and solved with the Lax–Wendorff method. The flux is implemented using the anonymous function `flux`. The variable `Hx`, computed using the function `H`, is the x-component of $\mathbf{H}^n$ and `Hxo` x-component of $\mathbf{H}^{n-1}$. Similarly, `Hy` and `Hyo` 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, `uo`, `vo`, `us` and `vs`

are u^{n-1}, v^{n-1} , u^* and v^*, respectively. The variable `phi` is a temporary variable used for $\Delta^{-1}\nabla \cdot \mathbf{u}^*$.

Ⓜ The MATLAB function `fft` implements the Fastest Fourier Transform in the West (FFTW) library. The inverse FFT can be computed using `ifft` and multidimensional FFTs can be computed using `fftn` and `ifftn`.

Exercises

16.1. Suppose that you use a fourth-order Runge–Kutta scheme to solve the Burgers equation using a Fourier spectral method

$$u_t + \left(\frac{u^2}{2}\right)_x = 0.$$

What order can you expect? What else can you say about the numerical solution?

✍ 16.2. The Kortweg–deVries (KdV) equation

$$u_t + 3(u^2)_x + u_{xxx} = 0 \tag{16.11}$$

is a nonlinear equation used to model wave motion in a dispersive medium such as shallow water and optic fiber. The purpose of this exercise is to explore the behavior the KdV equation by first developing a high-order numerical scheme to solve the equation and then using the scheme to simulate soliton interaction.

Dispersion means that different frequencies travel at different speeds causing a solution to break up and spread out over time. For certain class of initial conditions, the nonlinearity of the KdV equation counteracts the dispersion and the solution maintains its shape. We call such a solution a solitary wave or a soliton. It can be easily shown by substitution that

$$u(x,t) = \tfrac{1}{2}c\,\mathrm{sech}^2\left(\frac{\sqrt{c}}{2}(x-ct)\right)$$

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 (16.11) on the interval $[-3,3]$ for $t \in [0, 0.01]$ using the fourth-order Runge–Kutta method with integrating factors. Use perhaps $n = 256$ grid points in space over a domain $x \in [-20, 20]$. Take $\Delta t = O(100/n^2)$ to ensure stability. Let

$$\phi(x; x_0, c) = \tfrac{1}{2}c\,\mathrm{sech}^2\left(\frac{\sqrt{c}}{2}(x-x_0)\right).$$

Observe the solutions to (16.11) with initial conditions given by

1. $u(x,0) = \phi(x;-4,4)$
2. $u(x,0) = \phi(x;-9,9)$
3. $u(x,0) = \phi(x;-4,4) + \phi(x;-9,9)$.

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. Comment on the behavior. You may also want to observe a three-soliton collision by adding another soliton (say $\phi(x;-12,12)$). You may need to make the domain larger so that unstable oscillations at the domain boundaries do not pollute the results.

To implement the phase vector ξ in MATLAB use

```
k = [0:(n/2-1) 0 (-n/2+1):-1]*(2*pi/L);
```

where `n` is the number of grid points and `L` is the length of the domain.

You may want to use the following Runge–Kutta function:

```
function u = rk4(u,f,dt)
K1 = f(0,u);
K2 = f(0.5*dt,u+0.5*dt*K1);
K3 = f(0.5*dt,u+0.5*dt*K2);
K4 = f(dt,u+dt*K3);
u = u + dt*(K1+2*K2+2*K3+K4)/6;
```

✍ 16.3. The two-dimensional Swift–Hohenberg equation

$$u_t = \varepsilon u - u^3 - (\Delta + 1)^2 u$$

with $u \equiv u(x,y,t)$ models Rayleigh–Bernard 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 gridpoints in x and y. Choose $u(x,y)$ randomly from the uniform distribution $[-1,1]$. Output the solution at time $t = 100$ and several intermediate times.

Note: The Fourier transform of the Laplacian operator can be implemented as

```
k = [0:(n/2-1) 0 (-n/2+1):-1]*(2*pi/L);
[kx,ky] = meshgrid(k);
D2 = (1i*kx).^2 + (1i*ky).^2;
```

where `n` is the number of grid points and `L` is the length of the domain. A two dimensional FFT can be implemented in MATLAB with `fft2`

Note: the analytic solution to $y' = \varepsilon y - y^3$ is

$$y = \frac{y_0}{\sqrt{(1 - \varepsilon^{-1} y_0^2)\mathrm{e}^{-2\varepsilon t} + \varepsilon^{-1} y_0^2}}.$$

Exercise Solutions

Numerical Methods for Linear Algebra

1.3. For the Krylov supspace:

(a) If $\mathbf{x}$ is an eigenvector of $\mathbf{A}$, then $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, $\mathbf{A}^2\mathbf{x} = \lambda^2\mathbf{x}$ 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

$$\begin{aligned}
&\mathbf{A}\mathbf{x} = a\mathbf{A}\mathbf{v}_1 + b\mathbf{A}\mathbf{v}_2 = a\lambda_1\mathbf{v}_1 + b\lambda_2\mathbf{v}_2 \\
&\mathbf{A}^2\mathbf{x} = a\mathbf{A}^2\mathbf{v}_1 + b\mathbf{A}^2\mathbf{v}_2 = a\lambda_1^2\mathbf{v}_1 + b\lambda_2^2\mathbf{v}_2 \\
&\text{and so on...}
\end{aligned}$$

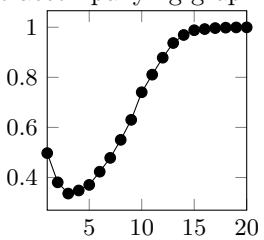
So, the Krylov subspace is spanned by linear combinations of the eigenvectors $\mathbf{v}_1$ and $\mathbf{v}_2$ and the dimension is 2.

(c) If $\mathbf{A}$ is a projection operator, then $\mathbf{A}^2 = \mathbf{A}$. It follows that the Krylov subspace $\mathcal{K}_r(\mathbf{A}, \mathbf{x}) = \text{span}\{\mathbf{x}, \mathbf{A}\mathbf{x}\}$. If $\mathbf{x}$ is in the null space of $\mathbf{A}$ then $\mathbf{A}\mathbf{x} = \mathbf{0}$. So, the maximum dimension is 2.

(d) For any vector $\mathbf{x}$, the vector $\mathbf{A}\mathbf{x}$ 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 $\mathbf{A}$). $\square$

1.4. The following MATLAB code generates the accompanying graph:

```
N = 10000; n = zeros([1 20]);
for i = 1:20
for j = 1:N
n(i) = n(i) + (det(rand(i)>0.5)~=0);
end
end
plot(1:20,n/N)
```



□

1.5. (a) Let $\mathbf{M}_n = \mathbf{D}_n + 2\mathbf{I}$. If $\{\lambda, \mathbf{x}\}$ is an eigenpair of $\mathbf{M}_n$ (that is, $\mathbf{M}_n\mathbf{x} = \lambda\mathbf{x}$), then

$$\mathbf{D}_n\mathbf{x} = \tfrac{1}{2}(\mathbf{M}_n - \mathbf{I})\mathbf{x} = \tfrac{1}{2}(\lambda - 1)\mathbf{x}.$$

So, $\{\frac{1}{2}(\lambda - 1), \mathbf{x}\}$ is an eigenpair of $\mathbf{D}_n$ The eigenvalues λ of $\mathbf{M}_n$ are zeros of the characteristic polynomial $p_n(\lambda) = \det(\mathbf{M}_n - \lambda\mathbf{I})$, which we can evaluate using the method of cofactors. In this case we have,

$$\begin{aligned} p_n(\lambda) &= \det(\mathbf{M}_n - \lambda\mathbf{I}) \\ &= -\lambda\det(\mathbf{M}_{n-1} - \lambda\mathbf{I}) - \det(\mathbf{M}_{n-2} - \lambda\mathbf{I}) \\ &= -\lambda p_{n-1}(\lambda) - p_{n-2}(\lambda) \end{aligned}$$

By making the change of variables $\lambda = -2\gamma$ and $p_n(\lambda) = T_n(\gamma)$ we have that

$$T_n(\gamma) = 2\gamma T_{n-1}(\gamma) - T_{n-2}(\gamma)$$

which is the three-term recurrence relation for the Chebyshev polynomial $T_n(\gamma)$. The Chebyshev nodes (the roots of the Chebyshev polynomial) are given by

$$\gamma_j = \cos\left(\frac{2j-1}{2n}\pi\right), \qquad j = 1, \ldots, n$$

So,

$$\lambda_j = -2\cos\left(\frac{2j-1}{2n}\pi\right), \qquad j = 1, \ldots, n$$

Therefore, the eigenvalues of $\mathbf{D}_n$ are given by

$$\tfrac{1}{2}(\lambda_j - 1) = -1 - \cos\left(\frac{2j-1}{2n}\pi\right), \qquad j = 1, \ldots, n$$

(b) The spectral radius of $\mathbf{D}_n + 2\mathbf{I}$ is

$$\rho(\mathbf{D}_n + 2\mathbf{I}) = \max_{j=1,\ldots,n} \left| -2\cos\left(\frac{2j-1}{2n}\pi\right)\right| = 2\cos\left(\frac{\pi}{2n}\right)$$

(c) The l_2 condition number for a symmetric matrix is the ratio of the largest and smallest eigenvalues.

$$\begin{aligned}\kappa_2(\mathbf{D}_n) &= \frac{1+\cos(\pi/2n)}{1-\cos(\pi/2n)} \\ &\approx \frac{2-\frac{1}{2}(\pi/2n)^2}{\frac{1}{2}(\pi/2n)^2} \approx \frac{16}{\pi^2}n^2 \text{ as } n \to \infty\end{aligned}$$

So, the condition number of D_n is $O(n^2)$. □

1.5. We'll start with (e)

(e) The ith diagonal element of $\mathbf{A}^T\mathbf{A}$ is given by $\sum_j a_{ij}^2$. The trace of a matrix is the sum of the diagonal elements. So, $\mathrm{tr}(\mathbf{A}^T\mathbf{A}) = \sum_{i,j} a_{ij}^2$, which equals $\|\mathbf{A}\|_F^2$.

(b) For an $n \times n$ orthogonal matrix $\mathbf{Q}$, $\mathbf{Q}^T\mathbf{Q}$ is $n \times n$ identity matrix $\mathbf{I}$. The trace of $\mathbf{I} = n$. Therefore, using the result from (e), $\|\mathbf{Q}\|_F = \sqrt{n}$

(c) Using the result from (e),

$$\|\mathbf{QA}\|_F^2 = \mathrm{tr}((\mathbf{QA})^T\mathbf{QA}) = \mathrm{tr}(\mathbf{A}^T\mathbf{Q}^T\mathbf{QA}) = \mathrm{tr}(\mathbf{A}^T\mathbf{A}) = \|\mathbf{A}\|_F^2.$$

(d) Suppose that $\mathbf{A}$ has the singular value decomposition $\mathbf{A} = \mathbf{U\Sigma V}^T$. Then using the result of part (c),

$$\|\mathbf{A}\|_F^2 = \|\mathbf{U\Sigma V}^T\|_F^2 = \|\mathbf{\Sigma V}^T\|_F^2 = \|\mathbf{V\Sigma}^T\|_F^2 = \|\mathbf{\Sigma}^T\|^2 = \textstyle\sum_i \sigma_i^2.$$

(a) The Frobenius norm sums over each element of $\mathbf{A}$ independent of the location of that element. By reshaped the $m \times n$ array $\mathbf{A}$ as an $mn \times 1$ array $\mathbf{a}$, we can see the Frobenius norm as simply the l_2-norm of the vector $\mathbf{a}$. The l_2-norm of a vector satisfies positivity, homogeneity and the triangle inequality, so the Frobenius norm also does.

(g) To show submultiplicitivity $\|\mathbf{AB}\|_F \le \|\mathbf{A}\|_F\|\mathbf{B}\|_F$:

$$\begin{aligned}\|\mathbf{AB}\|_F^2 &= \sum_{i,j}\sum_k (a_{ik}b_{kj})^2 = \sum_{i,j}\sum_k a_{ik}^2 b_{kj}^2 = \sum_k\Big(\sum_i a_{ik}^2\Big)\Big(\sum_j b_{kj}^2\Big) \\ &\le \sum_{i,k} a_{ik}^2 \sum_{k,j} b_{kj}^2 = \|\mathbf{A}\|_F^2\|\mathbf{B}\|_F^2.\end{aligned}$$

(f) $\|\mathbf{Ax}\|_F \leq \|\mathbf{Ax}\|_F\|\mathbf{x}\|$ follows from submultiplicitivity proved in (g) by taking $\mathbf{B} = \mathbf{x}$. □

2.4. The determinant of a product is given by the product of the determinants. We can use this identity along with LU factorization to efficiently determine the determinant of a matrix. With partial pivoting, we have that the LU factorization of a matrix $\mathbf{A}$ is given by $\mathbf{PA} = \mathbf{LU}$ where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is a lower triangular matrix with ones along its diagonal, and $\mathbf{U}$ is an upper triangular matrix. The determinant of a triangular matrix is the product of the diagonal elements. So, $\det \mathbf{L} = 1$. The determinant of a permutation matrix $\mathbf{P}$ is the sign of the permutation $\operatorname{sgn}(\mathbf{P})$ which equals $+1$ if $\mathbf{P}$ was built from $\mathbf{I}$ using even number of row exchanges (even parity) and -1 if $\mathbf{P}$ is built using an odd number of row exchanges (odd parity). So,

$$\det \mathbf{A} = \det \mathbf{P} \det \mathbf{L} \det \mathbf{U} = \operatorname{sgn}(\mathbf{P}) \prod_{i=1}^{N} \operatorname{diag} \mathbf{U}.$$

The MATLAB funtion `lu` will give us $\mathbf{U}$ and $\mathbf{P}$, so we just need to compute the sign of the permutation.

```
function D = det(A)
[L,U,P] = lu(A);
s = 1; N = length(P);
for i = 1:N
  m = find(P(i+1:N,i));
  if m, P([i m+i],:) = P([m+i i],:); s = -1*s; end
end
D = s * prod(diag(U));
```

□

2.5. We start the Cuthill–McKee algorithm by selecting a column of $\mathbf{A}$ that has the fewest off-diagonal nonzero elements—column 5 which has one nonzero in rows 3. We now find the nonzero elements in column 3–rows 2 and 10. We repeat the process, looking at columns 2 and 10 and skipping any positions that we've already counted. We continue in this fashion until we've run through all of the columns to build a tree. See Figure S.1 on the next page. To get the permutation, start with the root node and subsequently collect the siblings at each lower level to get

$$\{5, 3, 2, 10, 4, 8, 11, 1, 7, 12, 6, 9\}$$

The reverse Cuthill–McKee sorting simply reverses this order

$$\{9, 6, 12, 7, 1, 11, 8, 4, 10, 2, 3, 5\}.$$

□

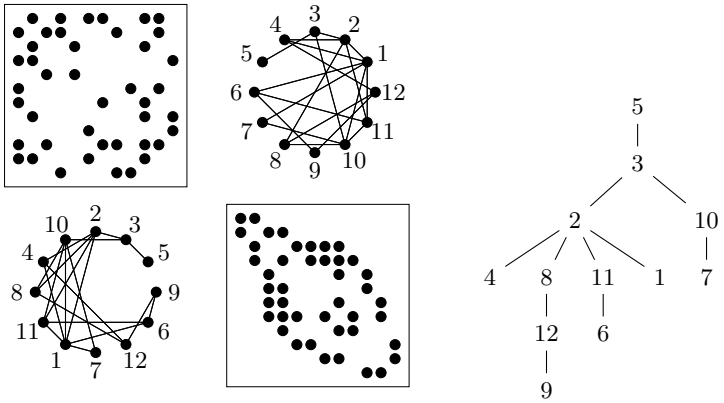


Figure S.1: Cuthill–McKee sorting

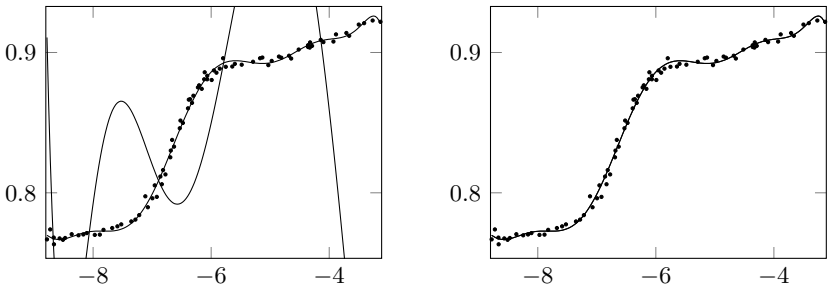
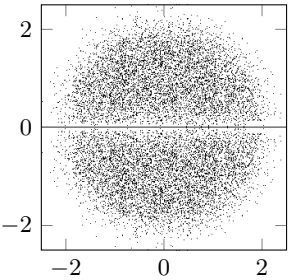


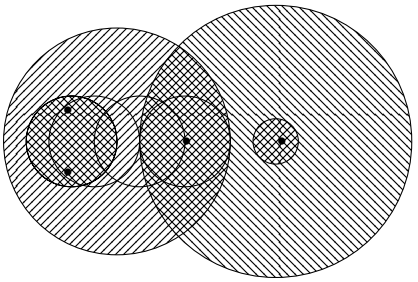
Figure S.2: Solutions to Exercise 3.6. The left plot shows solutions using the Vandermonde matrix built using the original x-data and the right plot shows solutions using the Vandermonde matrix built using the mean-centered x-data.

3.6. The 2-condition number of the Vandermonde matrix is 1.7×10^{15}, which makes it very ill-conditioned. The residuals are $\|\mathbf{r}\|_2 = 0.97998$ using the normal equations and $\|\mathbf{r}\|_2 = 0.028211$ using QR-decomposition, which matches residual for the coefficients provided by NIST. The relative errors are roughly 11 and 4×10^{-7}, respectively. The normal equation solution clearly fails, which is evident in Figure S.2.

After mean centering the x-data, the 2-condition number of the new Vandermonde is reduced to 1.3×10^5. By further scaling the x-data between -1 and 1, the 2-condition number is reduced to $\kappa_2 = 4610$. So, the matrix is well conditioned, even when using the normal equation method. The residuals are both methods are $\|\mathbf{r}\|_2 = 0.028211$ and both

Figure S.3: The eigenvalues of a thousand normally distributed random 16×16 real matrices with variance 1 in the complex plane. The distribution follows Girko's circular law which says that as $n \to \infty$, eigenvalues are uniformly distributed in a disk of radius $\sqrt{n}$.





$$\mathbf{A} = \begin{bmatrix} 9 & 0 & 0 & 1 \\ 2 & 5 & 0 & 0 \\ 1 & 2 & 0 & 1 \\ 3 & 0 & -2 & 0 \end{bmatrix}$$

Figure S.4: Gershgorin circles in the complex plane and the eigenvalues of $\mathbf{A}$. Eigenvalues lie in the region cross-hatched region ▩.

methods perform well. □

4.2. The solution is shown in Figure S.3. □

4.3. The solution is shown in Figure S.4. □

4.5. The following MATLAB code finds an eigenpair using the Rayleigh iteration starting with a random initial vector. By running repeatedly for different initial guesses, we can find all four eigenpairs.

```
function [rho,x] = rayleigh(A)
n = length(A); x = randn([n 1]);
while 1
  x = x/norm(x);
  rho = x'*A*x;
  M = A-rho*eye(n);
  if rcond(M)<eps, break; end
```

```
  x = M\x;
end
```

The method converges rapidly and $\mathbf{A}-\rho\mathbf{I}$ becomes badly conditioned as ρ approaches one of the eigenvalues. So, we break out of the iteration before committing the cardinal sin of using an ill-conditioned operator. □

4.6. The following MATLAB code implements the implicit QR method:

```
function ev = implicitqr(A)
n = length(A);
tolerance = 10E-10;
H = hess(A);
while 1
  if abs(H(n,n-1))<tolerance,
    n = n-1; if n<2, break; end;
  end
  r = rho(H(n-1,n-1),H(n-1,n),H(n,n-1),H(n,n));
  Q = givens(H(1,1)-r,H(2,1));
  H(1:2,1:n) = Q*H(1:2,1:n);
  H(1:n,1:2) = H(1:n,1:2)*Q';
  for i = 2:n-1
    Q = givens(H(i,i-1),H(i+1,i-1));
    H(i:i+1,1:n) = Q*H(i:i+1,1:n);
    H(1:n,i:i+1) = H(1:n,i:i+1)*Q';
  end
end
ev = diag(H);

function r = rho(a,b,c,d)
r = (d+a+(1-2*(d<a))*sqrt(4*b*c+(d-a)^2))/2;
```
□

5.1. The Gauss-Seidel method converges if and only if the spectral radius of the matrix $(\mathbf{L}+\mathbf{D})^{-1}\mathbf{U}$ is strictly less than one. The eigenvalues of

$$(\mathbf{L}+\mathbf{D})^{-1}\mathbf{U} = \begin{bmatrix} 1 & 0 \\ -\sigma & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 & \sigma \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \sigma \\ 0 & \sigma^2 \end{bmatrix}$$

are 0 and σ^2. So, the Gauss–Seidel method converges when $|\sigma| < 1$. □

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 $j = 0, \ldots, m-1$ and noting

$$\begin{aligned} \omega_m^{k(m+j)} &= \omega_m^{km}\omega_m^{kj} = \omega_m^{kj}, \\ \omega_m^{k(2m+j)} &= \omega_m^{2km}\omega_m^{kj} = \omega_m^{kj}, \end{aligned} \quad \text{and} \quad \begin{aligned} \omega_n^{m+j} &= \omega_n^m \omega_n^j = \omega_3 \omega_n^j, \\ \omega_n^{2m+j} &= \omega_n^{2m}\omega_n^j = \omega_3^2 \omega_n^j, \end{aligned}$$

where $\omega_3 = \mathrm{e}^{-2\mathrm{i}\pi/3}$, $\omega_3^2 = \mathrm{e}^{2\mathrm{i}\pi/3}$ and $\omega_3^4 = \mathrm{e}^{-2\mathrm{i}\pi/3}$, 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

$$\mathbf{F}_n \mathbf{c} = \left(\mathbf{F}_3 \otimes \mathbf{I}_{n/3}\right) \boldsymbol{\Omega}_n \left(\mathbf{I}_3 \otimes \mathbf{F}_{n/3}\right) \mathbf{P}\mathbf{c}$$

were $\boldsymbol{\Omega} = \mathrm{diag}(1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1})$.

```
function y = fftc(c)
n = length(c);
omega = exp(-2i*pi/n);
if mod(n,3) == 0
 %Continue recursion
  k = (0:n/3-1)';
  w = omega.^k;
  F = exp(-2i*pi/3).^((0:2)'*(0:2));
  u = [ fftc(c(1:3:n-2)),
        w.*fftc(c(2:3:n-1)),
        w.^2.*fftc(c(3:3:n))];
  y = [u*F(:,1);u*F(:,2);u*F(:,3)];
else
  %Compute DFT
  j = 0:n-1;
  k = j';
  F = omega.^(k*j);
  y = F*c;
end
```

□

6.3. When writing the code, keep in mind that the FFT operates on floating-point numbers, so we use `round` to convert the `ifft` to nearest integer. Putting everything together we get the following:

```
function [pq] = fftMult(p,q)
%Compute x*y = xy using FFTs
np = [fliplr(p-'0') zeros([1 length(q)])];
nq = [fliplr(q-'0') zeros([1 length(p)])];
pq = round(ifft(fft(np).*fft(nq)));
%Now to carry and remove the leading zeros
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');
```

□

Numerical Methods for Analysis

7.1. Truncation error of the central-difference approximation is bounded by $e_T = \frac{1}{3}h^2 M$ where $M = \sup_\xi |f''(\xi)|$. For $f(x) = \exp(x)$, we have that $M \approx 1$. The round-off error is bounded by $e_R = 2\,\mathrm{eps}/(2h)$. The total error $E(h) = e_T + e_R$ is minimized when $E'(h) = 0$:

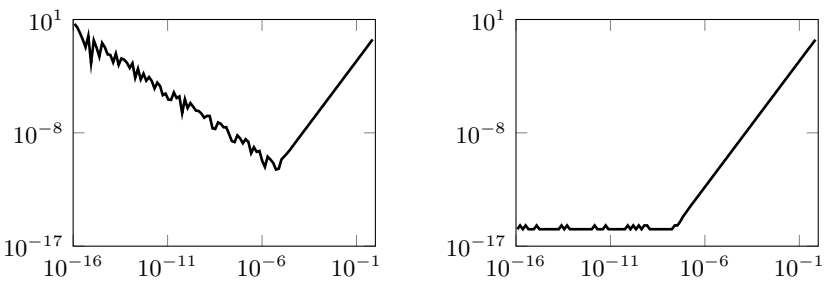
$$E' = \tfrac{2}{3}Mh - 2\,\mathrm{eps}/h^2 = 0$$

So, the error is minimum at $h \approx (3\,\mathrm{eps}/M)^{1/3}$. For $f(x) = \exp(x)$ at $x = 0$, the error reaches a minimum of about $\mathrm{eps}^{2/3} \approx 10^{-11}$ when $h = (3\,\mathrm{eps})^{1/3} \approx 8 \times 10^{-6}$.

Taking the Taylor series expansion of $f(x + \mathrm{i}h)$ we have that

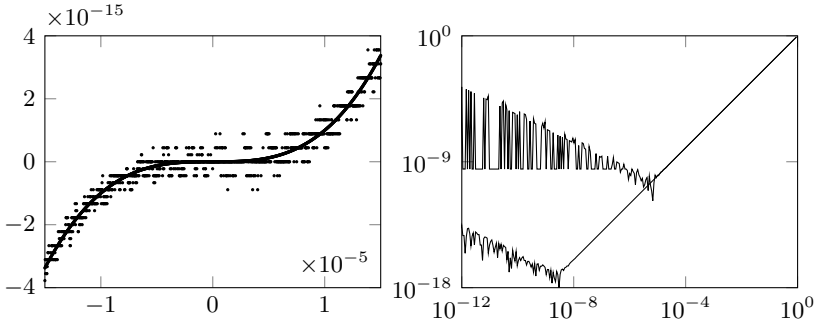
$$f(x + \mathrm{i}h) = f(x) + \mathrm{i}hf'(x) - \tfrac{1}{2}h^2 f''(x) - \tfrac{1}{2}\mathrm{i}h^3 f'''(\xi)$$

where $|\xi - x| < h$. The imaginary part $f'(x) = \mathrm{Im}\, f(x + \mathrm{i}h)/h + e_T$ with truncation error $e_T \approx \frac{1}{2}h^2|f'''(x)|$. The round-off error $e_R \approx \mathrm{eps}|f(x)|$. For $f(x) = \exp(x)$, the total error $E(h) = \frac{1}{2}h^2 + \mathrm{eps}$ which is smallest when $h < \sqrt{2\,\mathrm{eps}} \approx 10^{-8}$



The figure above shows total error as a function of the stepsize h for $(f(x + h) - f(x - h))/2h$ and $\mathrm{Im}\,(f(x + \mathrm{i}h))$. □

7.2. Below left: Plots of $(x-1)^3$ (forming a solid line) and x^3-3x^2+3x-1 (dots). The x-axis offset from $x=1$.



Above right: Log-log plots of the error in the central difference approximation as a function of stepsize h. Round-off error dominates total error when $h < 10^{-5}$ in x^3-3x^2+3x-1 and when $h < 10^{-9}$ in $(x-1)^3$. □

8.1. 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 is 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

$$2\frac{(1/f)'}{(1/f)''} = 2\frac{-\dfrac{2x}{(x^2-a)^2}}{\dfrac{2(3x^2+a)}{(x^2-a)^3}} = 2\frac{-x(x^2-a)}{3x^2+a}$$

from which we have that

$$x^{(k+1)} = \frac{x^{(k)}((x^{(2)})k+3a)}{3(x^{(2)})k+a}.$$

Starting with $x^{(0)}=1$, we get:

1.0000000000000000000000000000000000000000000000000000...
1.4000000000000000000000000000000000000000000000000000...
1.414213197969543147208121827411167512690355329949 23...
1.41421356237309504879564008075425994635423824014524...
1.414213562373095048801688724209698078569671875376 94...

where right number of decimals (1, 2, 7, 19, and 62, respectfully) roughly triples with each iteration. □

8.4. If $p(r_i) = 0$, then $r_i^n = -c_0 - c_1 r_i - c_2 r_i^2 - \cdots - c_{n-1} r_i^{n-1}$. Furthermore,

$$\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -c_0 & -c_1 & c_2 & \cdots & -c_{n-1} \end{bmatrix} \begin{bmatrix} 1 \\ r_i \\ \vdots \\ r_i^{n-2} \\ r_i^{n-1} \end{bmatrix} = \begin{bmatrix} r_i \\ r_i^2 \\ \vdots \\ r_i^{n-1} \\ r_i^n \end{bmatrix} = r_i \begin{bmatrix} 1 \\ r_i \\ \vdots \\ r_i^{n-2} \\ r_i^{n-1} \end{bmatrix}.$$

So, the left eigenvectors of $\mathbf{A}$ are $\begin{bmatrix} 1 & r_i & r_i^2 & \cdots & r_i^{n-1} \end{bmatrix}$. We have that $\mathbf{VAV}^{-1} = \mathbf{\Lambda}$ where

$$\mathbf{V} = \begin{bmatrix} 1 & r_1 & r_1^2 & \dots & r_1^{n-1} \\ 1 & r_2 & r_2^2 & \dots & r_2^{n-1} \\ 1 & r_3 & r_3^2 & \dots & r_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & r_n & r_n^2 & \dots & r_n^{n-1} \end{bmatrix}$$

is a Vandermonde matrix and $\mathbf{\Lambda} = \operatorname{diag}(r_1, r_2, r_3, \dots, r_n)$. □

8.5. For the fixed-point method to converge, we need $|\phi'(x)| < 1$ in a neighborhood of the fixed point x^*. In the case of $\phi(x) = x - \alpha f(x)$, we have that $|\phi'(x)| = |1 - \alpha f'(x)| < 1$ 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 α 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 $x^{(k+1)} = x^{(k)} - f(x^{(k)})/f'(x^{(k)})$ □

8.6. 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

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{x}(t) = -\left(\frac{\partial f}{\partial \mathbf{x}}\right)^{-1} f(\mathbf{x}_0)$$

with initial conditions $(x(0), y(0)) = \mathbf{x}_0$. We solve this problem using the following MATLAB code:

```
x0 = [-1;-1];
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];
dxdt = @(t,x) -inv(df(x(1),x(2)))*f(x0(1),x0(2));
[t,y] = ode45(dxdt,[0;1],x0);
```

We can implement Newton's method

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\frac{\partial f}{\partial \mathbf{x}^{(k)}}\right)^{-1} f(\mathbf{x}^{(k)})$$

by replacing the last two lines in code above with

```
for i = 1:100
  x1 = x0 - inv(df(x(1),x(2)))*f(x(1),x(2));
  if norm(x1-x0)<10e-8, break;end
  x0 = x1;
end
```

□

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 $x_0, x_1, \ldots, x_n$. Consider the relation

$$\det(\mathbf{V}) = \prod_{j=0}^{n} \prod_{i \neq j} (x_i - x_j) \tag{16.12}$$

The relation clearly holds for $n = 1$. Assume that (16.12) is true for n nodes, we will show that it also holds for $n+1$ nodes. We will prove this by using cofactor expansion with respect to the last column. Let $\mathbf{V}[x_k]$ be the Vandermonde matrix formed using all the nodes $\{x_0, x_1, \ldots, x_n\}$ *except* for x_k. Then

$$\det(\mathbf{V}[x_k]) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \prod_{i \neq j} (x_i - x_j) \tag{16.13}$$

By cofactor expansion $\det(\mathbf{V})$ equals

$$x_0^n \det(\mathbf{V}[x_0]) + \cdots + (-1)^k x_k^n \det(\mathbf{V}[x_k]) + \cdots + (-1)^n x_n^n \det(\mathbf{V}[x_n])$$

By (16.13),

$$\det(\mathbf{V}) = \sum_{k=0}^{n} (-1)^k x_k^n \prod_{\substack{j=0 \\ j \neq k}}^{n} \prod_{i \neq j} (x_i - x_j)$$

□

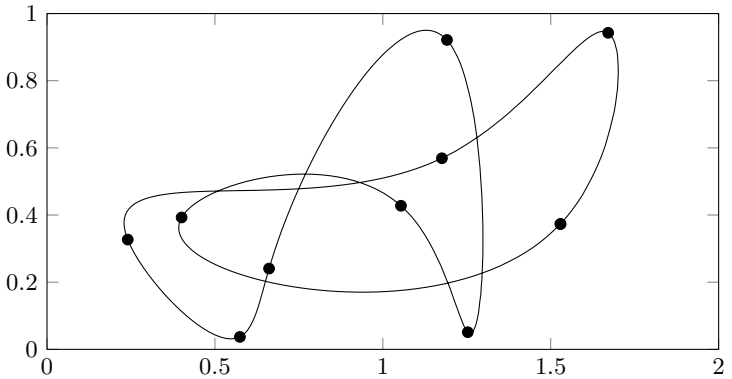


Figure S.5: Parametric spline through 10 knots.

9.2. The following MATLAB code picks random points and constructs a parametric spline through them using periodic boundary conditions. One solution is shown in Figure S.5.

```
function randomspline
x = 2*rand([10,1]); y = rand([10,1]);
x = [x;x(1)]; y = [y;y(1)];
t = [0;cumsum(sqrt(diff(x).^2+diff(y).^2))]
X = splinep(t,x,30*N);
Y = splinep(t,y,30*N);
clf; plot(X,Y,x,y,'.','markersize',16);

function [Y,X] = splinep(x,y,N)
n = length(x);
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)];
B = y(1:n-1) - m(1:n-1).*h.^2/6;
A = diff(y)./h-h./6.*diff(m);
X = linspace(x(1),x(end),N)';
[iX,ix] = meshgrid(X,x); i = sum(ix<=iX)'; i(end) = n-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);
```
□

9.3. This MATLAB code computes and plots the interpolating curves:

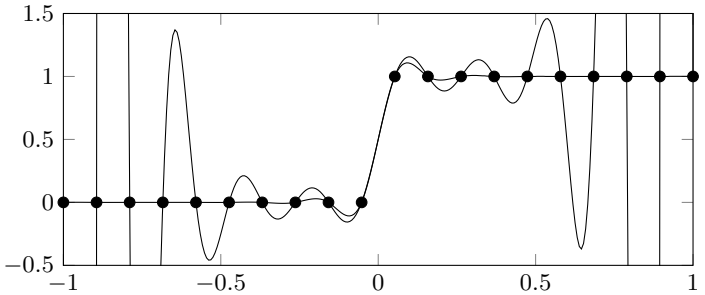


Figure S.6: Interpolation using polynomial and $|x|^3$ radial basis function .

```
function RBF
n = 20; N = 300;
x = linspace(-1,1,n)';
y = double(x>0);
X = linspace(-1,1,N)';
Y1 = interpolate(x,x,y,@(a,b) abs(a-b).^3,X);
Y2 = interpolate((0:n-1)',x,y,@(a,b) b.^a,X);
plot(x,y,X,Y1,X,Y2); ylim([-.5 1.5]);

function Y = interpolate(a,x,y,phi,X)
[ma,mx] = meshgrid(a,x);
[mA,mX] = meshgrid(a,X);
Y = phi(mA,mX)*(phi(ma,mx)\y);
```

Alternatively, you could use Newton basis coupled with the Horner algorithm to construct the polynomial interpolant

```
function Y = newton_interpolate(x,y,X)
n = length(x);
c = y;
for i = 1:n-1
    h = x(1+i:n)-x(1:n-i);
    c(i+1:n) = diff(c(i:n))./h;
end
Y = c(n).*ones(size(X));
for j = n-1:-1:1
    Y = c(j) + (X-x(j)).*Y;
end
```

As can be seen in Figure S.6, the polynomial interpolant suffers from the Runge phenomenon with a maximum error of about 500. The radial basis function on the other hand gives us a good approximation. It should

be noted that the radial basis function with $\phi(x) = |x|^3$ simply produces a cubic spline, because the resulting function is cubic on the subintervals and matches up to the second derivative at the nodes. $\square$

9.4. We solve the problem by solving the system of linear system of equations for $\{c_{-1}, c_0, \ldots, c_{n+1}\}$ and use the resulting coefficients to build the solution $u(x)$. The B-spline using a uniform mesh with separation h and nodes at $x_i = ih$ is given by

$$v_j(x) = B\left(\frac{x}{h} - j\right)$$

where

$$B(x) = \begin{cases} \frac{1}{6}(2+x)^3, & x \in [-2,1) \\ \frac{1}{6} + \frac{1}{2}(1+x) + \frac{1}{2}(1+x)^2 - \frac{1}{2}(1+x)^2, & x \in [-1,0) \\ \frac{1}{6} + \frac{1}{2}(1-x) + \frac{1}{2}(1-x)^2 - \frac{1}{2}(1-x)^2, & x \in [0,1) \\ \frac{1}{6}(1-x)^3, & x \in [1,2) \\ 0, & \text{otherwise} \end{cases}$$

Using the definition for the B-spline $v_j(x)$, we have that

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} v_j(x_i) = \begin{Bmatrix} 1/h^2 \\ -2/h^2 \\ 1/h^2 \\ 0 \end{Bmatrix}, \frac{\mathrm{d}}{\mathrm{d}x} v_j(x_i) = \begin{Bmatrix} -1/2h \\ 0 \\ 1/2h \\ 0 \end{Bmatrix}, v_j(x_i) = \begin{Bmatrix} 1/6 \\ 2/3 \\ 1/6 \\ 0 \end{Bmatrix}$$

for $\{j = i-1, j = i, j = i+1, \text{otherwise}\}$.

We solve a matrix equation $\mathbf{Ac} = \mathbf{w}$ where

$$w_i = w(x_i) = -12x_i^4 \mathrm{e}^{-3x_i^2}$$

and $A_{ij} = \mathrm{L}\, v_j(x_i)$ for $i = 1, \ldots, n$ is evaluated using

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} v_j(x_i) + (20x_i + 2)\frac{\mathrm{d}}{\mathrm{d}x} v_j(x_i) + (96x_i^2 + 24x_i + 2) v_j(x_i)$$

which is simply

$$\begin{Bmatrix} 1/h^2 \\ -2/h^2 \\ 1/h^2 \\ 0 \end{Bmatrix} + (20x_i + 2) \begin{Bmatrix} -1/2h \\ 0 \\ 1/2h \\ 0 \end{Bmatrix} + (96x_i^2 + 24x_i + 2) \begin{Bmatrix} 1/6 \\ 2/3 \\ 1/6 \\ 0 \end{Bmatrix}$$

for $\{j = i-1, j = i, j = i+1, \text{otherwise}\}$. The two boundary conditions can also be computed by using the definition of the B-spline $v_j(x_i)$ to give us the the first and last rows of $\mathbf{A}$. We now need to solve a matrix equation of the form

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

Once we have found c_j, we reconstruct the solution using

$$u(x) = \sum_{j=0}^{n} c_j v_j(x).$$

To get a smooth curve we use 200 or more values for x when evaluating $u(x)$. The MATLAB code is given below.

```
function error = collocation(n,N)
x = linspace(0,1,n); h = x(2)-x(1);
f = @(x) [96*x.^2+24*x+2;20*x+2;ones(size(x))];
w = @(x) -12*x.^4.*exp(-3*x.^2);
exact = @(x) (x - x.^2).*exp(-3*x.^2);
d = [1/6 -1/(2*h) 1/h^2; 2/3 0 -2/h^2; 1/6 1/(2*h) 1/h^2]*f(x);
M(2:n+1,1:n+2) = spdiags(d',[0 1 2],n,n+2);
M(1,1:3) = [1/6 2/3 1/6]; M(n+2,n:n+2) = [1/6 2/3 1/6];
c = M\[0;w(x');0];

%% Reconstruct Solution
X = linspace(0,1,N)';
i = floor(X/h)+1; i(N) = i(N-1);
C = [c(i)';c(i+1)';c(i+2)';c(i+3)'];
Y0 = sum(C.*B0((X'-x(i))/h));
Y1 = sum(C.*B1((X'-x(i))/h)/h);
Y2 = sum(C.*B2((X'-x(i))/h)/h^2);
R = sum(f(X').*[Y0;Y1;Y2]) - w(X');
error = Y0 - exact(X');

%% Plot Solutions
subplot(1,2,1)
plot(X,Y0,X,(X-X.^2).*exp(-3*X.^2)); title('Solution');
subplot(1,2,2)
plot(X,R); title('Residual');

%% B-Spline
function B0 = B0(x)
B0 = [(1-x).^3/6;1/6+(1-x)/2+(1-x).^2/2-(1-x).^3/2;...
        1/6+x/2+x.^2/2-x.^3/2;x.^3/6];
```

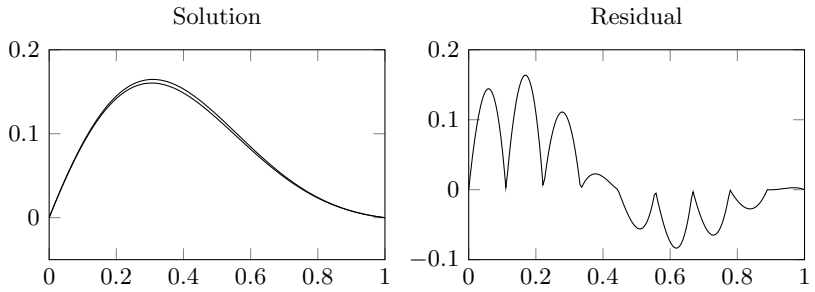


Figure S.7: Exact solution (top) and numerical solution (bottom) in the left figure and the residual in the right figure for 10 nodes.

```
%% First Derivative of B-Spline
function B1 = B1(x)
B1 = [-(1-x).^2/2;-1/2-(1-x)+3*(1-x).^2/2;...
        1/2+x-3*x.^2/2;x.^2/2];

%% Second Derivative of B-Spline
function B2 = B2(x)
B2 = [(1-x);1-3*(1-x);1-3*x;x];
```

The output is plotted in Figure S.7 using 10 nodes (`n = 10`) and 300 interpolating points (`N = 300`). Note that even for 10 nodes the numerical solution is fairly accurate. We can measure the convergence rate using the following analysis. A method has order p if the error is $O(h^p)$ were h is the mesh size—equivalently, if the error is $O(n^{-p})$ where n is the number of nodes. This means that the error $e \approx Mn^{-p}$ for some M. Solving for p we have

$$\log e = -p \log n + \log M$$

By computing the the slope of $(\log n, \log e)$ we will get an estimate for p. Using following MATLAB code, we get a slope of approx -2.0 which says that the method is second order convergent.

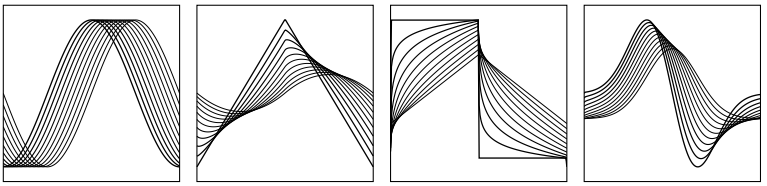
```
N = 1000;
n = 10*2.^(0:6);
y = zeros(size(n));
for i = 1:length(n)
    Y0 = collocation(n(i),N);
    y(i) = sqrt(sum(Y0.^2))/N;
end
clf;loglog(n,y)
```

```
convergence = diff(log(y))/log(2)
```

nodes	10	20	40	80	160
l^2-error ($\times 10^{-6}$)	89.1	20.4	4.85	1.18	0.29
convergence	-2.13	-2.07	-2.04	-2.02	

Typically, we do not have an exact solution to which to compare our numerical solution. Therefore, it is often useful to compute the residual. Consider the problem $\mathrm{L}\,u = w$. Let $u(x)$ be the exact solution and $U(x)$ be the approximate numerical solution. Then the error is given by $e(x) = u(x) - U(x)$. The residual is given by $w - \mathrm{L}\,U$. Note that the residual is simply $\mathrm{L}\,e$ and that the residual is zero at the nodes. The plot of the residual is given in Figure S.7 on the preceding page. □

10.3. Fractional derivatives of sine, piecewise-quadratic, piecewise-linear, and Gaussian functions for $p = 0, 0.1, \ldots, 1$ are



The pth derivative for $\sin(x)$ is $\sin(x + p\pi/2)$ which is simply $\sin x$, $\cos x$, $-\sin x$, and $-\cos x$ when p is integer values 0, 1, 2, and 3. The derivative is no longer a local operator in the Fourier domain. The function $\exp(-6x^2)$ is not periodic, but as long as the domain is sufficiently large, the function is $\exp(-6x^2)$ sufficiently smooth across the boundaries. Similarly, the piecewise-linear and piecewise-quadratic functions are not smooth, but as long as the have enough resolution to account for the slow decay of the Fourier transform, we minimize the effects of aliasing. Here is the code

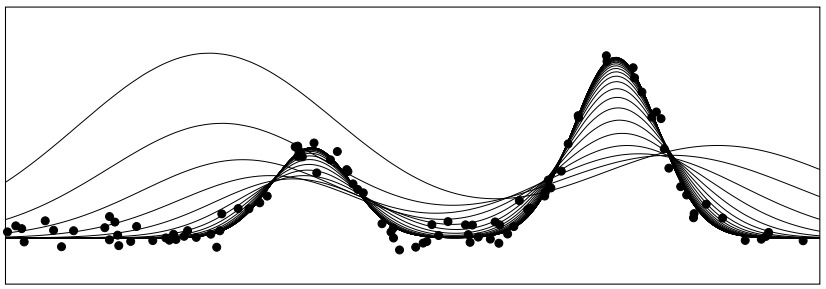
```
n = 128^2; L = 2;
x = (0:n-1)/n*L-L/2;
k = [0:(n/2-1) (-n/2):-1]*(2*pi/L);
f = exp(-6x.^2);
for p = 0:0.1:4
  d = real(ifft((1i*k).^p.*fft(f)));
  d = interp1(x,d,linspace(min(x),max(x),200);
end
```

□

10.4. Newton's method is $\mathbf{c}^{(k+1)} = \mathbf{c}^{(k)} - \alpha(\mathbf{J}^{(k)})^{+}(\mathbf{f}(\mathbf{x};\mathbf{c}^{(k)}) - \mathbf{y})$ where $(\mathbf{J}^{(k)})^{+}$ is the pseudoinverse of the Jacobian and where the learning rate α is chosen to be small enough to maintain stability. The following code implements the Levenberg–Marquardt method with $\alpha = 1$:

```
function NLS_solver
f = @(c,x) c(1)*exp(-c(2).*(x-c(3)).^2)+ ...
              c(4)*exp(-c(5).*(x-c(6)).^2);
rand('seed',8); randn('seed',8);
x = 8*rand([100 1]);
y = f([1 3 3 2 3 6],x) + 0.1*randn([100 1]);
c = [2 .3 2 1 .3 7]';
newton_solver(x,y,c,f);

function newton_solver(x,y,c,f)
for j = 1:80
  G = gradient(f,c,x);
  b = G'*(y - f(c,x));
  M = G'*G;
  c0 = c + (M+diag(diag(M)))\b;
  if norm(c0 - c)< 0.001, break; else c = c0; end
end

function g = gradient(f,c,x)
for i = 1:length(c)
  d = zeros(size(c));
  d(i) = eps(c(i)).^(1/3);
  g(:,i) = (f(c+d,x)-f(c-d,x))/(2*d(i));
end
```

The following plot shows each iteration, ending with the solution $\mathbf{c} = \{0.98, 3.25.98, 3.00, 1.96, 2.88, 5.99\}$. The input data is represented by $\bullet$.



Either of the following Newton steps could be substituted instead

```
c0 = c + 0.5*(M\b);
```

or

```
c0 = c + 0.5*pinv(G)*(y - f(c,x));
```

which uses the Moore–Penrose pseudoinverse as discussed in Section 3.6. In both of these methods, we had to take a smaller $\alpha = \frac{1}{2}$ to ensure stability. The Levenberg–Marquardt is more stable, and we are able to take a larger α (as big 2). □

11.1. The Vandermonde Matrix

$$V = \begin{bmatrix} 1 & c_1 & c_1^2 & \dots & c_1^{n-1} \\ 1 & c_1 & c_1^2 & \dots & c_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & c_m & c_m^2 & \dots & c_m^{n-1} \end{bmatrix}$$

$$F = \operatorname{diag}\left(\begin{bmatrix} \frac{1}{0!} & \frac{1}{1!} & \frac{1}{2!} & \dots & \frac{1}{(n-1)!} \end{bmatrix}\right)$$

```
V = fliplr(vander([0 1 2 3]))
F = diag(1./factorial([0:3]))
D = inv(V*F)
```

Then `rats(D(2,:))` returns `-11/6 3 -3/2 1/3` as the coefficients

$$f'(x) \approx \frac{1}{h}\left[-\tfrac{11}{6}f(x) + 3f(x+h) - \tfrac{3}{2}f(x+2h) + \tfrac{1}{3}f(x+3h)\right].$$

Note that `D(2,:)*(0:3)'`, `D(2,:)*(0:3).^2'` , `D(2,:)*(0:3).^3'` returns 1, 0, and 0, respectively. Finally, `D(2,:)*(0:3).^4'/factorial(4)` returns 1/4, from which we have have that the truncation error $\frac{1}{4}h^3 f^{(4)}$.

Using the method of undetermined coefficients,

$$\begin{aligned} af(x) &= af(x) \\ bf(x+h) &= bf(x) + bhf'(x) + \tfrac{1}{2}bh^2 f''(x) + \tfrac{1}{6}bh^3 f'''(x) + O(h^4) \\ bf(x+2h) &= cf(x) + 2chf'(x) + \tfrac{4}{2}ch^2 f''(x) + \tfrac{8}{6}ch^3 f'''(x) + O(h^4) \\ df(x+3h) &= df(x) + 3dhf'(x) + \tfrac{9}{2}dh^2 f''(x) + \tfrac{27}{6}dh^3 f'''(x) + O(h^4) \end{aligned}$$

To get a third-order finite-difference approximation for $f'(x)$, we must eliminate the $f(x)$, $f''(x)$ and $f'''(x)$ terms and keep the $f'(x)$ term:

$$\begin{aligned} a + b + c + d &= 0 \\ b + 2c + 3d &= 1 \\ \tfrac{1}{2}b + 2c + \tfrac{9}{2}d &= 0 \\ \tfrac{1}{6}b + \tfrac{4}{3}c + \tfrac{9}{2}d &= 0 \end{aligned}$$

Solving this equations gives us $a = -\frac{11}{6}$, $b = 3$, $c = -\frac{3}{2}$, and $d = -\frac{1}{3}$ from which

$$-\tfrac{2}{3}f(x) + \tfrac{1}{4}f(x+h) + \tfrac{1}{2}f(x+2h) - \tfrac{1}{12}f(x+3h) = hf'(x) + O(h^4).$$

Therefore,

$$f'(x) \approx \frac{1}{h}\left[-\tfrac{2}{3}f(x) + \tfrac{1}{4}f(x+h) + \tfrac{1}{2}f(x+2h) - \tfrac{1}{12}f(x+3h)\right].$$

The truncation

$$O(h^4) = \tfrac{1}{24}\left[b + 16c + 81d\right]h^4 f''''(x) = \tfrac{1}{16}h^4 f''''(x)$$

is bounded by $e_T = \frac{1}{16}h^4 M_T$ where $M_T = \sup|f''''(\xi)|$ for $\xi \in (x, x+h)$. The round-off error is bounded by $e_R = 8M_R\,\mathrm{eps}/h$ where $M_R = \sup|f(\xi)|$. The total error $E(h) = e_T + e_R$ is minimized when $E'(h) = 0$:

$$E' = \tfrac{1}{4}M_T h^3 - 8\,\mathrm{eps}M_R/h^2 = 0.$$

So, the error is minimum at $h \approx 2\left(\mathrm{eps}M_R/M_T\right)^{1/5}$.

For a second-order finite-difference approximation for $f''(x)$, we have `rats(D(3,:))` returns `2  -5  4  -1` the coefficients

$$f''(x) \approx \frac{1}{h^2}\left[2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)\right].$$

To get a second-order finite difference approximation for $f''(x)$, we must eliminate the $f(x)$, $f'(x)$ and $f''(x)$ terms and keep the $f''(x)$ term. It follows that $a = -\frac{10}{9}$, $b = \frac{8}{3}$, $c = -\frac{4}{3}$, and $d = \frac{1}{9}$, and therefore,

$$f''(x) \approx \frac{1}{h}\left[-\tfrac{10}{9}f(x) + \tfrac{8}{3}f(x+h) - \tfrac{4}{3}f(x+2h) + \tfrac{1}{9}f(x+3h)\right]. \quad \square$$

11.2. We can use Theorem 38 to compute the nodes and weights for the Gauss–Legendre quadrature. Define the implicit function

```
f = @(x) exp(-16*x.^2);
```

and the Gauss–Legendre quadrature as

```
function S = gauss_legendre(f,N)
b = (1:N).^2./(4*(1:N).^2-1);
[W,x] = eig(diag(sqrt(b),1)+diag(sqrt(b),-1));
A = 2*W(1,:).^2;
S = sum(A*f(diag(x)));
```

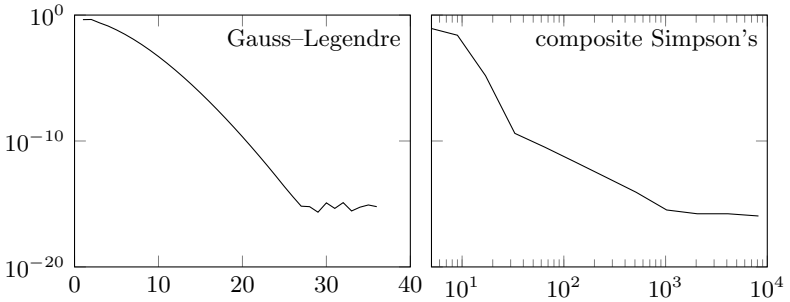


Figure S.8: Errors in Gauss–Legendre and composite Simpson's method.

For the composite Simpson's method (with N odd)

```
function S = simpsons(f,N)
h = 2/(N-1);
F = f(linspace(-1,1,N));
W = [1;repmat([4;2],(N-3)/2,1);4;1];
S = h/3*(F*W);
```

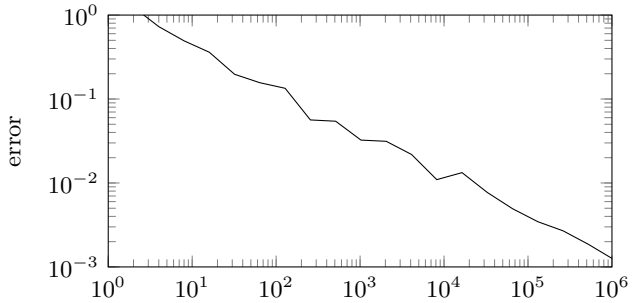
We confirm the order of the composite Simpson's method to be $O(h^4)$ by using `linspace` of the error. See Figure **??**. □

11.4. Using the MATLAB code

```
n = 10E4; p = sum(sum((rand(2,n).^2))<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;
for i = 1:20
  n(i) = 2^i;
  p = sum(sum(rand(2,n(i),m).^2)<1)*4/n(i);
  d(i) = sum(abs(pi - p))/m;
end
loglog(n,d); polyfit(log(n),log(d),1)
```



The slope of the log-log plot of error $m = -0.506$ agrees with an $O(N^{-1/2})$ convergence rate. We can compute the area of an eight dimensional unit hypersphere by a modifying our original code:

```
n = 10E6; p = sum(sum((rand(8,n).^2))<1)/n*2^8
```

We get 4.0479 which is close to the actual volume $\pi^4 r^8/24 \approx 4.059$. $\square$

Numerical Methods for Partial Differential Equations

12.1. Note that for $\theta = 1$ the θ-scheme is the forward Euler scheme, for $\theta = 0$ the θ-scheme is the backward Euler scheme, and for $\theta = \frac{1}{2}$ the θ-scheme is the trapezoidal scheme. So, the regions of stability of the θ-scheme be the same as the three schemes for $\theta = \{0, 1, \frac{1}{2}\}$. Take $f(U^n) = \gamma U^n)$ and take $r = U^{n+1}/U^n$. Then

$$\frac{r-1}{k} = (1-\theta)\gamma r + \theta\gamma$$

or equivalently (by solving for r)

$$r = \frac{1+\theta\gamma k}{1-(1-\theta)\gamma k}.$$

We want to find the conditions on the complex-valued γk so that $|r| \leq 1$. Let's find the boundary of the region γk such that $|r| = 1$. Take $\gamma = x+iy$. Then

$$\begin{aligned} 1 = |r|^2 = r\overline{r} = &= \frac{1+\theta(x+iy)}{1-(1-\theta)(x+iy)}\frac{1+\theta(x-iy)}{1-(1-\theta)(x-iy)} \\ &= \frac{1+2\theta x+\theta^2(x^2+y^2)}{1-2(1-\theta)x+(1-\theta)^2(x^2+y^2)} \end{aligned}$$

from which we have

$$1-2(1-\theta)x+(1-\theta)^2(x^2+y^2) = 1+2\theta x+\theta^2(x^2+y^2).$$

After rearranging and combining terms, we have

$$(1-2\theta)x^2 - 2x + (1-2\theta)y^2 = 0$$

or equivalently

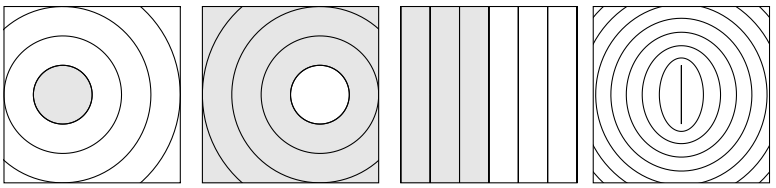
$$x^2 - \frac{2}{1-2\theta}x + y^2 = 0.$$

Completing the square give us

$$\left(x - \frac{1}{1-2\theta}\right)^2 + y^2 = \frac{1}{(1-2\theta)^2}.$$

So, the region of absolute stability is bounded by a circle centered on the real axis at $\gamma k = (1-2\theta)^{-1}$ with radius $(1-2\theta)^{-1}$, that is, tangential to the imaginary axis. Note that when $\theta \to \frac{1}{2}$, the boundary approaches the imaginary axis. □

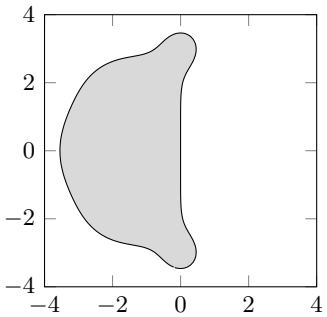
12.2. The stability contours in $[-3-3] \times [-3\mathrm{i}, 3\mathrm{i}]$ are shown below. Regions of absolute stability is are shaded.



Forward Euler Backward Euler Trapezoidal Leap Frog □

12.5. The following MATLAB 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];
n = length(b);
N = 100;
[x,y] = meshgrid(linspace(-4,4,N));
lk = x + 1i*y;
E = ones(n,1);
for i = 1:N, for j=1:N
    r(i,j) = 1+ lk(i,j) * b*(( eye(n) - lk(i,j)*A)\E);
end, end
contour(x,y,abs(r),[1 1],'k');
axis([-4 4 -4 4]);
```



□

12.9. To solve $u' = f(u) + g(u)$ with third-order L-stable IMEX method, we need to choose an L-stable implicit scheme, the BDF3, and then match it with an explicit scheme that uses the same time discretization as the BDF3. The BDF3 approximates the time derivative at U^n using U^n, U^{n-1}, U^{n-2} and U^{n-3} and uses $f(U^n)$ directly. For the explicit scheme, we'll need to approximate $f(U^n)$ using $f(U^{n-1})$, $f(U^{n-2})$, $f(U^{n-3})$ and $f(U^{n-4})$. Again, it is important to emphasize that we are evaluating the equation at time t_n in both the implicit and explicit schemes to avoid splitting error. The BDF scheme is well-known (and can be found with a quick search in books and online), but we can also easily reformulate it combining terms of the Taylor series expansion

$$\begin{aligned}
u(t) &= u \\
u(t-k) &= u - ku' + \tfrac{1}{2}k^2u'' - \tfrac{1}{6}k^3u''' + O(k^4) \\
u(t-2k) &= u - 2ku' + \tfrac{4}{2}k^2u'' - \tfrac{8}{6}k^3u''' + O(k^4) \\
u(t-3k) &= u - 3ku' + \tfrac{9}{2}k^2u'' - \tfrac{27}{6}k^3u''' + O(k^4) \\
u(t-4k) &= u - 4ku' + \tfrac{16}{2}k^2u'' - \tfrac{64}{6}k^3u''' + O(k^4)
\end{aligned}$$

to keep the u' term while eliminating the u, u'', and u''' terms. In this case, (taking the first four equations) we solve the system

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & -2 & -3 \\ 0 & 1/2 & 4/2 & 9/2 \\ 0 & -1/6 & -8/6 & -27/6 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{16.14}$$

to get the coefficients $\{11/6, -3, 3/2, -1/3\}$ for the BDF3

$$\tfrac{11}{6}U^n - 3U^{n-1} + \tfrac{3}{2}U^{n-2} - \tfrac{1}{3}U^{n-3} = kf(U^n).$$

To approximate $f(U^n)$ in the explicit scheme we keep the u term and eliminate the u', u'' and u''' terms in the Taylor series expansions. This time (taking the last four equations) we solve the system

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & -2 & -3 & -4 \\ 1/2 & 4/2 & 9/2 & 16/2 \\ -1/6 & -8/6 & -27/6 & -64/6 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{16.15}$$

to get the coefficients $\{180/61, -174/61, 52/61, 3/61\}$ giving

$$\begin{aligned}
&\tilde{g}(U^n, U^{n-2}, U^{n-3}, U^{n-4}) = \\
&\qquad g\left(\tfrac{180}{61}U^{n-1} - \tfrac{174}{61}U^{n-2} + \tfrac{52}{61}U^{n-3} - \tfrac{3}{61}U^{n-4}\right)
\end{aligned}$$

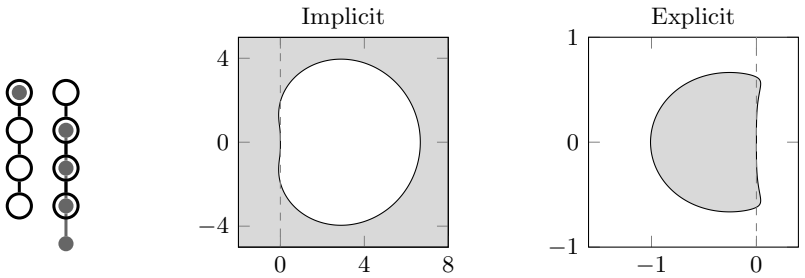


Figure S.9: The stencil and regions of absolute stability (in gray) for the implicit and explicit parts of the L-stable IMEX method.

Then

$$g(U^n) = \tilde{g}(U^{n-1}, U^{n-2}, U^{n-3}, U^{n-4}) + O(k^4)$$

Alternatively, we could use the following approximation for $g(U^n)$:

$$\tfrac{180}{61} f\left(U^{n-1}\right) - \tfrac{174}{61} f\left(U^{n-2}\right) + \tfrac{52}{61} f\left(U^{n-3}\right) - \tfrac{3}{61} f\left(U^{n-4}\right) + O(k^4).$$

Combining the implicit and explicit methods gives us

$$\begin{aligned} &\tfrac{11}{6} U^n - 3U^{n-1} + \tfrac{3}{2} U^{n-2} - \tfrac{1}{3} U^{n-3} = \\ &\qquad k\left[f(U^n) + \tilde{g}(U^{n-1}, U^{n-2}, U^{n-3}, U^{n-4})\right]. \end{aligned}$$

The region of stability for the BDF3 is known from Figure 12.2. To compute the regions of stability for the explicit scheme we take $g(u) = \gamma u$ and let $r = U^n/U^{n-1}$ and find when $|r| \le 1$. Then

$$\tfrac{11}{6} r^4 - 3r^3 + \tfrac{3}{2} r^2 - \tfrac{1}{3} r = \gamma k \left(\tfrac{180}{61} r^3 - \tfrac{174}{61} r^2 + \tfrac{52}{61} r - \tfrac{3}{61}\right)$$

Solving for γk gives

$$\gamma k = \frac{671r^4 - 1098r^3 + 549r^2 - 366r}{1080r^3 - 1044r^2 + 312r - 18}.$$

We find the boundary of the domain $|r| \le 1$ by taking $r = \mathrm{e}^{i\theta}$. Regions of stability are shown in Figure S.9. □

12.11. Taking $\alpha = \sum_{j=1}^{s} b_j U^{n-j}$, $\beta = (b_0^*)^{-1} \sum_{j=1}^{s} b_j^* U^{n-j}$, and $f(u) = \gamma u$ in the Adams–Bashforth–Moulton PECE equation (12.9) we have

$$\tilde{U}^n = U^{n-1} + \gamma k \alpha \tag{16.16a}$$

$$U^n = U^{n-1} + \gamma k b_0^* (\tilde{U}^n + \beta). \tag{16.16b}$$

Substituting (16.16a) into (16.16b) we have

$$U^n = U^{n-1} + M(U^n + M\alpha + \beta)$$

where $M = \gamma k b_0^*$. We continue by substituting this new expression in for $\tilde{U}^n$ in (16.16b) giving us

$$U^n = U^{n-1} + (M + M^2)(U^{n-1} + \beta) + M^3\alpha.$$

Continuing

$$U^n = U^{n-1} + (M + M^2 + \cdots + M^s)(U^{n-1} + \beta) + M^{s+1}\alpha.$$

Letting $r = U^{n-1}/U^n$, then

$$1 = r + (M + M^2 + \cdots + M^s)(r + \beta(r)) + M^{s+1}\alpha(r) \tag{16.17}$$

where $\alpha(r) = \sum_{j=1}^{s} b_j r^j$ and $\beta(r) = \sum_{j=1}^{s} b_j^* r^j$. We want to find the boundaries when $|r| = 1$. So, we take $r = \exp(\mathrm{i}\theta)$ and solving (16.17) by find the roots.

```
function z = PECE(n,m)
[AB,AM] = ABMcoeffs;
a = AB(1:10,n);
b = AM(1:10,n+1); b0 = b(1); b = b/b0; a = a/b0;
for i = 1:300
  r = 1*exp(2i*pi*(i/300));
  R = r.^[1:n];
  c(1) = r - 1;
  c(2:m) = r + R*b(2:n+1);
  c(m+1) = R*a(2:n+1);
  z(i,:) = roots(fliplr(c))'/b0;
end
function [AB,AM] = ABMcoeffs
j = [1:12];
for i = 1:12,
  V1(i,j) = -j.^i;
  V2(i,j) = i.*j.^(i-1);
end
for p = 1:10
  s = 1:p;
  AM(s,p) = V2(s,s)\V1(s,1:2)*[1;-1];
  AB(s+1,p) = V2(s,s+1)\V1(s,1:2)*[1;-1];
end
```

After splicing the solution together and snipping off unwanted sections of the curve, we get the regions in Figure S.10 on the next page. □

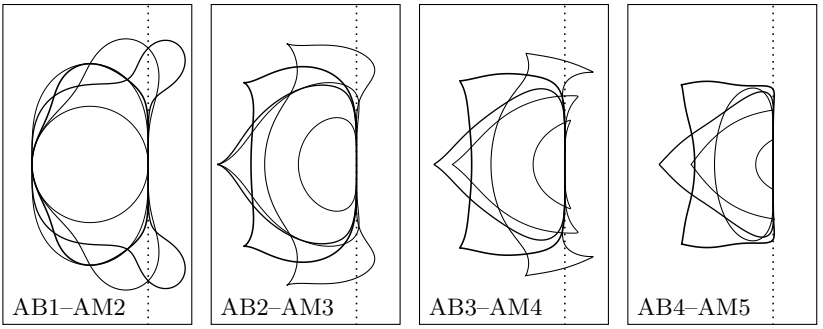


Figure S.10: Regions of stability for Adams–Bashforth–Moulton PE(CE)^m methods for $s = 0$ (thin curve) to $m = 4$ (thick curve). The domain is $[-2.5, .75] \times [-2.75, 2.75]$.

13.2. Take the Taylor series approximation

$$\begin{aligned}
U_j^n &: \quad u(t,x) = u \\
U_j^{n+1} &: \; u(t+k,x) = u + ku_t + \tfrac{1}{2}k^2 u_{tt} + O(k^3) \\
U_{j+1}^n &: \; u(t,x+k) = u + hu_x + \tfrac{1}{2}h^2 u_{xx} + \tfrac{1}{6}h^3 u_{xxx} + \tfrac{1}{24}h^4 u_{xxxx} + O(h^5) \\
U_{j-1}^n &: \; u(t,x-k) = u - hu_x + \tfrac{1}{2}h^2 u_{xx} - \tfrac{1}{6}h^3 u_{xxx} + \tfrac{1}{24}h^4 u_{xxxx} - O(h^5)
\end{aligned}$$

Substituting approximations into the Crank–Nicolson scheme

$$\frac{U_j^{n+1} - U_j^n}{k} = \frac{U_{j+1}^n - 2U_j^n + U_{j-1}^n}{h^2}$$

gives

$$u_t + \tfrac{1}{2}k + O(k^2) = u_{xx} + \tfrac{1}{12}h^2 u_{xxxx} + O(h^3)$$

which says that the Crank–Nicolson is $O(k + h^2)$ to $u_t = u_{xx}$. □

13.3. Making the substitution

$$U_{j+2}^n = \hat{u}(t_n, \xi)\mathrm{e}^{\mathrm{i}jh\xi}$$

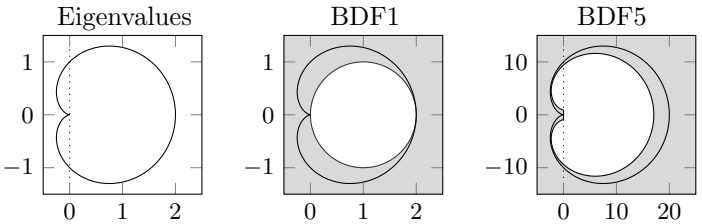
in the expression

$$\frac{U_{j+2} - 2U_{j+1} + U_{j-1}}{h^2}$$

gives us

$$\frac{1}{h^2}\left(\mathrm{e}^{\mathrm{i}2h\xi} - 2\mathrm{e}^{\mathrm{i}h\xi} + 1\right)\hat{u} = \frac{1}{h^2}\mathrm{e}^{\mathrm{i}h\xi}\left(\mathrm{e}^{\mathrm{i}h\xi} - \mathrm{e}^{-\mathrm{i}h\xi}\right)^2\hat{u} = -\frac{2}{h^2}\mathrm{e}^{\mathrm{i}h\xi}\sin^2\frac{\xi h}{2}\hat{u}$$

as the right hand side of $\hat{u}_t = \gamma\hat{u}$. The eigenvalues form a cardoid.



Examining the regions of stability for the methods in Figures 12.2, 12.3, and 12.5, we see that BDF1–BDF5 are the only time-difference schemes that would be absolutely stable and only if we take a sufficiently *large* timestep. For example, for the backward Euler (BDF1) we need to take $k > h^2/2$ and for BDF5 methods we need to take $k > 5h^2$. Every other method we've examined is unconditionally unstable. □

13.5. The constant-potential Schrödinger equation can be written as

$$\frac{\partial \psi}{\partial t} = \mathrm{i}\tfrac{1}{2}\varepsilon \frac{\partial^2 \psi}{\partial x^2} - \mathrm{i}\varepsilon^{-1} V \psi.$$

Using (13.9) we have that

$$\frac{\partial \hat{u}}{\partial t} = -\mathrm{i}\frac{2\varepsilon}{h^2}\sin^2\frac{\xi h}{2}\hat{u} - \mathrm{i}\varepsilon^{-1} V \hat{u}. \tag{16.18}$$

So, the eigenvalues μ_ξ of the system $\hat{u}_t = \mu_\xi \hat{u}$ are

$$\mu_\xi = -\mathrm{i}\left(\frac{2\varepsilon}{h^2}\sin^2\frac{\xi h}{2} + \varepsilon^{-1} V \hat{u}\right).$$

These eigenvalues all lie along the imaginary axis and are bounded by $|\mu_\xi| \le 2\varepsilon/h^2 + \varepsilon^{-1}|V|$. Note where the regions of absolute stability intersect the imaginary axis for different methods in Figure 12.1 on page 268. We see that backward Euler and trapezoidal methods are unconditionally stable, that the leapfrog method is conditionally stable with condition $|\mu_\xi| \le 1$, and that the forward Euler is unconditionally unstable. From Figure 12.5 on page 281 we see that RK4 is conditionally stable when $|\mu_\xi| \le 2.5$. From Figures 12.2 and 12.3, we see that BDF2 is unconditionally stable and that higher-order BDF and Adams methods are conditionally stable.

The following MATLAB code solves the Schrödinger equation with $V(x) = \frac{1}{2}x^2$ using a Crank–Nicolson method over the domain $[-3, 3]$ with initial conditions $\psi(0, x) = \mathrm{e}^{-(x-1)^2/2\varepsilon}$.

```
function psi = schrodinger(n,m,eps)
x = linspace(-3,3,n)'; dx = x(2)-x(1); dt = pi/m;
V = x.^2/2;
tridiag = repmat([1 -2 1],[n 1]);
D =(0.5i*eps)*spdiags(tridiag,[-1 0 1],n,n)/dx^2 - ...
   (1i/eps)*spdiags(V,0,n,n);
D(1,2)=2*D(1,2); D(end,end-1)=2*D(end,end-1);
B = (speye(n) + (dt/2)*D);
C = (speye(n) - (dt/2)*D);
psi = exp(-(x-1).^2/(2*eps));
for i = 1:m
  psi = C\(B*psi);
end
```

When the mesh spacing h or the time step k are about equal to or larger than ε the method no longer resolves the dynamics and we get a wrong solution. Specifically the numerical solution moves too slow and doesn't keep the form of a traveling wave. □

13.6. We'll start by expanding the derivative

$$\frac{\partial u}{\partial t} = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right)$$

to get

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r}. \tag{16.19}$$

At $r = 0$ we have division by zero in the right-hand side of the equation. We'll need to fix this. Taking the Taylor series expansion of $u(t, r)$ at $r = 0$ and noting the $u(t, r)$ is an even function, we have that

$$\begin{aligned} u(t,r) &= u(t,0) + ru_r(t,0) + \tfrac{1}{2}r^2 u_{rr}(t,0) + \tfrac{1}{6}r^3 u_{rrr}(t,0) + O(r^4) \\ &= u(t,0) + \tfrac{1}{2}r^2 u_{rr}(t,0) + O(r^4). \end{aligned}$$

Then

$$\left.\frac{1}{r}\frac{\partial u}{\partial r}\right| = u_{rr}(t,0) + O(r^4) \to u_{rr}(t,0) \text{ as } r \to 0.$$

So, at $r = 0$ we have

$$\frac{\partial u}{\partial t} = 2\frac{\partial^2 u}{\partial r^2}. \tag{16.20}$$

Using the method of lines we have that $\frac{\partial}{\partial t}U_j = \mathrm{A}\,U_j$ where

$$\mathrm{A}\,U_j = \frac{U_{j+1} - 2U_j + U_{j+1}}{h^2} + \frac{1}{r_j}\frac{U_{j+1} - U_{j-1}}{2h} \text{ from (16.19) and}$$

$$\mathrm{A}\,U_0 = 2\frac{U_1 - 2U_0 + U_{-1}}{h^2} \text{ from (16.20).}$$

Now, we worry about with the boundary conditions. An insulating boundary conditions at $r = 1$ means that $u_r(1) = 0$. Using a second-order centered-difference approximation fro the derivative, this means $U_{N+1} - U_{N-1} = 0$ where we have taken a ghost point at $j = N + 1$. We'll use this condition to replace U_{N+1} with U_{N-1} in the system. The derivative is also zero at the origin—this time because of symmetry. So, we will replace U_{-1} with U_1 in the system. All together,

$$\begin{aligned}
\mathrm{A}\,U_0 &= 4\frac{U_1 - U_0}{h^2} \\
\mathrm{A}\,U_j &= \frac{U_{j+1} - 2U_j + U_{j+1}}{h^2} + \frac{1}{r_j}\frac{U_{j+1} - U_{j+1}}{2h} \\
\mathrm{A}\,U_N &= 2\frac{U_{N-1} - U_N}{h^2}
\end{aligned}$$

The Crank–Nicolson method is then

$$U_j^{n+1} - U_j^n = k(\tfrac{1}{2}\,\mathrm{A}\,U_j^{n+1} + \tfrac{1}{2}\,\mathrm{A}\,U_j^n)$$

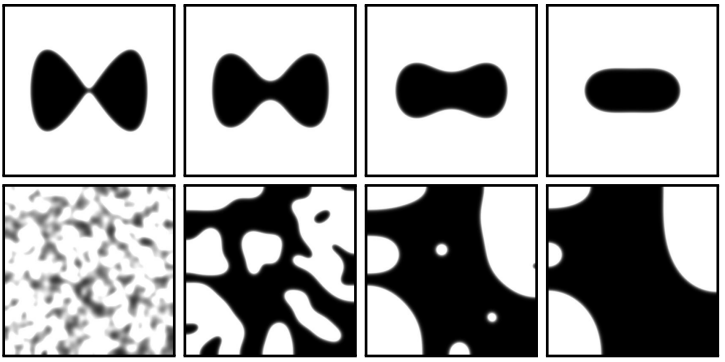
which can be solved as

$$U_j^{n+1} = (\mathrm{I} - \tfrac{1}{2}k\,\mathrm{A})^{-1}(\mathrm{I} + \tfrac{1}{2}k\,\mathrm{A})U_j^n.$$

The following MATLAB code implements the method with reflecting boundary conditions over the domain $[0, 1]$ with initial conditions $u(0, r) = \tanh(4 - 8r)$:

```
function u = polarheat(n,m)
r = linspace(0,1,n)'; dr = r(2)-r(1); dt = 1/m;
tridiag = repmat([1 -2 1],[n 1]);
A1 = spdiags(tridiag,[-1 0 1],n,n)/dr^2;
A2 = spdiags([-1./r 1./r],[-1 1],n,n)'/(2*dr);
A = A1 + A2;
A(1,[1 2])=4*[-1 1]/dr^2;
A(n,[n-1 n])=2*[1 -1]/dr^2;
B = (speye(n) + (dt/2)*A);
C = (speye(n) - (dt/2)*A);
u = (tanh(4-8*r));
for i = 1:m
  u = C\(B*u);
end
```

□



$t = 0.05$ $t = 0.5$ $t = 2.0$ $t = 4.0$

Figure S.11: Allen–Cahn solutions for two different initial conditions.

13.7. The following MATLAB code produces the solutions to the Allen–Cahn equation shown in Figure S.11.

```
function allencahn
L = 16; N = 400; M = 8000; T = 4; epsilon =1/50;
h = L/N; k = T/M; lambda = k/h^2;
[X,Y] = meshgrid(linspace(-L/2,L/2,N));
u = (X.^4 < 32*X.^2-16*Y.^2)*2-1;
%u = randn(size(X));
[n m] = size(u);
Q = spdiags(repmat([1 -2 1],[n 1]),-1:1,n,n)*lambda ;
A = (speye(n) + 0.5*Q);
B = (speye(n) - 0.5*Q);
A(1,2) = lambda; A(n,n-1) = lambda; %reflecting BC
B(1,2) = -lambda; B(n,n-1) = -lambda; %reflecting BC
for i = 1:M
  u = u./sqrt(u.^2 - (u.^2-1).*exp(-k/epsilon));
  u = B\(A*u); u = (B\(A*u'))';
  u = u./sqrt(u.^2 - (u.^2-1).*exp(-k/epsilon));
  image((u+1)/2*100);colormap(1-gray(100)); drawnow;
end
```

□

14.1. The Lax–Friedrichs scheme is given by

$$\frac{U_j^{n+1} - \frac{1}{2}(U_{j+1}^n + U_{j-1}^n)}{k} + c\frac{U_{j+1}^n - U_{j-1}^n}{2h} = 0.$$

Substitute the Taylor series approximation

$$\begin{aligned}
u(t,x) &= u \\
u(t+k,x) &= u + ku_t + \tfrac{1}{2}k^2u_{tt} + \tfrac{1}{6}k^3u_{ttt} + O(k^4) \\
u(t,x+h) &= u + hu_x + \tfrac{1}{2}h^2u_{xx} + \tfrac{1}{6}h^3u_{xxx} + O(h^4) \\
u(t,x-h) &= u - hu_x + \tfrac{1}{2}h^2u_{xx} - \tfrac{1}{6}h^3u_{xxx} + O(h^4)
\end{aligned}$$

for U_j^n, U_j^{n+1}, U_{j+1}^n, and U_{j-1}^n. The terms of the Lax–Friedrichs scheme are

$$\frac{U_j^{n+1} - \frac{1}{2}(U_{j+1}^n + U_{j-1}^n)}{k} = u_t + \tfrac{1}{2}ku_{tt} + O(k^2) - \tfrac{1}{2}\frac{h^2}{k}u_{xx} + O(h^4)$$

and

$$c\frac{U_{j+1}^n - U_{j-1}^n}{2h} = cu_x + \tfrac{1}{6}ch^2u_{xxx} + O(h^4).$$

Combining the terms gives us

$$u_t + cu_x + \tfrac{1}{2}ku_{tt} - \tfrac{1}{2}\frac{h^2}{k}u_{xx} + \tfrac{1}{6}ch^2u_{xxx} + \{\text{higher order terms}\} = 0$$

For $k \ll 1$, we have that $|c|h \ll h^2/k$ and so the truncation error is $O(k + h^2/k)$.

To find the dispersion relation, we take a look at the leading order truncation

$$u_t + cu_x + \tfrac{1}{2}ku_{tt} - \tfrac{1}{2}\frac{h^2}{k}u_{xx} = 0 \tag{16.21}$$

We will eliminate the u_{tt} term by using differentiating (16.21) with respect to t and to x and substituting the expression for u_{tt} back into the original equation and discarding the higher order terms.

$$\begin{aligned}
u_{tt} &= -cu_{xt} + O(k + h^2/k) \\
u_{tx} &= -cu_{xx} + O(k + h^2/k)
\end{aligned}$$

from which

$$u_{tt} = c^2u_{xx} + O(k + h^2/k)$$

After eliminating higher order terms, (16.21) becomes

$$u_t + cu_x + \tfrac{1}{2}c^2ku_{xx} - \tfrac{1}{2}\frac{h^2}{k}u_{xx} = 0.$$

or equivalently

$$u_t + cu_x + \tfrac{1}{2}k\left(c^2 - \frac{h^2}{k^2}\right)u_{xx} = 0.$$

Substituting the Fourier component

$$u(t,x) = \mathrm{e}^{\mathrm{i}(\omega t - \xi x)}$$

as an ansatz gives us

$$\left[\mathrm{i}\omega + c(-\mathrm{i}\xi) - \tfrac{1}{2}\left(c^2 k - \frac{h^2}{k^2}\right)\xi^2\right]\mathrm{e}^{\mathrm{i}(\omega t - \xi x)} = 0.$$

So,

$$i\omega + c(-\mathrm{i}\xi) - \tfrac{1}{2}\left(c^2 k - \frac{h^2}{k^2}\right)\xi^2 = 0$$

and the dispersion relation $\omega(\xi)$ is given by

$$\omega = c\xi - \mathrm{i}\tfrac{1}{2}k\left(c^2 - \frac{h^2}{k^2}\right)\xi^2.$$

Plugging this ω back into our ansatz gives us

$$\begin{aligned} u(t,x) &= \mathrm{e}^{\mathrm{i}(\omega t - \xi x)} \\ &= \mathrm{e}^{\mathrm{i}c\xi t}\mathrm{e}^{-\alpha\xi^2}\mathrm{e}^{-\mathrm{i}\xi x} \\ &= \underbrace{\mathrm{e}^{\mathrm{i}\xi(ct-x)}}_{\text{advection}} \cdot \underbrace{\mathrm{e}^{-\alpha\xi^2 t}}_{\text{dissipation}} \end{aligned}$$

where

$$\alpha = \tfrac{1}{2}k\left(\frac{h^2}{k^2} - c^2\right)$$

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 $\mathrm{e}^{-\alpha\xi^2 t}$ grows and the solution blows up, which is in agreement with 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

$$\begin{aligned}
u(t+k,x) &= u + ku_t + \tfrac{1}{2}k^2u_{tt} + \tfrac{1}{6}k^3u_{ttt} + O(k^4)\\
u(t-k,x) &= u - ku_t + \tfrac{1}{2}k^2u_{tt} - \tfrac{1}{6}k^3u_{ttt} + O(k^4)\\
u(t,x+h) &= u + hu_x + \tfrac{1}{2}h^2u_{xx} + \tfrac{1}{6}h^3u_{xxx} + O(h^4)\\
u(t,x-h) &= u - hu_x + \tfrac{1}{2}h^2u_{xx} - \tfrac{1}{6}h^3u_{xxx} + O(h^4)
\end{aligned}$$

From which we have that

$$u_t + \tfrac{1}{2}k^2u_{ttt} + cu_x + \tfrac{1}{2}h^2u_{ttt} = O(k^2+h^2).$$

So the method is $O(k^2+h^2)$.

The scheme is leap frog in time and centered-difference in space:

$$\frac{U_j^{n+1} - U_j^{n-1}}{2k} = -c\frac{U_{j+1}^n - U_{j-1}^n}{2h}$$

The Fourier transform of

$$\frac{\partial}{\partial t}U_j = -c\frac{U_{j+1} - U_{j-1}}{2h}$$

is

$$\frac{\partial}{\partial t}\hat{u}(t,\xi) = -c\left(\mathrm{e}^{\mathrm{i}\xi h} - \mathrm{e}^{-\mathrm{i}\xi h}\right)\hat{u}(t,\xi) = -\mathrm{i}\frac{c}{h}\sin(\xi h)\hat{u}(t,\xi)$$

Recall that the leapfrog scheme is absolute stability only on the imaginary axis $\gamma k \in [-\mathrm{i}, +\mathrm{i}]$. The factor $|\sin(\xi h)| \le 1$, so the scheme is stable when $(|c|/h)k \le 1$. That is, when $k \le h/|c|$. □

14.5. We can express the compressible Euler equations

$$\rho_t + (\rho u)_x = 0 \tag{16.22a}$$

$$(\rho u)_t + (\rho u^2 + p)_x = 0 \tag{16.22b}$$

$$E_t + ((E+p)u)_x = 0 \tag{16.22c}$$

in semilinear form using ρ, u and p as independent variables. First, rewrite (16.22a) and (16.22b) as

$$\rho_t + u\rho_x + \rho u_x = 0 \tag{16.23}$$

$$u\rho_t + \rho u_t + u^2\rho_x + 2\rho u u_x + p_x = 0 \tag{16.24}$$

We can eliminate the $u\rho_t$ term in (16.24) by multiplying (16.23) by u:

$$u\rho_t + u^2\rho_x + \rho u u_x = 0$$

and subtracting the equation from (16.24):

$$\rho u_t + \rho u u_x + p_x = 0.$$

Dividing by ρ gives

$$u_t + u u_x + \frac{1}{\rho} p_x = 0 \tag{16.25}$$

To derive an equation for p, we will use the equation of state

$$E = \tfrac{1}{2}\rho u^2 + \frac{p}{\gamma - 1} \tag{16.26}$$

to eliminate E from (16.22c). Differentiating (16.26) with respect to t:

$$E_t = \tfrac{1}{2} u^2 \rho_t + \rho u u_t + \frac{1}{\gamma - 1} p_t$$

and substituting (16.23) and(16.25) in for ρ_t and u_t:

$$E_t = \tfrac{1}{2} u^2 \left(-u\rho_x - \rho u_x\right) + \rho u \left(-u u_x - \frac{1}{\rho} p_x\right) + \frac{1}{\gamma - 1} p_t$$

which we can simplify to

$$E_t = -\tfrac{1}{2} u^3 \rho_x - \tfrac{3}{2}\rho u^2 u_x - u p_x + \frac{1}{\gamma - 1} p_t. \tag{16.27}$$

Expanding $((E + p)u)_x$ with the equation of state:

$$\begin{aligned} ((E + p)u)_x &= \left(\tfrac{1}{2} u^3 \rho + \frac{\gamma}{\gamma - 1} p u\right)_x \\ &= \tfrac{1}{2} u^3 \rho_x + \tfrac{3}{2}\rho u^2 u_x + \frac{\gamma}{\gamma - 1} p u_x + \frac{\gamma}{\gamma - 1} u p_x \end{aligned} \tag{16.28}$$

Combining (16.27) and (16.28):

$$E_t + ((E + p)u)_x = \frac{1}{\gamma - 1} p_t + \frac{\gamma}{\gamma - 1} p u_x + \frac{1}{\gamma - 1} u p_x$$

So,

$$p_t + \gamma p u_x + u p_x = 0 \tag{16.29}$$

We now can express (16.23), (16.25) and (16.29) 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 λ of the Jacobian matrix are real:

$$\begin{vmatrix} u-\lambda & \rho & 0 \\ 0 & u-\lambda & 1/\rho \\ 0 & \gamma p & u-\lambda \end{vmatrix} = (u-\lambda)^3 - \gamma p \rho^{-1}(u-\lambda)$$
$$= (u-\lambda)\left[(u-\lambda)^2 - c^2\right] = 0$$

with $c^2 = \gamma p/\rho$. So, the eigenvalues $\lambda = \{u,\, u+c,\, u-c\}$ where c is the sound speed. □

14.7. The solution will start with a shock starting at $x = 1$ moving with speed $\frac{1}{2}$ (half the height, as given by the Rankine–Hugoniot condition). At the same time a rarefaction propagates from $x = 0$ moving at speed 1 until the rarefaction catches up with the shock at location $x = 2$ and time $t = 2$. We can image a the trailing edge of a right-angled trapezoid catching up to the leading edge to form a right triangle. Now, the leading edge of the right triangle continues to move forward with speed equal to half the height (again by the Rankine–Hugoniot condition). To compute the position of the leading edge, we only need to know the height $u(t)$. The area of the triangle is conserved, so $A = \frac{1}{2}x(t)u(t,x) = 1$. That is, $u = 2/x$. The speed is given by

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \tfrac{1}{2}u(t,x) = \frac{1}{x}$$

The solution to this differential equation is

$$x(t) = \sqrt{2t + C}$$

for some constant C. At $t = 2$, the leading edge is at $x = 2$, and $C = 0$. Putting everything together we have

$$\text{when } t < 2 \qquad u(x,t) = \begin{cases} x/t, & 0 < x < t \\ 1, & t < x < 1 + \frac{1}{2}t \\ 0, & \text{otherwise} \end{cases}$$

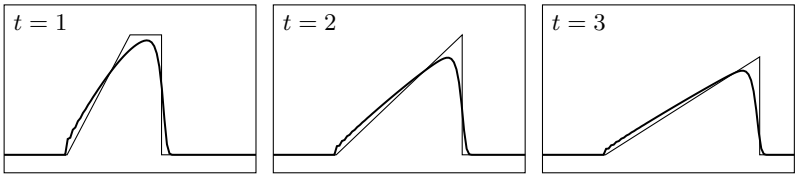
and

$$\text{when } t \geq 2 \qquad u(x,t) = \begin{cases} x/t, & 0 < x < \sqrt{2t} \\ 0, & \text{otherwise} \end{cases}$$

because the area is conserved. The local Lax–Friedrichs method can be implemented in MATLAB using the following code:

```
n = 100; x = linspace(-1,3,n); 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
```

The analytical solution and the numerical solution are as follows:



□

14.7. The Nessyahu–Tadmor scheme

$$
\begin{aligned}
U_j^{n+1/2} &= U_j^n - \frac{k}{2}\partial_x f(U_j^n) \\
U_{j+1/2}^{n+1} &= \tfrac{1}{2}(U_j^n + U_{j+1}^n) - \tfrac{1}{8}h(\partial_x U_{j+1}^n - \partial_x U_j^n) \\
&\qquad - \frac{k}{h}\left[f(U_{j+1}^{n+1/2}) - f(U_j^{n+1/2})\right]
\end{aligned}
$$

where we approximate ∂_x using the slope limiter σ_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

$$
\phi(\theta) = \frac{|\theta| + \theta}{1 + |\theta|}
$$

```
function dambreak
F = @(u) [u(:,1).*u(:,2), u(:,1).*u(:,2).^2+0.5*u(:,1).^2];
n = 1000; x = linspace(-.5,.5,n)'; dx = x(2)-x(1);
Lt = 0.25; N = ceil(Lt/(dx/2)); dt = (Lt/N)/2;
u = [0.8*(x<0)+0.2,0*x];
for i = 1:N
```

```
v = u-0.5*(dt/dx)*slope(F(u));
u(2:n,:)=(u(1:n-1,:)+u(2:n,:))/2-...
   diff(slope(u))/8-(dt/dx)*diff(F(v));
v = u-0.5*(dt/dx)*slope(F(u));
u(1:n-1,:)=(u(1:n-1,:)+u(2:n,:))/2-...
   diff(slope(u))/8-(dt/dx)*diff(F(v));
plot(x,u(:,1)); drawnow;
end

function s = slope(u)
limiter = @(t) (abs(t)+t)./(1+abs(t)); %van leer
%limiter = @(t) 0; %1st order
du = diff(u);
s = [[0 0];du(2:end,:).*limiter(du(1:end-1,:)./(du(2:end,:)+ ...
    (du(2:end,:)==0)));[0 0]];
```

Other slope limiters superbee and the minmod

```
limiter = @(t) max(0,max(min(2*t,1),min(t,2))); %superbee
limiter = @(t) max(0,min(1,t)); %minmod
```

□

15.1. While we we can get away with not using elements on the boundary for Dirichlet boundary conditions, we must put a half-element on the boundary for Neumann boundary conditions. By multiplying by a test function and integrating over the domain, we have

$$\int_0^1 -u'' - uv \,\mathrm{d}x = \int_0^1 vf \,\mathrm{d}x$$

where $f = -8x^2$. We can write the problem $a(u_h, v_h) = b(v_h)$ if we integrate by parts once where

$$a(u, v) = \int_0^1 u'v' - uv \,\mathrm{d}x \quad \text{and} \quad b(v) = \int_0^1 vf \,\mathrm{d}x + v(1)u'(1) - v(0)u'(0).$$

We take

$$u_h(x) = \sum_{i=0}^{m+1} \xi_j \phi_j(x) \quad \text{and} \quad v_h(x) = \sum_{i=0}^{m+1} v_j \phi_j(x).$$

Then after a bit of manipulation we are left with

$$\sum_{i=0}^{m+1} \xi_i a(\phi_j, \phi_i) = b(\phi_j)$$

for all j. Note that the boundary terms $v(1)u'(1)$ is zero except for ϕ_{m+1} because the every other basis element is zero at $x = 1$. That is, $v(1) = v_j\phi_j(1)$. A similar property holds for $v(0)u'(0)$. So, we have

$$b(\phi_j) = \int_0^1 \phi_j f \, \mathrm{d}x + u'(1)\delta_{j,m+1} - u'(0)\delta_{j,0}.$$

In our problem $u'(0) = 0$ and $u'(1) = 1$.

We now need to solve the system $\mathbf{A}\xi = \mathbf{f}$. We can compute the inner products in Mathematica

```
P1=1+x/h;P2=1-x/h;
f=-8*(x+s)^2;
Integrate[D[P1,x]^2-P1^2,{x,-h,0}]
Integrate[D[P1,x]*D[P2/.x->(x+h),x]-P1*(P2/.x->(x+h)),{x,-h,0}]
Integrate[P2*f/.s->0,{x,0,h}]
Integrate[P1*f,{x,-h,0}]+Integrate[P2*f,{x,0,h}]
Integrate[P1*f/.s->1,{x,-h,0}]
DSolve[{-u''[x]-u[x]==-8x^2,u'[0]==0,u'[1]==1},u,x]
```

Note that we must use half elements to compute a_{11}, a_{nn}, f_1, and f_n. We solve the problem in MATLAB using

```
n=10;
x=linspace(0,1,n)';h=x(2)-x(1);
A=spdiags(repmat([-1/h-h/62/h-2*h/3-1/h-h/6],[n1]),-1:1,n,n);
A(1,1)=A(1,1)/2;A(n,n)=A(n,n)/2;
b=[-2/3*h^3;-4/3*h^3-8*h*x(2:n-1).^2;-4*h+8*h^2/3-2*h^3/3];
b(n)=b(n)+1;
u=A\b;
s=(-16)+8.*x.^2+15.*cos(x).*csc(1);
plot(x,s,'o-',x,u,'.-');
```

The solution is plotted in Figure S.12. Note that we get very little error even with only 10 nodes. □

15.2. By multiplying by the test function v and integrating over the domain gives

$$\int_0^1 u''' v \, \mathrm{d}x = \int_0^1 f v \, \mathrm{d}x$$

We integrate by parts twice so that we have a symmetric bilinear form

$$\int_0^1 u'' v'' \, \mathrm{d}x = \int_0^1 f v \, \mathrm{d}x$$

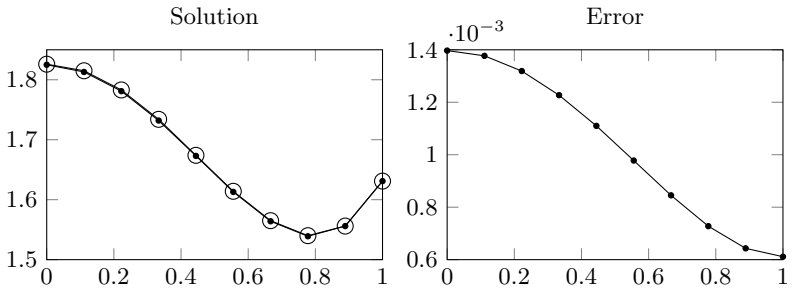


Figure S.12: Solutions to the Neumann problem 15.1. The marker • is used for the finite element solution and ∘ for the analytical solution.

Define $a(u, v) = \int_0^1 u''v'' \,\mathrm{d}x$ and $b(v) = \int_0^1 fv \,\mathrm{d}x$. Our finite element problem: find $u_h \in V_h$ such that $a(u_h, v_h) = b(v_h)$ for all $v_h \in V_h$ where

$$u_h(x) = \sum_{i=0}^{m+1} \xi_i \phi_i(x) + \eta_i \psi_i(x)$$
$$v_h(x) = \sum_{j=0}^{m+1} \alpha_j \phi_j(x) + \beta_j \psi_j(x)$$

where ξ_j and η_j are unknown and α_j and β_j are arbitrary. The basis elements $\phi_j(x)$ are used to prescribe the value of $u(x)$ at the nodes and the basis elements $\psi_j(x)$ are used to prescribe the value of the derivative of $u(x)$ at the nodes. From the boundary conditions we have that $\xi_0 = \xi_{m+1} = \eta_0 = \eta_{m+1} = 0$, so we can reduce the dimension of our problem

$$u_h(x) = \sum_{i=1}^{m} \xi_i \phi_i(x) + \eta_i \psi_i(x)$$
$$v_h(x) = \sum_{j=1}^{m} \alpha_j \phi_j(x) + \beta_j \psi_j(x).$$

Let's plug u_h and v_h into the finite element problem $a(u_h, v_h) = b(v_h)$:

$$a\left(\sum_{i=1}^{m} \xi_i \phi_i + \eta_i \psi_i, \sum_{j=1}^{m} \alpha_j \phi_j + \beta_j \psi_j\right) = b\left(\sum_{j=1}^{m} \alpha_j \phi_j + \beta_j \psi_j\right).$$

By bilinearity and homogeneity we have

$$\sum_{j=1}^{m} \alpha_j a\left(\sum_{i=1}^{m} \xi_i \phi_i + \eta_i \psi_i,\ \phi_j\right) + \beta_j a\left(\sum_{i=1}^{m} \xi_i \phi_i + \eta_i \psi_i,\ \eta_j\right) = \sum_{j=1}^{m} \alpha_j b(\phi_j) + \beta_j b(\psi_j)$$

Because α_j and β_j are arbitrary, we must have

$$a\left(\sum_{i=1}^{m} \xi_i \phi_i + \eta_i \psi_i,\ \phi_j\right) = b(\phi_j)$$
$$a\left(\sum_{i=1}^{m} \xi_i \phi_i + \eta_i \psi_i,\ \psi_j\right) = b(\psi_j)$$

for all $j = 1, 2, \ldots, m$. Finally, by bilinearity and homogeneity we have

$$\sum_{i=1}^{m} \xi_i a(\phi_i,\ \phi_j) + \eta_i a(\psi_i,\ \phi_j) = b(\phi_j)$$
$$\sum_{i=1}^{m} \xi_i a(\phi_i,\ \psi_j) + \eta_i a(\psi_i,\ \psi_j) = b(\psi_j)$$

We have a system of $2m$ equations which we can write as the block matrix

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{C} \end{bmatrix} \begin{bmatrix} \xi \\ \eta \end{bmatrix} = \begin{bmatrix} \mathbf{f}^{(1)} \\ \mathbf{f}^{(2)} \end{bmatrix}$$

where

$$a_{ij} = \int_0^1 \phi_i'' \phi_j''\, \mathrm{d}x,$$
$$b_{ij} = \int_0^1 \psi_i'' \phi_j''\, \mathrm{d}x,$$
$$c_{ij} = \int_0^1 \psi_i'' \psi_j''\, \mathrm{d}x,$$
$$f_j^{(1)} = \int_0^1 f \phi\, \mathrm{d}x = \frac{1}{h},$$
$$f_j^{(2)} = \int_0^1 f \psi\, \mathrm{d}x = 0$$

We can compute these inner products in Mathematica using

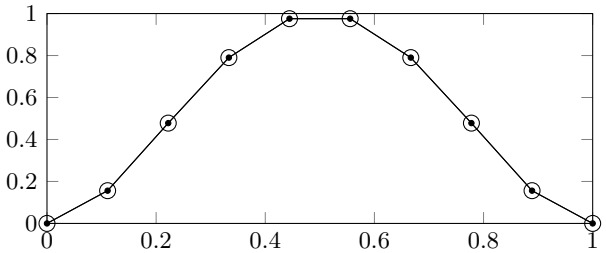


Figure S.13: Solutions to the beam problem. The marker • is used for the finite element solution and ∘ for the analytical solution.

```
P1=D[-2t^3-3t^2+1,{t,2}]/h^2;
P2=D[2t^3-3t^2+1,{t,2}]/h^2;
S1=D[t^3+2t^2+t,{t,2}]/h^2;
S2=D[t^3-2t^2+t,{t,2}]/h^2;
A11=2*Integrate[P2*P2,{t,0,1}]*h
A21=Integrate[(P1/.t->(t-1))*P2,{t,0,1}]*h
B11=Integrate[P1*S1,{t,-1,0}]*h+Integrate[P2*S2,{t,0,1}]*h
B21=Integrate[(S1/.t->(t-1))*P2,{t,0,1}]*h
B12=Integrate[(P1/.t->(t-1))*S2,{t,0,1}]*h
C11=2*Integrate[S2*S2,{t,0,1}]*h
C21=Integrate[(S1/.t->(t-1))*S2,{t,0,1}]*h
```

The values `A11` $= 24/h^3$, `A21` $= -12/h^3$, `B22` $= 0$, `B12` $= 6/h^3$, `B21` $= -6/h^3$, `C11` $= 8/h^3$, and `C21` $= 2/h^3$ give the elements along the diagonals and off-diagonals of $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$. Compute the solution in MATLAB with

```
n = 8;
x = linspace(0,1,n+2); h = x(2)-x(1);
A = spdiags(repmat([-12 24 -12],[n 1]),-1:1,n,n)/h^3;
B = spdiags(repmat([6 0 -6],[n 1]),-1:1,n,n)/h^3;
C = spdiags(repmat([2 8 2],[n 1]),-1:1,n,n)/h^3;
M = [A B;B' C];
b = [ones([n 1])*h*384;zeros([n 1])];
u = M\b;
plot(x,16*(x.^4 - 2.*x.^3 + x.^2),'o-',x,[0;u(1:n);0],'.-');
```

If we are only interested in plotting the solution at the nodes, we can simply take $u(x_j) = \xi_j$. The solution using 8 interior nodes is shown in Figure S.13. Note that because we have a constant load, the finite element solution matches the analytical solution exactly. □

16.2. The following MATLAB code solves the KdV equation:

```
function KdV
L = 30;
n = 512;
x = (1:n)/n*L-L/2;
k = [0:(n/2-1) 0 (-n/2+1):-1]*(2*pi/L);
k = repmat(k,[3 1])';
phi = @(z,c) 0.5*c*sech(sqrt(c)/2*(x-z)).^2;
u = phi(-9,9) + phi(-4,4);
u = [phi(-9,9);phi(-4,4);phi(-9,9) + phi(-4,4)]';

N = ceil(2/(80/n^2)); dt = 2/N;
G = @(t) exp(-1i.*k.^3*t);
f = @(t,w) -G(t).*(3i*k.*(fft(ifft(G(t).\w).^2)));

w = fft(u);
for i = 1:N
    w = G(dt).\rk4(w,f,dt);
    v = ifft(w);
    [x1,y1] = interpolate(x',v(:,1));
    [x2,y2] = interpolate(x',v(:,2));
    plot(x1,y1,'.r',x2,y2,'.b',x,v(:,3),'k');
    ylim([-.5 5]); drawnow;
end

function [xn,yn] = interpolate(x,y)
t = [0;cumsum(diff(x).^2+diff(y).^2)];
xn = interp1(t,x,linspace(0,max(t),300));
yn = interp1(t,y,linspace(0,max(t),300));

function u = rk4(u,f,dt)
K1 = f(0,u);
K2 = f(0.5*dt,u+0.5*dt*K1);
K3 = f(0.5*dt,u+0.5*dt*K2);
K4 = f(dt,u+dt*K3);
u = u + dt*(K1+2*K2+2*K3+K4)/6;
```

The soliton solution to the KdV equation for two solitons is shown in Figure S.15. The top row shows the solutions to $u(x,0) = \phi(x;-4,4)$ and $u(x,0) = \phi(x;-9,9)$ each moving independently of one another. The bottom row shows the two solution for $u(x,0) = \phi(x;-4,4)+ \phi(x;-9,9)$. After collision the large, fast taller soliton in the bottom row is about 1 unit in front of the corresponding soliton in the top row. The smaller, slow soliton in the bottom row is about 1.5 units behind the corresponding soliton in the top row. Each soliton has the same energy after collision as it had before collision—their positions are merely shifted. □

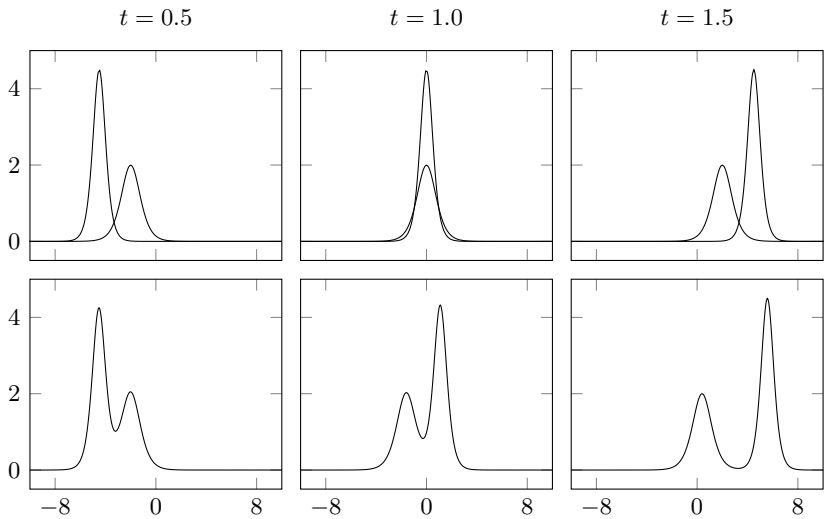


Figure S.14: Solution to Exercise **??**. Top row: independent solitons. Bottom row: interacting solitons.

16.3. The following MATLAB code produces the solution to the Swift–Hohenberg equation shown in Figure S.15 on the next page.

```
function swifthohenberg
epsilon=1; n = 256; L = 100; N = 2000; dt=100/N;
U=(rand(n)>.5)-0.5;
colormap(gray(256))
k = [0:(n/2-1) 0 (-n/2+1):-1]*(2*pi/L);
[kx,ky]=meshgrid(k);
D2=(1i*kx).^2+(1i*ky).^2;
E = exp(-(D2+1).^2*dt);
for i=1:N
   Y = U./sqrt(U.^2/epsilon + ...
               exp(-2*dt*epsilon)*( 1 - U.^2/epsilon));
   U = ifft2(fft2(Y).*E);
   image(real(U'+1)*128);
   axis equal; drawnow;
end
```

□

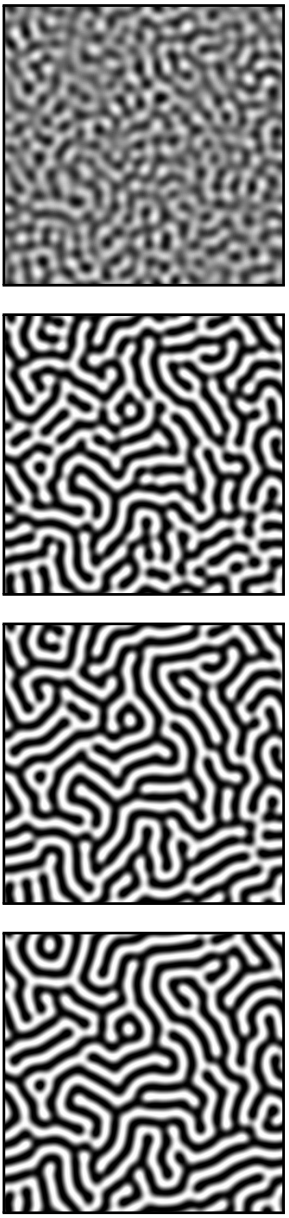


Figure S.15: Solution to the Swift–Hohenberg equation at times $t = 3$, 10, 25, and 100.

References

1. 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.

2. Kendall Atkinson and Weimin Han. *Theoretical numerical analysis*, volume 39 of *Texts in Applied Mathematics*. Springer, New York, second edition, 2005. ISBN 978-0387-25887-4; 0-387-25887-6.

3. Michael Blair, Sally Obenski, and Paula Bridickas. Patriot missile defense: Software problem led to system failure at Dhahran. Technical report, Saudi Arabia. Technical Report GAO/IMTEC-92-26, United States Department of Defense, General Accounting office, 1992.

4. John C Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2008.

5. Emmanuel J Candè and Michael B Wakin. An introduction to compressive sampling. *Signal Processing Magazine, IEEE*, 25(2):21–30, 2008.

6. 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.

7. James W. Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997. ISBN 0-89871-389-7.

8. 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. ISBN 0-387-95410-4.

9. Lawrence C Evans. *Partial differential equations.* American Mathematical Society, 2010.

10. Etienne Forest. Geometric integration for particle accelerators. *Journal of Physics A: Mathematical and General*, 39(19):5321, 2006.

11. Etienne Forest and Ronald D Ruth. Fourth-order symplectic integration. *Physica D: Nonlinear Phenomena*, 43(1):105–117, 1990.

12. Matteo Frigo and Steven G Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

13. John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992. ISSN 0895-4798.

14. 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. ISBN 0-8018-5413-X; 0-8018-5414-8.

15. Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*, volume 31. Springer Science & Business Media, 2006.

16. Godefroy Harold Hardy. Weierstrass's non-differentiable function. *Transactions of the American Mathematical Society*, 17(3):301–325, 1916.

17. Nicholas J Higham and Françoise 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.

18. Aly-Khan Kassam and Lloyd N Trefethen. Fourth-order time-stepping for stiff PDEs. *SIAM Journal on Scientific Computing*, 26(4):1214–1233, 2005.

19. David Kincaid and Ward Cheney. *Numerical analysis: Mathematics of scientific computing.* Brooks/Cole Publishing Co., Pacific Grove, CA, third edition, 2001. ISBN 0-534-38905-8.

20. John Denholm Lambert. *Numerical methods for ordinary differential systems: the initial value problem.* John Wiley & Sons, Inc., 1991.

21. Peter D. Lax. *Linear algebra and its applications.* Pure and Applied Mathematics (Hoboken). Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, second edition, 2007. ISBN 978-0-471-75156-4.

22. Randall J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002. ISBN 0-521-81087-6; 0-521-00924-3.

23. Cleve B. Moler. *Numerical computing with MATLAB*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2004. ISBN 0-89871-560-1.

24. K. W. Morton and D. F. Mayers. *Numerical solution of partial differential equations*. Cambridge University Press, Cambridge, second edition, 2005. ISBN 978-0-521-60793-0; 0-521-60793-0.

25. Haim Nessyahu and Eitan Tadmor. Non-oscillatory central differencing for hyperbolic conservation laws. *Journal of computational physics*, 87 (2):408–463, 1990.

26. Karlheinz Ochs. A comprehensive analytical solution of the nonlinear pendulum. *European Journal of Physics*, 32(2):479, 2011.

27. Per-Olof Persson. 18.335j Introduction to Numerical Methods, Fall 2006. MIT OpenCourseWare, 2006.

28. 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. ISBN 978-3-540-32612-0; 3-540-32612-X.

29. Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer-Verlag, Berlin, second edition, 2007. ISBN 978-3-540-34658-6; 3-540-34658-9.

30. Youcef Saad. *Numerical methods for large eigenvalue problems*. Algorithms and Architectures for Advanced Scientific Computing. Manchester University Press, Manchester, 1992. ISBN 0-7190-3386-1.

31. Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003. ISBN 0-89871-534-2.

32. Chi-Wang Shu and Stanley Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of Computational Physics*, 77(2):439–471, 1988.

33. Lawrence Sirovich and Michael Kirby. Low-dimensional procedure for the characterization of human faces. *Josa a*, 4(3):519–524, 1987.

34. Pavel Šolín. *Partial differential equations and the finite element method.* Pure and Applied Mathematics (New York). Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, 2006. ISBN 978-0-471-72070-6; 0-471-72070-4.

35. Gilbert Strang. 18.085 Mathematical Methods for Engineers I, Fall 2005. MIT OpenCourseWare, 2005a.

36. Gilbert Strang. 18.06 Linear Algebra, Spring 2005. MIT OpenCourseWare, 2005b.

37. Gilbert Strang. *Linear algebra and its applications.* Brooks Cole, fourth edition, 2005c. ISBN 978-0030105678.

38. Gilbert Strang. 18.086 Mathematical Methods for Engineers II, Spring 2006. MIT OpenCourseWare, 2006.

39. Johan Thim. Continuous nowhere differentiable functions. *Mémoire de DEA, Luleå University of Technology*, 2003.

40. Lloyd N. Trefethen. Finite difference and spectral methods for ordinary and partial differential equations. Unpublished text, 1996.

41. Lloyd N. Trefethen. *Spectral methods in MATLAB*, volume 10 of *Software, Environments, and Tools.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000. ISBN 0-89871-465-6.

42. Lloyd N Trefethen. A hundred-dollar, hundred-digit challenge. *SIAM news*, 35(1):1, 2002.

43. Lloyd N. Trefethen and David Bau, III. *Numerical linear algebra.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997. ISBN 0-89871-361-7.

44. LN Trefethen. Numerical analysis. princeton companion of mathematics, 2006.

45. Charles Van Loan. *Computational frameworks for the fast Fourier transform*, volume 10. Siam, 1992.

46. Loup Verlet. Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard–Jones molecules. *Physical review*, 159 (1):98, 1967.

47. G Wanner and E Hairer. *Solving ordinary differential equations II*, volume 1. Springer-Verlag, Berlin, 1991.

48. David S. Watkins. *Fundamentals of matrix computations*. Pure and Applied Mathematics (New York). Wiley-Interscience [John Wiley & Sons], New York, 2002. ISBN 0-471-21394-2.

49. X Wu, GJ Zwieten, and KG 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.

50. Haruo Yoshida. Construction of higher order symplectic integrators. *Physics Letters A*, 150(5):262–268, 1990.

Index

l^1-norm, 14, 16
l^2-norm, 17
l^∞-norm, 14, 17
l^p-norms, 16
nth root of unity, 125, 231
w-orthogonal, 222

A-conjugate, 116
A-stable, 269
absolute condition number, 153
absolutely stable, 266
Adams–Bashforth method, 275
Adams–Moulton method, 275
adaptive, 248
adjoint, 5
Aitken extrapolation, 178
algebraic multiplicity, 8
ansatz, 330
Arnoldi process, 89, 92
asymptotic convergence, 177
attractive fixed point, 177

B-spline, 214
banded, 7
banded matrix, 39
bandwidth, 7, 39
basin of attraction, 189
basis, 4, 198
basis spline, 214
BDF, 271
Bernstein polynomial, 218
best approximation, 221
bilinear functional, 359
boundary value problem, 355
bounded, 364
bracket, 169
Broyden's method, 190
Butcher tableau, 280
butterfly matrix, 129

Céa's lemma, 366
Cantor sets, 182
Cauchy–Schwarz, 14, 364
CFL condition, 307, 323
CFL number, 323
chaotic, 180
characteristic curves, 322
characteristic decomposition, 333
characteristic functions, 215
characteristic speed, 322, 331
Chebyshev abscissas, 205
Chebyshev approximation, 221
circulant, 132
coercive, 364
column space, 6
compact support, 336

companion matrix, 185
compatible, 16
complete orthogonal system, 226
complete pivoting, 35
composite Simpson's method, 247
composite trapezoid method, 247
compressed column format, 42
condition number, 21
congruent, 20, 37
conjugate gradient method, 116
conjugate transpose, 5
conservation law, 333
conservative, 347
consistent, 16, 157, 265, 293
continuous, 364
continuous Fourier transform, 230
contraction mapping, 103, 176
convergent, 158, 293
coset, 127
Courant–Friedrichs–Lewy, 307
cumulative length spline, 213
curse of dimensionality, 253

de Boor's algorithm, 215
de Casteljau's algorithm, 218
defective, 9
deflation, 83, 182
degree of the vertex, 40
depends continuously, 21, 153
DFT, 126
diagonal, 7
diagonalization, 9
dimension, 4, 198
direct problem, 152
direct sum, 234
Dirichlet boundary condition, 302
dispersion relation, 330
dispersive, 330
divided differences, 202
domain of dependence, 324
dominant eigenvalue, 73

eigenspace, 8
eigenvalue, 8
eigenvalue condition number, 70
eigenvector, 8
energy inner product, 13
energy method, 315
energy norm, 14
equivalent, 15
Euclidean inner product, 13
Euclidean norm, 14
explicit, 265

father wavelet, 233
Feigenbaum contant, 180
finite volume method, 349
fixed point, 175
fixed-point theorem, 176
flux, 333
Fourier transform, 126, 230

Galerkin, 357
Gauss–Christoffel quadrature, 248
Gauss–Newton method, 238
Gauss–Seidel method, 102
Gaussian elimination, 31
Gaussian quadrature, 248
generates, 4, 198
generators, 4, 198
geometric multiplicity, 8
ghost points, 302
global truncation error, 274
globally convergent, 170
Gram matrix, 223
group velocity, 330

Hölder norm, 14
Haar scaling function, 235
Hermite polynomial, 207
Hermitian, 7
Hilbert approximation, 221
Hilbert matrix, 224
homotopic, 191
Householder methods, 194
Householder reflection matrix, 54

Householder reflections, 53
hyperbolic, 341

identification problem, 152
IDFT, 126, 230
ill-conditioned, 22, 153
ill-posed, 20, 153
implicit, 265
incident matrix, 95
induced matrix norm, 16
inner product, 13
inner-product space, 221
invariant, 8
inverse problem, 152
isomorphic, 41

Jacobi method, 101
Julia set, 181, 189

kernel, 6
knots, 208
Kronecker product, 130
Krylov subspace, 88

L-stable, 286
Lanczos method, 93
Lax entropy condition, 340
Lax–Milgram Lemma, 364
left eigenvector, 70, 343
left nullspace, 6
Legendre polynomials, 228
Levenberg–Marquardt, 238
linear functional, 359
linear map, 5
linearly dependent, 4, 198
linearly independent, 4, 198
Lipschitz continuous, 262
load vector, 358
local Lax–Friedrichs, 348
local truncation error, 274
locally convergent, 170
LU decomposition, 31

Müller's method, 174
machine epsilon, 161
Mandelbrot set, 181
matrix norm, 15
maximum principle, 300
method of characteristics, 322
method of lines, 301
minimization problem, 355
Moore–Penrose pseudoinverse, 58
mother wavelet, 234
multiresolution, 233

Neumann boundary condition, 303
Newton's method, 171
Newton–Cotes, 245
Newton–Raphson method, 171
nodes, 197
normal, 7
normal equations, 46, 223, 236
normalizing, 161
nullity, 6
nullspace, 6
numerical dispersion, 329
numerical method of lines, 261
numerical viscosity, 329
numerically stable, 158

open boundary conditions, 305
optimal, 111
orbit, 177
order p, 170, 293
oriented graph, 40
orthogonal, 7, 13, 222
orthogonal projection, 7
orthogonal sum, 234

Padé approximation, 277
partial pivoting, 35
partition of unity, 215, 358
paths, 40
periodic boundary conditions, 305
permutation, 7
phase velocity, 330
Poincaré inequality, 364

positive-definite, 7, 19, 36
preconditioner, 102
principal components, 60
probability vector, 96
projection, 7
pseudo-inner product, 222
pseudo-inverse, 237

QR decomposition, 48
quadratic approximation, 174

Raleigh–Ritz, 357
rank, 6
Rankine–Hugoniot condition, 338
Rayleigh quotient, 76
Rayleigh quotient shifting, 83
reduced row echelon form, 30
region of absolute stability, 267
Reimann invariant, 322
relative condition number, 153
relaxation, 105
repulsive fixed point, 177
residual, 45, 105
reverse Cuthill–McKee, 41
Richardson extrapolation, 244
Riemann invariant, 343
Riemann problem, 338
Ritz estimate, 91
Ritz pair, 89
Robin boundary condition, 305
Romberg integration, 247
row space, 6
Runge's phenomenon, 205

scaling function, 233
secant method, 173
self-adjoint, 7
self-similar solution, 338
semi-norm, 222
semisimple, 9
Sherman–Morrison formula, 191
shift-and-invert, 75
shock, 335
similar, 8
similarity transform, 8
Simpson's rule, 246
singular value decomposition, 12
singular values, 12
Sobolev space, 359
span, 4, 198
sparse matrix, 39
spectral accuracy, 371
spectral decomposition, 12
spectral norm, 17
spectral radius, 8
spectrum, 8
splines, 208
splitting, 286
stable, 262, 266, 294
stencil, 265
stiff, 285
stiffness matrix, 358
Strang splitting, 288
strictly hyperbolic, 331
submultiplicative, 16
subspace, 3, 198
successive over-relaxation, 105
SVD, 12
symmetric, 7, 364

test function, 335
tetration, 150
Tikhonov regularization, 56, 238
time-marching, 302
Toeplitz, 132
total potential energy, 356
transpose, 5
trapezoidal rule, 246
tridiagonal, 7
truncation error, 293

ultimate shift, 87
unconditionally stable, 307
undetermined coefficients, 242
unit roundoff, 161
unitarily similar, 8

unitary, 7
upper Hessenberg, 7
upper triangular, 7
upwind method, 323

V-elliptic, 364
Vandermonde, 200
variational derivative, 356
variational problem, 355
vector norm, 14
vector space, 3, 197, 355
vectors, 3
vertices, 40
von Neumann analysis, 308

wavelet function, 234
weak solution, 337
weighted Jacobi method, 105
well-conditioned, 153
well-posed, 20, 153, 157, 262, 364
Wiener weight, 60
Wilkinson shift, 83
wisdom, 139

zero-stability, 270

Impress your friends with

THE DEFINITIVE MANUAL FOR MATH GEEKS

Scientists and engineers often use algorithms without fully knowing what's happening inside them. Such blind faith can lead to inefficient solutions and sometimes flat-out wrong ones. This book breaks open the algorithmic black boxes to help you understand how the algorithms work and why they can break down. Ideal for first-year graduate students, this book works to build both intuitive understanding of underlying mathematical theory and useful skills for their research. Examples are worked out in detail to provide a practical guide for using numerical methods in linear algebra, numerical analysis, and partial differential equations.

Kyle Novak has over twenty years of experience applying mathematics, decision analysis, modeling, and scientific computing on a wide range topics from autonomous navigation systems and cryptanalysis to superconducting materials and multiscale networks.

Cover illustration: Basins of attraction for the Rayleigh iteration for a three-dimensional matrix and the solution to the Swift–Hohenberg equation with random initial conditions.