

Contents

PREFACE

xiii

CHAPTER 0 Fundamentals

1

0.1	Evaluating a Polynomial	1
0.2	Binary Numbers	5
0.2.1	Decimal to binary	6
0.2.2	Binary to decimal	7
0.3	Floating Point Representation of Real Numbers	8
0.3.1	Floating point formats	8
0.3.2	Machine representation	11
0.3.3	Addition of floating point numbers	13
0.4	Loss of Significance	16
0.5	Review of Calculus	19
	Software and Further Reading	23

CHAPTER 1 Solving Equations

24

1.1	The Bisection Method	25
1.1.1	Bracketing a root	25
1.1.2	How accurate and how fast?	28
1.2	Fixed-Point Iteration	30
1.2.1	Fixed points of a function	31
1.2.2	Geometry of Fixed-Point Iteration	33
1.2.3	Linear convergence of Fixed-Point Iteration	34
1.2.4	Stopping criteria	40
1.3	Limits of Accuracy	43
1.3.1	Forward and backward error	44
1.3.2	The Wilkinson polynomial	47
1.3.3	Sensitivity of root-finding	48
1.4	Newton's Method	51
1.4.1	Quadratic convergence of Newton's Method	53
1.4.2	Linear convergence of Newton's Method	55
1.5	Root-Finding without Derivatives	61
1.5.1	Secant Method and variants	61
1.5.2	Brent's Method	64
	Reality Check 1: Kinematics of the Stewart platform	67
	Software and Further Reading	69

CHAPTER 2 Systems of Equations

71

2.1	Gaussian Elimination	71
2.1.1	Naive Gaussian elimination	72
2.1.2	Operation counts	74

2.2	The LU Factorization	79
2.2.1	Matrix form of Gaussian elimination	79
2.2.2	Back substitution with the LU factorization	81
2.2.3	Complexity of the LU factorization	83
2.3	Sources of Error	85
2.3.1	Error magnification and condition number	86
2.3.2	Swamping	91
2.4	The PA = LU Factorization	95
2.4.1	Partial pivoting	95
2.4.2	Permutation matrices	97
2.4.3	PA = LU factorization	98
	Reality Check 2: The Euler–Bernoulli Beam	102
2.5	Iterative Methods	106
2.5.1	Jacobi Method	106
2.5.2	Gauss–Seidel Method and SOR	108
2.5.3	Convergence of iterative methods	111
2.5.4	Sparse matrix computations	113
2.6	Methods for symmetric positive-definite matrices	117
2.6.1	Symmetric positive-definite matrices	117
2.6.2	Cholesky factorization	119
2.6.3	Conjugate Gradient Method	121
2.6.4	Preconditioning	126
2.7	Nonlinear Systems of Equations	130
2.7.1	Multivariate Newton’s Method	131
2.7.2	Broyden’s Method	133
	Software and Further Reading	137
CHAPTER 3 Interpolation		138
3.1	Data and Interpolating Functions	139
3.1.1	Lagrange interpolation	140
3.1.2	Newton’s divided differences	141
3.1.3	How many degree d polynomials pass through n points?	144
3.1.4	Code for interpolation	145
3.1.5	Representing functions by approximating polynomials	147
3.2	Interpolation Error	151
3.2.1	Interpolation error formula	151
3.2.2	Proof of Newton form and error formula	153
3.2.3	Runge phenomenon	155
3.3	Chebyshev Interpolation	158
3.3.1	Chebyshev’s theorem	158
3.3.2	Chebyshev polynomials	160
3.3.3	Change of interval	162
3.4	Cubic Splines	166
3.4.1	Properties of splines	167
3.4.2	Endpoint conditions	173
3.5	Bézier Curves	179
	Reality Check 3: Fonts from Bézier curves	183
	Software and Further Reading	187

CHAPTER 4	Least Squares	188
4.1	Least Squares and the Normal Equations	188
4.1.1	Inconsistent systems of equations	189
4.1.2	Fitting models to data	193
4.1.3	Conditioning of least squares	197
4.2	A Survey of Models	201
4.2.1	Periodic data	201
4.2.2	Data linearization	203
4.3	QR Factorization	212
4.3.1	Gram–Schmidt orthogonalization and least squares	212
4.3.2	Modified Gram–Schmidt orthogonalization	218
4.3.3	Householder reflectors	220
4.4	Generalized Minimum Residual (GMRES) Method	225
4.4.1	Krylov methods	226
4.4.2	Preconditioned GMRES	228
4.5	Nonlinear Least Squares	230
4.5.1	Gauss–Newton Method	230
4.5.2	Models with nonlinear parameters	233
4.5.3	The Levenberg–Marquardt Method.	235
	Reality Check 4: GPS, Conditioning, and Nonlinear Least Squares	238
	Software and Further Reading	242
CHAPTER 5	Numerical Differentiation and Integration	243
5.1	Numerical Differentiation	244
5.1.1	Finite difference formulas	244
5.1.2	Rounding error	247
5.1.3	Extrapolation	249
5.1.4	Symbolic differentiation and integration	250
5.2	Newton–Cotes Formulas for Numerical Integration	254
5.2.1	Trapezoid Rule	255
5.2.2	Simpson’s Rule	257
5.2.3	Composite Newton–Cotes formulas	259
5.2.4	Open Newton–Cotes Methods	262
5.3	Romberg Integration	265
5.4	Adaptive Quadrature	269
5.5	Gaussian Quadrature	273
	Reality Check 5: Motion Control in Computer-Aided Modeling	278
	Software and Further Reading	280
CHAPTER 6	Ordinary Differential Equations	281
6.1	Initial Value Problems	282
6.1.1	Euler’s Method	283
6.1.2	Existence, uniqueness, and continuity for solutions	287
6.1.3	First-order linear equations	290
6.2	Analysis of IVP Solvers	293
6.2.1	Local and global truncation error	293

6.2.2	The explicit Trapezoid Method	297
6.2.3	Taylor Methods	300
6.3	Systems of Ordinary Differential Equations	303
6.3.1	Higher order equations	304
6.3.2	Computer simulation: the pendulum	305
6.3.3	Computer simulation: orbital mechanics	309
6.4	Runge–Kutta Methods and Applications	314
6.4.1	The Runge–Kutta family	314
6.4.2	Computer simulation: the Hodgkin–Huxley neuron	317
6.4.3	Computer simulation: the Lorenz equations	319
	Reality Check 6: The Tacoma Narrows Bridge	322
6.5	Variable Step-Size Methods	325
6.5.1	Embedded Runge–Kutta pairs	325
6.5.2	Order 4/5 methods	328
6.6	Implicit Methods and Stiff Equations	332
6.7	Multistep Methods	336
6.7.1	Generating multistep methods	336
6.7.2	Explicit multistep methods	339
6.7.3	Implicit multistep methods	342
	Software and Further Reading	347
CHAPTER 7	Boundary Value Problems	348
7.1	Shooting Method	349
7.1.1	Solutions of boundary value problems	349
7.1.2	Shooting Method implementation	352
	Reality Check 7: Buckling of a Circular Ring	355
7.2	Finite Difference Methods	357
7.2.1	Linear boundary value problems	357
7.2.2	Nonlinear boundary value problems	359
7.3	Collocation and the Finite Element Method	365
7.3.1	Collocation	365
7.3.2	Finite elements and the Galerkin Method	367
	Software and Further Reading	373
CHAPTER 8	Partial Differential Equations	374
8.1	Parabolic Equations	375
8.1.1	Forward Difference Method	375
8.1.2	Stability analysis of Forward Difference Method	379
8.1.3	Backward Difference Method	380
8.1.4	Crank–Nicolson Method	385
8.2	Hyperbolic Equations	393
8.2.1	The wave equation	393
8.2.2	The CFL condition	395
8.3	Elliptic Equations	398
8.3.1	Finite Difference Method for elliptic equations	399
	Reality Check 8: Heat distribution on a cooling fin	403
8.3.2	Finite Element Method for elliptic equations	406

8.4	Nonlinear partial differential equations	417
8.4.1	Implicit Newton solver	417
8.4.2	Nonlinear equations in two space dimensions	423
	Software and Further Reading	430
CHAPTER 9 Random Numbers and Applications		431
9.1	Random Numbers	432
9.1.1	Pseudo-random numbers	432
9.1.2	Exponential and normal random numbers	437
9.2	Monte Carlo Simulation	440
9.2.1	Power laws for Monte Carlo estimation	440
9.2.2	Quasi-random numbers	442
9.3	Discrete and Continuous Brownian Motion	446
9.3.1	Random walks	447
9.3.2	Continuous Brownian motion	449
9.4	Stochastic Differential Equations	452
9.4.1	Adding noise to differential equations	452
9.4.2	Numerical methods for SDEs	456
	Reality Check 9: The Black–Scholes Formula	464
	Software and Further Reading	465
CHAPTER 10 Trigonometric Interpolation and the FFT		467
10.1	The Fourier Transform	468
10.1.1	Complex arithmetic	468
10.1.2	Discrete Fourier Transform	470
10.1.3	The Fast Fourier Transform	473
10.2	Trigonometric Interpolation	476
10.2.1	The DFT Interpolation Theorem	476
10.2.2	Efficient evaluation of trigonometric functions	479
10.3	The FFT and Signal Processing	483
10.3.1	Orthogonality and interpolation	483
10.3.2	Least squares fitting with trigonometric functions	485
10.3.3	Sound, noise, and filtering	489
	Reality Check 10: The Wiener Filter	492
	Software and Further Reading	494
CHAPTER 11 Compression		495
11.1	The Discrete Cosine Transform	496
11.1.1	One-dimensional DCT	496
11.1.2	The DCT and least squares approximation	498
11.2	Two-Dimensional DCT and Image Compression	501
11.2.1	Two-dimensional DCT	501
11.2.2	Image compression	505
11.2.3	Quantization	508
11.3	Huffman Coding	514
11.3.1	Information theory and coding	514
11.3.2	Huffman coding for the JPEG format	517

11.4	Modified DCT and Audio Compression	519
11.4.1	Modified Discrete Cosine Transform	520
11.4.2	Bit quantization	525
Reality Check 11:	A Simple Audio Codec	527
	Software and Further Reading	530

CHAPTER 12 Eigenvalues and Singular Values 531

12.1	Power Iteration Methods	531
12.1.1	Power Iteration	532
12.1.2	Convergence of Power Iteration	534
12.1.3	Inverse Power Iteration	535
12.1.4	Rayleigh Quotient Iteration	537
12.2	QR Algorithm	539
12.2.1	Simultaneous iteration	539
12.2.2	Real Schur form and the QR algorithm	542
12.2.3	Upper Hessenberg form	544
Reality Check 12:	How Search Engines Rate Page Quality	549
12.3	Singular Value Decomposition	552
12.3.1	Finding the SVD in general	554
12.3.2	Special case: symmetric matrices	555
12.4	Applications of the SVD	557
12.4.1	Properties of the SVD	557
12.4.2	Dimension reduction	559
12.4.3	Compression	560
12.4.4	Calculating the SVD	561
	Software and Further Reading	563

CHAPTER 13 Optimization 565

13.1	Unconstrained Optimization without Derivatives	566
13.1.1	Golden Section Search	566
13.1.2	Successive parabolic interpolation	569
13.1.3	Nelder–Mead search	571
13.2	Unconstrained Optimization with Derivatives	575
13.2.1	Newton's Method	576
13.2.2	Steepest Descent	577
13.2.3	Conjugate Gradient Search	578
Reality Check 13:	Molecular Conformation and Numerical Optimization	580
	Software and Further Reading	582

Appendix A 583

A.1	Matrix Fundamentals	583
A.2	Block Multiplication	585
A.3	Eigenvalues and Eigenvectors	586
A.4	Symmetric Matrices	587
A.5	Vector Calculus	588

Appendix B	590
B.1 Starting MATLAB	590
B.2 Graphics	591
B.3 Programming in MATLAB	593
B.4 Flow Control	594
B.5 Functions	595
B.6 Matrix Operations	597
B.7 Animation and Movies	597
ANSWERS TO SELECTED EXERCISES	599
BIBLIOGRAPHY	626
INDEX	637



Fundamentals

This introductory chapter provides basic building blocks necessary for the construction and understanding of the algorithms of the book. They include fundamental ideas of introductory calculus and function evaluation, the details of machine arithmetic as it is carried out on modern computers, and discussion of the loss of significant digits resulting from poorly-designed calculations.

After discussing efficient methods for evaluating polynomials, we study the binary number system, the representation of floating point numbers and the common protocols used for rounding. The effects of the small rounding errors on computations are magnified in ill-conditioned problems. The battle to limit these pernicious effects is a recurring theme throughout the rest of the chapters.

The goal of this book is to present and discuss methods of solving mathematical problems with computers. The most fundamental operations of arithmetic are addition and multiplication. These are also the operations needed to evaluate a polynomial $P(x)$ at a particular value x . It is no coincidence that polynomials are the basic building blocks for many computational techniques we will construct.

Because of this, it is important to know how to evaluate a polynomial. The reader probably already knows how and may consider spending time on such an easy problem slightly ridiculous! But the more basic an operation is, the more we stand to gain by doing it right. Therefore we will think about how to implement polynomial evaluation as efficiently as possible.

0.1 EVALUATING A POLYNOMIAL

What is the best way to evaluate

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1,$$

say, at $x = 1/2$? Assume that the coefficients of the polynomial and the number $1/2$ are stored in memory, and try to minimize the number of additions and multiplications required

to get $P(1/2)$. To simplify matters, we will not count time spent storing and fetching numbers to and from memory.

METHOD 1 The first and most straightforward approach is

$$P\left(\frac{1}{2}\right) = 2 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} + 3 * \frac{1}{2} * \frac{1}{2} * \frac{1}{2} - 3 * \frac{1}{2} * \frac{1}{2} + 5 * \frac{1}{2} - 1 = \frac{5}{4}. \quad (0.1)$$

The number of multiplications required is 10, together with 4 additions. Two of the additions are actually subtractions, but because subtraction can be viewed as adding a negative stored number, we will not worry about the difference.

There surely is a better way than (0.1). Effort is being duplicated—operations can be saved by eliminating the repeated multiplication by the input $1/2$. A better strategy is to first compute $(1/2)^4$, storing partial products as we go. That leads to the following method:

METHOD 2 Find the powers of the input number $x = 1/2$ first, and store them for future use:

$$\begin{aligned} \frac{1}{2} * \frac{1}{2} &= \left(\frac{1}{2}\right)^2 \\ \left(\frac{1}{2}\right)^2 * \frac{1}{2} &= \left(\frac{1}{2}\right)^3 \\ \left(\frac{1}{2}\right)^3 * \frac{1}{2} &= \left(\frac{1}{2}\right)^4. \end{aligned}$$

Now we can add up the terms:

$$P\left(\frac{1}{2}\right) = 2 * \left(\frac{1}{2}\right)^4 + 3 * \left(\frac{1}{2}\right)^3 - 3 * \left(\frac{1}{2}\right)^2 + 5 * \frac{1}{2} - 1 = \frac{5}{4}.$$

There are now 3 multiplications of $1/2$, along with 4 other multiplications. Counting up, we have reduced to 7 multiplications, with the same 4 additions. Is the reduction from 14 to 11 operations a significant improvement? If there is only one evaluation to be done, then probably not. Whether Method 1 or Method 2 is used, the answer will be available before you can lift your fingers from the computer keyboard. However, suppose the polynomial needs to be evaluated at different inputs x several times per second. Then the difference may be crucial to getting the information when it is needed.

Is this the best we can do for a degree 4 polynomial? It may be hard to imagine that we can eliminate three more operations, but we can. The best elementary method is the following one:

METHOD 3 (Nested Multiplication) Rewrite the polynomial so that it can be evaluated from the inside out:

$$\begin{aligned} P(x) &= -1 + x(5 - 3x + 3x^2 + 2x^3) \\ &= -1 + x(5 + x(-3 + 3x + 2x^2)) \\ &= -1 + x(5 + x(-3 + x(3 + 2x))) \\ &= -1 + x * (5 + x * (-3 + x * (3 + x * 2))). \end{aligned} \quad (0.2)$$

Here the polynomial is written backwards, and powers of x are factored out of the rest of the polynomial. Once you can see to write it this way—no computation is required to do the rewriting—the coefficients are unchanged. Now evaluate from the inside out:

$$\begin{aligned}
&\text{multiply } \frac{1}{2} * 2, & \text{add } + 3 \rightarrow 4 \\
&\text{multiply } \frac{1}{2} * 4, & \text{add } - 3 \rightarrow -1 \\
&\text{multiply } \frac{1}{2} * -1, & \text{add } + 5 \rightarrow \frac{9}{2} \\
&\text{multiply } \frac{1}{2} * \frac{9}{2}, & \text{add } - 1 \rightarrow \frac{5}{4}.
\end{aligned} \tag{0.3}$$

This method, called **nested multiplication** or **Horner's method**, evaluates the polynomial in 4 multiplications and 4 additions. A general degree d polynomial can be evaluated in d multiplications and d additions. Nested multiplication is closely related to synthetic division of polynomial arithmetic.

The example of polynomial evaluation is characteristic of the entire topic of computational methods for scientific computing. First, computers are very fast at doing very simple things. Second, it is important to do even simple tasks as efficiently as possible, since they may be executed many times. Third, the best way may not be the obvious way. Over the last half-century, the fields of numerical analysis and scientific computing, hand in hand with computer hardware technology, have developed efficient solution techniques to attack common problems.

While the standard form for a polynomial $c_1 + c_2x + c_3x^2 + c_4x^3 + c_5x^4$ can be written in nested form as

$$c_1 + x(c_2 + x(c_3 + x(c_4 + x(c_5))))), \tag{0.4}$$

some applications require a more general form. In particular, interpolation calculations in Chapter 3 will require the form

$$c_1 + (x - r_1)(c_2 + (x - r_2)(c_3 + (x - r_3)(c_4 + (x - r_4)(c_5)))), \tag{0.5}$$

where we call r_1, r_2, r_3 , and r_4 the **base points**. Note that setting $r_1 = r_2 = r_3 = r_4 = 0$ in (0.5) recovers the original nested form (0.4).

The following MATLAB code implements the general form of nested multiplication (compare with (0.3)):

```

%Program 0.1 Nested multiplication
%Evaluates polynomial from nested form using Horner's Method
%Input: degree d of polynomial,
%       array of d+1 coefficients c (constant term first),
%       x-coordinate x at which to evaluate, and
%       array of d base points b, if needed
%Output: value y of polynomial at x
function y=nest(d,c,x,b)
if nargin<4, b=zeros(d,1); end
y=c(d+1);
for i=d:-1:1
    y = y.*(x-b(i))+c(i);
end

```

Running this MATLAB function is a matter of substituting the input data, which consist of the degree, coefficients, evaluation points, and base points. For example, polynomial (0.2) can be evaluated at $x = 1/2$ by the MATLAB command

```
>> nest(4, [-1 5 -3 3 2], 1/2, [0 0 0 0])

ans =

    1.2500
```

as we found earlier by hand. The file `nest.m`, as the rest of the MATLAB code shown in this book, must be accessible from the MATLAB path (or in the current directory) when executing the command.

If the `nest` command is to be used with all base points 0 as in (0.2), the abbreviated form

```
>> nest(4, [-1 5 -3 3 2], 1/2)
```

may be used with the same result. This is due to the `nargin` statement in `nest.m`. If the number of input arguments is less than 4, the base points are automatically set to zero.

Because of MATLAB's seamless treatment of vector notation, the `nest` command can evaluate an array of x values at once. The following code is illustrative:

```
>> nest(4, [-1 5 -3 3 2], [-2 -1 0 1 2])

ans =

   -15   -10    -1     6    53
```

Finally, the degree 3 interpolating polynomial

$$P(x) = 1 + x \left(\frac{1}{2} + (x - 2) \left(\frac{1}{2} + (x - 3) \left(-\frac{1}{2} \right) \right) \right)$$

from Chapter 3 has base points $r_1 = 0, r_2 = 2, r_3 = 3$. It can be evaluated at $x = 1$ by

```
>> nest(3, [1 1/2 1/2 -1/2], 1, [0 2 3])

ans =

    0
```

► **EXAMPLE 0.1** Find an efficient method for evaluating the polynomial $P(x) = 4x^5 + 7x^8 - 3x^{11} + 2x^{14}$.

Some rewriting of the polynomial may help reduce the computational effort required for evaluation. The idea is to factor x^5 from each term and write as a polynomial in the quantity x^3 :

$$\begin{aligned} P(x) &= x^5(4 + 7x^3 - 3x^6 + 2x^9) \\ &= x^5 * (4 + x^3 * (7 + x^3 * (-3 + x^3 * (2))))). \end{aligned}$$

For each input x , we need to calculate $x * x = x^2$, $x * x^2 = x^3$, and $x^2 * x^3 = x^5$ first. These three multiplications, combined with the multiplication of x^5 , and the three multiplications and three additions from the degree 3 polynomial in the quantity x^3 give the total operation count of 7 multiplies and 3 adds per evaluation. ◀

0.1 Exercises

- Rewrite the following polynomials in nested form. Evaluate with and without nested form at $x = 1/3$.
 - $P(x) = 6x^4 + x^3 + 5x^2 + x + 1$
 - $P(x) = -3x^4 + 4x^3 + 5x^2 - 5x + 1$
 - $P(x) = 2x^4 + x^3 - x^2 + 1$
- Rewrite the following polynomials in nested form and evaluate at $x = -1/2$:
 - $P(x) = 6x^3 - 2x^2 - 3x + 7$
 - $P(x) = 8x^5 - x^4 - 3x^3 + x^2 - 3x + 1$
 - $P(x) = 4x^6 - 2x^4 - 2x + 4$
- Evaluate $P(x) = x^6 - 4x^4 + 2x^2 + 1$ at $x = 1/2$ by considering $P(x)$ as a polynomial in x^2 and using nested multiplication.
- Evaluate the nested polynomial with base points $P(x) = 1 + x(1/2 + (x - 2)(1/2 + (x - 3)(-1/2)))$ at (a) $x = 5$ and (b) $x = -1$.
- Evaluate the nested polynomial with base points $P(x) = 4 + x(4 + (x - 1)(1 + (x - 2)(3 + (x - 3)(2))))$ at (a) $x = 1/2$ and (b) $x = -1/2$.
- Explain how to evaluate the polynomial for a given input x , using as few operations as possible. How many multiplications and how many additions are required?
 - $P(x) = a_0 + a_5x^5 + a_{10}x^{10} + a_{15}x^{15}$
 - $P(x) = a_7x^7 + a_{12}x^{12} + a_{17}x^{17} + a_{22}x^{22} + a_{27}x^{27}$.
- How many additions and multiplications are required to evaluate a degree n polynomial with base points, using the general nested multiplication algorithm?

0.1 Computer Problems

- Use the function `nest` to evaluate $P(x) = 1 + x + \cdots + x^{50}$ at $x = 1.00001$. (Use the MATLAB `ones` command to save typing.) Find the error of the computation by comparing with the equivalent expression $Q(x) = (x^{51} - 1)/(x - 1)$.
- Use `nest.m` to evaluate $P(x) = 1 - x + x^2 - x^3 + \cdots + x^{98} - x^{99}$ at $x = 1.00001$. Find a simpler, equivalent expression, and use it to estimate the error of the nested multiplication.

0.2 BINARY NUMBERS

In preparation for the detailed study of computer arithmetic in the next section, we need to understand the binary number system. Decimal numbers are converted from base 10 to base 2 in order to store numbers on a computer and to simplify computer operations like addition and multiplication. To give output in decimal notation, the process is reversed. In this section, we discuss ways to convert between decimal and binary numbers.

Binary numbers are expressed as

$$\dots b_2b_1b_0.b_{-1}b_{-2}\dots,$$

where each binary digit, or **bit**, is 0 or 1. The base 10 equivalent to the number is

$$\dots b_2 2^2 + b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} \dots$$

For example, the decimal number 4 is expressed as $(100.)_2$ in base 2, and $3/4$ is represented as $(0.11)_2$.

0.2.1 Decimal to binary

The decimal number 53 will be represented as $(53)_{10}$ to emphasize that it is to be interpreted as base 10. To convert to binary, it is simplest to break the number into integer and fractional parts and convert each part separately. For the number $(53.7)_{10} = (53)_{10} + (0.7)_{10}$, we will convert each part to binary and combine the results.

Integer part. Convert decimal integers to binary by dividing by 2 successively and recording the remainders. The remainders, 0 or 1, are recorded by starting at the decimal point (or more accurately, **radix**) and moving away (to the left). For $(53)_{10}$, we would have

$$53 \div 2 = 26 \text{ R } 1$$

$$26 \div 2 = 13 \text{ R } 0$$

$$13 \div 2 = 6 \text{ R } 1$$

$$6 \div 2 = 3 \text{ R } 0$$

$$3 \div 2 = 1 \text{ R } 1$$

$$1 \div 2 = 0 \text{ R } 1.$$

Therefore, the base 10 number 53 can be written in bits as 110101, denoted as $(53)_{10} = (110101.)_2$. Checking the result, we have $110101 = 2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53$.

Fractional part. Convert $(0.7)_{10}$ to binary by reversing the preceding steps. Multiply by 2 successively and record the integer parts, moving away from the decimal point to the right.

$$.7 \times 2 = .4 + 1$$

$$.4 \times 2 = .8 + 0$$

$$.8 \times 2 = .6 + 1$$

$$.6 \times 2 = .2 + 1$$

$$.2 \times 2 = .4 + 0$$

$$.4 \times 2 = .8 + 0$$

$$\vdots$$

Notice that the process repeats after four steps and will repeat indefinitely exactly the same way. Therefore,

$$(0.7)_{10} = (.1011001100110\dots)_2 = (.1\overline{0110})_2,$$

where overbar notation is used to denote infinitely repeated bits. Putting the two parts together, we conclude that

$$(53.7)_{10} = (110101.1\overline{0110})_2.$$

0.2.2 Binary to decimal

To convert a binary number to decimal, it is again best to separate into integer and fractional parts.

Integer part. Simply add up powers of 2 as we did before. The binary number $(10101)_2$ is simply $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (21)_{10}$.

Fractional part. If the fractional part is finite (a terminating base 2 expansion), proceed the same way. For example,

$$(.1011)_2 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = \left(\frac{11}{16}\right)_{10}.$$

The only complication arises when the fractional part is not a finite base 2 expansion. Converting an infinitely repeating binary expansion to a decimal fraction can be done in several ways. Perhaps the simplest way is to use the shift property of multiplication by 2.

For example, suppose $x = (0.\overline{1011})_2$ is to be converted to decimal. Multiply x by 2^4 , which shifts 4 places to the left in binary. Then subtract the original x :

$$\begin{aligned} 2^4 x &= 1011.\overline{1011} \\ x &= 0000.\overline{1011}. \end{aligned}$$

Subtracting yields

$$(2^4 - 1)x = (1011)_2 = (11)_{10}.$$

Then solve for x to find $x = (\overline{1011})_2 = 11/15$ in base 10.

As another example, assume that the fractional part does not immediately repeat, as in $x = .\overline{10101}$. Multiplying by 2^2 shifts to $y = 2^2 x = 10.\overline{101}$. The fractional part of y , call it $z = .\overline{101}$, is calculated as before:

$$\begin{aligned} 2^3 z &= 101.\overline{101} \\ z &= 000.\overline{101}. \end{aligned}$$

Therefore, $7z = 5$, and $y = 2 + 5/7$, $x = 2^{-2}y = 19/28$ in base 10. It is a good exercise to check this result by converting $19/28$ to binary and comparing to the original x .

Binary numbers are the building blocks of machine computations, but they turn out to be long and unwieldy for humans to interpret. It is useful to use base 16 at times just to present numbers more easily. **Hexadecimal numbers** are represented by the 16 numerals $0, 1, 2, \dots, 9, A, B, C, D, E, F$. Each hex number can be represented by 4 bits. Thus $(1)_{16} = (0001)_2$, $(8)_{16} = (1000)_2$, and $(F)_{16} = (1111)_2 = (15)_{10}$. In the next section, MATLAB's `format hex` for representing machine numbers will be described.

0.2 Exercises

- Find the binary representation of the base 10 integers. (a) 64 (b) 17 (c) 79 (d) 227
- Find the binary representation of the base 10 numbers. (a) $1/8$ (b) $7/8$ (c) $35/16$ (d) $31/64$
- Convert the following base 10 numbers to binary. Use overbar notation for nonterminating binary numbers. (a) 10.5 (b) $1/3$ (c) $5/7$ (d) 12.8 (e) 55.4 (f) 0.1
- Convert the following base 10 numbers to binary. (a) 11.25 (b) $2/3$ (c) $3/5$ (d) 3.2 (e) 30.6 (f) 99.9

5. Find the first 15 bits in the binary representation of π .
6. Find the first 15 bits in the binary representation of e .
7. Convert the following binary numbers to base 10: (a) 1010101 (b) 1011.101 (c) 10111.01 (d) 110.10 (e) 10.110 (f) 110.1101 (g) 10.0101101 (h) 111.1
8. Convert the following binary numbers to base 10: (a) 11011 (b) 110111.001 (c) 111.001 (d) 1010.01 (e) 10111.10101 (f) 1111.010001

0.3 FLOATING POINT REPRESENTATION OF REAL NUMBERS

In this section, we present a model for computer arithmetic of floating point numbers. There are several models, but to simplify matters we will choose one particular model and describe it in detail. The model we choose is the so-called IEEE 754 Floating Point Standard. The Institute of Electrical and Electronics Engineers (IEEE) takes an active interest in establishing standards for the industry. Their floating point arithmetic format has become the common standard for single-precision and double precision arithmetic throughout the computer industry.

Rounding errors are inevitable when finite-precision computer memory locations are used to represent real, infinite precision numbers. Although we would hope that small errors made during a long calculation have only a minor effect on the answer, this turns out to be wishful thinking in many cases. **Simple algorithms, such as Gaussian elimination or methods for solving differential equations, can magnify microscopic errors to macroscopic size.** In fact, a main theme of this book is to help the reader to recognize when a calculation is at risk of being unreliable due to magnification of the small errors made by digital computers and to know how to avoid or minimize the risk.

0.3.1 Floating point formats

The IEEE standard consists of a set of binary representations of real numbers. A **floating point number** consists of three parts: the **sign** (+ or −), a **mantissa**, which contains the string of significant bits, and an **exponent**. The three parts are stored together in a single computer **word**.

There are three commonly used levels of precision for floating point numbers: single precision, double precision, and extended precision, also known as long-double precision. The number of bits allocated for each floating point number in the three formats is 32, 64, and 80, respectively. The bits are divided among the parts as follows:

precision	sign	exponent	mantissa
single	1	8	23
double	1	11	52
long double	1	15	64

All three types of precision work essentially the same way. The form of a **normalized** IEEE floating point number is

$$\pm 1.bbb \dots b \times 2^p, \quad (0.6)$$

where each of the N b 's is 0 or 1, and p is an M -bit binary number representing the exponent. Normalization means that, as shown in (0.6), the leading (leftmost) bit must be 1.

When a binary number is stored as a normalized floating point number, it is “left-justified,” meaning that the leftmost 1 is shifted just to the left of the radix point. The shift

is compensated by a change in the exponent. For example, the decimal number 9, which is 1001 in binary, would be stored as

$+1.001 \times 2^3,$

because a shift of 3 bits, or multiplication by 2^3 , is necessary to move the leftmost one to the correct position.

For concreteness, we will specialize to the double precision format for most of the discussion. Single and long-double precision are handled in the same way, with the exception of different exponent and mantissa lengths M and N . In double precision, used by many C compilers and by MATLAB, $M = 11$ and $N = 52$.

The double precision number 1 is

$$+1.\overline{000} \times 2^0,$$

where we have boxed the 52 bits of the mantissa. The next floating point number greater than 1 is

[illegible]

or $1 + 2^{-52}$.

DEFINITION 0.1 The number **machine epsilon**, denoted ϵ_{mach} , is the distance between 1 and the smallest floating point number greater than 1. For the IEEE double precision floating point standard,

$$\epsilon_{\text{mach}} = 2^{-52}. \quad \square$$

The decimal number $9.4 = (1001.\overline{0110})_2$ is left-justified as

$$+1. \overline{001011001100110011001100110011001100110011001100}110 \dots \times 2^3,$$

where we have boxed the first 52 bits of the mantissa. A new question arises: How do we fit the infinite binary number representing 9.4 in a finite number of bits?

We must truncate the number in some way, and in so doing we necessarily make a small error. One method, called **chopping**, is to simply throw away the bits that fall off the end—that is, those beyond the 52nd bit to the right of the decimal point. This protocol is simple, but it is biased in that it always moves the result toward zero.

The alternative method is **rounding**. In base 10, numbers are customarily rounded up if the next digit is 5 or higher, and rounded down otherwise. In binary, this corresponds to rounding up if the bit is 1. Specifically, the important bit in the double precision format is the 53rd bit to the right of the radix point, the first one lying outside of the box. The default rounding technique, implemented by the IEEE standard, is to add 1 to bit 52 (round up) if bit 53 is 1, and to do nothing (round down) to bit 52 if bit 53 is 0, with one exception: If the bits following bit 52 are 10000..., exactly halfway between up and down, we round up or round down according to which choice makes the final bit 52 equal to 0. (Here we are dealing with the mantissa only, since the sign does not play a role.)

Why is there the strange exceptional case? Except for this case, the rule means rounding to the normalized floating point number closest to the original number—hence its name, the Rounding to Nearest Rule. The error made in rounding will be equally likely to be up or down. Therefore, the exceptional case, the case where there are two equally distant floating point numbers to round to, should be decided in a way that doesn't prefer up or down systematically. This is to try to avoid the possibility of an unwanted slow drift in long calculations due simply to a biased rounding. The choice to make the final bit 52 equal to 0 in the case of a tie is somewhat arbitrary, but at least it does not display a preference up or down. Problem 8 sheds some light on why the arbitrary choice of 0 is made in case of a tie.

2. Convert the following base 10 numbers to binary and express each as a floating point number $\text{fl}(x)$ by using the Rounding to Nearest Rule: (a) 9.5 (b) 9.6 (c) 100.2 (d) 44/7
3. For which positive integers k can the number $5 + 2^{-k}$ be represented exactly (with no rounding error) in double precision floating point arithmetic?
4. Find the largest integer k for which $\text{fl}(19 + 2^{-k}) > \text{fl}(19)$ in double precision floating point arithmetic.
5. Do the following sums by hand in IEEE double precision computer arithmetic, using the Rounding to Nearest Rule. (Check your answers, using MATLAB.)
 - (a) $(1 + (2^{-51} + 2^{-53})) - 1$
 - (b) $(1 + (2^{-51} + 2^{-52} + 2^{-53})) - 1$
6. Do the following sums by hand in IEEE double precision computer arithmetic, using the Rounding to Nearest Rule:
 - (a) $(1 + (2^{-51} + 2^{-52} + 2^{-54})) - 1$
 - (b) $(1 + (2^{-51} + 2^{-52} + 2^{-60})) - 1$
7. Write each of the given numbers in MATLAB's `format hex`. Show your work. Then check your answers with MATLAB. (a) 8 (b) 21 (c) 1/8 (d) $\text{fl}(1/3)$ (e) $\text{fl}(2/3)$ (f) $\text{fl}(0.1)$ (g) $\text{fl}(-0.1)$ (h) $\text{fl}(-0.2)$
8. Is $1/3 + 2/3$ exactly equal to 1 in double precision floating point arithmetic, using the IEEE Rounding to Nearest Rule? You will need to use $\text{fl}(1/3)$ and $\text{fl}(2/3)$ from Exercise 1. Does this help explain why the rule is expressed as it is? Would the sum be the same if chopping after bit 52 were used instead of IEEE rounding?
9. (a) Explain why you can determine machine epsilon on a computer using IEEE double precision and the IEEE Rounding to Nearest Rule by calculating $(7/3 - 4/3) - 1$. (b) Does $(4/3 - 1/3) - 1$ also give ϵ_{mach} ? Explain by converting to floating point numbers and carrying out the machine arithmetic.
10. Decide whether $1 + x > 1$ in double precision floating point arithmetic, with Rounding to Nearest. (a) $x = 2^{-53}$ (b) $x = 2^{-53} + 2^{-60}$
11. Does the associative law hold for IEEE computer addition?
12. Find the IEEE double precision representation $\text{fl}(x)$, and find the exact difference $\text{fl}(x) - x$ for the given real numbers. Check that the relative rounding error is no more than $\epsilon_{\text{mach}}/2$. (a) $x = 1/3$ (b) $x = 3.3$ (c) $x = 9/7$
13. There are 64 double precision floating point numbers whose 64-bit machine representations have exactly one nonzero bit. Find the (a) largest (b) second-largest (c) smallest of these numbers.
14. Do the following operations by hand in IEEE double precision computer arithmetic, using the Rounding to Nearest Rule. (Check your answers, using MATLAB.) (a) $(4.3 - 3.3) - 1$ (b) $(4.4 - 3.4) - 1$ (c) $(4.9 - 3.9) - 1$
15. Do the following operations by hand in IEEE double precision computer arithmetic, using the Rounding to Nearest Rule. (a) $(8.3 - 7.3) - 1$ (b) $(8.4 - 7.4) - 1$ (c) $(8.8 - 7.8) - 1$

16. Find the IEEE double precision representation $\text{fl}(x)$, and find the exact difference $\text{fl}(x) - x$ for the given real numbers. Check that the relative rounding error is no more than $\epsilon_{\text{mach}}/2$.
 (a) $x = 2.75$ (b) $x = 2.7$ (c) $x = 10/3$

0.4 LOSS OF SIGNIFICANCE

An advantage of knowing the details of computer arithmetic is that we are therefore in a better position to understand potential pitfalls in computer calculations. One major problem that arises in many forms is the loss of significant digits that results from subtracting nearly equal numbers. In its simplest form, this is an obvious statement. Assume that through considerable effort, as part of a long calculation, we have determined two numbers correct to seven significant digits, and now need to subtract them:

$$\begin{array}{r} 123.4567 \\ - 123.4566 \\ \hline 000.0001 \end{array}$$

The subtraction problem began with two input numbers that we knew to seven-digit accuracy, and ended with a result that has only one-digit accuracy. Although this example is quite straightforward, there are other examples of loss of significance that are more subtle, and in many cases this can be avoided by restructuring the calculation.

► **EXAMPLE 0.5** Calculate $\sqrt{9.01} - 3$ on a three-decimal-digit computer.

This example is still fairly simple and is presented only for illustrative purposes. Instead of using a computer with a 52-bit mantissa, as in double precision IEEE standard format, we assume that we are using a three-decimal-digit computer. Using a three-digit computer means that storing each intermediate calculation along the way implies storing into a floating point number with a three-digit mantissa. The problem data (the 9.01 and 3.00) are given to three-digit accuracy. Since we are going to use a three-digit computer, being optimistic, we might hope to get an answer that is good to three digits. (Of course, we can't expect more than this because we only carry along three digits during the calculation.) Checking on a hand calculator, we see that the correct answer is approximately $0.0016662 = 1.6662 \times 10^{-3}$. How many correct digits do we get with the three-digit computer?

None, as it turns out. Since $\sqrt{9.01} \approx 3.0016662$, when we store this intermediate result to three significant digits we get 3.00. Subtracting 3.00, we get a final answer of 0.00. No significant digits in our answer are correct.

Surprisingly, there is a way to save this computation, even on a three-digit computer. What is causing the loss of significance is the fact that we are explicitly subtracting nearly equal numbers, $\sqrt{9.01}$ and 3. We can avoid this problem by using algebra to rewrite the expression:

$$\begin{aligned} \sqrt{9.01} - 3 &= \frac{(\sqrt{9.01} - 3)(\sqrt{9.01} + 3)}{\sqrt{9.01} + 3} \\ &= \frac{9.01 - 3^2}{\sqrt{9.01} + 3} \\ &= \frac{0.01}{3.00 + 3} = \frac{.01}{6} = 0.00167 \approx 1.67 \times 10^{-3}. \end{aligned}$$

Here, we have rounded the last digit of the mantissa up to 7 since the next digit is 6. Notice that we got all three digits correct this way, at least the three digits that the correct answer

rounds to. The lesson is that it is important to find ways to avoid subtracting nearly equal numbers in calculations, if possible. ◀

The method that worked in the preceding example was essentially a trick. Multiplying by the “conjugate expression” is one trick that can help restructure the calculation. Often, specific identities can be used, as with trigonometric expressions. For example, calculation of $1 - \cos x$ when x is close to zero is subject to loss of significance. Let’s compare the calculation of the expressions

$$E_1 = \frac{1 - \cos x}{\sin^2 x} \quad \text{and} \quad E_2 = \frac{1}{1 + \cos x}$$

for a range of input numbers x . We arrived at E_2 by multiplying the numerator and denominator of E_1 by $1 + \cos x$, and using the trig identity $\sin^2 x + \cos^2 x = 1$. In infinite precision, the two expressions are equal. Using the double precision of MATLAB computations, we get the following table:

x	E_1	E_2
1.000000000000000	0.64922320520476	0.64922320520476
0.100000000000000	0.50125208628858	0.50125208628857
0.010000000000000	0.50001250020848	0.50001250020834
0.001000000000000	0.50000012499219	0.50000012500002
0.000100000000000	0.49999999862793	0.50000000125000
0.000010000000000	0.50000004138685	0.50000000001250
0.000001000000000	0.50004445029134	0.50000000000013
0.000000100000000	0.49960036108132	0.50000000000000
0.000000010000000	0.00000000000000	0.50000000000000
0.000000001000000	0.00000000000000	0.50000000000000
0.000000000100000	0.00000000000000	0.50000000000000
0.000000000010000	0.00000000000000	0.50000000000000
0.000000000001000	0.00000000000000	0.50000000000000
0.000000000000100	0.00000000000000	0.50000000000000
0.000000000000010	0.00000000000000	0.50000000000000

The right column E_2 is correct up to the digits shown. The E_1 computation, due to the subtraction of nearly equal numbers, is having major problems below $x = 10^{-5}$ and has no correct significant digits for inputs $x = 10^{-8}$ and below.

The expression E_1 already has several incorrect digits for $x = 10^{-4}$ and gets worse as x decreases. The equivalent expression E_2 does not subtract nearly equal numbers and has no such problems.

The quadratic formula is often subject to loss of significance. Again, it is easy to avoid as long as you know it is there and how to restructure the expression.

► **EXAMPLE 0.6** Find both roots of the quadratic equation $x^2 + 9^{12}x = 3$.

Try this one in double precision arithmetic, for example, using MATLAB. Neither one will give the right answer unless you are aware of loss of significance and know how to counteract it. The problem is to find both roots, let’s say, with four-digit accuracy. So far it looks like an easy problem. The roots of a quadratic equation of form $ax^2 + bx + c = 0$ are given by the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \quad (0.12)$$

For our problem, this translates to

$$x = \frac{-9^{12} \pm \sqrt{9^{24} + 4(3)}}{2}.$$

Using the minus sign gives the root

$$x_1 = -2.824 \times 10^{11},$$

correct to four significant digits. For the plus sign root


$$x_2 = \frac{-9^{12} + \sqrt{9^{24} + 4(3)}}{2},$$

MATLAB calculates 0. Although the correct answer is close to 0, the answer has no correct significant digits—even though the numbers defining the problem were specified exactly (essentially with infinitely many correct digits) and despite the fact that MATLAB computes with approximately 16 significant digits (an interpretation of the fact that the machine epsilon of MATLAB is $2^{-52} \approx 2.2 \times 10^{-16}$). How do we explain the total failure to get accurate digits for x_2 ?

The answer is loss of significance. It is clear that 9^{12} and $\sqrt{9^{24} + 4(3)}$ are nearly equal, relatively speaking. More precisely, as stored floating point numbers, their mantissas not only start off similarly, but also are actually identical. When they are subtracted, as directed by the quadratic formula, of course the result is zero.

Can this calculation be saved? We must fix the loss of significance problem. The correct way to compute x_2 is by restructuring the quadratic formula:

$$\begin{aligned} x_2 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \\ &= \frac{(-b + \sqrt{b^2 - 4ac})(b + \sqrt{b^2 - 4ac})}{2a(b + \sqrt{b^2 - 4ac})} \\ &= \frac{-4ac}{2a(b + \sqrt{b^2 - 4ac})} \\ &= \frac{-2c}{(b + \sqrt{b^2 - 4ac})}. \end{aligned}$$

Substituting a, b, c for our example yields, according to MATLAB, $x_2 = 1.062 \times 10^{-11}$, which is correct to four significant digits of accuracy, as required. 

This example shows us that the quadratic formula (0.12) must be used with care in cases where a and/or c are small compared with b . More precisely, if $4|ac| \ll b^2$, then b and $\sqrt{b^2 - 4ac}$ are nearly equal in magnitude, and one of the roots is subject to loss of significance. If b is positive in this situation, then the two roots should be calculated as

$$x_1 = -\frac{b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_2 = -\frac{2c}{(b + \sqrt{b^2 - 4ac})}. \quad (0.13)$$

Note that neither formula suffers from subtracting nearly equal numbers. On the other hand, if b is negative and $4|ac| \ll b^2$, then the two roots are best calculated as

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad x_2 = \frac{2c}{(-b + \sqrt{b^2 - 4ac})}. \quad (0.14)$$

0.4 Exercises

- Identify for which values of x there is subtraction of nearly equal numbers, and find an alternate form that avoids the problem.

$$(a) \frac{1 - \sec x}{\tan^2 x} \quad (b) \frac{1 - (1 - x)^3}{x} \quad (c) \frac{1}{1 + x} - \frac{1}{1 - x}$$

- Find the roots of the equation $x^2 + 3x - 8^{-14} = 0$ with three-digit accuracy.
- Explain how to most accurately compute the two roots of the equation $x^2 + bx - 10^{-12} = 0$, where b is a number greater than 100.
- Prove formula 0.14.

0.4 Computer Problems

- Calculate the expressions that follow in double precision arithmetic (using MATLAB, for example) for $x = 10^{-1}, \dots, 10^{-14}$. Then, using an alternative form of the expression that doesn't suffer from subtracting nearly equal numbers, repeat the calculation and make a table of results. Report the number of correct digits in the original expression for each x .

$$(a) \frac{1 - \sec x}{\tan^2 x} \quad (b) \frac{1 - (1 - x)^3}{x}$$

- Find the smallest value of p for which the expression calculated in double precision arithmetic at $x = 10^{-p}$ has no correct significant digits. (Hint: First find the limit of the expression as $x \rightarrow 0$.)

$$(a) \frac{\tan x - x}{x^3} \quad (b) \frac{e^x + \cos x - \sin x - 2}{x^3}$$

- Evaluate the quantity $a + \sqrt{a^2 + b^2}$ to four correct significant digits, where $a = -12345678987654321$ and $b = 123$.
- Evaluate the quantity $\sqrt{c^2 + d} - c$ to four correct significant digits, where $c = 246886422468$ and $d = 13579$.
- Consider a right triangle whose legs are of length 3344556600 and 1.2222222. How much longer is the hypotenuse than the longer leg? Give your answer with at least four correct digits.

0.5 REVIEW OF CALCULUS

Some important basic facts from calculus will be necessary later. The Intermediate Value Theorem and the Mean Value Theorem are important for solving equations in Chapter 1. Taylor's Theorem is important for understanding interpolation in Chapter 3 and becomes of paramount importance for solving differential equations in Chapters 6, 7, and 8.

The graph of a continuous function has no gaps. For example, if the function is positive for one x -value and negative for another, it must pass through zero somewhere. This fact is basic for getting equation solvers to work in the next chapter. The first theorem, illustrated in Figure 0.1(a), generalizes this notion.

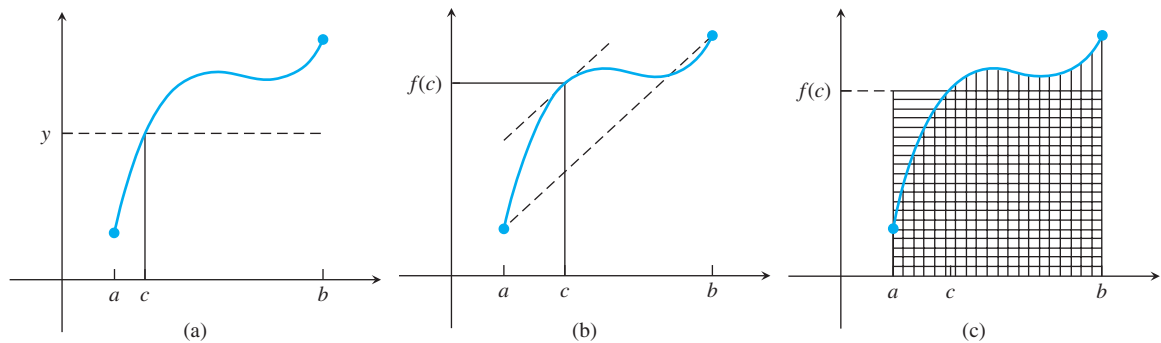


Figure 0.1 Three important theorems from calculus. There exist numbers c between a and b such that: (a) $f(c) = y$, for any given y between $f(a)$ and $f(b)$, by Theorem 0.4, the Intermediate Value Theorem (b) the instantaneous slope of f at c equals $(f(b) - f(a))/(b - a)$ by Theorem 0.6, the Mean Value Theorem (c) the vertically shaded region is equal in area to the horizontally shaded region, by Theorem 0.9, the Mean Value Theorem for Integrals, shown in the special case $g(x) = 1$.

THEOREM 0.4 (Intermediate Value Theorem) Let f be a continuous function on the interval $[a, b]$. Then f realizes every value between $f(a)$ and $f(b)$. More precisely, if y is a number between $f(a)$ and $f(b)$, then there exists a number c with $a \leq c \leq b$ such that $f(c) = y$. ■

► **EXAMPLE 0.7** Show that $f(x) = x^2 - 3$ on the interval $[1, 3]$ must take on the values 0 and 1.

Because $f(1) = -2$ and $f(3) = 6$, all values between -2 and 6 , including 0 and 1, must be taken on by f . For example, setting $c = \sqrt{3}$, note that $f(c) = f(\sqrt{3}) = 0$, and secondly, $f(2) = 1$. ◀

THEOREM 0.5 (Continuous Limits) Let f be a continuous function in a neighborhood of x_0 , and assume $\lim_{n \rightarrow \infty} x_n = x_0$. Then

$$\lim_{n \rightarrow \infty} f(x_n) = f\left(\lim_{n \rightarrow \infty} x_n\right) = f(x_0).$$

In other words, limits may be brought inside continuous functions.

THEOREM 0.6 (Mean Value Theorem) Let f be a continuously differentiable function on the interval $[a, b]$. Then there exists a number c between a and b such that $f'(c) = (f(b) - f(a))/(b - a)$. ■

► **EXAMPLE 0.8** Apply the Mean Value Theorem to $f(x) = x^2 - 3$ on the interval $[1, 3]$.

The content of the theorem is that because $f(1) = -2$ and $f(3) = 6$, there must exist a number c in the interval $(1, 3)$ satisfying $f'(c) = (6 - (-2))/(3 - 1) = 4$. It is easy to find such a c . Since $f'(x) = 2x$, the correct $c = 2$. ◀

The next statement is a special case of the Mean Value Theorem.

THEOREM 0.7 (Rolle's Theorem) Let f be a continuously differentiable function on the interval $[a, b]$, and assume that $f(a) = f(b)$. Then there exists a number c between a and b such that $f'(c) = 0$. ■

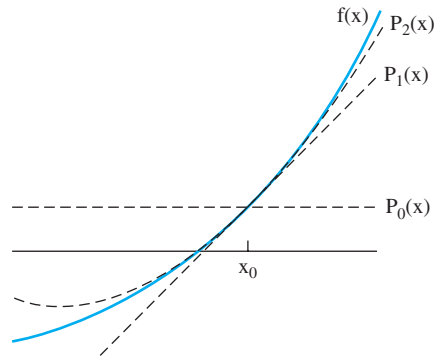


Figure 0.2 Taylor's Theorem with Remainder. The function $f(x)$, denoted by the solid curve, is approximated successively better near x_0 by the degree 0 Taylor polynomial (horizontal dashed line), the degree 1 Taylor polynomial (slanted dashed line), and the degree 2 Taylor polynomial (dashed parabola). The difference between $f(x)$ and its approximation at x is the Taylor remainder.

Taylor approximation underlies many simple computational techniques that we will study. If a function f is known well at a point x_0 , then a lot of information about f at nearby points can be learned. If the function is continuous, then for points x near x_0 , the function value $f(x)$ will be approximated reasonably well by $f(x_0)$. However, if $f'(x_0) > 0$, then f has greater values for nearby points to the right, and lesser values for points to the left, since the slope near x_0 is approximately given by the derivative. The line through $(x_0, f(x_0))$ with slope $f'(x_0)$, shown in Figure 0.2, is the Taylor approximation of degree 1. Further small corrections can be extracted from higher derivatives, and give the higher degree Taylor approximations. Taylor's Theorem uses the entire set of derivatives at x_0 to give a full accounting of the function values in a small neighborhood of x_0 .

THEOREM 0.8 (Taylor's Theorem with Remainder) Let x and x_0 be real numbers, and let f be $k + 1$ times continuously differentiable on the interval between x and x_0 . Then there exists a number c between x and x_0 such that

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \cdots + \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{f^{(k+1)}(c)}{(k+1)!}(x - x_0)^{k+1}.$$


■

The polynomial part of the result, the terms up to degree k in $x - x_0$, is called the **degree k Taylor polynomial** for f centered at x_0 . The final term is called the **Taylor remainder**. To the extent that the Taylor remainder term is small, Taylor's Theorem gives a way to approximate a general, smooth function with a polynomial. This is very convenient in solving problems with a computer, which, as mentioned earlier, can evaluate polynomials very efficiently.

► **EXAMPLE 0.9** Find the degree 4 Taylor polynomial $P_4(x)$ for $f(x) = \sin x$ centered at the point $x_0 = 0$. Estimate the maximum possible error when using $P_4(x)$ to estimate $\sin x$ for $|x| \leq 0.0001$.


The polynomial is easily calculated to be $P_4(x) = x - x^3/6$. Note that the degree 4 term is absent, since its coefficient is zero. The remainder term is

$$\frac{x^5}{120} \cos c,$$

which in absolute value cannot be larger than $|x|^5/120$. For $|x| \leq 0.0001$, the remainder is at most $10^{-20}/120$ and will be invisible when, for example, $x - x^3/6$ is used in double precision to approximate $\sin 0.0001$. Check this by computing both in MATLAB. 

Finally, the integral version of the Mean Value Theorem is illustrated in Figure 0.1(c).

THEOREM 0.9 (Mean Value Theorem for Integrals) Let f be a continuous function on the interval $[a, b]$, and let g be an integrable function that does not change sign on $[a, b]$. Then there exists a number c between a and b such that

$$\int_a^b f(x)g(x) dx = f(c) \int_a^b g(x) dx.$$


0.5 Exercises

- Use the Intermediate Value Theorem to prove that $f(c) = 0$ for some $0 < c < 1$.
(a) $f(x) = x^3 - 4x + 1$ (b) $f(x) = 5 \cos \pi x - 4$ (c) $f(x) = 8x^4 - 8x^2 + 1$
- Find c satisfying the Mean Value Theorem for $f(x)$ on the interval $[0, 1]$. (a) $f(x) = e^x$
(b) $f(x) = x^2$ (c) $f(x) = 1/(x + 1)$
- Find c satisfying the Mean Value Theorem for Integrals with $f(x), g(x)$ in the interval $[0, 1]$.
(a) $f(x) = x, g(x) = x$ (b) $f(x) = x^2, g(x) = x$ (c) $f(x) = x, g(x) = e^x$
- Find the Taylor polynomial of degree 2 about the point $x = 0$ for the following functions:
(a) $f(x) = e^{x^2}$ (b) $f(x) = \cos 5x$ (c) $f(x) = 1/(x + 1)$
- Find the Taylor polynomial of degree 5 about the point $x = 0$ for the following functions:
(a) $f(x) = e^{x^2}$ (b) $f(x) = \cos 2x$ (c) $f(x) = \ln(1 + x)$ (d) $f(x) = \sin^2 x$
- (a) Find the Taylor polynomial of degree 4 for $f(x) = x^{-2}$ about the point $x = 1$.
(b) Use the result of (a) to approximate $f(0.9)$ and $f(1.1)$.
(c) Use the Taylor remainder to find an error formula for the Taylor polynomial. Give error bounds for each of the two approximations made in part (b). Which of the two approximations in part (b) do you expect to be closer to the correct value?
(d) Use a calculator to compare the actual error in each case with your error bound from part (c).
- Carry out Exercise 6 (a)–(d) for $f(x) = \ln x$.
- (a) Find the degree 5 Taylor polynomial $P(x)$ centered at $x = 0$ for $f(x) = \cos x$. (b) Find an upper bound for the error in approximating $f(x) = \cos x$ for x in $[-\pi/4, \pi/4]$ by $P(x)$.
- A common approximation for $\sqrt{1+x}$ is $1 + \frac{1}{2}x$, when x is small. Use the degree 1 Taylor polynomial of $f(x) = \sqrt{1+x}$ with remainder to determine a formula of form $\sqrt{1+x} = 1 + \frac{1}{2}x \pm E$. Evaluate E for the case of approximating $\sqrt{1.02}$. Use a calculator to compare the actual error to your error bound E .

Software and Further Reading

The IEEE standard for floating point computation is published as IEEE Standard 754 [1985]. Goldberg [1991] and Stallings [2003] discuss floating point arithmetic in great detail, and Overton [2001] emphasizes the IEEE 754 standard. The texts Wilkinson [1994] and Knuth [1981] had great influence on the development of both hardware and software.

There are several software packages that specialize in general-purpose scientific computing, the bulk of it done in floating point arithmetic. Netlib (<http://www.netlib.org>) is a collection of free software maintained by AT&T Bell Laboratories, the University of Tennessee, and Oak Ridge National Laboratory. The collection consists of high-quality programs available in Fortran, C, and Java, but it comes with little support. The comments in the code are meant to be sufficiently instructive for the user to operate the program.

The Numerical Algorithms Group (NAG) (<http://www.nag.co.uk>) markets a library containing over 1400 user-callable subroutines for solving general applied math problems. The programs are available in Fortran and C and are callable from Java programs. NAG includes libraries for shared memory and distributed memory computing.

The International Mathematics and Statistics Library (IMSL) is a product of Rogue Wave Software (www.roguewave.com), and covers areas similar to those covered by the NAG library. Fortran, C, and Java programs are available. It also provides PV-WAVE, a powerful programming language with data analysis and visualization capabilities.

The computing environments Mathematica, Maple, and MATLAB have grown to encompass many of the same computational methods previously described and have built-in editing and graphical interfaces. Mathematica (<http://www.wolframresearch.com>) and Maple (www.maplesoft.com) came to prominence due to novel symbolic computing engines. MATLAB has grown to serve many science and engineering applications through “tool-boxes,” which leverage the basic high-quality software into diverse directions.

In this text, we frequently illustrate basic algorithms with MATLAB implementations. The MATLAB code given is meant to be instructional only. Quite often, speed and reliability are sacrificed for clarity and readability. Readers who are new to MATLAB should begin with the tutorial in Appendix B; they will soon be doing their own implementations.



Solving Equations

A recently excavated cuneiform tablet shows that the Babylonians calculated the square root of 2 correctly to within five decimal places. Their technique is unknown, but in this chapter we introduce iterative methods that they may have used and that are still used by modern calculators to find square roots.

The Stewart platform, a six-degree-of-freedom robot that can be located with extreme precision, was originally developed by Eric Gough of Dunlop Tire Corporation in the 1950s to test airplane tires. Today its

applications range from flight simulators, which are often of considerable mass, to medical and surgical applications, where precision is very important. Solving the forward kinematics problem requires determining the position and orientation of the platform, given the strut lengths.

Reality Check ✓

Reality Check 1 on page 67 uses the methods developed in this chapter to solve the forward kinematics of a planar version of the Stewart platform.

Equation solving is one of the most basic problems in scientific computing. This chapter introduces a number of iterative methods for locating solutions x of the equation $f(x) = 0$. These methods are of great practical importance. In addition, they illustrate the central roles of convergence and complexity in scientific computing.

Why is it necessary to know more than one method for solving equations? Often, the choice of method will depend on the cost of evaluating the function f and perhaps its derivative. If $f(x) = e^x - \sin x$, it may take less than one-millionth of a second to determine $f(x)$, and its derivative is available if needed. If $f(x)$ denotes the freezing temperature of an ethylene glycol solution under x atmospheres of pressure, each function evaluation may require considerable time in a well-equipped laboratory, and determining the derivative may be infeasible.

In addition to introducing methods such as the Bisection Method, Fixed-Point Iteration, and Newton's Method, we will analyze their rates of convergence and discuss their computational complexity. Later, more sophisticated equation solvers are presented, including Brent's Method, that combines the best properties of several solvers.

1.1 THE BISECTION METHOD

How do you look up a name in an unfamiliar phone book? To look up “Smith,” you might begin by opening the book at your best guess, say, the letter Q. Next you may turn a sheaf of pages and end up at the letter U. Now you have “bracketed” the name Smith and need to hone in on it by using smaller and smaller brackets that eventually converge to the name. The Bisection Method represents this type of reasoning, done as efficiently as possible.

1.1.1 Bracketing a root

DEFINITION 1.1 The function $f(x)$ has a **root** at $x = r$ if $f(r) = 0$. □

The first step to solving an equation is to verify that a root exists. One way to ensure this is to bracket the root: to find an interval $[a, b]$ on the real line for which one of the pair $\{f(a), f(b)\}$ is positive and the other is negative. This can be expressed as $f(a)f(b) < 0$. If f is a continuous function, then there will be a root: an r between a and b for which $f(r) = 0$. This fact is summarized in the following corollary of the Intermediate Value Theorem 0.4:

THEOREM 1.2 Let f be a continuous function on $[a, b]$, satisfying $f(a)f(b) < 0$. Then f has a root between a and b , that is, there exists a number r satisfying $a < r < b$ and $f(r) = 0$. ■

In Figure 1.1, $f(0)f(1) = (-1)(1) < 0$. There is a root just to the left of 0.7. How can we refine our first guess of the root’s location to more decimal places?

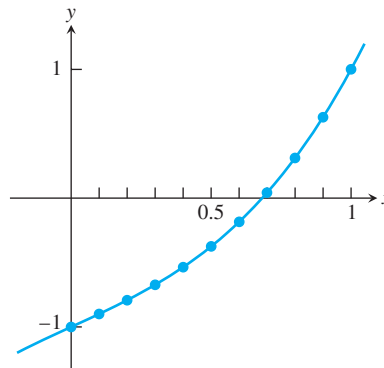


Figure 1.1 A plot of $f(x) = x^3 + x - 1$. The function has a root between 0.6 and 0.7.

We’ll take a cue from the way our eye finds a solution when given a plot of a function. It is unlikely that we start at the left end of the interval and move to the right, stopping at the root. Perhaps a better model of what happens is that the eye first decides the general location, such as whether the root is toward the left or the right of the interval. It then follows that up by deciding more precisely just how far right or left the root lies and gradually improves its accuracy, just like looking up a name in the phone book. This general approach is made quite specific in the Bisection Method, shown in Figure 1.2.

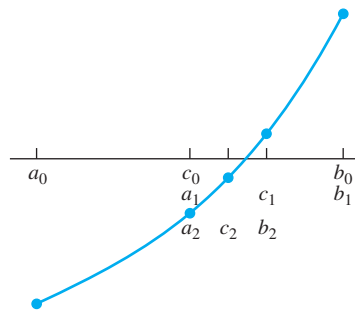


Figure 1.2 The Bisection Method. On the first step, the sign of $f(c_0)$ is checked. Since $f(c_0)f(b_0) < 0$, set $a_1 = c_0, b_1 = b_0$, and the interval is replaced by the right half $[a_1, b_1]$. On the second step, the subinterval is replaced by its left half $[a_2, b_2]$.

Bisection Method

Given initial interval $[a, b]$ such that $f(a)f(b) < 0$

while $(b - a)/2 > \text{TOL}$

$c = (a + b)/2$

if $f(c) = 0$, **stop, end**

if $f(a)f(c) < 0$

$b = c$

else

$a = c$

end

end

The final interval $[a, b]$ contains a root.

The approximate root is $(a + b)/2$.

Check the value of the function at the midpoint $c = (a + b)/2$ of the interval. Since $f(a)$ and $f(b)$ have opposite signs, either $f(c) = 0$ (in which case we have found a root and are done), or the sign of $f(c)$ is opposite the sign of either $f(a)$ or $f(b)$. If $f(c)f(a) < 0$, for example, we are assured a solution in the interval $[a, c]$, whose length is half that of the original interval $[a, b]$. If instead $f(c)f(b) < 0$, we can say the same of the interval $[c, b]$. In either case, one step reduces the problem to finding a root on an interval of one-half the original size. This step can be repeated to locate the function more and more accurately.

A solution is bracketed by the new interval at each step, reducing the uncertainty in the location of the solution as the interval becomes smaller. An entire plot of the function f is not needed. We have reduced the work of function evaluation to only what is necessary.


► **EXAMPLE 1.1** Find a root of the function $f(x) = x^3 + x - 1$ by using the Bisection Method on the interval $[0, 1]$.

As noted, $f(a_0)f(b_0) = (-1)(1) < 0$, so a root exists in the interval. The interval midpoint is $c_0 = 1/2$. The first step consists of evaluating $f(1/2) = -3/8 < 0$ and choosing the new interval $[a_1, b_1] = [1/2, 1]$, since $f(1/2)f(1) < 0$. The second step consists of

evaluating $f(c_1) = f(3/4) = 11/64 > 0$, leading to the new interval $[a_2, b_2] = [1/2, 3/4]$. Continuing in this way yields the following intervals:

i	a_i	$f(a_i)$	c_i	$f(c_i)$	b_i	$f(b_i)$
0	0.0000	—	0.5000	—	1.0000	+
1	0.5000	—	0.7500	+	1.0000	+
2	0.5000	—	0.6250	—	0.7500	+
3	0.6250	—	0.6875	+	0.7500	+
4	0.6250	—	0.6562	—	0.6875	+
5	0.6562	—	0.6719	—	0.6875	+
6	0.6719	—	0.6797	—	0.6875	+
7	0.6797	—	0.6836	+	0.6875	+
8	0.6797	—	0.6816	—	0.6836	+
9	0.6816	—	0.6826	+	0.6836	+

We conclude from the table that the solution is bracketed between $a_9 \approx 0.6816$ and $c_9 \approx 0.6826$. The midpoint of that interval $c_{10} \approx 0.6821$ is our best guess for the root.

Although the problem was to find a root, what we have actually found is an interval $[0.6816, 0.6826]$ that contains a root; in other words, the root is $r = 0.6821 \pm 0.0005$. We will have to be satisfied with an approximation. Of course, the approximation can be improved, if needed, by completing more steps of the Bisection Method. 

At each step of the Bisection Method, we compute the midpoint $c_i = (a_i + b_i)/2$ of the current interval $[a_i, b_i]$, calculate $f(c_i)$, and compare signs. If $f(c_i)f(a_i) < 0$, we set $a_{i+1} = a_i$ and $b_{i+1} = c_i$. If, instead, $f(c_i)f(a_i) > 0$, we set $a_{i+1} = c_i$ and $b_{i+1} = b_i$. Each step requires one new evaluation of the function f and bisects the interval containing a root, reducing its length by a factor of 2. After n steps of calculating c and $f(c)$, we have done $n + 2$ function evaluations, and our best estimate of the solution is the midpoint of the latest interval. The algorithm can be written in the following MATLAB code:

```
%Program 1.1 Bisection Method
%Computes approximate solution of f(x)=0
%Input: function handle f; a,b such that f(a)*f(b)<0,
%       and tolerance tol
%Output: Approximate solution xc
function xc=bisect(f,a,b,tol)
if sign(f(a))*sign(f(b)) >= 0
    error('f(a)f(b)<0 not satisfied!') %ceases execution
end
fa=f(a);
fb=f(b);
while (b-a)/2>tol
    c=(a+b)/2;
    fc=f(c);
    if fc == 0
        %c is a solution, done
```

```

        break
    end
    if sign(fc)*sign(fa)<0 %a and c make the new interval
        b=c;fb=fc;
    else %c and b make the new interval
        a=c;fa=fc;
    end
end
xc=(a+b)/2; %new midpoint is best estimate

```

To use `bisect.m`, first define a MATLAB function by:

```
>> f=@(x) x^3+x-1;
```

This command actually defines a “function handle” `f`, which can be used as input for other MATLAB functions. See Appendix B for more details on MATLAB functions and function handles. Then the command

```
» xc=bisect (f,0,1,0.00005)
```

returns a solution correct to a tolerance of 0.00005.

1.1.2 How accurate and how fast?

If $[a, b]$ is the starting interval, then after n bisection steps, the interval $[a_n, b_n]$ has length $(b - a)/2^n$. Choosing the midpoint $x_c = (a_n + b_n)/2$ gives a best estimate of the solution r , which is within half the interval length of the true solution. Summarizing, after n steps of the Bisection Method, we find that

$$\text{Solution error} = |x_c - r| < \frac{b - a}{2^{n+1}} \quad (1.1)$$

and

$$\text{Function evaluations} = n + 2. \quad (1.2)$$

A good way to assess the efficiency of the Bisection Method is to ask how much accuracy can be bought per function evaluation. Each step, or each function evaluation, cuts the uncertainty in the root by a factor of two.

DEFINITION 1.3 A solution is **correct within p decimal places** if the error is less than 0.5×10^{-p} . □

► **EXAMPLE 1.2** Use the Bisection Method to find a root of $f(x) = \cos x - x$ in the interval $[0, 1]$ to within six correct places.

First we decide how many steps of bisection are required. According to (1.1), the error after n steps is $(b - a)/2^{n+1} = 1/2^{n+1}$. From the definition of p decimal places, we require that

$$\begin{aligned} \frac{1}{2^{n+1}} &< 0.5 \times 10^{-6} \\ n &> \frac{6}{\log_{10} 2} \approx \frac{6}{0.301} = 19.9. \end{aligned}$$

Therefore, $n = 20$ steps will be needed. Proceeding with the Bisection Method, the following table is produced:

k	a_k	$f(a_k)$	c_k	$f(c_k)$	b_k	$f(b_k)$
0	0.000000	+	0.500000	+	1.000000	—
1	0.500000	+	0.750000	—	1.000000	—
2	0.500000	+	0.625000	+	0.750000	—
3	0.625000	+	0.687500	+	0.750000	—
4	0.687500	+	0.718750	+	0.750000	—
5	0.718750	+	0.734375	+	0.750000	—
6	0.734375	+	0.742188	—	0.750000	—
7	0.734375	+	0.738281	+	0.742188	—
8	0.738281	+	0.740234	—	0.742188	—
9	0.738281	+	0.739258	—	0.740234	—
10	0.738281	+	0.738770	+	0.739258	—
11	0.738769	+	0.739014	+	0.739258	—
12	0.739013	+	0.739136	—	0.739258	—
13	0.739013	+	0.739075	+	0.739136	—
14	0.739074	+	0.739105	—	0.739136	—
15	0.739074	+	0.739090	—	0.739105	—
16	0.739074	+	0.739082	+	0.739090	—
17	0.739082	+	0.739086	—	0.739090	—
18	0.739082	+	0.739084	+	0.739086	—
19	0.739084	+	0.739085	—	0.739086	—
20	0.739084	+	0.739085	—	0.739085	—

The approximate root to six correct places is 0.739085.

For the Bisection Method, the question of how many steps to run is a simple one—just choose the desired precision and find the number of necessary steps, as in (1.1). We will see that more high-powered algorithms are often less predictable and have no analogue to (1.1). In those cases, we will need to establish definite “stopping criteria” that govern the circumstances under which the algorithm terminates. Even for the Bisection Method, the finite precision of computer arithmetic will put a limit on the number of possible correct digits. We will look into this issue further in Section 1.3.

1.1 Exercises

- Use the Intermediate Value Theorem to find an interval of length one that contains a root of the equation. (a) $x^3 = 9$ (b) $3x^3 + x^2 = x + 5$ (c) $\cos^2 x + 6 = x$
- Use the Intermediate Value Theorem to find an interval of length one that contains a root of the equation. (a) $x^5 + x = 1$ (b) $\sin x = 6x + 5$ (c) $\ln x + x^2 = 3$
- Consider the equations in Exercise 1. Apply two steps of the Bisection Method to find an approximate root within $1/8$ of the true root.
- Consider the equations in Exercise 2. Apply two steps of the Bisection Method to find an approximate root within $1/8$ of the true root.
- Consider the equation $x^4 = x^3 + 10$.
 - Find an interval $[a, b]$ of length one inside which the equation has a solution.
 - Starting with $[a, b]$, how many steps of the Bisection Method are required to calculate the solution within 10^{-10} ? Answer with an integer.
- Suppose that the Bisection Method with starting interval $[-2, 1]$ is used to find a root of the function $f(x) = 1/x$. Does the method converge to a real number? Is it the root?

1.1 Computer Problems

- Use the Bisection Method to find the root to six correct decimal places. (a) $x^3 = 9$
(b) $3x^3 + x^2 = x + 5$ (c) $\cos^2 x + 6 = x$
- Use the Bisection Method to find the root to eight correct decimal places. (a) $x^5 + x = 1$
(b) $\sin x = 6x + 5$ (c) $\ln x + x^2 = 3$
- Use the Bisection Method to locate all solutions of the following equations. Sketch the function by using MATLAB's `plot` command and identify three intervals of length one that contain a root. Then find the roots to six correct decimal places. (a) $2x^3 - 6x - 1 = 0$
(b) $e^{x-2} + x^3 - x = 0$ (c) $1 + 5x - 6x^3 - e^{2x} = 0$
- Calculate the square roots of the following numbers to eight correct decimal places by using the Bisection Method to solve $x^2 - A = 0$, where A is (a) 2 (b) 3 (c) 5. State your starting interval and the number of steps needed.
- Calculate the cube roots of the following numbers to eight correct decimal places by using the Bisection Method to solve $x^3 - A = 0$, where A is (a) 2 (b) 3 (c) 5. State your starting interval and the number of steps needed.
- Use the Bisection Method to calculate the solution of $\cos x = \sin x$ in the interval $[0, 1]$ within six correct decimal places.
- Use the Bisection Method to find the two real numbers x , within six correct decimal places, that make the determinant of the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 & x \\ 4 & 5 & x & 6 \\ 7 & x & 8 & 9 \\ x & 10 & 11 & 12 \end{bmatrix}$$

equal to 1000. For each solution you find, test it by computing the corresponding determinant and reporting how many correct decimal places (after the decimal point) the determinant has when your solution x is used. (In Section 1.2, we will call this the “backward error” associated with the approximate solution.) You may use the MATLAB command `det` to compute the determinants.

- The **Hilbert matrix** is the $n \times n$ matrix whose ij th entry is $1/(i + j - 1)$. Let A denote the 5×5 Hilbert matrix. Its largest eigenvalue is about 1.567. Use the Bisection Method to decide how to change the upper left entry A_{11} to make the largest eigenvalue of A equal to π . Determine A_{11} within six correct decimal places. You may use the MATLAB commands `hilb`, `pi`, `eig`, and `max` to simplify your task.
- Find the height reached by 1 cubic meter of water stored in a spherical tank of radius 1 meter. Give your answer ± 1 mm. (Hint: First note that the sphere will be less than half full. The volume of the bottom H meters of a hemisphere of radius R is $\pi H^2(R - 1/3H)$.)

1.2 FIXED-POINT ITERATION

Use a calculator or computer to apply the `cos` function repeatedly to an arbitrary starting number. That is, apply the `cos` function to the starting number, then apply `cos` to the result, then to the new result, and so forth. (If you use a calculator, be sure it is in radian

mode.) Continue until the digits no longer change. The resulting sequence of numbers converges to 0.7390851332, at least to the first 10 decimal places. In this section, our goal is to explain why this calculation, an instance of Fixed-Point Iteration (FPI), converges. While we do this, most of the major issues of algorithm convergence will come under discussion.

1.2.1 Fixed points of a function

The sequence of numbers produced by iterating the cosine function appears to converge to a number r . Subsequent applications of cosine do not change the number. For this input, the output of the cosine function is equal to the input, or $\cos r = r$.

DEFINITION 1.4 The real number r is a **fixed point** of the function g if $g(r) = r$. □

The number $r = 0.7390851332$ is an approximate fixed point for the function $g(x) = \cos x$. The function $g(x) = x^3$ has three fixed points, $r = -1, 0$, and 1 .

We used the Bisection Method in Example 1.2 to solve the equation $\cos x - x = 0$. The fixed-point equation $\cos x = x$ is the same problem from a different point of view. When the output equals the input, that number is a fixed point of $\cos x$, and simultaneously a solution of the equation $\cos x - x = 0$.

Once the equation is written as $g(x) = x$, Fixed-Point Iteration proceeds by starting with an initial guess x_0 and iterating the function g .

Fixed-Point Iteration

$$\begin{aligned} x_0 &= \text{initial guess} \\ x_{i+1} &= g(x_i) \text{ for } i = 0, 1, 2, \dots \end{aligned}$$

Therefore,

$$\begin{aligned} x_1 &= g(x_0) \\ x_2 &= g(x_1) \\ x_3 &= g(x_2) \\ &\vdots \end{aligned}$$

and so forth. The sequence x_i may or may not converge as the number of steps goes to infinity. However, if g is continuous and the x_i converge, say, to a number r , then r is a fixed point. In fact, Theorem 0.5 implies that

$$g(r) = g\left(\lim_{i \rightarrow \infty} x_i\right) = \lim_{i \rightarrow \infty} g(x_i) = \lim_{i \rightarrow \infty} x_{i+1} = r. \quad (1.3)$$

The Fixed-Point Iteration algorithm applied to a function g is easily written in MATLAB code:

```
%Program 1.2 Fixed-Point Iteration
%Computes approximate solution of g(x)=x
%Input: function handle g, starting guess x0,
%       number of iteration steps k
%Output: Approximate solution xc
function xc=fpi(g, x0, k)
x(1)=x0;
```

```

for i=1:k
    x(i+1)=g(x(i));
end
xc=x(k+1);

```

After defining a MATLAB function by

```
>> g=@(x) cos(x)
```

the code of Program 1.2 can be called as

```
>> xc=fpi(g,0,10)
```

to run 10 steps of Fixed-Point Iteration with initial guess 0.

Fixed-Point Iteration solves the fixed-point problem $g(x) = x$, but we are primarily interested in solving equations. Can every equation $f(x) = 0$ be turned into a fixed-point problem $g(x) = x$? Yes, and in many different ways. For example, the root-finding equation of Example 1.1,

$$x^3 + x - 1 = 0, \quad (1.4)$$

can be rewritten as

$$x = 1 - x^3, \quad (1.5)$$

and we may define $g(x) = 1 - x^3$. Alternatively, the x^3 term in (1.4) can be isolated to yield

$$x = \sqrt[3]{1 - x}, \quad (1.6)$$

where $g(x) = \sqrt[3]{1 - x}$. As a third and not very obvious approach, we might add $2x^3$ to both sides of (1.4) to get

$$\begin{aligned}
 3x^3 + x &= 1 + 2x^3 \\
 (3x^2 + 1)x &= 1 + 2x^3 \\
 x &= \frac{1 + 2x^3}{1 + 3x^2}
 \end{aligned} \quad (1.7)$$

and define $g(x) = (1 + 2x^3)/(1 + 3x^2)$.

Next, we demonstrate Fixed-Point Iteration for the preceding three choices of $g(x)$. The underlying equation to be solved is $x^3 + x - 1 = 0$. First we consider the form $x = g(x) = 1 - x^3$. The starting point $x_0 = 0.5$ is chosen somewhat arbitrarily. Applying FPI gives the following result:

i	x_i
0	0.50000000
1	0.87500000
2	0.33007813
3	0.96403747
4	0.10405419
5	0.99887338
6	0.00337606
7	0.99999996
8	0.00000012
9	1.00000000
10	0.00000000
11	1.00000000
12	0.00000000

Instead of converging, the iteration tends to alternate between the numbers 0 and 1. Neither is a fixed point, since $g(0) = 1$ and $g(1) = 0$. The Fixed-Point Iteration fails. With the Bisection Method, we know that if f is continuous and $f(a)f(b) < 0$ on the original interval, we must see convergence to the root. This is not so for FPI.

The second choice is $g(x) = \sqrt[3]{1-x}$. We will keep the same initial guess, $x_0 = 0.5$.

i	x_i	i	x_i
0	0.50000000	13	0.68454401
1	0.79370053	14	0.68073737
2	0.59088011	15	0.68346460
3	0.74236393	16	0.68151292
4	0.63631020	17	0.68291073
5	0.71380081	18	0.68191019
6	0.65900615	19	0.68262667
7	0.69863261	20	0.68211376
8	0.67044850	21	0.68248102
9	0.69072912	22	0.68221809
10	0.67625892	23	0.68240635
11	0.68664554	24	0.68227157
12	0.67922234	25	0.68236807

This time FPI is successful. The iterates are apparently converging to a number near 0.6823.

Finally, let's use the rearrangement $x = g(x) = (1 + 2x^3)/(1 + 3x^2)$. As in the previous case, there is convergence, but in a much more striking way.

i	x_i
0	0.50000000
1	0.71428571
2	0.68317972
3	0.68232842
4	0.68232780
5	0.68232780
6	0.68232780
7	0.68232780

Here we have four correct digits after four iterations of Fixed-Point Iteration, and many more correct digits soon after. Compared with the previous attempts, this is an astonishing result. Our next goal is to try to explain the differences between the three outcomes.

1.2.2 Geometry of Fixed-Point Iteration

In the previous section, we found three different ways to rewrite the equation $x^3 + x - 1 = 0$ as a fixed-point problem, with varying results. To find out why the FPI method converges in some situations and not in others, it is helpful to look at the geometry of the method.

Figure 1.3 shows the three different $g(x)$ discussed before, along with an illustration of the first few steps of FPI in each case. The fixed point r is the same for each $g(x)$. It is represented by the point where the graphs $y = g(x)$ and $y = x$ intersect. Each step of FPI can be sketched by drawing line segments (1) **vertically to the function** and then (2) **horizontally to the diagonal** line $y = x$. The vertical and horizontal arrows in Figure 1.3 follow the steps made by FPI. The vertical arrow moving from the x -value to the function g represents $x_i \rightarrow g(x_i)$. The horizontal arrow represents turning the output $g(x_i)$ on the y -axis and transforming it into the same number x_{i+1} on the x -axis, ready to be input into g in the next step. This is done by drawing the horizontal line segment from the output

height $g(x_i)$ across to the diagonal line $y = x$. This geometric illustration of a Fixed-Point Iteration is called a **cobweb diagram**.

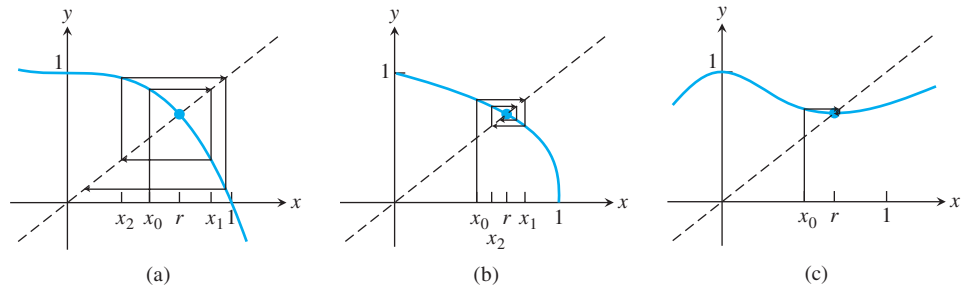


Figure 1.3 Geometric view of FPI. The fixed point is the intersection of $g(x)$ and the diagonal line. Three examples of $g(x)$ are shown together with the first few steps of FPI. (a) $g(x) = 1 - x^3$ (b) $g(x) = (1 - x)^{1/3}$ (c) $g(x) = (1 + 2x^3)/(1 + 3x^2)$

In Figure 1.3(a), the path starts at $x_0 = 0.5$, and moves up to the function and horizontal to the point $(0.875, 0.875)$ on the diagonal, which is (x_1, x_1) . Next, x_1 should be substituted into $g(x)$. This is done the same way it was done for x_0 , by moving vertically to the function. This yields $x_2 \approx 0.3300$, and after moving horizontally to move the y -value to an x -value, we continue the same way to get x_3, x_4, \dots . As we saw earlier, the result of FPI for this $g(x)$ is not successful—the iterates eventually tend toward alternating between 0 and 1, neither of which are fixed points.

Fixed-Point Iteration is more successful in Figure 1.3(b). Although the $g(x)$ here looks roughly similar to the $g(x)$ in part (a), there is a significant difference, which we will clarify in the next section. You may want to speculate on what the difference is. What makes FPI spiral in toward the fixed point in (b), and spiral out away from the fixed point in (a)? Figure 1.3(c) shows an example of very fast convergence. Does this picture help with your speculation? If you guessed that it has something to do with the slope of $g(x)$ near the fixed point, you are correct.

1.2.3 Linear convergence of Fixed-Point Iteration

The convergence properties of FPI can be easily explained by a careful look at the algorithm in the simplest possible situation. Figure 1.4 shows Fixed-Point Iteration for two linear functions $g_1(x) = -\frac{3}{2}x + \frac{5}{2}$ and $g_2(x) = -\frac{1}{2}x + \frac{3}{2}$. In each case, the fixed point is $x = 1$, but $|g'_1(1)| = |-\frac{3}{2}| > 1$ while $|g'_2(1)| = |-\frac{1}{2}| < 1$. Following the vertical and horizontal arrows that describe FPI, we see the reason for the difference. Because the slope of g_1 at the fixed point is greater than one, the vertical segments, the ones that represent the change from x_n to x_{n+1} , are increasing in length as FPI proceeds. As a result, the iteration “spirals out” from the fixed point $x = 1$, even if the initial guess x_0 was quite near. For g_2 , the situation is reversed: The slope of g_2 is less than one, the vertical segments decrease in length, and FPI “spirals in” toward the solution. Thus, $|g'(r)|$ makes the crucial difference between divergence and convergence.

That’s the geometric view. In terms of equations, it helps to write $g_1(x)$ and $g_2(x)$ in terms of $x - r$, where $r = 1$ is the fixed point:

$$\begin{aligned} g_1(x) &= -\frac{3}{2}(x - 1) + 1 \\ g_1(x) - 1 &= -\frac{3}{2}(x - 1) \\ x_{i+1} - 1 &= -\frac{3}{2}(x_i - 1). \end{aligned} \tag{1.8}$$

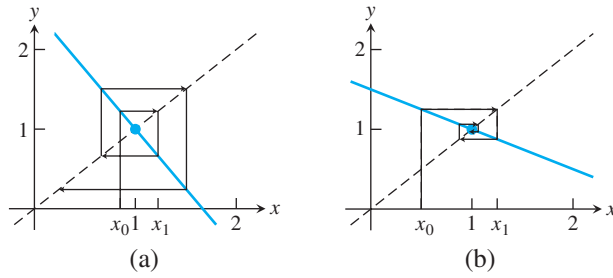


Figure 1.4 Cobweb diagram for linear functions. (a) If the linear function has slope greater than one in absolute value, nearby guesses move farther from the fixed point as FPI progresses, leading to failure of the method. (b) For slope less than one in absolute value, the reverse happens, and the fixed point is found.

If we view $e_i = |r - x_i|$ as the error at step i (meaning the distance from the best guess at step n to the fixed point), we see from (1.8) that $e_{i+1} = 3e_i/2$, implying that errors increase at each step by a factor of approximately $3/2$. This is divergence.

Repeating the preceding algebra for g_2 , we have

$$\begin{aligned} g_2(x) &= -\frac{1}{2}(x - 1) + 1 \\ g_2(x) - 1 &= -\frac{1}{2}(x - 1) \\ x_{i+1} - 1 &= -\frac{1}{2}(x_i - 1). \end{aligned}$$

The result is $e_{i+1} = e_i/2$, implying that the error, the distance to the fixed point, is multiplied by $1/2$ on each step. The error decreases to zero as the number of steps increases. This is convergence of a particular type.

DEFINITION 1.5 Let e_i denote the error at step i of an iterative method. If

$$\lim_{i \rightarrow \infty} \frac{e_{i+1}}{e_i} = S < 1,$$

the method is said to obey **linear convergence** with rate S . □

Fixed-Point Iteration for g_2 is linearly convergent to the root $r = 1$ with rate $S = 1/2$. Although the previous discussion was simplified because g_1 and g_2 are linear, the same reasoning applies to a general continuously differentiable function $g(x)$ with fixed point $g(r) = r$, as shown in the next theorem.

THEOREM 1.6 Assume that g is continuously differentiable, that $g(r) = r$, and that $S = |g'(r)| < 1$. Then Fixed-Point Iteration converges linearly with rate S to the fixed point r for initial guesses sufficiently close to r . ■

Proof. Let x_i denote the iterate at step i . According to the Mean Value Theorem, there exists a number c_i between x_i and r such that

$$x_{i+1} - r = g'(c_i)(x_i - r), \quad (1.9)$$

where we have substituted $x_{i+1} = g(x_i)$ and $r = g(r)$. Defining $e_i = |x_i - r|$, (1.9) can be written as

$$e_{i+1} = |g'(c_i)|e_i. \quad (1.10)$$

If $S = |g'(r)|$ is less than one, then by the continuity of g' , there is a small neighborhood around r for which $|g'(x)| < (S + 1)/2$, slightly larger than S , but still less than one. If x_i happens to lie in this neighborhood, then c_i does, too (it is trapped between x_i and r), and so

$$e_{i+1} \leq \frac{S+1}{2} e_i.$$

Thus, the error decreases by a factor of $(S + 1)/2$ or better on this and every future step. That means $\lim_{i \rightarrow \infty} x_i = r$, and taking the limit of (1.10) yields

$$\lim_{i \rightarrow \infty} \frac{e_{i+1}}{e_i} = \lim_{i \rightarrow \infty} |g'(c_i)| = |g'(r)| = S. \quad \square$$

According to Theorem 1.6, the approximate error relationship

$$e_{i+1} \approx S e_i \quad (1.11)$$

holds in the limit as convergence is approached, where $S = |g'(r)|$. See Exercise 25 for a variant of this theorem.

DEFINITION 1.7 An iterative method is called **locally convergent** to r if the method converges to r for initial guesses sufficiently close to r . \square

In other words, the method is locally convergent to the root r if there exists a neighborhood $(r - \epsilon, r + \epsilon)$, where $\epsilon > 0$, such that convergence to r follows from all initial guesses from the neighborhood. The conclusion of Theorem 1.6 is that Fixed-Point Iteration is locally convergent if $|g'(r)| < 1$.

Theorem 1.6 explains what happened in the previous Fixed-Point Iteration runs for $f(x) = x^3 + x - 1 = 0$. We know the root $r \approx 0.6823$. For $g(x) = 1 - x^3$, the derivative is $g'(x) = -3x^2$. Near the root r , FPI behaves as $e_{i+1} \approx S e_i$, where $S = |g'(r)| = |-3(0.6823)^2| \approx 1.3966 > 1$, so errors increase, and there can be no convergence. This error relationship between e_{i+1} and e_i is only guaranteed to hold near r , but it does mean that no convergence to r can occur.

For the second choice, $g(x) = \sqrt[3]{1-x}$, the derivative is $g'(x) = 1/3(1-x)^{-2/3}(-1)$, and $S = |(1 - 0.6823)^{-2/3}/3| \approx 0.716 < 1$. Theorem 1.6 implies convergence, agreeing with our previous calculation.

For the third choice, $g(x) = (1 + 2x^3)/(1 + 3x^2)$,

$$\begin{aligned} g'(x) &= \frac{6x^2(1 + 3x^2) - (1 + 2x^3)6x}{(1 + 3x^2)^2} \\ &= \frac{6x(x^3 + x - 1)}{(1 + 3x^2)^2}, \end{aligned}$$

and $S = |g'(r)| = 0$. This is as small as S can get, leading to the very fast convergence seen in Figure 1.3(c).

► **EXAMPLE 1.3** Explain why the Fixed-Point Iteration $g(x) = \cos x$ converges.

This is the explanation promised early in the chapter. Applying the cosine button repeatedly corresponds to FPI with $g(x) = \cos x$. According to Theorem 1.6, the solution $r \approx 0.74$ attracts nearby guesses because $g'(r) = -\sin r \approx -\sin 0.74 \approx -0.67$ is less than 1 in absolute value. \blacktriangleleft

► **EXAMPLE 1.4** Use Fixed-Point Iteration to find a root of $\cos x = \sin x$.

The simplest way to convert the equation to a fixed-point problem is to add x to each side of the equation. We can rewrite the problem as

$$x + \cos x - \sin x = x$$

and define

$$g(x) = x + \cos x - \sin x. \quad (1.12)$$

The result of applying the Fixed-Point Iteration method to this $g(x)$ is shown in the table.

i	x_i	$g(x_i)$	$e_i = x_i - r $	e_i/e_{i-1}
0	0.0000000	1.0000000	0.7853982	
1	1.0000000	0.6988313	0.2146018	0.273
2	0.6988313	0.8211025	0.0865669	0.403
3	0.8211025	0.7706197	0.0357043	0.412
4	0.7706197	0.7915189	0.0147785	0.414
5	0.7915189	0.7828629	0.0061207	0.414
6	0.7828629	0.7864483	0.0025353	0.414
7	0.7864483	0.7849632	0.0010501	0.414
8	0.7849632	0.7855783	0.0004350	0.414
9	0.7855783	0.7853235	0.0001801	0.414
10	0.7853235	0.7854291	0.0000747	0.415
11	0.7854291	0.7853854	0.0000309	0.414
12	0.7853854	0.7854035	0.0000128	0.414
13	0.7854035	0.7853960	0.0000053	0.414
14	0.7853960	0.7853991	0.0000022	0.415
15	0.7853991	0.7853978	0.0000009	0.409
16	0.7853978	0.7853983	0.0000004	0.444
17	0.7853983	0.7853981	0.0000001	0.250
18	0.7853981	0.7853982	0.0000001	1.000
19	0.7853982	0.7853982	0.0000000	

There are several interesting things to notice in the table. First, the iteration appears to converge to 0.7853982. Since $\cos \pi/4 = \sqrt{2}/2 = \sin \pi/4$, the true solution to the equation $\cos x - \sin x = 0$ is $r = \pi/4 \approx 0.7853982$. The fourth column is the “error column.” It shows the absolute value of the difference between the best guess x_i at step i and the actual fixed point r . This difference becomes small near the bottom of the table, indicating convergence toward a fixed point.

Notice the pattern in the error column. The errors seem to decrease by a constant factor, each error being somewhat less than half the previous error. To be more precise, the ratio between successive errors is shown in the final column. In most of the table, we are seeing the ratio e_{k+1}/e_k of successive errors to approach a constant number, about 0.414. In other words, we are seeing the linear convergence relation

$$e_i \approx 0.414e_{i-1}. \quad (1.13)$$

This is exactly what is expected, since Theorem 1.6 implies that

$$S = |g'(r)| = |1 - \sin r - \cos r| = \left| 1 - \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2} \right| = |1 - \sqrt{2}| \approx 0.414. \quad \blacktriangleleft$$

The careful reader will notice a discrepancy toward the end of the table. We have used only seven correct digits for the correct fixed point r in computing the errors e_i . As a result,

the relative accuracy of the e_i is poor as the e_i near 10^{-8} , and the ratios e_i/e_{i-1} become inaccurate. This problem would disappear if we used a much more accurate value for r .

► **EXAMPLE 1.5** Find the fixed points of $g(x) = 2.8x - x^2$.

The function $g(x) = 2.8x - x^2$ has two fixed points 0 and 1.8, which can be determined by solving $g(x) = x$ by hand, or alternatively, by noting where the graphs of $y = g(x)$ and $y = x$ intersect. Figure 1.5 shows a cobweb diagram for FPI with initial guess $x = 0.1$. For this example, the iterates

$$x_0 = 0.1000$$

$$x_1 = 0.2700$$

$$x_2 = 0.6831$$

$$x_3 = 1.4461$$

$$x_4 = 1.9579,$$

and so on, can be read as the intersections along the diagonal.

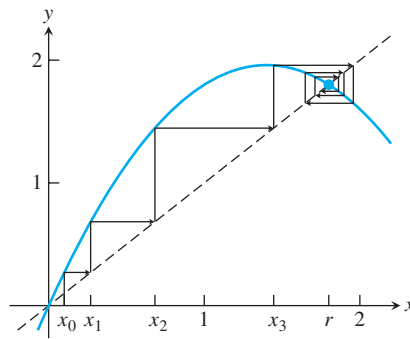


Figure 1.5 Cobweb diagram for Fixed-Point Iteration. Example 1.5 has two fixed points, 0 and 1.8. An iteration with starting guess 0.1 is shown. Only 1.8 will be converged to by FPI.

Even though the initial point $x_0 = 0.1$ is near the fixed point 0, FPI moves toward the other fixed point $x = 1.8$ and converges there. The difference between the two fixed points is that the slope of g at $x = 1.8$, given by $g'(1.8) = -0.8$, is smaller than one in absolute value. On the other hand, the slope of g at the other fixed point $x = 0$, the one that repels points, is $g'(0) = 2.8$, which is larger than one in absolute value. ◀

Theorem 1.6 is useful *a posteriori*—at the end of the FPI calculation, we know the root and can calculate the step-by-step errors. The theorem helps explain why the rate of convergence S turned out as it did. It would be much more useful to have that information before the calculation starts. In some cases, we are able to do this, as the next example shows.

► **EXAMPLE 1.6** Calculate $\sqrt{2}$ by using FPI.

An ancient method for determining square roots can be expressed as an FPI. Suppose we want to find the first 10 digits of $\sqrt{2}$. Start with the initial guess $x_0 = 1$. This guess is obviously too low; therefore, $2/1 = 2$ is too high. In fact, any initial guess $0 < x_0 < 2$, together with $2/x_0$, form a bracket for $\sqrt{2}$. Because of that, it is reasonable to average the two to get a better guess:

$$x_1 = \frac{1 + \frac{2}{1}}{2} = \frac{3}{2}.$$

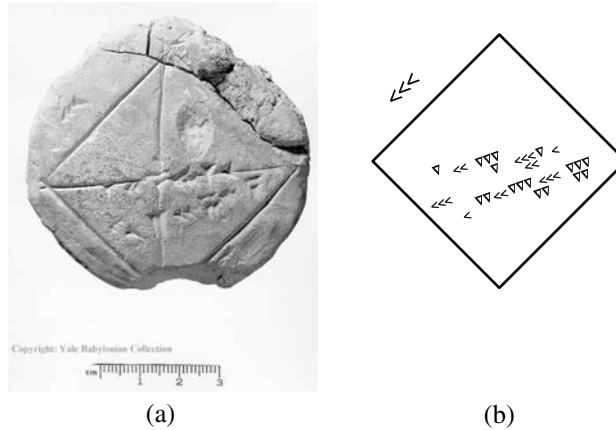


Figure 1.6 Ancient calculation of $\sqrt{2}$. (a) Tablet YBC7289 (b) Schematic of tablet. The Babylonians calculated in base 60, but used some base 10 notation. The < denotes 10, and the ∇ denotes 1. In the upper left is 30, the length of the side. Along the middle are 1, 24, 51, and 10, which represents the square root of 2 to five correct decimal places (see Spotlight on page 39). Below, the numbers 42, 25, and 35 represent $30\sqrt{2}$ in base 60.

Now repeat. Although $3/2$ is closer, it is too large to be $\sqrt{2}$, and $2/(3/2) = 4/3$ is too small. As before, average to get

$$x_2 = \frac{\frac{3}{2} + \frac{4}{3}}{2} = \frac{17}{12} = 1.41\bar{6},$$

which is even closer to $\sqrt{2}$. Once again, x_2 and $2/x_2$ bracket $\sqrt{2}$.

The next step yields

$$x_3 = \frac{\frac{17}{12} + \frac{24}{17}}{2} = \frac{577}{408} \approx 1.414215686.$$

Check with a calculator to see that this guess agrees with $\sqrt{2}$ within 3×10^{-6} . The FPI we are executing is

$$x_{i+1} = \frac{x_i + \frac{2}{x_i}}{2}. \quad (1.14)$$

Note that $\sqrt{2}$ is a fixed point of the iteration.

SPOTLIGHT ON

Convergence

The ingenious method of Example 1.6 converges to $\sqrt{2}$ within five decimal places after only three steps. This simple method is one of the oldest in the history of mathematics. The cuneiform tablet YBC7289 shown in Figure 1.6(a) was discovered near Baghdad in 1962, dating from around 1750 B.C. It contains the base 60 approximation (1)(24)(51)(10) for the side length of a square of area 2. In base 10, this is


$$1 + \frac{24}{60} + \frac{51}{60^2} + \frac{10}{60^3} = 1.41421296.$$

The Babylonians' method of calculation is not known, but some speculate it is the computation of Example 1.6, in their customary base 60. In any case, this method appears in Book 1 of *Metrica*, written by Heron of Alexandria in the first century A.D., to calculate $\sqrt{720}$.

Before finishing the calculation, let's decide whether it will converge. According to Theorem 1.6, we need $S < 1$. For this iteration, $g(x) = 1/2(x + 2/x)$ and $g'(x) = 1/2(1 - 2/x^2)$. Evaluating at the fixed point yields

$$g'(\sqrt{2}) = \frac{1}{2} \left(1 - \frac{2}{(\sqrt{2})^2} \right) = 0, \quad (1.15)$$

so $S = 0$. We conclude that the FPI will converge, and very fast.

Exercise 18 asks whether this method will be successful in finding the square root of an arbitrary positive number. 

1.2.4 Stopping criteria

Unlike the case of bisection, the number of steps required for FPI to converge within a given tolerance is rarely predictable beforehand. In the absence of an error formula like (1.1) for the Bisection Method, a decision must be made about terminating the algorithm, called a **stopping criterion**.

For a set tolerance, TOL, we may ask for an absolute error stopping criterion

$$|x_{i+1} - x_i| < \text{TOL} \quad (1.16)$$

or, in case the solution is not too near zero, the relative error stopping criterion

$$\frac{|x_{i+1} - x_i|}{|x_{i+1}|} < \text{TOL}. \quad (1.17)$$

A hybrid absolute/relative stopping criterion such as

$$\frac{|x_{i+1} - x_i|}{\max(|x_{i+1}|, \theta)} < \text{TOL} \quad (1.18)$$

for some $\theta > 0$ is often useful in cases where the solution is near 0. In addition, good FPI code sets a limit on the maximum number of steps in case convergence fails. The issue of stopping criteria is important, and will be revisited in a more sophisticated way when we study forward and backward error in Section 1.3.

The Bisection Method is guaranteed to converge linearly. Fixed-Point Iteration is only locally convergent, and when it converges it is linearly convergent. Both methods require one function evaluation per step. The bisection cuts uncertainty by $1/2$ for each step, compared with approximately $S = |g'(r)|$ for FPI. Therefore, Fixed-Point Iteration may be faster or slower than bisection, depending on whether S is smaller or larger than $1/2$. In Section 1.4, we study Newton's Method, a particularly refined version of FPI, where S is designed to be zero.

1.2 Exercises

- Find all fixed points of the following $g(x)$.
 (a) $\frac{3}{x}$ (b) $x^2 - 2x + 2$ (c) $x^2 - 4x + 2$
- Find all fixed points of the following $g(x)$.
 (a) $\frac{x+6}{3x-2}$ (b) $\frac{8+2x}{2+x^2}$ (c) x^5
- Show that 1, 2, and 3 are fixed points of the following $g(x)$.
 (a) $\frac{x^3 + x - 6}{6x - 10}$ (b) $\frac{6 + 6x^2 - x^3}{11}$

4. Show that $-1, 0$, and 1 are fixed points of the following $g(x)$.
 (a) $\frac{4x}{x^2 + 3}$ (b) $\frac{x^2 - 5x}{x^2 + x - 6}$
5. For which of the following $g(x)$ is $r = \sqrt{3}$ a fixed point?
 (a) $g(x) = \frac{x}{\sqrt{3}}$ (b) $g(x) = \frac{2x}{3} + \frac{1}{x}$ (c) $g(x) = x^2 - x$ (d) $g(x) = 1 + \frac{2}{x + 1}$
6. For which of the following $g(x)$ is $r = \sqrt{5}$ a fixed point?
 (a) $g(x) = \frac{5 + 7x}{x + 7}$ (b) $g(x) = \frac{10}{3x} + \frac{x}{3}$ (c) $g(x) = x^2 - 5$ (d) $g(x) = 1 + \frac{4}{x + 1}$
7. Use Theorem 1.6 to determine whether Fixed-Point Iteration of $g(x)$ is locally convergent to the given fixed point r . (a) $g(x) = (2x - 1)^{1/3}, r = 1$ (b) $g(x) = (x^3 + 1)/2, r = 1$ (c) $g(x) = \sin x + x, r = 0$
8. Use Theorem 1.6 to determine whether Fixed-Point Iteration of $g(x)$ is locally convergent to the given fixed point r . (a) $g(x) = (2x - 1)/x^2, r = 1$ (b) $g(x) = \cos x + \pi + 1, r = \pi$ (c) $g(x) = e^{2x} - 1, r = 0$
9. Find each fixed point and decide whether Fixed-Point Iteration is locally convergent to it.
 (a) $g(x) = \frac{1}{2}x^2 + \frac{1}{2}x$ (b) $g(x) = x^2 - \frac{1}{4}x + \frac{3}{8}$
10. Find each fixed point and decide whether Fixed-Point Iteration is locally convergent to it.
 (a) $g(x) = x^2 - \frac{3}{2}x + \frac{3}{2}$ (b) $g(x) = x^2 + \frac{1}{2}x - \frac{1}{2}$
11. Express each equation as a fixed-point problem $x = g(x)$ in three different ways.
 (a) $x^3 - x + e^x = 0$ (b) $3x^{-2} + 9x^3 = x^2$
12. Consider the Fixed-Point Iteration $x \rightarrow g(x) = x^2 - 0.24$. (a) Do you expect Fixed-Point Iteration to calculate the root -0.2 , say, to 10 or to correct decimal places, faster or slower than the Bisection Method? (b) Find the other fixed point. Will FPI converge to it?
13. (a) Find all fixed points of $g(x) = 0.39 - x^2$. (b) To which of the fixed-points is Fixed-Point Iteration locally convergent? (c) Does FPI converge to this fixed point faster or slower than the Bisection Method?
14. Which of the following three Fixed-Point Iterations converge to $\sqrt{2}$? Rank the ones that converge from fastest to slowest.
 (A) $x \rightarrow \frac{1}{2}x + \frac{1}{x}$ (B) $x \rightarrow \frac{2}{3}x + \frac{2}{3x}$ (C) $x \rightarrow \frac{3}{4}x + \frac{1}{2x}$
15. Which of the following three Fixed-Point Iterations converge to $\sqrt{5}$? Rank the ones that converge from fastest to slowest.
 (A) $x \rightarrow \frac{4}{5}x + \frac{1}{x}$ (B) $x \rightarrow \frac{x}{2} + \frac{5}{2x}$ (C) $x \rightarrow \frac{x + 5}{x + 1}$
16. Which of the following three Fixed-Point Iterations converge to the cube root of 4? Rank the ones that converge from fastest to slowest.
 (A) $g(x) = \frac{2}{\sqrt{x}}$ (B) $g(x) = \frac{3x}{4} + \frac{1}{x^2}$ (C) $g(x) = \frac{2}{3}x + \frac{4}{3x^2}$
17. Check that $1/2$ and -1 are roots of $f(x) = 2x^2 + x - 1 = 0$. Isolate the x^2 term and solve for x to find two candidates for $g(x)$. Which of the roots will be found by the two Fixed-Point Iterations?
18. Prove that the method of Example 1.6 will calculate the square root of any positive number.

19. Explore the idea of Example 1.6 for cube roots. If x is a guess that is smaller than $A^{1/3}$, then A/x^2 will be larger than $A^{1/3}$, so that the average of the two will be a better approximation than x . Suggest a Fixed-Point Iteration on the basis of this fact, and use Theorem 1.6 to decide whether it will converge to the cube root of A .
20. Improve the cube root algorithm of Exercise 19 by reweighting the average. Setting $g(x) = wx + (1 - w)A/x^2$ for some fixed number $0 < w < 1$, what is the best choice for w ?
21. Consider Fixed-Point Iteration applied to $g(x) = 1 - 5x + \frac{15}{2}x^2 - \frac{5}{2}x^3$. (a) Show that $1 - \sqrt{3/5}$, 1, and $1 + \sqrt{3/5}$ are fixed points. (b) Show that none of the three fixed points is locally convergent. (Computer Problem 7 investigates this example further.)
22. Show that the initial guesses 0, 1, and 2 lead to a fixed point in Exercise 21. What happens to other initial guesses close to those numbers?
23. Assume that $g(x)$ is continuously differentiable and that the Fixed-Point Iteration $g(x)$ has exactly three fixed points, $r_1 < r_2 < r_3$. Assume also that $|g'(r_1)| = 0.5$ and $|g'(r_3)| = 0.5$. List all values of $|g'(r_2)|$ that are possible under these conditions.
24. Assume that g is a continuously differentiable function and that the Fixed-Point Iteration $g(x)$ has exactly three fixed points, -3 , 1, and 2. Assume that $g'(-3) = 2.4$ and that FPI started sufficiently near the fixed point 2 converges to 2. Find $g'(1)$.
25. Prove the variant of Theorem 1.6: If g is continuously differentiable and $|g'(x)| \leq B < 1$ on an interval $[a, b]$ containing the fixed point r , then FPI converges to r from any initial guess in $[a, b]$.
26. Prove that a continuously differentiable function $g(x)$ satisfying $|g'(x)| < 1$ on a closed interval cannot have two fixed points on that interval.
27. Consider Fixed-Point Iteration with $g(x) = x - x^3$. (a) Show that $x = 0$ is the only fixed point. (b) Show that if $0 < x_0 < 1$, then $x_0 > x_1 > x_2 \dots > 0$. (c) Show that FPI converges to $r = 0$, while $g'(0) = 1$. (Hint: Use the fact that every bounded monotonic sequence converges to a limit.)
28. Consider Fixed-Point Iteration with $g(x) = x + x^3$. (a) Show that $x = 0$ is the only fixed point. (b) Show that if $0 < x_0 < 1$, then $x_0 < x_1 < x_2 < \dots$. (c) Show that FPI fails to converge to a fixed point, while $g'(0) = 1$. Together with Exercise 27, this shows that FPI may converge to a fixed point r or diverge from r when $|g'(r)| = 1$.
29. Consider the equation $x^3 + x - 2 = 0$, with root $r = 1$. Add the term cx to both sides and divide by c to obtain $g(x)$. (a) For what c is FPI locally convergent to $r = 1$? (b) For what c will FPI converge fastest?
30. Assume that Fixed-Point Iteration is applied to a twice continuously differentiable function $g(x)$ and that $g'(r) = 0$ for a fixed point r . Show that if FPI converges to r , then the error obeys $\lim_{i \rightarrow \infty} (e_{i+1})/e_i^2 = M$, where $M = |g''(r)|/2$.
31. Define Fixed-Point Iteration on the equation $x^2 + x = 5/16$ by isolating the x term. Find both fixed points, and determine which initial guesses lead to each fixed point under iteration. (Hint: Plot $g(x)$, and draw cobweb diagrams.)
32. Find the set of all initial guesses for which the Fixed-Point Iteration $x \rightarrow 4/9 - x^2$ converges to a fixed point.

33. Let $g(x) = a + bx + cx^2$ for constants a, b , and c . (a) Specify one set of constants a, b , and c for which $x = 0$ is a fixed-point of $x = g(x)$ and Fixed-Point Iteration is locally convergent to 0. (b) Specify one set of constants a, b , and c for which $x = 0$ is a fixed-point of $x = g(x)$ but Fixed-Point Iteration is not locally convergent to 0.

1.2 Computer Problems

1. Apply Fixed-Point Iteration to find the solution of each equation to eight correct decimal places. (a) $x^3 = 2x + 2$ (b) $e^x + x = 7$ (c) $e^x + \sin x = 4$.
2. Apply Fixed-Point Iteration to find the solution of each equation to eight correct decimal places. (a) $x^5 + x = 1$ (b) $\sin x = 6x + 5$ (c) $\ln x + x^2 = 3$
3. Calculate the square roots of the following numbers to eight correct decimal places by using Fixed-Point Iteration as in Example 1.6: (a) 3 (b) 5. State your initial guess and the number of steps needed.
4. Calculate the cube roots of the following numbers to eight correct decimal places, by using Fixed-Point Iteration with $g(x) = (2x + A/x^2)/3$, where A is (a) 2 (b) 3 (c) 5. State your initial guess and the number of steps needed.
5. Example 1.3 showed that $g(x) = \cos x$ is a convergent FPI. Is the same true for $g(x) = \cos^2 x$? Find the fixed point to six correct decimal places, and report the number of FPI steps needed. Discuss local convergence, using Theorem 1.6.
6. Derive three different $g(x)$ for finding roots to six correct decimal places of the following $f(x) = 0$ by Fixed-Point Iteration. Run FPI for each $g(x)$ and report results, convergence or divergence. Each equation $f(x) = 0$ has three roots. Derive more $g(x)$ if necessary until all roots are found by FPI. For each convergent run, determine the value of S from the errors e_{i+1}/e_i , and compare with S determined from calculus as in (1.11). (a) $f(x) = 2x^3 - 6x - 1$ (b) $f(x) = e^{x-2} + x^3 - x$ (c) $f(x) = 1 + 5x - 6x^3 - e^{2x}$
7. Exercise 21 considered Fixed-Point Iteration applied to $g(x) = 1 - 5x + \frac{15}{2}x^2 - \frac{5}{2}x^3 = x$. Find initial guesses for which FPI (a) cycles endlessly through numbers in the interval $(0, 1)$ (b) the same as (a), but the interval is $(1, 2)$ (c) diverges to infinity. Cases (a) and (b) are examples of chaotic dynamics. In all three cases, FPI is unsuccessful.

1.3 LIMITS OF ACCURACY

One of the goals of numerical analysis is to compute answers within a specified level of accuracy. Working in double precision means that we store and operate on numbers that are kept to 52-bit accuracy, about 16 decimal digits.

Can answers always be computed to 16 correct significant digits? In Chapter 0, it was shown that, with a naive algorithm for computing roots of a quadratic equation, it was possible to lose some or all significant digits. An improved algorithm eliminated the problem. In this section, we will see something new—a calculation that a double-precision computer cannot make to anywhere near 16 correct digits, even with the best algorithm.

1.3.1 Forward and backward error

The first example shows that, in some cases, pencil and paper can still outperform a computer.

► **EXAMPLE 1.7** Use the Bisection Method to find the root of $f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27}$ to within six correct significant digits.

Note that $f(0)f(1) = (-8/27)(1/27) < 0$, so the Intermediate Value Theorem guarantees a solution in $[0, 1]$. According to Example 1.2, 20 bisection steps should be sufficient for six correct places.

In fact, it is easy to check without a computer that $r = 2/3 = 0.66666666\dots$ is a root:

$$f(2/3) = \frac{8}{27} - 2\left(\frac{4}{9}\right) + \left(\frac{4}{3}\right)\left(\frac{2}{3}\right) - \frac{8}{27} = 0.$$

How many of these digits can the Bisection Method obtain?

i	a_i	$f(a_i)$	c_i	$f(c_i)$	b_i	$f(b_i)$
0	0.0000000	—	0.5000000	—	1.0000000	+
1	0.5000000	—	0.7500000	+	1.0000000	+
2	0.5000000	—	0.6250000	—	0.7500000	+
3	0.6250000	—	0.6875000	+	0.7500000	+
4	0.6250000	—	0.6562500	—	0.6875000	+
5	0.6562500	—	0.6718750	+	0.6875000	+
6	0.6562500	—	0.6640625	—	0.6718750	+
7	0.6640625	—	0.6679688	+	0.6718750	+
8	0.6640625	—	0.6660156	—	0.6679688	+
9	0.6660156	—	0.6669922	+	0.6679688	+
10	0.6660156	—	0.6665039	—	0.6669922	+
11	0.6665039	—	0.6667480	+	0.6669922	+
12	0.6665039	—	0.6666260	—	0.6667480	+
13	0.6666260	—	0.6666870	+	0.6667480	+
14	0.6666260	—	0.6666565	—	0.6666870	+
15	0.6666565	—	0.6666718	+	0.6666870	+
16	0.6666565	—	0.6666641	0	0.6666718	+

Surprisingly, the Bisection Method stops after 16 steps, when it computes $f(0.6666641) = 0$. This is a serious failure if we care about six or more digits of precision. Figure 1.7 shows the difficulty. As far as IEEE double precision is concerned, there are many floating point numbers within 10^{-5} of the correct root $r = 2/3$ that are evaluated to machine zero, and therefore have an equal right to be called the root! To make matters worse, although the function f is monotonically increasing, part (b) of the figure shows that even the sign of the double precision value of f is often wrong.

Figure 1.7 shows that the problem lies not with the Bisection Method, but with the inability of double precision arithmetic to compute the function f accurately enough near the root. Any other solution method that relies on this computer arithmetic is bound to fail. For this example, 16-digit precision cannot even check whether a candidate solution is correct to six places. ◀

To convince you that it's not the fault of the Bisection Method, we apply MATLAB's most high-powered multipurpose rootfinder, `fzero.m`. We will discuss its details later in

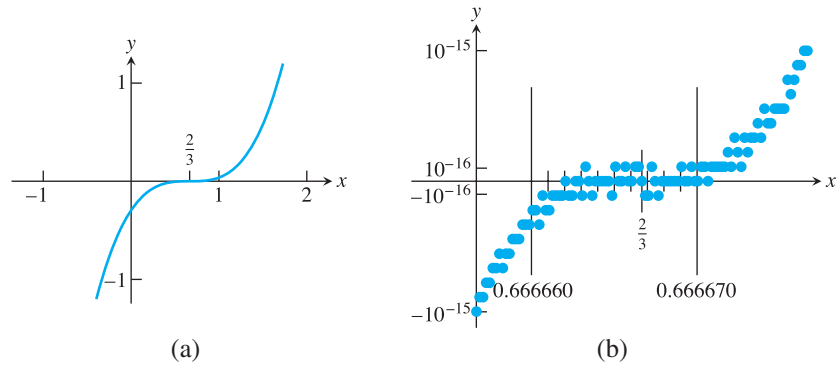


Figure 1.7 The shape of a function near a multiple root. (a) Plot of $f(x) = x^3 - 2x^2 + 4/3x - 8/27$. (b) Magnification of (a), near the root $r = 2/3$. There are many floating point numbers within 10^{-5} of $2/3$ that are roots as far as the computer is concerned. We know from calculus that $2/3$ is the only root.

this chapter; for now, we just need to feed it the function and a starting guess. It has no better luck:

```
>> fzero('x.^3-2*x.^2+4*x/3-8/27', 1)

ans =

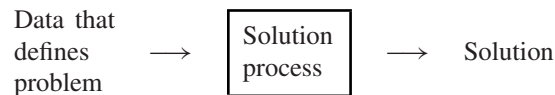
    0.66666250845989
```

The reason that all methods fail to get more than five correct digits for this example is clear from Figure 1.7. The only information any method has is the function, computed in double precision. If the computer arithmetic is showing the function to be zero at a nonroot, there is no way the method can recover. Another way to state the difficulty is to say that an approximate solution can be as close as possible to a solution as far as the y -axis is concerned, but not so close on the x -axis.

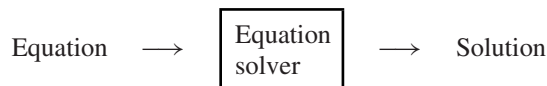
These observations motivate some key definitions.

DEFINITION 1.8 Assume that f is a function and that r is a root, meaning that it satisfies $f(r) = 0$. Assume that x_a is an approximation to r . For the root-finding problem, the **backward error** of the approximation x_a is $|f(x_a)|$ and the **forward error** is $|r - x_a|$. \square

The usage of “backward” and “forward” may need some explanation. Our viewpoint considers the process of finding a solution as central. The problem is the input, and the solution is the output:



In this chapter, the “problem” is an equation in one variable, and the “solution process” is an algorithm that solves equations:



Backward error is on the left or input (problem data) side. It is the amount we would need to change the problem (the function f) to make the equation balance with the output

approximation x_a . This amount is $|f(x_a)|$. Forward error is the error on the right or output (problem solution) side. It is the amount we would need to change the approximate solution to make it correct, which is $|r - x_a|$.

The difficulty with Example 1.7 is that, according to Figure 1.7, the backward error is near $\epsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$, while forward error is approximately 10^{-5} . Double precision numbers cannot be computed reliably below a relative error of machine epsilon. Since the backward error cannot be decreased further with reliability, neither can the forward error.

Example 1.7 is rather special because the function has a triple root at $r = 2/3$. Note that

$$f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27} = \left(x - \frac{2}{3}\right)^3.$$

This is an example of a multiple root.

DEFINITION 1.9 Assume that r is a root of the differentiable function f ; that is, assume that $f(r) = 0$. Then if $0 = f(r) = f'(r) = f''(r) = \dots = f^{(m-1)}(r)$, but $f^{(m)}(r) \neq 0$, we say that f has a **root** of **multiplicity** m at r . We say that f has a **multiple root** at r if the multiplicity is greater than one. The root is called **simple** if the multiplicity is one. \square


For example, $f(x) = x^2$ has a multiplicity two, or double, root at $r = 0$, because $f(0) = 0$, $f'(0) = 2(0) = 0$, but $f''(0) = 2 \neq 0$. Likewise, $f(x) = x^3$ has a multiplicity three, or triple, root at $r = 0$, and $f(x) = x^m$ has a multiplicity m root there. Example 1.7 has a multiplicity three, or triple, root at $r = 2/3$.

Because the graph of the function is relatively flat near a multiple root, a great disparity exists between backward and forward errors for nearby approximate solutions. The backward error, measured in the vertical direction, is often much smaller than the forward error, measured in the horizontal direction.

► **EXAMPLE 1.8** The function $f(x) = \sin x - x$ has a triple root at $r = 0$. Find the forward and backward error of the approximate root $x_c = 0.001$.

The root at 0 has multiplicity three because

$$\begin{aligned} f(0) &= \sin 0 - 0 = 0 \\ f'(0) &= \cos 0 - 1 = 0 \\ f''(0) &= -\sin 0 - 0 = 0 \\ f'''(0) &= -\cos 0 = -1. \end{aligned}$$

The forward error is $\text{FE} = |r - x_a| = 10^{-3}$. The backward error is the constant that would need to be added to $f(x)$ to make x_a a root, namely $\text{BE} = |f(x_a)| = |\sin(0.001) - 0.001| \approx 1.6667 \times 10^{-10}$. 

The subject of backward and forward error is relevant to stopping criteria for equation solvers. The goal is to find the root r satisfying $f(r) = 0$. Suppose our algorithm produces an approximate solution x_a . How do we decide whether it is good enough?

Two possibilities come to mind: (1) to make $|x_a - r|$ small and (2) to make $|f(x_a)|$ small. In case $x_a = r$, there is no decision to be made—both ways of looking at it are the same. However, we are rarely lucky enough to be in this situation. In the more typical case, approaches (1) and (2) are different and correspond to forward and backward error.

Whether forward or backward error is more appropriate depends on the circumstances surrounding the problem. If we are using the Bisection Method, both errors are easily observable. For an approximate root x_a , we can find the backward error by evaluating

$f(x_a)$, and the forward error can be no more than half the length of the current interval. For FPI, our choices are more limited, since we have no bracketing interval. As before, the backward error is known as $f(x_a)$, but to know the forward error would require knowing the true root, which we are trying to find.

Stopping criteria for equation-solving methods can be based on either forward or backward error. There are other stopping criteria that may be relevant, such as a limit on computation time. The context of the problem must guide our choice.

Functions are flat in the vicinity of a multiple root, since the derivative f' is zero there. Because of this, we can expect some trouble in isolating a multiple root, as we have demonstrated. But multiplicity is only the tip of the iceberg. Similar difficulties can arise where no multiple roots are in sight, as shown in the next section.

1.3.2 The Wilkinson polynomial

A famous example with simple roots that are hard to determine numerically is discussed in Wilkinson [1994]. The **Wilkinson polynomial** is

$$W(x) = (x - 1)(x - 2) \cdots (x - 20), \quad (1.19)$$

which, when multiplied out, is

$$\begin{aligned} W(x) = & x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + 53327946x^{16} - 1672280820x^{15} \\ & + 40171771630x^{14} - 756111184500x^{13} + 11310276995381x^{12} \\ & - 135585182899530x^{11} + 1307535010540395x^{10} - 10142299865511450x^9 \\ & + 63030812099294896x^8 - 311333643161390640x^7 \\ & + 1206647803780373360x^6 - 3599979517947607200x^5 \\ & + 8037811822645051776x^4 - 12870931245150988800x^3 \\ & + 13803759753640704000x^2 - 8752948036761600000x \\ & + 2432902008176640000. \end{aligned} \quad (1.20)$$

The roots are the integers from 1 to 20. However, when $W(x)$ is defined according to its unfactored form (1.20), its evaluation suffers from cancellation of nearly equal, large numbers. To see the effect on root-finding, define the MATLAB m-file `wilkpoly.m` by typing in the nonfactored form (1.20), or obtaining it from the textbook website.

Again we will try MATLAB's `fzero`. To make it as easy as possible, we feed it an actual root $x = 16$ as a starting guess:

```
>> fzero(@wilkpoly,16)
```

```
ans =
```

```
16.01468030580458
```

The surprising result is that MATLAB's double precision arithmetic could not get the second decimal place correct, even for the simple root $r = 16$. It is not due to a deficiency of the algorithm—both `fzero` and Bisection Method have the same problem, as do Fixed-Point Iteration and any other floating point method. Referring to his work with this polynomial, Wilkinson wrote in 1984: “Speaking for myself I regard it as the most traumatic experience in my career as a numerical analyst.” The roots of $W(x)$ are clear: the integers $x = 1, \dots, 20$. To Wilkinson, the surprise had to do with the huge error magnification in the roots caused by small relative errors in storing the coefficients, which we have just seen in action.

The difficulty of getting accurate roots of the Wilkinson polynomial disappears if factored form (1.19) is used instead of (1.20). Of course, if the polynomial is factored before we start, there is no need to compute roots.

1.3.3 Sensitivity of root-finding

The Wilkinson polynomial and Example 1.7 with the triple root cause difficulties for similar reasons—small floating point errors in the equation translate into large errors in the root. A problem is called **sensitive** if small errors in the input, in this case the equation to be solved, lead to large errors in the output, or solution. In this section, we will quantify sensitivity and introduce the concepts of error magnification factor and condition number.

To understand what causes this magnification of error, we will establish a formula predicting how far a root moves when the equation is changed. Assume that the problem is to find a root r of $f(x) = 0$, but that a small change $\epsilon g(x)$ is made to the input, where ϵ is small. Let Δr be the corresponding change in the root, so that

$$f(r + \Delta r) + \epsilon g(r + \Delta r) = 0.$$

Expanding f and g in degree-one Taylor polynomials implies that

$$f(r) + (\Delta r)f'(r) + \epsilon g(r) + \epsilon(\Delta r)g'(r) + O((\Delta r)^2) = 0,$$

where we use the “big O” notation $O((\Delta r)^2)$ to stand for terms involving $(\Delta r)^2$ and higher powers of Δr . For small Δr , the $O((\Delta r)^2)$ terms can be neglected to get

$$(\Delta r)(f'(r) + \epsilon g'(r)) \approx -f(r) - \epsilon g(r) = -\epsilon g(r)$$

or

$$\Delta r \approx \frac{-\epsilon g(r)}{f'(r) + \epsilon g'(r)} \approx -\epsilon \frac{g(r)}{f'(r)},$$

assuming that ϵ is small compared with $f'(r)$, and in particular, that $f'(r) \neq 0$.

Sensitivity Formula for Roots

Assume that r is a root of $f(x)$ and $r + \Delta r$ is a root of $f(x) + \epsilon g(x)$. Then

$$\Delta r \approx -\frac{\epsilon g(r)}{f'(r)} \tag{1.21}$$

if $\epsilon \ll f'(r)$.

► **EXAMPLE 1.9** Estimate the largest root of $P(x) = (x - 1)(x - 2)(x - 3)(x - 4)(x - 5)(x - 6) - 10^{-6}x^7$.

Set $f(x) = (x - 1)(x - 2)(x - 3)(x - 4)(x - 5)(x - 6)$, $\epsilon = -10^{-6}$ and $g(x) = x^7$. Without the $\epsilon g(x)$ term, the largest root is $r = 6$. The question is, how far does the root move when we add the extra term?

The Sensitivity Formula yields

$$\Delta r \approx -\frac{\epsilon 6^7}{5!} = -2332.8\epsilon,$$

meaning that input errors of relative size ϵ in $f(x)$ are magnified by a factor of over 2000 into the output root. We estimate the largest root of $P(x)$ to be $r + \Delta r = 6 - 2332.8\epsilon = 6.0023328$. Using `fzero` on $P(x)$, we get the correct value 6.0023268. ◀

The estimate in Example 1.9 is good enough to tell us how errors propagate in the root-finding problem. An error in the sixth digit of the problem data caused an error in the third digit of the answer, meaning that three decimal digits were lost due to the factor of 2332.8. It is useful to have a name for this factor. For a general algorithm that produces an approximation x_c , we define its

$$\text{error magnification factor} = \frac{\text{relative forward error}}{\text{relative backward error}}.$$

The forward error is the change in the solution that would make x_a correct, which for root-finding problems is $|x_a - r|$. The backward error is a change in input that makes x_c the correct solution. There is a wider variety of choices, depending on what sensitivity we want to investigate. Changing the constant term by $|f(x_a)|$ is the choice that was used earlier in this section, corresponding to $g(x) = 1$ in the Sensitivity Formula (1.21). More generally, any change in the input data can be used as the backward error, such as the choice $g(x) = x^7$ in Example 1.9. The error magnification factor for root-finding is

$$\text{error magnification factor} = \left| \frac{\Delta r / r}{\epsilon g(r) / g(r)} \right| = \left| \frac{-\epsilon g(r) / (r f'(r))}{\epsilon} \right| = \frac{|g(r)|}{|r f'(r)|}, \quad (1.22)$$

which in Example 1.9 is $6^7 / (5!6) = 388.8$.

► **EXAMPLE 1.10** Use the Sensitivity Formula for Roots to investigate the effect of changes in the x^{15} term of the Wilkinson polynomial on the root $r = 16$. Find the error magnification factor for this problem.

Define the perturbed function $W_\epsilon(x) = W(x) + \epsilon g(x)$, where $g(x) = -1,672,280,820x^{15}$. Note that $W'(16) = 15!4!$ (see Exercise 7). Using (1.21), the change in the root can be approximated by

$$\Delta r \approx \frac{16^{15} 1,672,280,820 \epsilon}{15!4!} \approx 6.1432 \times 10^{13} \epsilon. \quad (1.23)$$

Practically speaking, we know from Chapter 0 that a relative error on the order of machine epsilon must be assumed for every stored number. A relative change in the x^{15} term of machine epsilon ϵ_{mach} will cause the root $r = 16$ to move by

$$\Delta r \approx (6.1432 \times 10^{13})(\pm 2.22 \times 10^{-16}) \approx \pm 0.0136$$

to $r + \Delta r \approx 16.0136$, not far from what was observed on page 47. Of course, many other powers of x in the Wilkinson polynomial are making their own contributions, so the complete picture is complicated. However, the Sensitivity Formula allows us to see the mechanism for the huge magnification of error.

Finally, the error magnification factor is computed from (1.22) as

$$\frac{|g(r)|}{|r f'(r)|} = \frac{16^{15} 1,672,280,820}{15!4!16} \approx 3.8 \times 10^{12}.$$

The significance of the error magnification factor is that it tells us how many of the 16 digits of operating precision are lost from input to output. For a problem with error magnification factor of 10^{12} , we expect to lose 12 of the 16 and have about four correct significant digits left in the root, which is the case for the Wilkinson approximation $x_c = 16.014\dots$

SPOTLIGHT ON

Conditioning

This is the first appearance of the concept of condition number, a measure of error magnification. Numerical analysis is the study of algorithms, which take data defining the problem as input and deliver an answer as output. Condition number refers to the part of this magnification that is inherent in the theoretical problem itself, irrespective of the particular algorithm used to solve it.

It is important to note that the error magnification factor measures only magnification due to the problem. Along with conditioning, there is a parallel concept, stability, that refers to the magnification of small input errors due to the algorithm, not the problem itself. An algorithm is called stable if it always provides an approximate solution with small backward error. If the problem is well-conditioned and the algorithm is stable, we can expect both small backward and forward error.

The preceding error magnification examples show the sensitivity of root-finding to a particular input change. The problem may be more or less sensitive, depending on how the input change is designed. The **condition number** of a problem is defined to be the maximum error magnification over all input changes, or at least all changes of a prescribed type. A problem with high condition number is called **ill-conditioned**, and a problem with a condition number near 1 is called **well-conditioned**. We will return to this concept when we discuss matrix problems in Chapter 2.

1.3 Exercises

- Find the forward and backward error for the following functions, where the root is $3/4$ and the approximate root is $x_a = 0.74$: (a) $f(x) = 4x - 3$ (b) $f(x) = (4x - 3)^2$ (c) $f(x) = (4x - 3)^3$ (d) $f(x) = (4x - 3)^{1/3}$
- Find the forward and backward error for the following functions, where the root is $1/3$ and the approximate root is $x_a = 0.3333$: (a) $f(x) = 3x - 1$ (b) $f(x) = (3x - 1)^2$ (c) $f(x) = (3x - 1)^3$ (d) $f(x) = (3x - 1)^{1/3}$
- (a) Find the multiplicity of the root $r = 0$ of $f(x) = 1 - \cos x$. (b) Find the forward and backward errors of the approximate root $x_a = 0.0001$.
- (a) Find the multiplicity of the root $r = 0$ of $f(x) = x^2 \sin x^2$. (b) Find the forward and backward errors of the approximate root $x_a = 0.01$.
- Find the relation between forward and backward error for finding the root of the linear function $f(x) = ax - b$.
- Let n be a positive integer. The equation defining the n th root of a positive number A is $x^n - A = 0$. (a) Find the multiplicity of the root. (b) Show that, for an approximate n th root with small forward error, the backward error is approximately $nA^{(n-1)/n}$ times the forward error.
- Let $W(x)$ be the Wilkinson polynomial. (a) Prove that $W'(16) = 15!4!$ (b) Find an analogous formula for $W'(j)$, where j is an integer between 1 and 20.
- Let $f(x) = x^n - ax^{n-1}$, and set $g(x) = x^n$. (a) Use the Sensitivity Formula to give a prediction for the nonzero root of $f_\epsilon(x) = x^n - ax^{n-1} + \epsilon x^n$ for small ϵ . (b) Find the nonzero root and compare with the prediction.

1.3 Computer Problems

1. Let $f(x) = \sin x - x$. (a) Find the multiplicity of the root $r = 0$. (b) Use MATLAB's `fzero` command with initial guess $x = 0.1$ to locate a root. What are the forward and backward errors of `fzero`'s response?
2. Carry out Computer Problem 1 for $f(x) = \sin x^3 - x^3$.
3. (a) Use `fzero` to find the root of $f(x) = 2x \cos x - 2x + \sin x^3$ on $[-0.1, 0.2]$. Report the forward and backward errors. (b) Run the Bisection Method with initial interval $[-0.1, 0.2]$ to find as many correct digits as possible, and report your conclusion.
4. (a) Use (1.21) to approximate the root near 3 of $f_\epsilon(x) = (1 + \epsilon)x^3 - 3x^2 + x - 3$ for a constant ϵ . (b) Setting $\epsilon = 10^{-3}$, find the actual root and compare with part (a).
5. Use (1.21) to approximate the root of $f(x) = (x - 1)(x - 2)(x - 3)(x - 4) - 10^{-6}x^6$ near $r = 4$. Find the error magnification factor. Use `fzero` to check your approximation.
6. Use the MATLAB command `fzero` to find the root of the Wilkinson polynomial near $x = 15$ with a relative change of $\epsilon = 2 \times 10^{-15}$ in the x^{15} coefficient, making the coefficient slightly more negative. Compare with the prediction made by (1.21).

1.4 NEWTON'S METHOD

Newton's Method, also called the Newton–Raphson Method, usually converges much faster than the linearly convergent methods we have seen previously. The geometric picture of Newton's Method is shown in Figure 1.8. To find a root of $f(x) = 0$, a starting guess x_0 is given, and the tangent line to the function f at x_0 is drawn. The tangent line will approximately follow the function down to the x -axis toward the root. The intersection point of the line with the x -axis is an approximate root, but probably not exact if f curves. Therefore, this step is iterated.

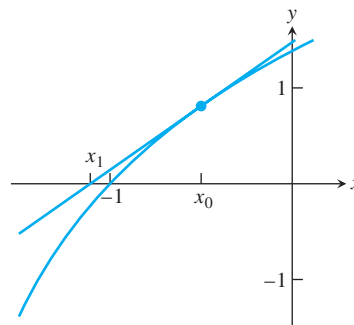


Figure 1.8 One step of Newton's Method. Starting with x_0 , the tangent line to the curve $y = f(x)$ is drawn. The intersection point with the x -axis is x_1 , the next approximation to the root.

From the geometric picture, we can develop an algebraic formula for Newton's Method. The tangent line at x_0 has slope given by the derivative $f'(x_0)$. One point on the tangent line is $(x_0, f(x_0))$. The point-slope formula for the equation of a line is

$y - f(x_0) = f'(x_0)(x - x_0)$, so that looking for the intersection point of the tangent line with the x -axis is the same as substituting $y = 0$ in the line:

$$\begin{aligned} f'(x_0)(x - x_0) &= 0 - f(x_0) \\ x - x_0 &= -\frac{f(x_0)}{f'(x_0)} \\ x &= x_0 - \frac{f(x_0)}{f'(x_0)}. \end{aligned}$$

Solving for x gives an approximation for the root, which we call x_1 . Next, the entire process is repeated, beginning with x_1 , to produce x_2 , and so on, yielding the following iterative formula:

Newton's Method

$$\begin{aligned} x_0 &= \text{initial guess} \\ x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \quad \text{for } i = 0, 1, 2, \dots \end{aligned}$$

► **EXAMPLE 1.11** Find the Newton's Method formula for the equation $x^3 + x - 1 = 0$.

Since $f'(x) = 3x^2 + 1$, the formula is given by

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^3 + x_i - 1}{3x_i^2 + 1} \\ &= \frac{2x_i^3 + 1}{3x_i^2 + 1}. \end{aligned}$$

Iterating this formula from initial guess $x_0 = -0.7$ yields

$$\begin{aligned} x_1 &= \frac{2x_0^3 + 1}{3x_0^2 + 1} = \frac{2(-0.7)^3 + 1}{3(-0.7)^2 + 1} \approx 0.1271 \\ x_2 &= \frac{2x_1^3 + 1}{3x_1^2 + 1} \approx 0.9577. \end{aligned}$$

These steps are shown geometrically in Figure 1.9. Further steps are given in the following table:

i	x_i	$e_i = x_i - r $	e_i/e_{i-1}^2
0	-0.70000000	1.38232780	
1	0.12712551	0.55520230	0.2906
2	0.95767812	0.27535032	0.8933
3	0.73482779	0.05249999	0.6924
4	0.68459177	0.00226397	0.8214
5	0.68233217	0.00000437	0.8527
6	0.68232780	0.00000000	0.8541
7	0.68232780	0.00000000	

After only six steps, the root is known to eight correct digits. There is a bit more we can say about the error and how fast it becomes small. Note in the table that once convergence starts to take hold, the number of correct places in x_i approximately doubles on each iteration. This is characteristic of “quadratically convergent” methods, as we shall see next.

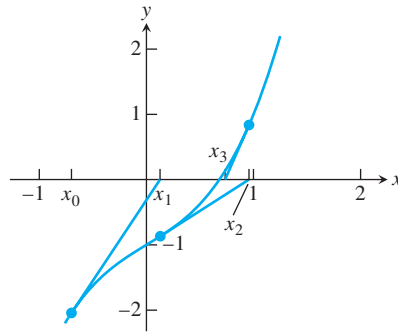


Figure 1.9 Three steps of Newton's method. Illustration of Example 1.11. Starting with $x_0 = -0.7$, the Newton's Method iterates are plotted along with the tangent lines. The method appears to be converging to the root.

1.4.1 Quadratic convergence of Newton's Method

The convergence in Example 1.11 is qualitatively faster than the linear convergence we have seen for the Bisection Method and Fixed-Point Iteration. A new definition is needed.

DEFINITION 1.10 Let e_i denote the error after step i of an iterative method. The iteration is **quadratically convergent** if

$$M = \lim_{i \rightarrow \infty} \frac{e_{i+1}}{e_i^2} < \infty.$$

□

THEOREM 1.11 Let f be twice continuously differentiable and $f(r) = 0$. If $f'(r) \neq 0$, then Newton's Method is locally and quadratically convergent to r . The error e_i at step i satisfies

$$\lim_{i \rightarrow \infty} \frac{e_{i+1}}{e_i^2} = M,$$

where

$$M = \frac{f''(r)}{2f'(r)}.$$

■

Proof. To prove local convergence, note that Newton's Method is a particular form of Fixed-Point Iteration, where

$$g(x) = x - \frac{f(x)}{f'(x)},$$

with derivative

$$g'(x) = 1 - \frac{f'(x)^2 - f(x)f''(x)}{f'(x)^2} = \frac{f(x)f''(x)}{f'(x)^2}.$$

Since $g'(r) = 0$, Newton's Method is locally convergent according to Theorem 1.6.

To prove quadratic convergence, we derive Newton's Method a second way, this time keeping a close eye on the error at each step. By error, we mean the difference between the correct root and the current best guess.

Taylor's formula in Theorem 0.8 tells us the difference between the values of a function at a given point and another nearby point. For the two points, we will use the root r and the current guess x_i after i steps, and we will stop and take a remainder after two terms:

$$f(r) = f(x_i) + (r - x_i)f'(x_i) + \frac{(r - x_i)^2}{2}f''(c_i).$$

Here, c_i is between x_i and r . Because r is the root, we have

$$\begin{aligned} 0 &= f(x_i) + (r - x_i)f'(x_i) + \frac{(r - x_i)^2}{2}f''(c_i) \\ -\frac{f(x_i)}{f'(x_i)} &= r - x_i + \frac{(r - x_i)^2}{2} \frac{f''(c_i)}{f'(x_i)}, \end{aligned}$$

assuming that $f'(x_i) \neq 0$. With some rearranging, we can compare the next Newton iterate with the root:

$$\begin{aligned} x_i - \frac{f(x_i)}{f'(x_i)} - r &= \frac{(r - x_i)^2}{2} \frac{f''(c_i)}{f'(x_i)} \\ x_{i+1} - r &= e_i^2 \frac{f''(c_i)}{2f'(x_i)} \\ e_{i+1} &= e_i^2 \left| \frac{f''(c_i)}{2f'(x_i)} \right|. \end{aligned} \quad (1.24)$$

In this equation, we have defined the error at step i to be $e_i = |x_i - r|$. Since c_i lies between r and x_i , it converges to r just as x_i does, and

$$\lim_{i \rightarrow \infty} \frac{e_{i+1}}{e_i^2} = \left| \frac{f''(r)}{2f'(r)} \right|,$$

the definition of quadratic convergence. □

The error formula (1.24) we have developed can be viewed as

$$e_{i+1} \approx M e_i^2, \quad (1.25)$$

where $M = |f''(r)/2f'(r)|$, under the assumption that $f'(r) \neq 0$. The approximation gets better as Newton's Method converges, since the guesses x_i move toward r , and because c_i is caught between x_i and r . This error formula should be compared with $e_{i+1} \approx S e_i$ for the linearly convergent methods, where $S = |g'(r)|$ for FPI and $S = 1/2$ for bisection.

Although the value of S is critical for linearly convergent methods, the value of M is less critical, because the formula involves the square of the previous error. Once the error gets significantly below 1, squaring will cause a further decrease; and as long as M is not too large, the error according to (1.25) will decrease as well.

Returning to Example 1.11, we can analyze the output table to demonstrate this error rate. The right column shows the ratio e_i/e_{i-1}^2 , which, according to the Newton's Method error formula (1.25), should tend toward M as convergence to the root takes place. For $f(x) = x^3 + x - 1$, the derivatives are $f'(x) = 3x^2 + 1$ and $f''(x) = 6x$; evaluating at $x_c \approx 0.6823$ yields $M \approx 0.85$, which agrees with the error ratio in the right column of the table.

With our new understanding of Newton's Method, we can more fully explain the square root calculator of Example 1.6. Let a be a positive number, and consider finding roots of $f(x) = x^2 - a$ by Newton's Method. The iteration is

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - a}{2x_i} \\ &= \frac{x_i^2 + a}{2x_i} = \frac{x_i + \frac{a}{x_i}}{2}, \end{aligned} \quad (1.26)$$

which is the method from Example 1.6, for arbitrary a .

To study its convergence, evaluate the derivatives at the root \sqrt{a} :

$$\begin{aligned} f'(\sqrt{a}) &= 2\sqrt{a} \\ f''(\sqrt{a}) &= 2. \end{aligned} \quad (1.27)$$

Newton is quadratically convergent, since $f'(\sqrt{a}) = 2\sqrt{a} \neq 0$, and the convergence rate is

$$e_{i+1} \approx M e_i^2, \quad (1.28)$$

where $M = 2/(2 \cdot 2\sqrt{a}) = 1/(2\sqrt{a})$.

1.4.2 Linear convergence of Newton's Method

Theorem 1.11 does not say that Newton's Method always converges quadratically. Recall that we needed to divide by $f'(r)$ for the quadratic convergence argument to make sense. This assumption turns out to be crucial. The following example shows an instance where Newton's Method does not converge quadratically:

► **EXAMPLE 1.12** Use Newton's Method to find a root of $f(x) = x^2$.

This may seem like a trivial problem, since we know there is one root: $r = 0$. But often it is instructive to apply a new method to an example we understand thoroughly. The Newton's Method formula is

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \\ &= x_i - \frac{x_i^2}{2x_i} \\ &= \frac{x_i}{2}. \end{aligned}$$

The surprising result is that Newton's Method simplifies to dividing by two. Since the root is $r = 0$, we have the following table of Newton iterates for initial guess $x_0 = 1$:

i	x_i	$e_i = x_i - r $	e_i/e_{i-1}
0	1.000	1.000	
1	0.500	0.500	0.500
2	0.250	0.250	0.500
3	0.125	0.125	0.500
\vdots	\vdots	\vdots	\vdots

Newton's Method does converge to the root $r = 0$. The error formula is $e_{i+1} = e_i/2$, so the convergence is linear with convergence proportionality constant $S = 1/2$. ◀

A similar result exists for x^m for any positive integer m , as the next example shows.

► **EXAMPLE 1.13** Use Newton's Method to find a root of $f(x) = x^m$.

The Newton formula is

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^m}{m x_i^{m-1}} \\ &= \frac{m-1}{m} x_i. \end{aligned}$$


SPOTLIGHT ON

Convergence

Equations (1.28) and (1.29) express the two different rates of convergence to the root r possible in Newton's Method. At a simple root, $f'(r) \neq 0$, and the convergence is quadratic, or fast convergence, which obeys (1.28). At a multiple root, $f'(r) = 0$, and the convergence is linear and obeys (1.29). In the latter case of linear convergence, the slower rate puts Newton's Method in the same category as bisection and FPI.

Again, the only root is $r = 0$, so defining $e_i = |x_i - r| = x_i$ yields


$$e_{i+1} = S e_i,$$

where $S = (m - 1)/m$. 

This is an example of the general behavior of Newton's Method at multiple roots. Note that Definition 1.9 of multiple root is equivalent to $f(r) = f'(r) = 0$, exactly the case where we could not make our derivation of the Newton's Method error formula work. There is a separate error formula for multiple roots. The pattern that we saw for multiple roots of monomials is representative of the general case, as summarized in Theorem 1.12.

THEOREM 1.12 Assume that the $(m + 1)$ -times continuously differentiable function f on $[a, b]$ has a multiplicity m root at r . Then Newton's Method is locally convergent to r , and the error e_i at step i satisfies

$$\lim_{i \rightarrow \infty} \frac{e_{i+1}}{e_i} = S, \quad (1.29)$$

where $S = (m - 1)/m$. 

► EXAMPLE 1.14 Find the multiplicity of the root $r = 0$ of $f(x) = \sin x + x^2 \cos x - x^2 - x$, and estimate the number of steps of Newton's Method required to converge within six correct places (use $x_0 = 1$).

It is easy to check that

$$\begin{aligned} f(x) &= \sin x + x^2 \cos x - x^2 - x \\ f'(x) &= \cos x + 2x \cos x - x^2 \sin x - 2x - 1 \\ f''(x) &= -\sin x + 2 \cos x - 4x \sin x - x^2 \cos x - 2 \end{aligned}$$

and that each evaluates to 0 at $r = 0$. The third derivative,

$$f'''(x) = -\cos x - 6 \sin x - 6x \cos x + x^2 \sin x, \quad (1.30)$$

satisfies $f'''(0) = -1$, so the root $r = 0$ is a triple root, meaning that the multiplicity is $m = 3$. By Theorem 1.12, Newton should converge linearly with $e_{i+1} \approx 2e_i/3$.

Using starting guess $x_0 = 1$, we have $e_0 = 1$. Near convergence, the error will decrease by $2/3$ on each step. Therefore, a rough approximation to the number of steps needed to get the error within six decimal places, or smaller than 0.5×10^{-6} , can be found by solving

$$\begin{aligned} \left(\frac{2}{3}\right)^n &< 0.5 \times 10^{-6} \\ n &> \frac{\log_{10}(0.5) - 6}{\log_{10}(2/3)} \approx 35.78. \end{aligned} \quad (1.31)$$

Approximately 36 steps will be needed. The first 20 steps are shown in the table.

i	x_i	$e_i = x_i - r $	e_i/e_{i-1}
1	1.000000000000000	1.000000000000000	
2	0.72159023986075	0.72159023986075	0.72159023986075
3	0.52137095182040	0.52137095182040	0.72253049309677
4	0.37530830859076	0.37530830859076	0.71984890466250
5	0.26836349052713	0.26836349052713	0.71504809348561
6	0.19026161369924	0.19026161369924	0.70896981301561
7	0.13361250532619	0.13361250532619	0.70225676492686
8	0.09292528672517	0.09292528672517	0.69548345417455
9	0.06403926677734	0.06403926677734	0.68914790617474
10	0.04377806216009	0.04377806216009	0.68361279513559
11	0.02972805552423	0.02972805552423	0.67906284694649
12	0.02008168373777	0.02008168373777	0.67551285759009
13	0.01351212730417	0.01351212730417	0.67285828621786
14	0.00906579564330	0.00906579564330	0.67093770205249
15	0.00607029292263	0.00607029292263	0.66958192766231
16	0.00405885109627	0.00405885109627	0.66864171927113
17	0.00271130367793	0.00271130367793	0.66799781850081
18	0.00180995966250	0.00180995966250	0.66756065624029
19	0.00120772384467	0.00120772384467	0.66726561353325
20	0.00080563307149	0.00080563307149	0.66706728946460

Note the convergence of the error ratio in the right column to the predicted $2/3$. ◀

If the multiplicity of a root is known in advance, convergence of Newton's Method can be improved with a small modification.

THEOREM 1.13 If f is $(m + 1)$ -times continuously differentiable on $[a, b]$, which contains a root r of multiplicity $m > 1$, then **Modified Newton's Method**

$$x_{i+1} = x_i - \frac{mf(x_i)}{f'(x_i)} \quad (1.32)$$

converges locally and quadratically to r . ■

Returning to Example 1.14, we can apply Modified Newton's Method to achieve quadratic convergence. After five steps, convergence to the root $r = 0$ has taken place to about eight digits of accuracy:

i	x_i
0	1.000000000000000
1	0.16477071958224
2	0.01620733771144
3	0.00024654143774
4	0.00000006072272
5	-0.00000000633250

There are several points to note in the table. First, the quadratic convergence to the approximate root is observable, as the number of correct places in the approximation more or less doubles at each step, up to Step 4. Steps 6, 7, ... are identical to Step 5. The reason Newton's Method lacks convergence to machine precision is familiar to us from Section 1.3.

We know that 0 is a multiple root. While the backward error is driven near ϵ_{mach} by Newton's Method, the forward error, equal to x_i , is several orders of magnitude larger.

Newton's Method, like FPI, may not converge to a root. The next example shows just one of its possible nonconvergent behaviors.

► **EXAMPLE 1.15** Apply Newton's Method to $f(x) = 4x^4 - 6x^2 - 11/4$ with starting guess $x_0 = 1/2$.

This function has roots, since it is continuous, negative at $x = 0$, and goes to positive infinity for large positive and large negative x . However, no root will be found for the starting guess $x_0 = 1/2$, as shown in Figure 1.10. The Newton formula is

$$x_{i+1} = x_i - \frac{4x_i^4 - 6x_i^2 - \frac{11}{4}}{16x_i^3 - 12x_i}. \quad (1.33)$$

Substitution gives $x_1 = -1/2$, and then $x_2 = 1/2$ again. Newton's Method alternates on this example between the two nonroots $1/2$ and $-1/2$, and fails to find a root.

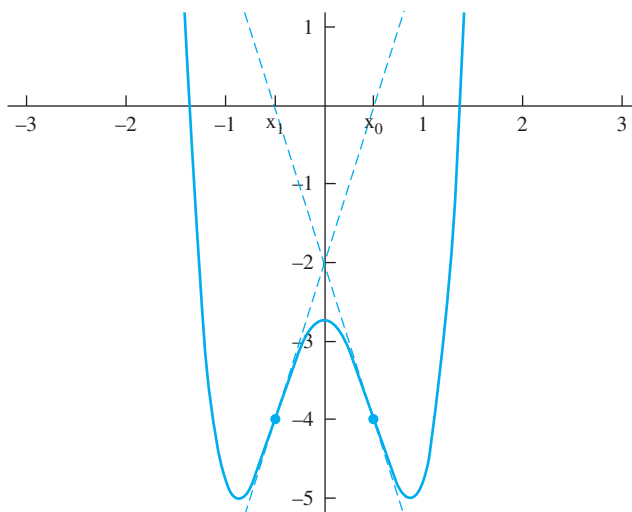


Figure 1.10 Failure of Newton's Method in Example 1.15. The iteration alternates between $1/2$ and $-1/2$, and does not converge to a root.

Newton's Method can fail in other ways. Obviously, if $f'(x_i) = 0$ at any iteration step, the method cannot continue. There are other examples where the iteration diverges to infinity (see Exercise 6) or mimics a random number generator (see Computer Problem 13). Although not every initial guess leads to convergence to a root, Theorems 1.11 and 1.12 guarantee a neighborhood of initial guesses surrounding each root for which convergence to that root is assured.

1.4 Exercises

- Apply two steps of Newton's Method with initial guess $x_0 = 0$. (a) $x^3 + x - 2 = 0$
(b) $x^4 - x^2 + x - 1 = 0$ (c) $x^2 - x - 1 = 0$
- Apply two steps of Newton's Method with initial guess $x_0 = 1$. (a) $x^3 + x^2 - 1 = 0$
(b) $x^2 + 1/(x + 1) - 3x = 0$ (c) $5x - 10 = 0$
- Use Theorem 1.11 or 1.12 to estimate the error e_{i+1} in terms of the previous error e_i as Newton's Method converges to the given roots. Is the convergence linear or quadratic?

- (a) $x^5 - 2x^4 + 2x^2 - x = 0$; $r = -1, r = 0, r = 1$ (b) $2x^4 - 5x^3 + 3x^2 + x - 1 = 0$; $r = -1/2, r = 1$
- Estimate e_{i+1} as in Exercise 3. (a) $32x^3 - 32x^2 - 6x + 9 = 0$; $r = -1/2, r = 3/4$
(b) $x^3 - x^2 - 5x - 3 = 0$; $r = -1, r = 3$
 - Consider the equation $8x^4 - 12x^3 + 6x^2 - x = 0$. For each of the two solutions $x = 0$ and $x = 1/2$, decide which will converge faster (say, to eight-place accuracy), the Bisection Method or Newton's Method, without running the calculation.
 - Sketch a function f and initial guess for which Newton's Method diverges.
 - Let $f(x) = x^4 - 7x^3 + 18x^2 - 20x + 8$. Does Newton's Method converge quadratically to the root $r = 2$? Find $\lim_{i \rightarrow \infty} e_{i+1}/e_i$, where e_i denotes the error at step i .
 - Prove that Newton's Method applied to $f(x) = ax + b$ converges in one step.
 - Show that applying Newton's Method to $f(x) = x^2 - A$ produces the iteration of Example 1.6.
 - Find the Fixed-Point Iteration produced by applying Newton's Method to $f(x) = x^3 - A$. See Exercise 1.2.10.
 - Use Newton's Method to produce a quadratically convergent method for calculating the n th root of a positive number A , where n is a positive integer. Prove quadratic convergence.
 - Suppose Newton's Method is applied to the function $f(x) = 1/x$. If the initial guess is $x_0 = 1$, find x_{50} .
 - (a) The function $f(x) = x^3 - 4x$ has a root at $r = 2$. If the error $e_i = x_i - r$ after four steps of Newton's Method is $e_4 = 10^{-6}$, estimate e_5 . (b) Apply the same question as (a) to the root $r = 0$. (Caution: The usual formula is not useful.)
 - Let $g(x) = x - f(x)/f'(x)$ denote the Newton's Method iteration for the function f . Define $h(x) = g(g(x))$ to be the result of two successive steps of Newton's Method. Then $h'(x) = g'(g(x))g'(x)$ according to the Chain Rule of calculus. (a) Assume that c is a fixed point of h , but not of g , as in Example 1.15. Show that if c is an inflection point of $f(x)$, that is, $f''(x) = 0$, then the fixed point iteration h is locally convergent to c . It follows that for initial guesses near c , Newton's Method itself does not converge to a root of f , but tends toward the oscillating sequence $\{c, g(c)\}$ (b) Verify that the stable oscillation described in (a) actually occurs in Example 1.15. Computer Problem 14 elaborates on this example.

1.4 Computer Problems

- Each equation has one root. Use Newton's Method to approximate the root to eight correct decimal places. (a) $x^3 = 2x + 2$ (b) $e^x + x = 7$ (c) $e^x + \sin x = 4$
- Each equation has one real root. Use Newton's Method to approximate the root to eight correct decimal places. (a) $x^5 + x = 1$ (b) $\sin x = 6x + 5$ (c) $\ln x + x^2 = 3$
- Apply Newton's Method to find the only root to as much accuracy as possible, and find the root's multiplicity. Then use Modified Newton's Method to converge to the root quadratically. Report the forward and backward errors of the best approximation obtained from each method. (a) $f(x) = 27x^3 + 54x^2 + 36x + 8$ (b) $f(x) = 36x^4 - 12x^3 + 37x^2 - 12x + 1$

4. Carry out the steps of Computer Problem 3 for (a) $f(x) = 2e^{x-1} - x^2 - 1$
(b) $f(x) = \ln(3 - x) + x - 2$.
5. A silo composed of a right circular cylinder of height 10 m surmounted by a hemispherical dome contains 400 m^3 of volume. Find the base radius of the silo to four correct decimal places.
6. A 10-cm-high cone contains 60 cm^3 of ice cream, including a hemispherical scoop on top. Find the radius of the scoop to four correct decimal places.
7. Consider the function $f(x) = e^{\sin^3 x} + x^6 - 2x^4 - x^3 - 1$ on the interval $[-2, 2]$. Plot the function on the interval, and find all three roots to six correct decimal places. Determine which roots converge quadratically, and find the multiplicity of the roots that converge linearly.
8. Carry out the steps of Computer Problem 7 for the function $f(x) = 94\cos^3 x - 24\cos x + 177\sin^2 x - 108\sin^4 x - 72\cos^3 x \sin^2 x - 65$ on the interval $[0, 3]$.
9. Apply Newton's Method to find both roots of the function $f(x) = 14xe^{x-2} - 12e^{x-2} - 7x^3 + 20x^2 - 26x + 12$ on the interval $[0, 3]$. For each root, print out the sequence of iterates, the errors e_i , and the relevant error ratio e_{i+1}/e_i^2 or e_{i+1}/e_i that converges to a nonzero limit. Match the limit with the expected value M from Theorem 1.11 or S from Theorem 1.12.
10. Set $f(x) = 54x^6 + 45x^5 - 102x^4 - 69x^3 + 35x^2 + 16x - 4$. Plot the function on the interval $[-2, 2]$, and use Newton's Method to find all five roots in the interval. Determine for which roots Newton converges linearly and for which the convergence is quadratic.
11. The ideal gas law for a gas at low temperature and pressure is $PV = nRT$, where P is pressure (in atm), V is volume (in L), T is temperature (in K), n is the number of moles of the gas, and $R = 0.0820578$ is the molar gas constant. The van der Waals equation

$$\left(P + \frac{n^2 a}{V^2}\right)(V - nb) = nRT$$

covers the nonideal case where these assumptions do not hold. Use the ideal gas law to compute an initial guess, followed by Newton's Method applied to the van der Waals equation to find the volume of one mole of oxygen at 320 K and a pressure of 15 atm. For oxygen, $a = 1.36 \text{ L}^2\text{-atm/mole}^2$ and $b = 0.003183 \text{ L/mole}$. State your initial guess and solution with three significant digits.

12. Use the data from Computer Problem 11 to find the volume of 1 mole of benzene vapor at 700 K under a pressure of 20 atm. For benzene, $a = 18.0 \text{ L}^2\text{-atm/mole}^2$ and $b = 0.1154 \text{ L/mole}$.
13. (a) Find the root of the function $f(x) = (1 - 3/(4x))^{1/3}$. (b) Apply Newton's Method using an initial guess near the root, and plot the first 50 iterates. This is another way Newton's Method can fail, by producing a chaotic trajectory. (c) Why are Theorems 1.11 and 1.12 not applicable?
14. (a) Fix real numbers $a, b > 0$ and plot the graph of $f(x) = a^2 x^4 - 6abx^2 - 11b^2$ for your chosen values. Do not use $a = 2, b = 1/2$, since that case already appears in Example 1.15. (b) Apply Newton's method to find both the negative root and the positive root of $f(x)$. Then find intervals of positive initial guesses $[d_1, d_2]$, where $d_2 > d_1$, for which Newton's Method: (c) converges to the positive root, (d) converges to the negative root, (e) is defined, but does not converge to any root. Your intervals should not contain any initial guess where $f'(x) = 0$, at which Newton's Method is not defined.

1.5 ROOT-FINDING WITHOUT DERIVATIVES

Apart from multiple roots, Newton's Method converges at a faster rate than the bisection and FPI methods. It achieves this faster rate because it uses more information—in particular, information about the tangent line of the function, which comes from the function's derivative. In some circumstances, the derivative may not be available.

The Secant Method is a good substitute for Newton's Method in this case. It replaces the tangent line with an approximation called the secant line, and converges almost as quickly. Variants of the Secant Method replace the line with an approximating parabola, whose axis is either vertical (Muller's Method) or horizontal (inverse quadratic interpolation). The section ends with the description of Brent's Method, a hybrid method which combines the best features of iterative and bracketing methods.

1.5.1 Secant Method and variants

The Secant Method is similar to the Newton's Method, but replaces the derivative by a difference quotient. Geometrically, the tangent line is replaced with a line through the two last known guesses. The intersection point of the “secant line” is the new guess.

An approximation for the derivative at the current guess x_i is the difference quotient

$$\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}.$$

A straight replacement of this approximation for $f'(x_i)$ in Newton's Method yields the Secant Method.

Secant Method

$x_0, x_1 = \text{initial guesses}$

$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})} \text{ for } i = 1, 2, 3, \dots$$

Unlike Fixed-Point Iteration and Newton's Method, two starting guesses are needed to begin the Secant Method.

It can be shown that under the assumption that the Secant Method converges to r and $f'(r) \neq 0$, the approximate error relationship

$$e_{i+1} \approx \left| \frac{f''(r)}{2f'(r)} \right| e_i e_{i-1}$$

holds and that this implies that

$$e_{i+1} \approx \left| \frac{f''(r)}{2f'(r)} \right|^{\alpha-1} e_i^\alpha,$$

where $\alpha = (1 + \sqrt{5})/2 \approx 1.62$. (See Exercise 6.) The convergence of the Secant Method to simple roots is called **superlinear**, meaning that it lies between linearly and quadratically convergent methods.

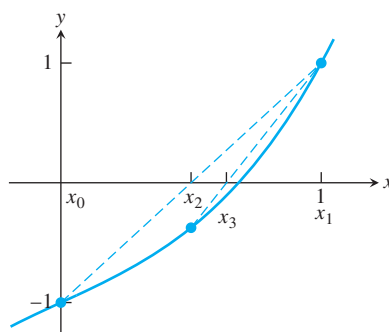


Figure 1.11 Two steps of the Secant Method. Illustration of Example 1.16. Starting with $x_0 = 0$ and $x_1 = 1$, the Secant Method iterates are plotted along with the secant lines.

► **EXAMPLE 1.16** Apply the Secant Method with starting guesses $x_0 = 0, x_1 = 1$ to find the root of $f(x) = x^3 + x - 1$.

The formula gives

$$x_{i+1} = x_i - \frac{(x_i^3 + x_i - 1)(x_i - x_{i-1})}{x_i^3 + x_i - (x_{i-1}^3 + x_{i-1} - 1)}. \quad (1.34)$$

Starting with $x_0 = 0$ and $x_1 = 1$, we compute

$$x_2 = 1 - \frac{(1)(1 - 0)}{1 + 1 - 0} = \frac{1}{2}$$

$$x_3 = \frac{1}{2} - \frac{-\frac{3}{8}(1/2 - 1)}{-\frac{3}{8} - 1} = \frac{7}{11},$$

as shown in Figure 1.11. Further iterates form the following table:

i	x_i
0	0.000000000000000
1	1.000000000000000
2	0.500000000000000
3	0.636363636363636
4	0.69005235602094
5	0.68202041964819
6	0.68232578140989
7	0.68232780435903
8	0.68232780382802
9	0.68232780382802

There are three generalizations of the Secant Method that are also important. The **Method of False Position**, or **Regula Falsi**, is similar to the Bisection Method, but where the midpoint is replaced by a Secant Method–like approximation. Given an interval $[a, b]$ that brackets a root (assume that $f(a)f(b) < 0$), define the next point

$$c = a - \frac{f(a)(a - b)}{f(a) - f(b)} = \frac{bf(a) - af(b)}{f(a) - f(b)}$$

as in the Secant Method, but unlike the Secant Method, the new point is guaranteed to lie in $[a, b]$, since the points $(a, f(a))$ and $(b, f(b))$ lie on separate sides of the x -axis.

The new interval, either $[a, c]$ or $[c, b]$, is chosen according to whether $f(a)f(c) < 0$ or $f(c)f(b) < 0$, respectively, and still brackets a root.

Method of False Position

Given interval $[a, b]$ such that $f(a)f(b) < 0$

for $i = 1, 2, 3, \dots$

$$c = \frac{bf(a) - af(b)}{f(a) - f(b)}$$

if $f(c) = 0$, **stop**, **end**

if $f(a)f(c) < 0$

$$b = c$$

else

$$a = c$$

end

end

The Method of False Position at first appears to be an improvement on both the Bisection Method and the Secant Method, taking the best properties of each. However, while the Bisection Method guarantees cutting the uncertainty by $1/2$ on each step, False Position makes no such promise, and for some examples can converge very slowly.

► **EXAMPLE 1.17** Apply the Method of False Position on initial interval $[-1, 1]$ to find the root $r = 0$ of $f(x) = x^3 - 2x^2 + \frac{3}{2}x$.

Given $x_0 = -1, x_1 = 1$ as the initial bracketing interval, we compute the new point

$$x_2 = \frac{x_1 f(x_0) - x_0 f(x_1)}{f(x_0) - f(x_1)} = \frac{1(-9/2) - (-1)1/2}{-9/2 - 1/2} = \frac{4}{5}.$$

Since $f(-1)f(4/5) < 0$, the new bracketing interval is $[x_0, x_2] = [-1, 0.8]$. This completes the first step. Note that the uncertainty in the solution has decreased by far less than a factor of $1/2$. As Figure 1.12(b) shows, further steps continue to make slow progress toward the root at $x = 0$.

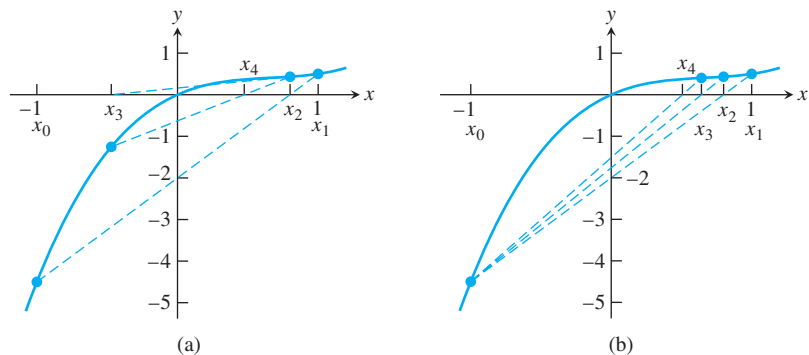


Figure 1.12 Slow convergence in Example 1.17. Both the (a) Secant Method and (b) Method of False Position converge slowly to the root $r=0$.

Muller's Method is a generalization of the Secant Method in a different direction. Instead of intersecting the line through two previous points with the x -axis, we use three previous points x_0, x_1, x_2 , draw the parabola $y = p(x)$ through them, and intersect the parabola

with the x -axis. The parabola will generally intersect in 0 or 2 points. If there are two intersection points, the one nearest to the last point x_2 is chosen to be x_3 . It is a simple matter of the quadratic formula to determine the two possibilities. If the parabola misses the x -axis, there are complex number solutions. This enables software that can handle complex arithmetic to locate complex roots. We will not pursue this idea further, although there are several sources in the literature that follow this direction.

Inverse Quadratic Interpolation (IQI) is a similar generalization of the Secant Method to parabolas. However, the parabola is of form $x = p(y)$ instead of $y = p(x)$, as in Muller's Method. One problem is solved immediately: This parabola will intersect the x -axis in a single point, so there is no ambiguity in finding x_{i+3} from the three previous guesses, x_i, x_{i+1} , and x_{i+2} .

The second-degree polynomial $x = P(y)$ that passes through the three points (a, A) , (b, B) , (c, C) is

$$P(y) = a \frac{(y - B)(y - C)}{(A - B)(A - C)} + b \frac{(y - A)(y - C)}{(B - A)(B - C)} + c \frac{(y - A)(y - B)}{(C - A)(C - B)}. \quad (1.35)$$

This is an example of Lagrange interpolation, one of the topics of Chapter 3. For now, it is enough to notice that $P(A) = a$, $P(B) = b$, and $P(C) = c$. Substituting $y = 0$ gives a formula for the intersection point of the parabola with the x -axis. After some rearrangement and substitution, we have

$$P(0) = c - \frac{r(r - q)(c - b) + (1 - r)s(c - a)}{(q - 1)(r - 1)(s - 1)}, \quad (1.36)$$

where $q = f(a)/f(b)$, $r = f(c)/f(b)$, and $s = f(c)/f(a)$.

For IQI, after setting $a = x_i$, $b = x_{i+1}$, $c = x_{i+2}$, and $A = f(x_i)$, $B = f(x_{i+1})$, $C = f(x_{i+2})$, the next guess $x_{i+3} = P(0)$ is

$$x_{i+3} = x_{i+2} - \frac{r(r - q)(x_{i+2} - x_{i+1}) + (1 - r)s(x_{i+2} - x_i)}{(q - 1)(r - 1)(s - 1)}, \quad (1.37)$$

where $q = f(x_i)/f(x_{i+1})$, $r = f(x_{i+2})/f(x_{i+1})$, and $s = f(x_{i+2})/f(x_i)$. Given three initial guesses, the IQI method proceeds by iterating (1.37), using the new guess x_{i+3} to replace the oldest guess x_i . An alternative implementation of IQI uses the new guess to replace one of the previous three guesses with largest backward error.

Figure 1.13 compares the geometry of Muller's Method with Inverse Quadratic Interpolation. Both methods converge faster than the Secant Method due to the higher-order interpolation. We will study interpolation in more detail in Chapter 3. The concepts of the Secant Method and its generalizations, along with the Bisection Method, are key ingredients of Brent's Method, the subject of the next section.

1.5.2 Brent's Method

Brent's Method [Brent, 1973] is a hybrid method—it uses parts of solving techniques introduced earlier to develop a new approach that retains the most useful properties of each. It is most desirable to combine the property of guaranteed convergence, from the Bisection Method, with the property of fast convergence from the more sophisticated methods. It was originally proposed by Dekker and Van Wijngaarden in the 1960s.

The method is applied to a continuous function f and an interval bounded by a and b , where $f(a)f(b) < 0$. Brent's Method keeps track of a current point x_i that is best in the sense of backward error, and a bracket $[a_i, b_i]$ of the root. Roughly speaking, the Inverse

Quadratic Interpolation method is attempted, and the result is used to replace one of x_i, a_i, b_i if (1) the backward error improves and (2) the bracketing interval is cut at least in half. If not, the Secant Method is attempted with the same goal. If it fails as well, a Bisection Method step is taken, guaranteeing that the uncertainty is cut at least in half.

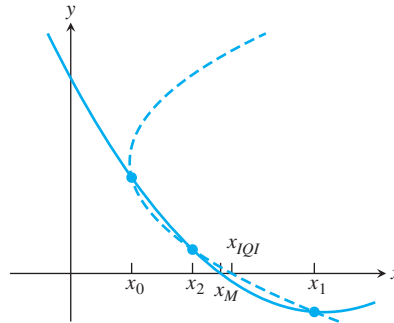


Figure 1.13 Comparison of Muller's Method step with Inverse Quadratic Iteration step. The former is determined by an interpolating parabola $y = p(x)$; the latter, by an interpolating parabola $x = p(y)$.

MATLAB's command `fzero` implements a version of Brent's Method, along with a preprocessing step, to discover a good initial bracketing interval if one is not provided by the user. The stopping criterion is of a mixed forward/backward error type. The algorithm terminates when the change from x_i to the new point x_{i+1} is less than $2\epsilon_{\text{mach}} \max(1, x_i)$, or when the backward error $|f(x_i)|$ achieves machine zero.

The preprocessing step is not triggered if the user provides an initial bracketing interval. The following use of the command enters the function $f(x) = x^3 + x - 1$ and the initial bracketing interval $[0, 1]$ and asks MATLAB to display partial results on each iteration:

```
>> f=@(x) x^3+x-1;
>> fzero(f, [0 1], optimset('Display', 'iter'))
```

Func-count	x	f(x)	Procedure
1	0	-1	initial
2	1	1	initial
3	0.5	-0.375	bisection
4	0.636364	-0.105935	interpolation
5	0.684910	0.00620153	interpolation
6	0.682225	-0.000246683	interpolation
7	0.682328	-5.43508e-007	interpolation
8	0.682328	1.50102e-013	interpolation
9	0.682328	0	interpolation

Zero found in the interval: $[0, 1]$.

ans=

0.68232780382802

Alternatively, the command

```
>> fzero(f, 1)
```

looks for a root of $f(x)$ near $x = 1$ by first locating a bracketing interval and then applying Brent's Method.

1.5 Exercises

1. Apply two steps of the Secant Method to the following equations with initial guesses $x_0 = 1$ and $x_1 = 2$. (a) $x^3 = 2x + 2$ (b) $e^x + x = 7$ (c) $e^x + \sin x = 4$
2. Apply two steps of the Method of False Position with initial bracket $[1, 2]$ to the equations of Exercise 1.
3. Apply two steps of Inverse Quadratic Interpolation to the equations of Exercise 1. Use initial guesses $x_0 = 1$, $x_1 = 2$, and $x_2 = 0$, and update by retaining the three most recent iterates.
4. A commercial fisher wants to set the net at a water depth where the temperature is 10 degrees C. By dropping a line with a thermometer attached, she finds that the temperature is 8 degrees at a depth of 9 meters, and 15 degrees at a depth of 5 meters. Use the Secant Method to determine a best estimate for the depth at which the temperature is 10.
5. Derive equation (1.36) by substituting $y = 0$ into (1.35).
6. If the Secant Method converges to r , $f'(r) \neq 0$, and $f''(r) \neq 0$, then the approximate error relationship $e_{i+1} \approx |f''(r)/(2f'(r))|e_i e_{i-1}$ can be shown to hold. Prove that if in addition $\lim_{i \rightarrow \infty} e_{i+1}/e_i^\alpha$ exists and is nonzero for some $\alpha > 0$, then $\alpha = (1 + \sqrt{5})/2$ and $e_{i+1} \approx |(f''(r)/2f'(r))|^{\alpha-1} e_i^\alpha$.
7. Consider the following four methods for calculating $2^{1/4}$, the fourth root of 2. (a) Rank them for speed of convergence, from fastest to slowest. Be sure to give reasons for your ranking.
 - (A) Bisection Method applied to $f(x) = x^4 - 2$
 - (B) Secant Method applied to $f(x) = x^4 - 2$
 - (C) Fixed Point Iteration applied to $g(x) = \frac{x}{2} + \frac{1}{x^3}$
 - (D) Fixed Point Iteration applied to $g(x) = \frac{x}{3} + \frac{1}{3x^3}$
 (b) Are there any methods that will converge faster than all above suggestions?

1.5 Computer Problems

1. Use the Secant Method to find the (single) solution of each equation in Exercise 1.
2. Use the Method of False Position to find the solution of each equation in Exercise 1.
3. Use Inverse Quadratic Interpolation to find the solution of each equation in Exercise 1.
4. Set $f(x) = 54x^6 + 45x^5 - 102x^4 - 69x^3 + 35x^2 + 16x - 4$. Plot the function on the interval $[-2, 2]$, and use the Secant Method to find all five roots in the interval. To which of the roots is the convergence linear, and to which is it superlinear?
5. In Exercise 1.1.6, you were asked what the outcome of the Bisection Method would be for $f(x) = 1/x$ on the interval $[-2, 1]$. Now compare that result with applying `fzero` to the problem.
6. What happens if `fzero` is asked to find the root of $f(x) = x^2$ near 1 (do not use a bracketing interval)? Explain the result. (b) Apply the same question to $f(x) = 1 + \cos x$ near -1 .



1 Kinematics of the Stewart platform

A Stewart platform consists of six variable length struts, or prismatic joints, supporting a payload. Prismatic joints operate by changing the length of the strut, usually pneumatically or hydraulically. As a six-degree-of-freedom robot, the Stewart platform can be placed at any point and inclination in three-dimensional space that is within its reach.

To simplify matters, the project concerns a two-dimensional version of the Stewart platform. It will model a manipulator composed of a triangular platform in a fixed plane controlled by three struts, as shown in Figure 1.14. The inner triangle represents the planar Stewart platform whose dimensions are defined by the three lengths L_1 , L_2 , and L_3 . Let γ denote the angle across from side L_1 . The position of the platform is controlled by the three numbers p_1 , p_2 , and p_3 , the variable lengths of the three struts.

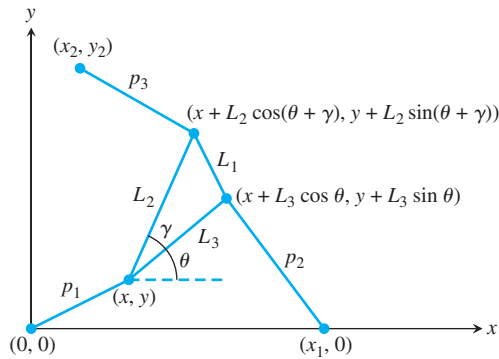


Figure 1.14 Schematic of planar Stewart platform. The forward kinematics problem is to use the lengths p_1 , p_2 , p_3 to determine the unknowns x , y , θ .

Finding the position of the platform, given the three strut lengths, is called the forward, or direct, kinematics problem for this manipulator. Namely, the problem is to compute (x, y) and θ for each given p_1 , p_2 , p_3 . Since there are three degrees of freedom, it is natural to expect three numbers to specify the position. For motion planning, it is important to solve this problem as fast as possible, often in real time. Unfortunately, no closed-form solution of the planar Stewart platform forward kinematics problem is known.

The best current methods involve reducing the geometry of Figure 1.14 to a single equation and solving it by using one of the solvers explained in this chapter. Your job is to complete the derivation of this equation and write code to carry out its solution.

Simple trigonometry applied to Figure 1.14 implies the following three equations:

$$\begin{aligned} p_1^2 &= x^2 + y^2 \\ p_2^2 &= (x + A_2)^2 + (y + B_2)^2 \\ p_3^2 &= (x + A_3)^2 + (y + B_3)^2. \end{aligned} \quad (1.38)$$

In these equations,

$$\begin{aligned} A_2 &= L_3 \cos \theta - x_1 \\ B_2 &= L_3 \sin \theta \\ A_3 &= L_2 \cos(\theta + \gamma) - x_2 = L_2[\cos \theta \cos \gamma - \sin \theta \sin \gamma] - x_2 \\ B_3 &= L_2 \sin(\theta + \gamma) - y_2 = L_2[\cos \theta \sin \gamma + \sin \theta \cos \gamma] - y_2. \end{aligned}$$

Note that (1.38) solves the inverse kinematics problem of the planar Stewart platform, which is to find p_1 , p_2 , p_3 , given x , y , θ . Your goal is to solve the forward problem, namely, to find x , y , θ , given p_1 , p_2 , p_3 .

Multiplying out the last two equations of (1.38) and using the first yields

$$\begin{aligned} p_2^2 &= x^2 + y^2 + 2A_2x + 2B_2y + A_2^2 + B_2^2 = p_1^2 + 2A_2x + 2B_2y + A_2^2 + B_2^2 \\ p_3^2 &= x^2 + y^2 + 2A_3x + 2B_3y + A_3^2 + B_3^2 = p_1^2 + 2A_3x + 2B_3y + A_3^2 + B_3^2, \end{aligned}$$

which can be solved for x and y as

$$\begin{aligned} x &= \frac{N_1}{D} = \frac{B_3(p_2^2 - p_1^2 - A_2^2 - B_2^2) - B_2(p_3^2 - p_1^2 - A_3^2 - B_3^2)}{2(A_2B_3 - B_2A_3)} \\ y &= \frac{N_2}{D} = \frac{-A_3(p_2^2 - p_1^2 - A_2^2 - B_2^2) + A_2(p_3^2 - p_1^2 - A_3^2 - B_3^2)}{2(A_2B_3 - B_2A_3)}, \end{aligned} \quad (1.39)$$

as long as $D = 2(A_2B_3 - B_2A_3) \neq 0$.

Substituting these expressions for x and y into the first equation of (1.38), and multiplying through by D^2 , yields one equation, namely,

$$f = N_1^2 + N_2^2 - p_1^2 D^2 = 0 \quad (1.40)$$

in the single unknown θ . (Recall that $p_1, p_2, p_3, L_1, L_2, L_3, \gamma, x_1, x_2, y_2$ are known.) If the roots of $f(\theta)$ can be found, the corresponding x - and y - values follow immediately from (1.39).

Note that $f(\theta)$ is a polynomial in $\sin \theta$ and $\cos \theta$, so, given any root θ , there are other roots $\theta + 2\pi k$ that are equivalent for the platform. For that reason, we can restrict attention to θ in $[-\pi, \pi]$. It can be shown that $f(\theta)$ has at most six roots in that interval.

Suggested activities:

1. Write a MATLAB function file for $f(\theta)$. The parameters $L_1, L_2, L_3, \gamma, x_1, x_2, y_2$ are fixed constants, and the strut lengths p_1, p_2, p_3 will be known for a given pose. Check Appendix B.5 if you are new to MATLAB function files. Here, for free, are the first and last lines:

```
function out=f(theta)
:
:
out=N1^2+N2^2-p1^2*D^2;
```

To test your code, set the parameters $L_1 = 2, L_2 = L_3 = \sqrt{2}, \gamma = \pi/2, p_1 = p_2 = p_3 = \sqrt{5}$ from Figure 1.15. Then, substituting $\theta = -\pi/4$ or $\theta = \pi/4$, corresponding to Figures 1.15(a, b), respectively, should make $f(\theta) = 0$.

2. Plot $f(\theta)$ on $[-\pi, \pi]$. You may use the `@` symbol as described in Appendix B.5 to assign a function handle to your function file in the plotting command. You may also need to precede arithmetic operations with the `“.”` character to vectorize the operations, as explained in Appendix B.2. As a check of your work, there should be roots at $\pm\pi/4$.
3. Reproduce Figure 1.15. The MATLAB commands

```
>> plot([u1 u2 u3 u1],[v1 v2 v3 v1], 'r'); hold on
>> plot([0 x1 x2],[0 0 y2], 'bo')
```

will plot a red triangle with vertices $(u_1, v_1), (u_2, v_2), (u_3, v_3)$ and place small circles at the strut anchor points $(0, 0), (0, x_1), (x_2, y_2)$. In addition, draw the struts.

4. Solve the forward kinematics problem for the planar Stewart platform specified by $x_1 = 5, (x_2, y_2) = (0, 6), L_1 = L_3 = 3, L_2 = 3\sqrt{2}, \gamma = \pi/4, p_1 = p_2 = 5, p_3 = 3$. Begin

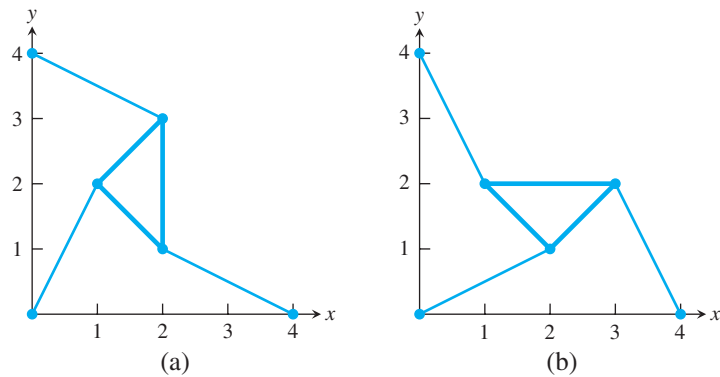


Figure 1.15 Two poses of the planar Stewart platform with identical arm lengths.

Each pose corresponds to a solution of (1.38) with strut lengths $p_1 = p_2 = p_3 = \sqrt{5}$. The shape of the triangle is defined by $L_1 = 2$, $L_2 = L_3 = \sqrt{2}$, $\gamma = \pi/2$.

- by plotting $f(\theta)$. Use an equation solver to find all four poses, and plot them. Check your answers by verifying that p_1 , p_2 , p_3 are the lengths of the struts in your plot.
5. Change strut length to $p_2 = 7$ and re-solve the problem. For these parameters, there are six poses.
 6. Find a strut length p_2 , with the rest of the parameters as in Step 4, for which there are only two poses.
 7. Calculate the intervals in p_2 , with the rest of the parameters as in Step 4, for which there are 0, 2, 4, and 6 poses, respectively.
 8. Derive or look up the equations representing the forward kinematics of the three-dimensional, six-degrees-of-freedom Stewart platform. Write a MATLAB program and demonstrate its use to solve the forward kinematics. See Merlet [2000] for a good introduction to prismatic robot arms and platforms. ✓

Software and Further Reading

There are many algorithms for locating solutions of nonlinear equations. The slow, but always convergent, algorithms like the Bisection Method contrast with routines with faster convergence, but without guarantees of convergence, including Newton's Method and variants. Equation solvers can also be divided into two groups, depending on whether or not derivative information is needed from the equation. The Bisection Method, the Secant Method, and Inverse Quadratic Interpolation are examples of methods that need only a black box providing a function value for a given input, while Newton's Method requires derivatives. Brent's Method is a hybrid that combines the best aspects of slow and fast algorithms and does not require derivative calculations. For this reason, it is heavily used as a general-purpose equation solver and is included in many comprehensive software packages.

MATLAB's `fzero` command implements Brent's Method and needs only an initial interval or one initial guess as input. The ZBREN program of IMSL, the NAG routine `c05adc`, and `netlib` FORTRAN program `fzero.f` all rely on this basic approach.

The MATLAB `roots` command finds all roots of a polynomial with an entirely different approach, computing all eigenvalues of the companion matrix, constructed to have eigenvalues identical to all roots of the polynomial.

Other often-cited algorithms are based on Muller's Method and Laguerre's Method, which, under the right conditions, is cubically convergent. For more details, consult the classic texts on equation solving by Traub [1964], Ostrowski [1966], and Householder [1970].



Systems of Equations

Physical laws govern every engineered structure, from skyscrapers and bridges to diving boards and medical devices. Static and dynamic loads cause materials to deform, or bend. Mathematical models of bending are basic tools in the structural engineer's workbench. The degree to which a structure bends under a load depends on the stiffness of the material, as measured by its Young's modulus. The competition between stress and stiffness is modeled by a differential equation, which, after discretization, is reduced to a system of linear equations for solution.

To increase accuracy, a fine discretization is used, making the system of linear equations large and usually sparse. Gaussian elimination methods are efficient for moderately sized matrices, but special iterative algorithms are necessary for large, sparse systems.

Reality Check

Reality Check 2 on page 102 studies solution methods applicable to the Euler–Bernoulli model for pinned and cantilever beams.

In the previous chapter, we studied methods for solving a single equation in a single variable. In this chapter, we consider the problem of solving several simultaneous equations in several variables. Most of our attention will be paid to the case where the number of equations and the number of unknown variables are the same.

Gaussian elimination is the workhorse for reasonably sized systems of linear equations. The chapter begins with the development of efficient and stable versions of this well-known technique. Later in the chapter our attention shifts to iterative methods, required for very large systems. Finally, we develop methods for systems of nonlinear equations.

2.1 GAUSSIAN ELIMINATION

Consider the system

$$\begin{aligned}x + y &= 3 \\ 3x - 4y &= 2.\end{aligned}\tag{2.1}$$

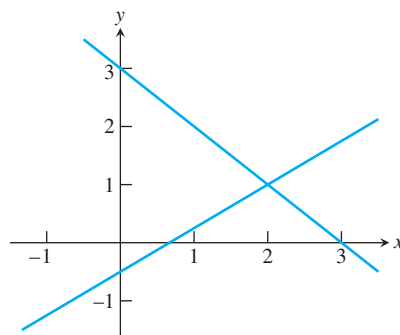


Figure 2.1 Geometric solution of a system of equations. Each equation of (2.1) corresponds to a line in the plane. The intersection point is the solution.

A system of two equations in two unknowns can be considered in terms either of algebra or of geometry. From the geometric point of view, each linear equation represents a line in the xy -plane, as shown in Figure 2.1. The point $x = 2$, $y = 1$ at which the lines intersect satisfies both equations and is the solution we are looking for.

The geometric view is very helpful for visualizing solutions of systems, but for computing the solution with a great deal of accuracy we return to algebra. The method known as Gaussian elimination is an efficient way to solve n equations in n unknowns. In the next few sections, we will explore implementations of Gaussian elimination that work best for typical problems.

2.1.1 Naive Gaussian elimination

We begin by describing the simplest form of Gaussian elimination. In fact, it is so simple that it is not guaranteed to proceed to completion, let alone find an accurate solution. The modifications that will be needed to improve the “naive” method will be introduced beginning in the next section.

Three useful operations can be applied to a linear system of equations that yield an equivalent system, meaning one that has the same solutions. These operations are as follows:

- (1) Swap one equation for another.
- (2) Add or subtract a multiple of one equation from another.
- (3) Multiply an equation by a nonzero constant.

For equation (2.1), we can subtract 3 times the first equation from the second equation to eliminate the x variable from the second equation. Subtracting $3 \cdot [x + y = 3]$ from the second equation leaves us with the system

$$\begin{aligned} x + y &= 3 \\ -7y &= -7. \end{aligned} \tag{2.2}$$

Starting with the bottom equation, we can “backsolve” our way to a full solution, as in

$$-7y = -7 \longrightarrow y = 1$$

and

$$x + y = 3 \longrightarrow x + (1) = 3 \longrightarrow x = 2.$$

Therefore, the solution of (2.1) is $(x, y) = (2, 1)$.

The same elimination work can be done in the absence of variables by writing the system in so-called tableau form:

$$\left[\begin{array}{cc|c} 1 & 1 & 3 \\ 3 & -4 & 2 \end{array} \right] \xrightarrow{\substack{\text{subtract } 3 \times \text{row 1} \\ \text{from row 2}}} \left[\begin{array}{cc|c} 1 & 1 & 3 \\ 0 & -7 & -7 \end{array} \right]. \quad (2.3)$$

The advantage of the tableau form is that the variables are hidden during elimination. When the square array on the left of the tableau is “triangular,” we can backsolve for the solution, starting at the bottom.

► **EXAMPLE 2.1** Apply Gaussian elimination in tableau form for the system of three equations in three unknowns:

$$\begin{aligned} x + 2y - z &= 3 \\ 2x + y - 2z &= 3 \\ -3x + y + z &= -6. \end{aligned} \quad (2.4)$$

This is written in tableau form as

$$\left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 2 & 1 & -2 & 3 \\ -3 & 1 & 1 & -6 \end{array} \right]. \quad (2.5)$$

Two steps are needed to eliminate column 1:

$$\begin{aligned} \left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 2 & 1 & -2 & 3 \\ -3 & 1 & 1 & -6 \end{array} \right] &\xrightarrow{\substack{\text{subtract } 2 \times \text{row 1} \\ \text{from row 2}}} \left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 0 & -3 & 0 & -3 \\ -3 & 1 & 1 & -6 \end{array} \right] \\ &\xrightarrow{\substack{\text{subtract } -3 \times \text{row 1} \\ \text{from row 3}}} \left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 0 & -3 & 0 & -3 \\ 0 & 7 & -2 & 3 \end{array} \right] \end{aligned}$$

and one more step to eliminate column 2:

$$\left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 0 & -3 & 0 & -3 \\ 0 & 7 & -2 & 3 \end{array} \right] \xrightarrow{\substack{\text{subtract } -\frac{7}{3} \times \text{row 2} \\ \text{from row 3}}} \left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 0 & -3 & 0 & -3 \\ 0 & 0 & -2 & -4 \end{array} \right]$$

Returning to the equations

$$\begin{aligned} x + 2y - z &= 3 \\ -3y &= -3 \\ -2z &= -4, \end{aligned} \quad (2.6)$$

we can solve for the variables

$$\begin{aligned} x &= 3 - 2y + z \\ -3y &= -3 \\ -2z &= -4 \end{aligned} \quad (2.7)$$

and solve for z , y , x in that order. The latter part is called **back substitution**, or **backsolving** because, after elimination, the equations are readily solved from the bottom up. The solution is $x = 3$, $y = 1$, $z = 2$. ◀

2.1.2 Operation counts

In this section, we do an approximate operation count for the two parts of Gaussian elimination: the elimination step and the back-substitution step. In order to do this, it will help to write out for the general case the operations that were carried out in the preceding two examples. To begin, recall two facts about sums of integers.

LEMMA 2.1 For any positive integer n , (a) $1 + 2 + 3 + 4 + \cdots + n = n(n + 1)/2$ and (b) $1^2 + 2^2 + 3^2 + 4^2 + \cdots + n^2 = n(n + 1)(2n + 1)/6$. ■

The general form of the tableau for n equations in n unknowns is

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array} \right].$$

To carry out the elimination step, we need to put zeros in the lower triangle, using the allowed row operations.

We can write the elimination step as the loop

```
for j = 1 : n-1
    eliminate column j
end
```

where, by “eliminate column j ,” we mean “use row operations to put a zero in each location below the main diagonal, which are the locations $a_{j+1,j}, a_{j+2,j}, \dots, a_{nj}$.” For example, to carry out elimination on column 1, we need to put zeros in a_{21}, \dots, a_{n1} . This can be written as the following loop within the former loop:

```
for j = 1 : n-1
    for i = j+1 : n
        eliminate entry a(i,j)
    end
end
```

It remains to fill in the inner step of the double loop, to apply a row operation that sets the a_{ij} entry to zero. For example, the first entry to be eliminated is the a_{21} entry. To accomplish this, we subtract a_{21}/a_{11} times row 1 from row 2, assuming that $a_{11} \neq 0$. That is, the first two rows change from

$$\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \end{array}$$

to

$$\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a_{22} - \frac{a_{21}}{a_{11}}a_{12} & \cdots & a_{2n} - \frac{a_{21}}{a_{11}}a_{1n} & b_2 - \frac{a_{21}}{a_{11}}b_1 \end{array}.$$

Accounting for the operations, this requires one division (to find the multiplier a_{21}/a_{11}), plus n multiplications and n additions. The row operation used to eliminate entry a_{i1} of the first column, namely,

$$\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & a_{i2} - \frac{a_{i1}}{a_{11}}a_{12} & \cdots & a_{in} - \frac{a_{i1}}{a_{11}}a_{1n} & b_i - \frac{a_{i1}}{a_{11}}b_1 \end{array}$$

requires similar operations.

The procedure just described works as long as the number a_{11} is nonzero. This number and the other numbers a_{ii} that are eventually divisors in Gaussian elimination are called **pivots**. A zero pivot will cause the algorithm to halt, as we have explained it so far. This issue will be ignored for now and taken up more carefully in Section 2.4.

Returning to the operation count, note that eliminating each entry a_{i1} in the first column uses one division, n multiplications, and n addition/subtractions, or $2n + 1$ operations when counted together. Putting zeros into the first column requires a repeat of these $2n + 1$ operations a total of $n - 1$ times.

After the first column is eliminated, the pivot a_{22} is used to eliminate the second column in the same way and the remaining columns after that. For example, the row operation used to eliminate entry a_{ij} is

$$\begin{array}{cccc|c} 0 & 0 & a_{jj} & a_{j,j+1} & \dots & a_{jn} & b_j \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & a_{i,j+1} - \frac{a_{ij}}{a_{jj}}a_{j,j+1} & \dots & a_{in} - \frac{a_{ij}}{a_{jj}}a_{jn} & b_i - \frac{a_{ij}}{a_{jj}}b_j. \end{array}$$

In our notation, a_{22} , for example, refers to the revised number in that position after the elimination of column 1, which is not the original a_{22} . The row operation to eliminate a_{ij} requires one division, $n - j + 1$ multiplications, and $n - j + 1$ addition/subtractions.

Inserting this step into the same double loop results in

```
for j = 1 : n-1
    if abs(a(j,j)) < eps; error('zero pivot encountered'); end
    for i = j+1 : n
        mult = a(i,j)/a(j,j);
        for k = j+1:n
            a(i,k) = a(i,k) - mult*a(j,k);
        end
        b(i) = b(i) - mult*b(j);
    end
end
```

Two comments on this code fragment are called for: First, asking the index k to move from j to n will put a zero in the a_{ij} location; however, moving from $j + 1$ to n is the most efficient coding. The latter will not place a zero in the a_{ij} entry, which was the entry we are trying to eliminate! Although this seems to be a mistake, note that we will never return to this entry in the remainder of the Gaussian elimination or back-substitution process, so actually putting a zero there represents a wasted step from the point of view of efficiency. Second, we ask the code to shut down, using MATLAB's `error` command, if a zero pivot is encountered. As mentioned, this possibility will be considered more seriously when row exchanges are discussed in Section 2.4.

We can make a total count of operations for the elimination step of Gaussian elimination. The elimination of each a_{ij} requires the following number of operations, including divisions, multiplication, and addition/subtractions:

$$\begin{bmatrix} 0 \\ 2n+1 & 0 \\ 2n+1 & 2(n-1)+1 & 0 \\ 2n+1 & 2(n-1)+1 & 2(n-2)+1 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 2n+1 & 2(n-1)+1 & 2(n-2)+1 & \dots & 2(3)+1 & 0 \\ 2n+1 & 2(n-1)+1 & 2(n-2)+1 & \dots & 2(3)+1 & 2(2)+1 & 0 \end{bmatrix}.$$

It is convenient to add up the operations in reverse order of how they are applied. Starting on the right, we total up the operations as

$$\begin{aligned}
 \sum_{j=1}^{n-1} \sum_{i=1}^j 2(j+1) + 1 &= \sum_{j=1}^{n-1} 2j(j+1) + j \\
 &= 2 \sum_{j=1}^{n-1} j^2 + 3 \sum_{j=1}^{n-1} j = 2 \frac{(n-1)n(2n-1)}{6} + 3 \frac{(n-1)n}{2} \\
 &= (n-1)n \left[\frac{2n-1}{3} + \frac{3}{2} \right] = \frac{n(n-1)(4n+7)}{6} \\
 &= \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n,
 \end{aligned}$$

where Lemma 2.1 has been applied.

Operation count for the elimination step of Gaussian elimination

The elimination step for a system of n equations in n variables can be completed in $\frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n$ operations.

Normally, the exact operation count is less important than order-of-magnitude estimates, since the details of implementation on various computer processors differ. The main point is that the number of operations is approximately proportional to the execution time of the algorithm. We will commonly make the approximation of $\frac{2}{3}n^3$ operations for elimination, which is a reasonably accurate approximation when n is large.

After the elimination is completed, the tableau is upper triangular:

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

In equation form,

$$\begin{aligned}
 a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
 a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
 &\vdots \\
 a_{nn}x_n &= b_n,
 \end{aligned} \tag{2.8}$$

where, again, the a_{ij} refer to the revised, not original, entries. To complete the computation of the solution x , we must carry out the back-substitution step, which is simply a rewriting of (2.8):

$$\begin{aligned}
 x_1 &= \frac{b_1 - a_{12}x_2 - \cdots - a_{1n}x_n}{a_{11}} \\
 x_2 &= \frac{b_2 - a_{23}x_3 - \cdots - a_{2n}x_n}{a_{22}} \\
 &\vdots \\
 x_n &= \frac{b_n}{a_{nn}}.
 \end{aligned} \tag{2.9}$$

SPOTLIGHT ON

Complexity

The operation count shows that direct solution of n equations in n unknowns by Gaussian elimination is an $O(n^3)$ process. This is a useful fact for estimating time required for solving large systems. For example, to estimate the time needed to solve a system of $n = 500$ equations on a particular computer, we could get a fair guess by solving a system of $n = 50$ equations and then scaling the elapsed time by $10^3 = 1000$.

Because of the triangular shape of the nonzero coefficients of the equations, we start at the bottom and work our way up to the top equation. In this way, the required x_i 's are known when they are needed to compute the next one. Counting operations yields

$$1 + 3 + 5 + \cdots + (2n - 1) = \sum_{i=1}^n 2i - 1 = 2 \sum_{i=1}^n i - \sum_{i=1}^n 1 = 2 \frac{n(n+1)}{2} - n = n^2.$$

In MATLAB syntax, the back-substitution step is

```
for i = n : -1 : 1
    for j = i+1 : n
        b(i) = b(i) - a(i,j)*x(j);
    end
    x(i) = b(i)/a(i,i);
end
```

Operation count for the back-substitution step of Gaussian elimination

The back-substitution step for a triangular system of n equations in n variables can be completed in n^2 operations.

The two operation counts, taken together, show that Gaussian elimination is made up of two unequal parts: the relatively expensive elimination step and the relatively cheap back-substitution step. If we ignore the lower order terms in the expressions for the number of multiplication/divisions, we find that elimination takes on the order of $2n^3/3$ operations and that back substitution takes on the order of n^2 .

We will often use the shorthand terminology of “big-O” to mean “on the order of,” saying that elimination is an $O(n^3)$ algorithm and that back substitution is $O(n^2)$.

This usage implies that the emphasis is on large n , where lower powers of n become negligible by comparison. For example, if $n = 100$, only about 1 percent or so of the calculation time of Gaussian elimination goes into the back-substitution step. Overall, Gaussian elimination takes $2n^3/3 + n^2 \approx 2n^3/3$ operations. In other words, for large n , the lower order terms in the complexity count will not have a large effect on the estimate for running time of the algorithm and can be ignored if only an estimated time is required.

► **EXAMPLE 2.2** Estimate the time required to carry out back substitution on a system of 500 equations in 500 unknowns, on a computer where elimination takes 1 second.

Since we have just established that elimination is far more time consuming than back substitution, the answer will be a fraction of a second. Using the approximate number $2(500)^3/3$ for the number of multiply/divide operations for the elimination step, and $(500)^2$ for the back-substitution step, we estimate the time for back substitution to be

$$\frac{(500)^2}{2(500)^3/3} = \frac{3}{2(500)} = .003 \text{ sec.}$$

The example shows two points: (1) Smaller powers of n in operation counts can often be safely neglected, and (2) the two parts of Gaussian elimination can be very unequal

in running time—the total computation time is 1.003 seconds, almost all of which would be taken by the elimination step. The next example shows a third point. While the back-substitution time may sometimes be negligible, it may factor into an important calculation.

- **EXAMPLE 2.3** On a particular computer, back substitution of a 5000×5000 triangular matrix takes 0.1 seconds. Estimate the time needed to solve a general system of 3000 equations in 3000 unknowns by Gaussian elimination.

The computer can carry out $(5000)^2$ operations in 0.1 seconds, or $(5000)^2(10) = 2.5 \times 10^8$ operations/second. Solving a general (nontriangular) system requires about $2(3000)^3/3$ operations, which can be done in approximately

$$\frac{2(3000)^3/3}{(5000)^2(10)} \approx 72 \text{ sec.}$$

2.1 Exercises

1. Use Gaussian elimination to solve the systems:

$$\begin{array}{lll} \text{(a)} & \begin{array}{l} 2x - 3y = 2 \\ 5x - 6y = 8 \end{array} & \begin{array}{l} \text{(b)} \quad \begin{array}{l} x + 2y = -1 \\ 2x + 3y = 1 \end{array} \\ \text{(c)} \quad \begin{array}{l} -x + y = 2 \\ 3x + 4y = 15 \end{array} \end{array}$$

2. Use Gaussian elimination to solve the systems:

$$\begin{array}{lll} \text{(a)} & \begin{array}{l} 2x - 2y - z = -2 \\ 4x + y - 2z = 1 \\ -2x + y - z = -3 \end{array} & \begin{array}{l} \text{(b)} \quad \begin{array}{l} x + 2y - z = 2 \\ 3y + z = 4 \\ 2x - y + z = 2 \end{array} \\ \text{(c)} \quad \begin{array}{l} 2x + y - 4z = -7 \\ x - y + z = -2 \\ -x + 3y - 2z = 6 \end{array} \end{array}$$

3. Solve by back substitution:

$$\begin{array}{ll} \text{(a)} & \begin{array}{l} 3x - 4y + 5z = 2 \\ 3y - 4z = -1 \\ 5z = 5 \end{array} \\ \text{(b)} & \begin{array}{l} x - 2y + z = 2 \\ 4y - 3z = 1 \\ -3z = 3 \end{array} \end{array}$$

4. Solve the tableau form

$$\begin{array}{ll} \text{(a)} & \left[\begin{array}{ccc|c} 3 & -4 & -2 & 3 \\ 6 & -6 & 1 & 2 \\ -3 & 8 & 2 & -1 \end{array} \right] \\ \text{(b)} & \left[\begin{array}{ccc|c} 2 & 1 & -1 & 2 \\ 6 & 2 & -2 & 8 \\ 4 & 6 & -3 & 5 \end{array} \right] \end{array}$$

5. Use the approximate operation count $2n^3/3$ for Gaussian elimination to estimate how much longer it takes to solve n equations in n unknowns if n is tripled.
6. Assume that your computer completes a 5000 equation back substitution in 0.005 seconds. Use the approximate operation counts n^2 for back substitution and $2n^3/3$ for elimination to estimate how long it will take to do a complete Gaussian elimination of this size. Round your answer to the nearest second.
7. Assume that a given computer requires 0.002 seconds to complete back substitution on a 4000×4000 upper triangular matrix equation. Estimate the time needed to solve a general system of 9000 equations in 9000 unknowns. Round your answer to the nearest second.
8. If a system of 3000 equations in 3000 unknowns can be solved by Gaussian elimination in 5 seconds on a given computer, how many back substitutions of the same size can be done per second?

2.1 Computer Problems

1. Put together the code fragments in this section to create a MATLAB program for “naive” Gaussian elimination (meaning no row exchanges allowed). Use it to solve the systems of Exercise 2.
2. Let H denote the $n \times n$ Hilbert matrix, whose (i, j) entry is $1/(i + j - 1)$. Use the MATLAB program from Computer Problem 1 to solve $Hx = b$, where b is the vector of all ones, for (a) $n = 2$ (b) $n = 5$ (c) $n = 10$.

2.2 THE LU FACTORIZATION

Carrying the idea of tableau form one step farther brings us to the matrix form of a system of equations. Matrix form will save time in the long run by simplifying the algorithms and their analysis.

2.2.1 Matrix form of Gaussian elimination

The system (2.1) can be written as $Ax = b$ in matrix form, or

$$\begin{bmatrix} 1 & 1 \\ 3 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}. \quad (2.10)$$

We will usually denote the **coefficient matrix** by A and the **right-hand-side** vector as b . In the matrix form of the systems of equations, we interpret x as a column vector and Ax as matrix-vector multiplication. We want to find x such that the vector Ax is equal to the vector b . Of course, this is equivalent to having Ax and b agree in all components, which is exactly what is required by the original system (2.1).

The advantage of writing systems of equations in matrix form is that we can use matrix operations, like matrix multiplication, to keep track of the steps of Gaussian elimination. The LU factorization is a matrix representation of Gaussian elimination. It consists of writing the coefficient matrix A as a product of a lower triangular matrix L and an upper triangular matrix U . The LU factorization is the Gaussian elimination version of a long tradition in science and engineering—breaking down a complicated object into simpler parts.

DEFINITION 2.2 An $m \times n$ matrix L is **lower triangular** if its entries satisfy $l_{ij} = 0$ for $i < j$. An $m \times n$ matrix U is **upper triangular** if its entries satisfy $u_{ij} = 0$ for $i > j$. \square

► **EXAMPLE 2.4** Find the LU factorization for the matrix A in (2.10).

The elimination steps are the same as for the tableau form seen earlier:

$$\begin{bmatrix} 1 & 1 \\ 3 & -4 \end{bmatrix} \xrightarrow{\text{subtract } 3 \times \text{row } 1 \text{ from row } 2} \begin{bmatrix} 1 & 1 \\ 0 & -7 \end{bmatrix} = U. \quad (2.11)$$

The difference is that now we store the multiplier 3 used in the elimination step. Note that we have defined U to be the upper triangular matrix showing the result of Gaussian elimination. Define L to be the 2×2 lower triangular matrix with 1's on the main diagonal and the multiplier 3 in the $(2,1)$ location:

$$\begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}.$$

Then check that

$$LU = \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & -7 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 3 & -4 \end{bmatrix} = A. \quad (2.12)$$

We will discuss the reason this works soon, but first we demonstrate the steps with a 3×3 example.

► **EXAMPLE 2.5** Find the LU factorization of

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & -2 \\ -3 & 1 & 1 \end{bmatrix}. \quad (2.13)$$

This matrix is the matrix of coefficients of system (2.4). The elimination steps proceed as before:

$$\begin{aligned} \begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & -2 \\ -3 & 1 & 1 \end{bmatrix} &\xrightarrow{\substack{\text{subtract } 2 \times \text{row 1} \\ \text{from row 2}}} \begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ -3 & 1 & 1 \end{bmatrix} \\ &\xrightarrow{\substack{\text{subtract } -3 \times \text{row 1} \\ \text{from row 3}}} \begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ 0 & 7 & -2 \end{bmatrix} \\ &\xrightarrow{\substack{\text{subtract } -\frac{7}{3} \times \text{row 2} \\ \text{from row 3}}} \begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ 0 & 0 & -2 \end{bmatrix} = U. \end{aligned}$$

The lower triangular L matrix is formed, as in the previous example, by putting 1's on the main diagonal and the multipliers in the lower triangle—in the specific places they were used for elimination. That is,

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -\frac{7}{3} & 1 \end{bmatrix}. \quad (2.14)$$

Notice that, for example, 2 is the (2,1) entry of L , because it was the multiplier used to eliminate the (2,1) entry of A . Now check that

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -\frac{7}{3} & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ 0 & 0 & -2 \end{bmatrix} = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & -2 \\ -3 & 1 & 1 \end{bmatrix} = A. \quad (2.15)$$

The reason that this procedure gives the LU factorization follows from three facts about lower triangular matrices.

FACT 1 Let $L_{ij}(-c)$ denote the lower triangular matrix whose only nonzero entries are 1's on the main diagonal and $-c$ in the (i, j) position. Then $A \rightarrow L_{ij}(-c)A$ represents the row operation “subtracting c times row j from row i .”

For example, multiplication by $L_{21}(-c)$ yields

$$\begin{aligned} A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} &\longrightarrow \begin{bmatrix} 1 & 0 & 0 \\ -c & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} - ca_{11} & a_{22} - ca_{12} & a_{23} - ca_{13} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}. \quad \square \end{aligned}$$

FACT 2 $L_{ij}(-c)^{-1} = L_{ij}(c)$.

For example,

$$\begin{bmatrix} 1 & 0 & 0 \\ -c & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Using Facts 1 and 2, we can understand the LU factorization of Example 2.4. Since the elimination step can be represented by

$$L_{21}(-3)A = \begin{bmatrix} 1 & 0 \\ -3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 3 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & -7 \end{bmatrix},$$

we can multiply both sides on the left by $L_{21}(-3)^{-1}$ to get

$$A = \begin{bmatrix} 1 & 1 \\ 3 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & -7 \end{bmatrix},$$

which is the LU factorization of A . □

To handle $n \times n$ matrices for $n > 2$, we need one more fact.

FACT 3 The following matrix product equation holds.

$$\begin{bmatrix} 1 & & \\ c_1 & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ c_2 & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & c_3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & & \\ c_1 & 1 & \\ c_2 & c_3 & 1 \end{bmatrix}.$$

This fact allows us to collect the inverse L_{ij} 's into one matrix, which becomes the L of the LU factorization. For Example 2.5, this amounts to

$$\begin{aligned} \begin{bmatrix} 1 & & \\ & 1 & \\ \frac{7}{3} & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ 3 & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ & 2 & 1 & -2 \\ & -3 & 1 & 1 \end{bmatrix} &= \begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ 0 & 0 & -2 \end{bmatrix} = U \\ A = \begin{bmatrix} 1 & & \\ 2 & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ -3 & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & 1 & \\ & -\frac{7}{3} & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ 0 & 0 & -2 \end{bmatrix} \\ &= \begin{bmatrix} 1 & & \\ 2 & 1 & \\ -3 & -\frac{7}{3} & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ 0 & 0 & -2 \end{bmatrix} = LU. \quad (2.16) \end{aligned} \quad \square$$

2.2.2 Back substitution with the LU factorization

Now that we have expressed the elimination step of Gaussian elimination as a matrix product LU , how do we translate the back-substitution step? More importantly, how do we actually get the solution x ?

Once L and U are known, the problem $Ax = b$ can be written as $LUx = b$. Define a new “auxiliary” vector $c = Ux$. Then back substitution is a two-step procedure:

- (a) Solve $Lc = b$ for c .
- (b) Solve $Ux = c$ for x .

Both steps are straightforward since L and U are triangular matrices. We demonstrate with the two examples used earlier.

► **EXAMPLE 2.6** Solve system (2.10), using the LU factorization (2.12).

The system has LU factorization

$$\begin{bmatrix} 1 & 1 \\ 3 & -4 \end{bmatrix} = LU = \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & -7 \end{bmatrix}$$

from (2.12), and the right-hand side is $b = [3, 2]$. Step (a) is

$$\begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix},$$

which corresponds to the system

$$\begin{aligned} c_1 + 0c_2 &= 3 \\ 3c_1 + c_2 &= 2. \end{aligned}$$


Starting at the top, the solutions are $c_1 = 3, c_2 = -7$.

Step (b) is

$$\begin{bmatrix} 1 & 1 \\ 0 & -7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -7 \end{bmatrix},$$

which corresponds to the system

$$\begin{aligned} x_1 + x_2 &= 3 \\ -7x_2 &= -7. \end{aligned}$$

Starting at the bottom, the solutions are $x_2 = 1, x_1 = 2$. This agrees with the “classical” Gaussian elimination computation done earlier. 

► **EXAMPLE 2.7** Solve system (2.4), using the LU factorization (2.15).

The system has LU factorization

$$\begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & -2 \\ -3 & 1 & 1 \end{bmatrix} = LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -\frac{7}{3} & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

from (2.15), and $b = (3, 3, -6)$. The $Lc = b$ step is

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -\frac{7}{3} & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ -6 \end{bmatrix},$$

which corresponds to the system

$$\begin{aligned} c_1 &= 3 \\ 2c_1 + c_2 &= 3 \\ -3c_1 - \frac{7}{3}c_2 + c_3 &= -6. \end{aligned}$$

Starting at the top, the solutions are $c_1 = 3, c_2 = -3, c_3 = -4$.

The $Ux = c$ step is

$$\begin{bmatrix} 1 & 2 & -1 \\ 0 & -3 & 0 \\ 0 & 0 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ -4 \end{bmatrix},$$

which corresponds to the system

$$\begin{aligned} x_1 + 2x_2 - x_3 &= 3 \\ -3x_2 &= -3 \\ -2x_3 &= -4, \end{aligned}$$

and is solved from the bottom up to give $x = [3, 1, 2]$. 

2.2.3 Complexity of the LU factorization

Now that we have learned the “how” of the LU factorization, here are a few words about “why.” Classical Gaussian elimination involves both A and b in the elimination step of the computation. This is by far the most expensive part of the process, as we have seen. Now, suppose that we need to solve a number of different problems with the same A and different b . That is, we are presented with the set of problems

$$\begin{aligned} Ax &= b_1 \\ Ax &= b_2 \\ &\vdots \\ Ax &= b_k \end{aligned}$$

with various right-hand side vectors b_i . Classical Gaussian elimination will require approximately $2kn^3/3$ operations, where A is an $n \times n$ matrix, since we must start over at the beginning for each problem. With the LU approach, on the other hand, the right-hand-side b doesn't enter the calculations until the elimination (the $A = LU$ factorization) is finished. By insulating the calculations involving A from b , we can solve the previous set of equations with only one elimination, followed by two back substitutions ($Lc = b, Ux = c$) for each new b . The approximate number of operations with the LU approach is, therefore, $2n^3/3 + 2kn^2$. When n^2 is small compared with n^3 (i.e., when n is large), this is a significant difference.

Even when $k = 1$, there is no extra computational work done by the $A = LU$ approach, compared with classical Gaussian elimination. Although there appears to be

SPOTLIGHT ON

Complexity

The main reason for the LU factorization approach to Gaussian elimination is the ubiquity of problems of form $Ax = b_1, Ax = b_2, \dots$. Often, A is a so-called structure matrix, depending only on the design of a mechanical or dynamic system, and b corresponds to a “loading vector.” In structural engineering, the loading vector gives the applied forces at various points on the structure. The solution x then corresponds to the stresses on the structure induced by that particular combination of loadings. Repeated solution of $Ax = b$ for various b 's would be used to test potential structural designs. Reality Check 2 carries out this analysis for the loading of a beam.

an extra back substitution that was not part of classical Gaussian elimination, these “extra” calculations exactly replace the calculations that were saved during elimination because the right-hand-side b was absent.

If all b_i were available at the outset, we could solve all k problems simultaneously in the same number of operations. But in typical applications, we are asked to solve some of the $Ax = b_i$ problems before other b_i 's are available. The LU approach allows efficient handling of all present and future problems that involve the same coefficient matrix A .

► **EXAMPLE 2.8** Assume that it takes one second to factorize the 300×300 matrix A into $A = LU$. How many problems $Ax = b_1, \dots, Ax = b_k$ can be solved in the next second?

The two back substitutions for each b_i require a total of $2n^2$ operations. Therefore, the approximate number of b_i that can be handled per second is

$$\frac{\frac{2n^3}{3}}{2n^2} = \frac{n}{3} = 100.$$

The LU factorization is a significant step forward in our quest to run Gaussian elimination efficiently. Unfortunately, not every matrix allows such a factorization.

► **EXAMPLE 2.9** Prove that $A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ does not have an LU factorization.

The factorization must have the form

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a & 1 \end{bmatrix} \begin{bmatrix} b & c \\ 0 & d \end{bmatrix} = \begin{bmatrix} b & c \\ ab & ac + d \end{bmatrix}.$$

Equating coefficients yields $b = 0$ and $ab = 1$, a contradiction.

The fact that not all matrices have an LU factorization means that more work is required before we can declare the LU factorization a general Gaussian elimination algorithm. The related problem of swamping is described in the next section. In Section 2.4, the $PA = LU$ factorization is introduced, which will overcome both problems.

2.2 Exercises

- Find the LU factorization of the given matrices. Check by matrix multiplication.

$$(a) \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix} \quad (c) \begin{bmatrix} 3 & -4 \\ -5 & 2 \end{bmatrix}$$

- Find the LU factorization of the given matrices. Check by matrix multiplication.

$$(a) \begin{bmatrix} 3 & 1 & 2 \\ 6 & 3 & 4 \\ 3 & 1 & 5 \end{bmatrix} \quad (b) \begin{bmatrix} 4 & 2 & 0 \\ 4 & 4 & 2 \\ 2 & 2 & 3 \end{bmatrix} \quad (c) \begin{bmatrix} 1 & -1 & 1 & 2 \\ 0 & 2 & 1 & 0 \\ 1 & 3 & 4 & 4 \\ 0 & 2 & 1 & -1 \end{bmatrix}$$

- Solve the system by finding the LU factorization and then carrying out the two-step back substitution.

$$(a) \begin{bmatrix} 3 & 7 \\ 6 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -11 \end{bmatrix} \quad (b) \begin{bmatrix} 2 & 3 \\ 4 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

4. Solve the system by finding the LU factorization and then carrying out the two-step back substitution.

$$(a) \begin{bmatrix} 3 & 1 & 2 \\ 6 & 3 & 4 \\ 3 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 3 \end{bmatrix} \quad (b) \begin{bmatrix} 4 & 2 & 0 \\ 4 & 4 & 2 \\ 2 & 2 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

5. Solve the equation $Ax = b$, where

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 4 & 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } b = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 0 \end{bmatrix}.$$

6. Given the 1000×1000 matrix A , your computer can solve the 500 problems $Ax = b_1, \dots, Ax = b_{500}$ in exactly one minute, using $A = LU$ factorization methods. How much of the minute was the computer working on the $A = LU$ factorization? Round your answer to the nearest second.
7. Assume that your computer can solve 1000 problems of type $Ux = c$, where U is an upper-triangular 500×500 matrix, per second. Estimate how long it will take to solve a full 5000×5000 matrix problem $Ax = b$. Answer in minutes and seconds.
8. Assume that your computer can solve a 2000×2000 linear system $Ax = b$ in 0.1 second. Estimate the time required to solve 100 systems of 8000 equations in 8000 unknowns with the same coefficient matrix, using the LU factorization method.
9. Let A be an $n \times n$ matrix. Assume that your computer can solve 100 problems $Ax = b_1, \dots, Ax = b_{100}$ by the LU method in the same amount of time it takes to solve the first problem $Ax = b_0$. Estimate n .

2.2 Computer Problems

- Use the code fragments for Gaussian elimination in the previous section to write a MATLAB script to take a matrix A as input and output L and U . No row exchanges are allowed—the program should be designed to shut down if it encounters a zero pivot. Check your program by factoring the matrices in Exercise 2.
- Add two-step back substitution to your script from Computer Problem 1, and use it to solve the systems in Exercise 4.

2.3 SOURCES OF ERROR

There are two major potential sources of error in Gaussian elimination as we have described it so far. The concept of ill-conditioning concerns the sensitivity of the solution to the input data. We will discuss condition number, using the concepts of backward and forward error from Chapter 1. Very little can be done to avoid errors in computing the solution of ill-conditioned matrix equations, so it is important to try to recognize and avoid ill-conditioned matrices when possible. The second source of error is swamping, which can be avoided in the large majority of problems by a simple fix called partial pivoting, the subject of Section 2.4.

The concept of vector and matrix norms are introduced next to measure the size of errors, which are now vectors. We will give the main emphasis to the so-called infinity norm.

2.3.1 Error magnification and condition number

In Chapter 1, we found that some equation-solving problems show a great difference between backward and forward error. The same is true for systems of linear equations. In order to quantify the errors, we begin with a definition of the infinity norm of a vector.

DEFINITION 2.3 The **infinity norm**, or **maximum norm**, of the vector $x = (x_1, \dots, x_n)$ is $\|x\|_\infty = \max |x_i|, i = 1, \dots, n$, that is, the maximum of the absolute values of the components of x . \square

The backward and forward errors are defined in analogy with Definition 1.8. Backward error represents differences in the input, or problem data side, and forward error represents differences in the output, solution side of the algorithm.

DEFINITION 2.4 Let x_a be an approximate solution of the linear system $Ax = b$. The **residual** is the vector $r = b - Ax_a$. The **backward error** is the norm of the residual $\|b - Ax_a\|_\infty$, and the **forward error** is $\|x - x_a\|_\infty$. \square

► **EXAMPLE 2.10** Find the backward and forward errors for the approximate solution $x_a = [1, 1]$ of the system

$$\begin{bmatrix} 1 & 1 \\ 3 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}.$$

The correct solution is $x = [2, 1]$. In the infinity norm, the backward error is

$$\begin{aligned} \|b - Ax_a\|_\infty &= \left\| \begin{bmatrix} 3 \\ 2 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 3 & -4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\|_\infty \\ &= \left\| \begin{bmatrix} 1 \\ 3 \end{bmatrix} \right\|_\infty = 3, \end{aligned}$$

and the forward error is

$$\|x - x_a\|_\infty = \left\| \begin{bmatrix} 2 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\|_\infty = \left\| \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\|_\infty = 1. \quad \blacktriangleleft$$

In other cases, the backward and forward errors can be of different orders of magnitude.

► **EXAMPLE 2.11** Find the forward and backward errors for the approximate solution $[-1, 3.0001]$ of the system

$$\begin{aligned} x_1 + x_2 &= 2 \\ 1.0001x_1 + x_2 &= 2.0001. \end{aligned} \tag{2.17}$$

First, find the exact solution $[x_1, x_2]$. Gaussian elimination consists of the steps

$$\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 1.0001 & 1 & 2.0001 \end{array} \right] \xrightarrow{\text{subtract } 1.0001 \times \text{row 1} \\ \text{from row 2}} \left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & -0.0001 & -0.0001 \end{array} \right].$$

Solving the resulting equations

$$\begin{aligned}x_1 + x_2 &= 2 \\ -0.0001x_2 &= -0.0001\end{aligned}$$

yields the solution $[x_1, x_2] = [1, 1]$.

The backward error is the infinity norm of the vector

$$\begin{aligned}b - Ax_a &= \begin{bmatrix} 2 \\ 2.0001 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 1.0001 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 3.0001 \end{bmatrix} \\ &= \begin{bmatrix} 2 \\ 2.0001 \end{bmatrix} - \begin{bmatrix} 2.0001 \\ 2 \end{bmatrix} = \begin{bmatrix} -0.0001 \\ 0.0001 \end{bmatrix},\end{aligned}$$

which is 0.0001. The forward error is the infinity norm of the difference

$$x - x_a = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} -1 \\ 3.0001 \end{bmatrix} = \begin{bmatrix} 2 \\ -2.0001 \end{bmatrix},$$

which is 2.0001. ◀

Figure 2.2 helps to clarify how there can be a small backward error and large forward error at the same time. Even though the “approximate root” $(-1, 3.0001)$ is relatively far from the exact root $(1, 1)$, it nearly lies on both lines. This is possible because the two lines are almost parallel. If the lines are far from parallel, the forward and backward errors will be closer in magnitude.

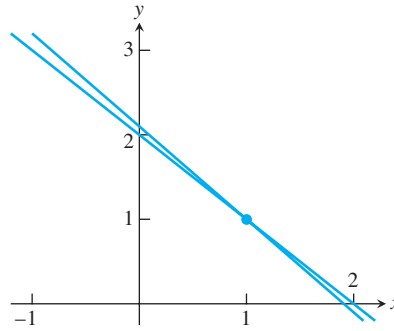


Figure 2.2 The geometry behind Example 2.11. System (2.17) is represented by the lines $x_2 = 2 - x_1$ and $x_2 = 2.0001 - 1.0001x_1$, which intersect at $(1, 1)$. The point $(-1, 3.0001)$ nearly misses lying on both lines and being a solution. The differences between the lines is exaggerated in the figure—they are actually much closer.

Denote the residual by $r = b - Ax_a$. The **relative backward error** of system $Ax = b$ is defined to be

$$\frac{\|r\|_\infty}{\|b\|_\infty},$$

and the **relative forward error** is

$$\frac{\|x - x_a\|_\infty}{\|x\|_\infty}.$$

SPOTLIGHT ON

Conditioning

Condition number is a theme that runs throughout numerical analysis. In the discussions of the Wilkinson polynomial in Chapter 1, we found how to compute the error magnification factor for root-finding, given small perturbations of an equation $f(x) = 0$. For matrix equations $Ax = b$, there is a similar error magnification factor, and the maximum possible factor is given by $\text{cond}(A) = \|A\| \|A^{-1}\|$.

The **error magnification factor** for $Ax = b$ is the ratio of the two, or

$$\text{error magnification factor} = \frac{\text{relative forward error}}{\text{relative backward error}} = \frac{\frac{\|x - x_a\|_\infty}{\|x\|_\infty}}{\frac{\|r\|_\infty}{\|b\|_\infty}}. \quad (2.18)$$

For system (2.17), the relative backward error is

$$\frac{0.0001}{2.0001} \approx 0.00005 = 0.005\%,$$

and the relative forward error is

$$\frac{2.0001}{1} = 2.0001 \approx 200\%.$$

The error magnification factor is $2.0001/(0.0001/2.0001) = 40004.0001$.

In Chapter 1, we defined the concept of condition number to be the maximum error magnification over a prescribed range of input errors. The “prescribed range” depends on the context. Now we will be more precise about it for the current context of systems of linear equations. For a fixed matrix A , consider solving $Ax = b$ for various vectors b . In this context, b is the input and the solution x is the output. A small change in input is a small change in b , which has an error magnification factor. We therefore make the following definition:

DEFINITION 2.5 The **condition number** of a square matrix A , $\text{cond}(A)$, is the maximum possible error magnification factor for solving $Ax = b$, over all right-hand sides b . \square

Surprisingly, there is a compact formula for the condition number of a square matrix. Analogous to the norm of a vector, define the **matrix norm** of an $n \times n$ matrix A as

$$\|A\|_\infty = \text{maximum absolute row sum}, \quad (2.19)$$

that is, total the absolute values of each row, and assign the maximum of these n numbers to be the norm of A .

THEOREM 2.6 The condition number of the $n \times n$ matrix A is

$$\text{cond}(A) = \|A\| \cdot \|A^{-1}\|. \quad \blacksquare$$

Theorem 2.6, proved below, allows us to calculate the condition number of the coefficient matrix in Example 2.11. The norm of

$$A = \begin{bmatrix} 1 & 1 \\ 1.0001 & 1 \end{bmatrix}$$

is $\|A\| = 2.0001$, according to (2.19). The inverse of A is

$$A^{-1} = \begin{bmatrix} -10000 & 10000 \\ 10001 & -10000 \end{bmatrix},$$

which has norm $\|A^{-1}\| = 20001$. The condition number of A is

$$\text{cond}(A) = (2.0001)(20001) = 40004.0001.$$

This is exactly the error magnification we found in Example 2.11, which evidently achieves the worst case, defining the condition number. The error magnification factor for any other b in this system will be less than or equal to 40004.0001. Exercise 3 asks for the computation of some of the other error magnification factors.

The significance of the condition number is the same as in Chapter 1. Error magnification factors of the magnitude $\text{cond}(A)$ are possible. In floating point arithmetic, the relative backward error cannot be expected to be less than ϵ_{mach} , since storing the entries of b already causes errors of that size. According to (2.18), relative forward errors of size $\epsilon_{\text{mach}} \cdot \text{cond}(A)$ are possible in solving $Ax = b$. In other words, if $\text{cond}(A) \approx 10^k$, we should prepare to lose k digits of accuracy in computing x .

In Example 2.11, $\text{cond}(A) \approx 4 \times 10^4$, so in double precision we should expect about $16 - 4 = 12$ correct digits in the solution x . We can test this by introducing MATLAB's best general-purpose linear equation solver: `\`.

In MATLAB, the backslash command `x = A\b` solves the linear system by using an advanced version of the LU factorization that we will explore in Section 2.4. For now, we will use it as an example of what we can expect from the best possible algorithm operating in floating point arithmetic. The following MATLAB commands deliver the computer solution x_a of Example 2.10:

```
>> A = [1 1; 1.0001 1]; b = [2; 2.0001];
>> xa = A\b
xa =
    1.000000000000222
    0.999999999999778
```

Compared with the correct solution $x = [1, 1]$, the computed solution has about 11 correct digits, close to the prediction from the condition number.

The Hilbert matrix H , with entries $H_{ij} = 1/(i + j - 1)$, is notorious for its large condition number.

► **EXAMPLE 2.12** Let H denote the $n \times n$ Hilbert matrix. Use MATLAB's `\` to compute the solution of $Hx = b$, where $b = H \cdot [1, \dots, 1]^T$, for $n = 6$ and 10.

The right-hand side b is chosen to make the correct solution the vector of n ones, for ease of checking the forward error. MATLAB finds the condition number (in the infinity norm) and computes the solution:

```
>> n=6; H=hilb(n);
>> cond(H, inf)
ans =
    2.907027900294064e+007
>> b=H*ones(n,1);
>> xa=H\b
xa =
    0.999999999999923
    1.000000000002184
    0.999999999985267
```



```

1.00000000038240
0.99999999957855
1.00000000016588

```

The condition number of about 10^7 predicts $16 - 7 = 9$ correct digits in the worst case; there are about 9 correct in the computed solution. Now repeat with $n = 10$:

```

>> n=10;H=hilb(n);
>> cond(H,inf)
ans =
    3.535371683074594e+013
>> b=H*ones(n,1);
>> xa=H\b
xa =
    0.99999999875463
    1.00000010746631
    0.99999771299818
    1.00002077769598
    0.99990094548472
    1.00027218303745
    0.99955359665722
    1.00043125589482
    0.99977366058043
    1.00004976229297

```

Since the condition number is 10^{13} , only $16 - 13 = 3$ correct digits appear in the solution.

For n slightly larger than 10, the condition number of the Hilbert matrix is larger than 10^{16} , and no correct digits can be guaranteed in the computed x_a . ◀

Even excellent software may have no defense against an ill-conditioned problem. Increased precision helps; in extended precision, $\epsilon_{\text{mach}} = 2^{-64} \approx 5.42 \times 10^{-20}$, and we start with 20 digits instead of 16. However, the condition number of the Hilbert matrix grows fast enough with n to eventually disarm any reasonable finite precision.

Fortunately, the large condition numbers of the Hilbert matrix are unusual. Well-conditioned linear systems of n equations in n unknowns are routinely solved in double precision for $n = 10^4$ and larger. However, it is important to know that ill-conditioned problems exist, and that the condition number is useful for diagnosing that possibility. See Computer Problems 1–4 for more examples of error magnification and condition numbers.

The infinity vector norm was used in this section as a simple way to assign a length to a vector. It is an example of a **vector norm** $\|x\|$, which satisfies three properties:

- (i) $\|x\| \geq 0$ with equality if and only if $x = [0, \dots, 0]$
- (ii) for each scalar α and vector x , $\|\alpha x\| = |\alpha| \cdot \|x\|$
- (iii) for vectors x, y , $\|x + y\| \leq \|x\| + \|y\|$.

In addition, $\|A\|_\infty$ is an example of a **matrix norm**, which satisfies three similar properties:

- (i) $\|A\| \geq 0$ with equality if and only if $A = 0$
- (ii) for each scalar α and matrix A , $\|\alpha A\| = |\alpha| \cdot \|A\|$
- (iii) for matrices A, B , $\|A + B\| \leq \|A\| + \|B\|$.

As a different example, the vector **1-norm** of the vector $x = [x_1, \dots, x_n]$ is $\|x\|_1 = |x_1| + \dots + |x_n|$. The matrix 1-norm of the $n \times n$ matrix A is $\|A\|_1 = \text{maximum absolute column sum}$ —that is, the maximum of the 1-norms of the column vectors. See Exercises 9 and 10 for verification that these definitions define norms.

The error magnification factor, condition number, and matrix norm just discussed can be defined for any vector and matrix norm. We will restrict our attention to matrix norms that are **operator norms**, meaning that they can be defined in terms of a particular vector norm as

$$\|A\| = \max \frac{\|Ax\|}{\|x\|},$$

where the maximum is taken over all nonzero vectors x . Then, by definition, the matrix norm is consistent with the associated vector norm, in the sense that

$$\|Ax\| \leq \|A\| \cdot \|x\| \quad (2.20)$$

for any matrix A and vector x . See Exercises 10 and 11 for verification that the norm $\|A\|_\infty$ defined by (2.20) is not only a matrix norm, but also the operator norm for the infinity vector norm.

This fact allows us to prove the aforementioned simple expression for $\text{cond}(A)$. The proof works for the infinity norm and any other operator norm.

Proof of Theorem 2.6. We use the equalities $A(x - x_a) = r$ and $Ax = b$. By consistency property (2.20),

$$\|x - x_a\| \leq \|A^{-1}\| \cdot \|r\|$$

and

$$\frac{1}{\|b\|} \geq \frac{1}{\|A\| \|x\|}.$$

Putting the two inequalities together yields

$$\frac{\|x - x_a\|}{\|x\|} \leq \frac{\|A\|}{\|b\|} \|A^{-1}\| \cdot \|r\|,$$

showing that $\|A\| \|A^{-1}\|$ is an upper bound for all error magnification factors. Second, we can show that the quantity is always attainable. Choose x such that $\|A\| = \|Ax\|/\|x\|$ and r such that $\|A^{-1}\| = \|A^{-1}r\|/\|r\|$, both possible by the definition of operator matrix norm. Set $x_a = x - A^{-1}r$ so that $x - x_a = A^{-1}r$. Then it remains to check the equality

$$\frac{\|x - x_a\|}{\|x\|} = \frac{\|A^{-1}r\|}{\|x\|} = \frac{\|A^{-1}\| \|r\| \|A\|}{\|Ax\|}$$

for this particular choice of x and r .

2.3.2 Swamping

A second significant source of error in classical Gaussian elimination is much easier to fix. We demonstrate swamping with the next example.

► **EXAMPLE 2.13** Consider the system of equations

$$\begin{aligned} 10^{-20}x_1 + x_2 &= 1 \\ x_1 + 2x_2 &= 4. \end{aligned}$$

We will solve the system three times: once with complete accuracy, second where we mimic a computer following IEEE double precision arithmetic, and once more where we exchange the order of the equations first.

1. **Exact solution.** In tableau form, Gaussian elimination proceeds as

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 2 & 4 \end{array} \right] \xrightarrow{\substack{\text{subtract } 10^{20} \times \text{row 1} \\ \text{from row 2}}} \left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & 2 - 10^{20} & 4 - 10^{20} \end{array} \right].$$

The bottom equation is

$$(2 - 10^{20})x_2 = 4 - 10^{20} \longrightarrow x_2 = \frac{4 - 10^{20}}{2 - 10^{20}},$$

and the top equation yields

$$\begin{aligned} 10^{-20}x_1 + \frac{4 - 10^{20}}{2 - 10^{20}} &= 1 \\ x_1 &= 10^{20} \left(1 - \frac{4 - 10^{20}}{2 - 10^{20}} \right) \\ x_1 &= \frac{-2 \times 10^{20}}{2 - 10^{20}}. \end{aligned}$$

The exact solution is

$$[x_1, x_2] = \left[\frac{2 \times 10^{20}}{10^{20} - 2}, \frac{4 - 10^{20}}{2 - 10^{20}} \right] \approx [2, 1].$$

2. **IEEE double precision.** The computer version of Gaussian elimination proceeds slightly differently:

$$\left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 1 & 2 & 4 \end{array} \right] \xrightarrow{\substack{\text{subtract } 10^{20} \times \text{row 1} \\ \text{from row 2}}} \left[\begin{array}{cc|c} 10^{-20} & 1 & 1 \\ 0 & 2 - 10^{20} & 4 - 10^{20} \end{array} \right].$$

In IEEE double precision, $2 - 10^{20}$ is the same as -10^{20} , due to rounding. Similarly, $4 - 10^{20}$ is stored as -10^{20} . Now the bottom equation is

$$-10^{20}x_2 = -10^{20} \longrightarrow x_2 = 1.$$

The machine arithmetic version of the top equation becomes

$$10^{-20}x_1 + 1 = 1,$$

so $x_1 = 0$. The computed solution is exactly

$$[x_1, x_2] = [0, 1].$$

This solution has large relative error compared with the exact solution.

3. **IEEE double precision, after row exchange.** We repeat the computer version of Gaussian elimination, after changing the order of the two equations:


$$\begin{aligned} \left[\begin{array}{cc|c} 1 & 2 & 4 \\ 10^{-20} & 1 & 1 \end{array} \right] &\xrightarrow{\substack{\text{subtract } 10^{-20} \times \text{row 1} \\ \text{from row 2}}} \\ &\longrightarrow \left[\begin{array}{cc|c} 1 & 2 & 4 \\ 0 & 1 - 2 \times 10^{-20} & 1 - 4 \times 10^{-20} \end{array} \right]. \end{aligned}$$

In IEEE double precision, $1 - 2 \times 10^{-20}$ is stored as 1 and $1 - 4 \times 10^{-20}$ is stored as 1. The equations are now

$$\begin{aligned} x_1 + 2x_2 &= 4 \\ x_2 &= 1, \end{aligned}$$

which yield the computed solution $x_1 = 2$ and $x_2 = 1$. Of course, this is not the exact answer, but it is correct up to approximately 16 digits, which is the most we can ask from a computation that uses 52-bit floating point numbers.

The difference between the last two calculations is significant. Version 3 gave us an acceptable solution, while version 2 did not. An analysis of what went wrong with version 2 leads to considering the multiplier 10^{20} that was used for the elimination step. The effect of subtracting 10^{20} times the top equation from the bottom equation was to overpower, or “swamp,” the bottom equation. While there were originally two independent equations, or sources of information, after the elimination step in version 2, there are essentially two copies of the top equation. Since the bottom equation has disappeared, for all practical purposes, we cannot expect the computed solution to satisfy the bottom equation; and it does not.

Version 3, on the other hand, completes elimination without swamping, because the multiplier is 10^{-20} . After elimination, the original two equations are still largely existent, slightly changed into triangular form. The result is an approximate solution that is much more accurate. 

The moral of Example 2.13 is that multipliers in Gaussian elimination should be kept as small as possible to avoid swamping. Fortunately, there is a simple modification to naive Gaussian elimination that forces the absolute value of multipliers to be no larger than 1. This new protocol, which involves judicious row exchanges in the tableau, is called partial pivoting, the topic of the next section.

2.3 Exercises

- Find the norm $\|A\|_\infty$ of each of the following matrices:

$$(a) \quad A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad (b) \quad A = \begin{bmatrix} 1 & 5 & 1 \\ -1 & 2 & -3 \\ 1 & -7 & 0 \end{bmatrix}.$$

- Find the (infinity norm) condition number of

$$(a) \quad A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad (b) \quad A = \begin{bmatrix} 1 & 2.01 \\ 3 & 6 \end{bmatrix} \quad (c) \quad A = \begin{bmatrix} 6 & 3 \\ 4 & 2 \end{bmatrix}.$$

- Find the forward and backward errors, and the error magnification factor (in the infinity norm) for the following approximate solutions x_a of the system in Example 2.11: (a) $[-1, 3]$ (b) $[0, 2]$ (c) $[2, 2]$ (d) $[-2, 4]$ (e) $[-2, 4.0001]$.
- Find the forward and backward errors and error magnification factor for the following approximate solutions of the system $x_1 + 2x_2 = 1, 2x_1 + 4.01x_2 = 2$: (a) $[-1, 1]$ (b) $[3, -1]$ (c) $[2, -1/2]$.
- Find the relative forward and backward errors and error magnification factor for the following approximate solutions of the system $x_1 - 2x_2 = 3, 3x_1 - 4x_2 = 7$: (a) $[-2, -4]$ (b) $[-2, -3]$ (c) $[0, -2]$ (d) $[-1, -1]$ (e) What is the condition number of the coefficient matrix?
- Find the relative forward and backward errors and error magnification factor for the following approximate solutions of the system $x_1 + 2x_2 = 3, 2x_1 + 4.01x_2 = 6.01$: (a) $[-10, 6]$ (b) $[-100, 52]$ (c) $[-600, 301]$ (d) $[-599, 301]$ (e) What is the condition number of the coefficient matrix?

7. Find the norm $\|H\|_\infty$ of the 5×5 Hilbert matrix.
8. (a) Find the condition number of the coefficient matrix in the system $\begin{bmatrix} 1 & 1 \\ 1 + \delta & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 + \delta \end{bmatrix}$ as a function of $\delta > 0$. (b) Find the error magnification factor for the approximate root $x_a = [-1, 3 + \delta]$.
9. (a) Prove that the infinity norm $\|x\|_\infty$ is a vector norm. (b) Prove that the 1-norm $\|x\|_1$ is a vector norm.
10. (a) Prove that the infinity norm $\|A\|_\infty$ is a matrix norm. (b) Prove that the 1-norm $\|A\|_1$ is a matrix norm.
11. Prove that the matrix infinity norm is the operator norm of the vector infinity norm.
12. Prove that the matrix 1-norm is the operator norm of the vector 1-norm.
13. For the matrices in Exercise 1, find a vector x satisfying $\|A\|_\infty = \|Ax\|_\infty / \|x\|_\infty$.
14. For the matrices in Exercise 1, find a vector x satisfying $\|A\|_1 = \|Ax\|_1 / \|x\|_1$.
15. Find the LU factorization of

$$A = \begin{bmatrix} 10 & 20 & 1 \\ 1 & 1.99 & 6 \\ 0 & 50 & 1 \end{bmatrix}.$$

What is the largest magnitude multiplier l_{ij} needed?

2.3 Computer Problems

1. For the $n \times n$ matrix with entries $A_{ij} = 5/(i + 2j - 1)$, set $x = [1, \dots, 1]^T$ and $b = Ax$. Use the MATLAB program from Computer Problem 2.1.1 or MATLAB's backslash command to compute x_c , the double precision computed solution. Find the infinity norm of the forward error and the error magnification factor of the problem $Ax = b$, and compare it with the condition number of A : (a) $n = 6$ (b) $n = 10$.
2. Carry out Computer Problem 1 for the matrix with entries $A_{ij} = 1/(|i - j| + 1)$.
3. Let A be the $n \times n$ matrix with entries $A_{ij} = |i - j| + 1$. Define $x = [1, \dots, 1]^T$ and $b = Ax$. For $n = 100, 200, 300, 400$, and 500 , use the MATLAB program from Computer Problem 2.1.1 or MATLAB's backslash command to compute x_c , the double precision computed solution. Calculate the infinity norm of the forward error for each solution. Find the five error magnification factors of the problems $Ax = b$, and compare with the corresponding condition numbers.
4. Carry out the steps of Computer Problem 3 for the matrix with entries $A_{ij} = \sqrt{(i - j)^2 + n/10}$.
5. For what values of n does the solution in Computer Problem 1 have no correct significant digits?
6. Use the MATLAB program from Computer Problem 2.1.1 to carry out double precision implementations of versions 2 and 3 of Example 2.13, and compare with the theoretical results found in the text.

2.4 THE PA = LU FACTORIZATION

The form of Gaussian elimination considered so far is often called “naive,” because of two serious difficulties: encountering a zero pivot and swamping. For a nonsingular matrix, both can be avoided with an improved algorithm. The key to this improvement is an efficient protocol for exchanging rows of the coefficient matrix, called partial pivoting.

2.4.1 Partial pivoting

At the start of classical Gaussian elimination of n equations in n unknowns, the first step is to use the diagonal element a_{11} as a pivot to eliminate the first column. The **partial pivoting** protocol consists of comparing numbers before carrying out each elimination step. The largest entry of the first column is located, and its row is swapped with the pivot row, in this case the top row.

In other words, at the start of Gaussian elimination, partial pivoting asks that we select the p th row, where

$$|a_{p1}| \geq |a_{i1}| \quad (2.21)$$

for all $1 \leq i \leq n$, and exchange rows 1 and p . Next, elimination of column 1 proceeds as usual, using the “new” version of a_{11} as the pivot. The multiplier used to eliminate a_{i1} will be

$$m_{i1} = \frac{a_{i1}}{a_{11}}$$

and $|m_{i1}| \leq 1$.

The same check is applied to every choice of pivot during the algorithm. When deciding on the second pivot, we start with the current a_{22} and check all entries directly below. We select the row p such that

$$|a_{p2}| \geq |a_{i2}|$$

for all $2 \leq i \leq n$, and if $p \neq 2$, rows 2 and p are exchanged. Row 1 is never involved in this step. If $|a_{22}|$ is already the largest, no row exchange is made.

The protocol applies to each column during elimination. Before eliminating column k , the p with $k \leq p \leq n$ and largest $|a_{pk}|$ is located, and rows p and k are exchanged if necessary before continuing with the elimination. Note that using partial pivoting ensures that all multipliers, or entries of L , will be no greater than 1 in absolute value. With this minor change in the implementation of Gaussian elimination, the problem of swamping illustrated in Example 2.13 is completely avoided.


► **EXAMPLE 2.14** Apply Gaussian elimination with partial pivoting to solve the system (2.1).

The equations can be written in tableau form as

$$\left[\begin{array}{cc|c} 1 & 1 & 3 \\ 3 & -4 & 2 \end{array} \right].$$

According to partial pivoting, we compare $|a_{11}| = 1$ with all entries below it, in this case the single entry $a_{21} = 3$. Since $|a_{21}| > |a_{11}|$, we must exchange rows 1 and 2. The new tableau is

$$\left[\begin{array}{cc|c} 3 & -4 & 2 \\ 1 & 1 & 3 \end{array} \right] \xrightarrow{\text{subtract } \frac{1}{3} \times \text{row 1} \text{ from row 2}} \left[\begin{array}{cc|c} 3 & -4 & 2 \\ 0 & \frac{7}{3} & \frac{7}{3} \end{array} \right].$$

After back substitution, the solution is $x_2 = 1$ and then $x_1 = 2$, as we found earlier. When we solved this system the first time, the multiplier was 3, but under partial pivoting this would never occur. 

► **EXAMPLE 2.15** Apply Gaussian elimination with partial pivoting to solve the system

$$x_1 - x_2 + 3x_3 = -3$$

$$-x_1 - 2x_3 = 1$$

$$2x_1 + 2x_2 + 4x_3 = 0.$$

This example is written in tableau form as

$$\left[\begin{array}{ccc|c} 1 & -1 & 3 & -3 \\ -1 & 0 & -2 & 1 \\ 2 & 2 & 4 & 0 \end{array} \right].$$

Under partial pivoting we compare $|a_{11}| = 1$ with $|a_{21}| = 1$ and $|a_{31}| = 2$, and choose a_{31} for the new pivot. This is achieved through an exchange of rows 1 and 3:

$$\begin{aligned} \left[\begin{array}{ccc|c} 1 & -1 & 3 & -3 \\ -1 & 0 & -2 & 1 \\ 2 & 2 & 4 & 0 \end{array} \right] &\xrightarrow{\text{exchange row 1 and row 3}} \left[\begin{array}{ccc|c} 2 & 2 & 4 & 0 \\ -1 & 0 & -2 & 1 \\ 1 & -1 & 3 & -3 \end{array} \right] \\ &\xrightarrow{\text{subtract } -\frac{1}{2} \times \text{row 1 from row 2}} \left[\begin{array}{ccc|c} 2 & 2 & 4 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & -1 & 3 & -3 \end{array} \right] \\ &\xrightarrow{\text{subtract } \frac{1}{2} \times \text{row 1 from row 3}} \left[\begin{array}{ccc|c} 2 & 2 & 4 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & -2 & 1 & -3 \end{array} \right]. \end{aligned}$$

Before eliminating column 2 we must compare the current $|a_{22}|$ with the current $|a_{32}|$. Because the latter is larger, we again switch rows:

$$\begin{aligned} \left[\begin{array}{ccc|c} 2 & 2 & 4 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & -2 & 1 & -3 \end{array} \right] &\xrightarrow{\text{exchange row 2 and row 3}} \left[\begin{array}{ccc|c} 2 & 2 & 4 & 0 \\ 0 & -2 & 1 & -3 \\ 0 & 1 & 0 & 1 \end{array} \right] \\ &\xrightarrow{\text{subtract } -\frac{1}{2} \times \text{row 2 from row 3}} \left[\begin{array}{ccc|c} 2 & 2 & 4 & 0 \\ 0 & -2 & 1 & -3 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} \end{array} \right]. \end{aligned}$$


Note that all three multipliers are less than 1 in absolute value.

The equations are now simple to solve. From

$$\frac{1}{2}x_3 = -\frac{1}{2}$$

$$-2x_2 + x_3 = -3$$

$$2x_1 + 2x_2 + 4x_3 = 0,$$

we find that $x = [1, 1, -1]$. 

Notice that partial pivoting also solves the problem of zero pivots. When a potential zero pivot is encountered, for example, if $a_{11} = 0$, it is immediately exchanged for a nonzero pivot somewhere in its column. If there is no such nonzero entry at or below the diagonal entry, then the matrix is singular and Gaussian elimination will fail to provide a solution anyway.

2.4.2 Permutation matrices

Before showing how row exchanges can be used with the LU factorization approach to Gaussian elimination, we will discuss the fundamental properties of permutation matrices.

DEFINITION 2.7 A **permutation matrix** is an $n \times n$ matrix consisting of all zeros, except for a single 1 in every row and column. \square

Equivalently, a permutation matrix P is created by applying arbitrary row exchanges to the $n \times n$ identity matrix (or arbitrary column exchanges). For example,

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

are the only 2×2 permutation matrices, and

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

are the six 3×3 permutation matrices.

The next theorem tells us at a glance what action a permutation matrix causes when multiplied on the left of another matrix.

THEOREM 2.8 Fundamental Theorem of Permutation Matrices. Let P be the $n \times n$ permutation matrix formed by a particular set of row exchanges applied to the identity matrix. Then, for any $n \times n$ matrix A , PA is the matrix obtained by applying exactly the same set of row exchanges to A . \blacksquare

For example, the permutation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

is formed by exchanging rows 2 and 3 of the identity matrix. Multiplying an arbitrary matrix on the left with P has the effect of exchanging rows 2 and 3:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} a & b & c \\ g & h & i \\ d & e & f \end{bmatrix}.$$

A good way to remember Theorem 2.8 is to imagine multiplying P times the identity matrix I :

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

There are two different ways to view this equality: first, as multiplication by the identity matrix (so we get the permutation matrix on the right); second, as the permutation matrix acting on the rows of the identity matrix. The content of Theorem 2.8 is that the row exchanges caused by multiplication by P are exactly the ones involved in the construction of P .

2.4.3 PA = LU factorization

In this section, we put together everything we know about Gaussian elimination into the PA = LU factorization. This is the matrix formulation of elimination with partial pivoting. The PA = LU factorization is the established workhorse for solving systems of linear equations.

As its name implies, the PA = LU factorization is simply the LU factorization of a row-exchanged version of A . Under partial pivoting, the rows that need exchanging are not known at the outset, so we must be careful about fitting the row exchange information into the factorization. In particular, we need to keep track of previous multipliers when a row exchange is made. We begin with an example.

► **EXAMPLE 2.16** Find the PA = LU factorization of the matrix

$$A = \begin{bmatrix} 2 & 1 & 5 \\ 4 & 4 & -4 \\ 1 & 3 & 1 \end{bmatrix}.$$

First, rows 1 and 2 need to be exchanged, according to partial pivoting:

$$\begin{bmatrix} 2 & 1 & 5 \\ 4 & 4 & -4 \\ 1 & 3 & 1 \end{bmatrix} \xrightarrow{\substack{P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \text{exchange rows 1 and 2}}} \begin{bmatrix} 4 & 4 & -4 \\ 2 & 1 & 5 \\ 1 & 3 & 1 \end{bmatrix}.$$

We will use the permutation matrix P to keep track of the cumulative permutation of rows that have been done along the way. Now we perform two row operations, namely,

$$\begin{array}{c} \text{subtract } \frac{1}{2} \times \text{row 1} \\ \text{from row 2} \end{array} \rightarrow \begin{bmatrix} 4 & 4 & -4 \\ \textcircled{\frac{1}{2}} & -1 & 7 \\ 1 & 3 & 1 \end{bmatrix} \xrightarrow{\substack{\text{subtract } \frac{1}{4} \times \text{row 1} \\ \text{from row 3}}} \begin{bmatrix} 4 & 4 & -4 \\ \textcircled{\frac{1}{2}} & -1 & 7 \\ \textcircled{\frac{1}{4}} & 2 & 2 \end{bmatrix},$$

to eliminate the first column. We have done something new—instead of putting only a zero in the eliminated position, we have made the zero a storage location. Inside the zero at the (i, j) position, we store the multiplier m_{ij} that we used to eliminate that position. We do this for a reason. This is the mechanism by which the multipliers will stay with their row, in case future row exchanges are made.

Next we must make a comparison to choose the second pivot. Since $|a_{22}| = 1 < 2 = |a_{32}|$, a row exchange is required before eliminating the second column. Notice that the previous multipliers move along with the row exchange:


$$\begin{array}{c} P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \\ \rightarrow \text{exchange rows 2 and 3} \end{array} \rightarrow \begin{bmatrix} 4 & 4 & -4 \\ \textcircled{\frac{1}{4}} & 2 & 2 \\ \textcircled{\frac{1}{2}} & -1 & 7 \end{bmatrix}$$

Finally, the elimination ends with one more row operation:

$$\begin{array}{c} \text{subtract } -\frac{1}{2} \times \text{row 2} \\ \text{from row 3} \end{array} \rightarrow \begin{bmatrix} 4 & 4 & -4 \\ \textcircled{\frac{1}{4}} & 2 & 2 \\ \textcircled{\frac{1}{2}} & \textcircled{-\frac{1}{2}} & 8 \end{bmatrix}.$$

This is the finished elimination. Now we can read off the PA = LU factorization:

$$\begin{array}{c} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 5 \\ 4 & 4 & -4 \\ 1 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{4} & 1 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 4 & 4 & -4 \\ 0 & 2 & 2 \\ 0 & 0 & 8 \end{bmatrix} \\ P \qquad \qquad A \qquad \qquad L \qquad \qquad U \end{array} \quad (2.22)$$

The entries of L are sitting inside the zeros in the lower triangle of the matrix (below the main diagonal), and U comes from the upper triangle. The final (cumulative) permutation matrix serves as P . 

Using the PA = LU factorization to solve a system of equations $Ax = b$ is just a slight variant of the $A = LU$ version. Multiply through the equation $Ax = b$ by P on the left, and then proceed as before:

$$\begin{aligned} PAx &= Pb \\ LUx &= Pb. \end{aligned} \quad (2.23)$$

Solve

$$\begin{aligned} 1. \quad Lc &= Pb \text{ for } c. \\ 2. \quad Ux &= c \text{ for } x. \end{aligned} \quad (2.24)$$

The important point, as mentioned earlier, is that the expensive part of the calculation, determining PA = LU, can be done without knowing b . Since the resulting LU factorization is of PA , a row-permuted version of the equation coefficients, it is necessary to permute the right-hand-side vector b in precisely the same way before proceeding with the back-substitution stage. That is achieved by using Pb in the first step of back substitution. The value of the matrix formulation of Gaussian elimination is apparent: All of the bookkeeping details of elimination and pivoting are automatic and contained in the matrix equations.

► **EXAMPLE 2.17** Use the PA = LU factorization to solve the system $Ax = b$, where

$$A = \begin{bmatrix} 2 & 1 & 5 \\ 4 & 4 & -4 \\ 1 & 3 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 5 \\ 0 \\ 6 \end{bmatrix}.$$

The PA = LU factorization is known from (2.22). It remains to complete the two back substitutions.

1. $Lc = Pb$:

$$\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{4} & 1 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \\ 6 \end{bmatrix} = \begin{bmatrix} 0 \\ 6 \\ 5 \end{bmatrix}.$$

Starting at the top, we have

$$\begin{aligned} c_1 &= 0 \\ \frac{1}{4}(0) + c_2 &= 6 \Rightarrow c_2 = 6 \\ \frac{1}{2}(0) - \frac{1}{2}(6) + c_3 &= 5 \Rightarrow c_3 = 8. \end{aligned}$$

2. $Ux = c$:

$$\begin{bmatrix} 4 & 4 & -4 \\ 0 & 2 & 2 \\ 0 & 0 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 6 \\ 8 \end{bmatrix}$$

Starting at the bottom,

$$\begin{aligned} 8x_3 &= 8 \Rightarrow x_3 = 1 \\ 2x_2 + 2(1) &= 6 \Rightarrow x_2 = 2 \\ 4x_1 + 4(2) - 4(1) &= 0 \Rightarrow x_1 = -1. \end{aligned} \quad (2.25)$$

Therefore, the solution is $x = [-1, 2, 1]$. 

► **EXAMPLE 2.18** Solve the system $2x_1 + 3x_2 = 4$, $3x_1 + 2x_2 = 1$ using the $PA = LU$ factorization with partial pivoting.

In matrix form, this is the equation

$$\begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}.$$

We begin by ignoring the right-hand-side b . According to partial pivoting, rows 1 and 2 must be exchanged (because $a_{21} > a_{11}$). The elimination step is

$$\begin{aligned} A = \begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix} &\xrightarrow{\text{exchange rows 1 and 2}} \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix} \\ &\xrightarrow{\substack{\text{subtract } \frac{2}{3} \times \text{row 1} \\ \text{from row 2}}} \begin{bmatrix} 3 & 2 \\ \textcircled{\frac{2}{3}} & \frac{5}{3} \end{bmatrix}. \end{aligned}$$

Therefore, the $PA = LU$ factorization is

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 0 & \frac{5}{3} \end{bmatrix}.$$

$P \qquad A \qquad L \qquad U$

The first back substitution $Lc = Pb$ is

$$\begin{bmatrix} 1 & 0 \\ \frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}.$$

Starting at the top, we have

$$\begin{aligned} c_1 &= 1 \\ \frac{2}{3}(1) + c_2 &= 4 \Rightarrow c_2 = \frac{10}{3}. \end{aligned}$$

The second back substitution $Ux = c$ is

$$\begin{bmatrix} 3 & 2 \\ 0 & \frac{5}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{10}{3} \end{bmatrix}.$$

Starting at the bottom, we have

$$\begin{aligned} \frac{5}{3}x_2 &= \frac{10}{3} \Rightarrow x_2 = 2 \\ 3x_1 + 2(2) &= 1 \Rightarrow x_1 = -1. \end{aligned} \quad (2.26)$$

Therefore, the solution is $x = [-1, 2]$. 

Every $n \times n$ matrix has a PA = LU factorization. We simply follow the partial pivoting rule, and if the resulting pivot is zero, it means that all entries that need to be eliminated are already zero, so the column is done.

All of the techniques described so far are implemented in MATLAB. The most sophisticated form of Gaussian elimination we have discussed is the PA = LU factorization. MATLAB's `lu` command accepts a square coefficient matrix A and returns P , L , and U . The following MATLAB script defines the matrix of Example 2.16 and computes its factorization:

```
>> A=[2 1 5; 4 4 -4; 1 3 1];
>> [L,U,P]=lu(A)
```

L=

```
1.0000    0    0
0.2500    1.0000    0
0.5000   -0.5000    1.0000
```

U=

```
4    4   -4
0    2    2
0    0    8
```

P=

```
0    1    0
0    0    1
1    0    0
```

2.4 Exercises

- Find the PA = LU factorization (using partial pivoting) of the following matrices:

$$(a) \begin{bmatrix} 1 & 3 \\ 2 & 3 \end{bmatrix} \quad (b) \begin{bmatrix} 2 & 4 \\ 1 & 3 \end{bmatrix} \quad (c) \begin{bmatrix} 1 & 5 \\ 5 & 12 \end{bmatrix} \quad (d) \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

- Find the PA = LU factorization (using partial pivoting) of the following matrices:

$$(a) \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ -1 & 1 & -1 \end{bmatrix} \quad (b) \begin{bmatrix} 0 & 1 & 3 \\ 2 & 1 & 1 \\ -1 & -1 & 2 \end{bmatrix} \quad (c) \begin{bmatrix} 1 & 2 & -3 \\ 2 & 4 & 2 \\ -1 & 0 & 3 \end{bmatrix} \quad (d) \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 2 \\ -2 & 1 & 0 \end{bmatrix}$$

- Solve the system by finding the PA = LU factorization and then carrying out the two-step back substitution.

$$(a) \begin{bmatrix} 3 & 7 \\ 6 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -11 \end{bmatrix} \quad (b) \begin{bmatrix} 3 & 1 & 2 \\ 6 & 3 & 4 \\ 3 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 3 \end{bmatrix}$$

- Solve the system by finding the PA = LU factorization and then carrying out the two-step back substitution.

$$(a) \begin{bmatrix} 4 & 2 & 0 \\ 4 & 4 & 2 \\ 2 & 2 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \quad (b) \begin{bmatrix} -1 & 0 & 1 \\ 2 & 1 & 1 \\ -1 & 2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2 \\ 17 \\ 3 \end{bmatrix}$$

5. Write down a 5×5 matrix P such that multiplication of another matrix by P on the left causes rows 2 and 5 to be exchanged.
6. (a) Write down the 4×4 matrix P such that multiplying a matrix on the left by P causes the second and fourth rows of the matrix to be exchanged. (b) What is the effect of multiplying on the right by P ? Demonstrate with an example.
7. Change four entries of the leftmost matrix to make the matrix equation correct:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 5 & 6 \\ 5 & 6 & 7 & 8 \\ 7 & 8 & 9 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 6 & 7 & 8 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 0 \\ 1 & 2 & 3 & 4 \end{bmatrix}.$$

8. Find the $PA=LU$ factorization of the matrix A in Exercise 2.3.15. What is the largest multiplier l_{ij} needed?

9. (a) Find the $PA=LU$ factorization of $A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$. (b) Let A be the $n \times n$

matrix of the same form as in (a). Describe the entries of each matrix of its $PA=LU$ factorization.

10. (a) Assume that A is an $n \times n$ matrix with entries $|a_{ij}| \leq 1$ for $1 \leq i, j \leq n$. Prove that the matrix U in its $PA=LU$ factorization satisfies $|u_{ij}| \leq 2^{n-1}$ for all $1 \leq i, j \leq n$. See Exercise 9(b). (b) Formulate and prove an analogous fact for an arbitrary $n \times n$ matrix A .



2 The Euler–Bernoulli Beam

The Euler–Bernoulli beam is a fundamental model for a material bending under stress. Discretization converts the differential equation model into a system of linear equations. The smaller the discretization size, the larger is the resulting system of equations. This example will provide us an interesting case study of the roles of system size and ill-conditioning in scientific computation.

The vertical displacement of the beam is represented by a function $y(x)$, where $0 \leq x \leq L$ along the beam of length L . We will use MKS units in the calculation: meters, kilograms, seconds. The displacement $y(x)$ satisfies the Euler–Bernoulli equation

$$EIy'''' = f(x) \quad (2.27)$$

where E , the Young's modulus of the material, and I , the area moment of inertia, are constant along the beam. The right-hand-side $f(x)$ is the applied load, including the weight of the beam, in force per unit length.

Techniques for discretizing derivatives are found in Chapter 5, where it will be shown that a reasonable approximation for the fourth derivative is

$$y''''(x) \approx \frac{y(x-2h) - 4y(x-h) + 6y(x) - 4y(x+h) + y(x+2h)}{h^4} \quad (2.28)$$

for a small increment h . The discretization error of this approximation is proportional to h^2 (see Exercise 5.1.21.). Our strategy will be to consider the beam as the union of many segments of length h , and to apply the discretized version of the differential equation on each segment.

For a positive integer n , set $h = L/n$. Consider the evenly spaced grid $0 = x_0 < x_1 < \dots < x_n = L$, where $h = x_i - x_{i-1}$ for $i = 1, \dots, n$. Replacing the differential equation (2.27) with the difference approximation (2.28) to get the system of linear equations for the displacements $y_i = y(x_i)$ yields

$$y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2} = \frac{h^4}{EI} f(x_i). \quad (2.29)$$

We will develop n equations in the n unknowns y_1, \dots, y_n . The coefficient matrix, or structure matrix, will have coefficients from the left-hand side of this equation. However, notice that we must alter the equations near the ends of the beam to take the boundary conditions into account.

A diving board is a beam with one end clamped at the support, and the opposite end free. This is called the **clamped-free** beam or sometimes the **cantilever** beam. The boundary conditions for the clamped (left) end and free (right) end are

$$y(0) = y'(0) = y''(L) = y'''(L) = 0.$$

In particular, $y_0 = 0$. Note that finding y_1 , however, presents us with a problem, since applying the approximation (2.29) to the differential equation (2.27) at x_1 results in

$$y_{-1} - 4y_0 + 6y_1 - 4y_2 + y_3 = \frac{h^4}{EI} f(x_1), \quad (2.30)$$

and y_{-1} is not defined. Instead, we must use an alternate derivative approximation at the point x_1 near the clamped end. Exercise 5.1.22(a) derives the approximation

$$y'''(x_1) \approx \frac{16y(x_1) - 9y(x_1 + h) + \frac{8}{3}y(x_1 + 2h) - \frac{1}{4}y(x_1 + 3h)}{h^4} \quad (2.31)$$

which is valid when $y(x_0) = y'(x_0) = 0$.

Calling the approximation “valid,” for now, means that the discretization error of the approximation is proportional to h^2 , the same as for equation (2.28). In theory, this means that the error in approximating the derivative in this way will decrease toward zero in the limit of small h . This concept will be the focal point of the discussion of numerical differentiation in Chapter 5. The result for us is that we can use approximation (2.31) to take the endpoint condition into account for $i = 1$, yielding

$$16y_1 - 9y_2 + \frac{8}{3}y_3 - \frac{1}{4}y_4 = \frac{h^4}{EI} f(x_1).$$

The free right end of the beam requires a little more work because we must compute y_i all the way to the end of the beam. Again, we need alternative derivative approximations at the last two points x_{n-1} and x_n . Exercise 5.1.22 gives the approximations

$$y'''(x_{n-1}) \approx \frac{-28y_n + 72y_{n-1} - 60y_{n-2} + 16y_{n-3}}{17h^4} \quad (2.32)$$

$$y'''(x_n) \approx \frac{72y_n - 156y_{n-1} + 96y_{n-2} - 12y_{n-3}}{17h^4} \quad (2.33)$$

which are valid under the assumption $y''(x_n) = y'''(x_n) = 0$.

Now we can write down the system of n equations in n unknowns for the diving board. This matrix equation summarizes our approximate versions of the original differential equation (2.27) at each point x_1, \dots, x_n , accurate within terms of order h^2 :

$$\begin{bmatrix} 16 & -9 & \frac{8}{3} & -\frac{1}{4} & & & & \\ -4 & 6 & -4 & 1 & & & & \\ & 1 & -4 & 6 & -4 & 1 & & \\ & & 1 & -4 & 6 & -4 & 1 & \\ & & & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & & 1 & -4 & 6 & -4 & 1 \\ & & & & & 1 & -4 & 6 & -4 & 1 \\ & & & & & & \frac{16}{17} & -\frac{60}{17} & \frac{72}{17} & -\frac{28}{17} \\ & & & & & & -\frac{12}{17} & \frac{96}{17} & -\frac{156}{17} & \frac{72}{17} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} = \frac{h^4}{EI} \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ f(x_{n-1}) \\ f(x_n) \end{bmatrix}. \quad (2.34)$$

The structure matrix A in (2.34) is a **banded matrix**, meaning that all entries sufficiently far from the main diagonal are zero. Specifically, the matrix entries $a_{ij} = 0$, except for $|i - j| \leq 3$. The **bandwidth** of this banded matrix is 7, since $i - j$ takes on 7 values for nonzero a_{ij} .

Finally, we are ready to model the clamped-free beam. Let us consider a solid wood diving board composed of Douglas fir. Assume that the diving board is $L = 2$ meters long, 30 cm wide, and 3 cm thick. The density of Douglas fir is approximately 480 kg/m^3 . One Newton of force is 1 kg-m/sec^2 , and the Young's modulus of this wood is approximately $E = 1.3 \times 10^{10}$ Pascals, or Newton/m^2 . The area moment of inertia I around the center of mass of a beam is $wd^3/12$, where w is the width and d the thickness of the beam.

You will begin by calculating the displacement of the beam with no payload, so that $f(x)$ represents only the weight of the beam itself, in units of force per meter. Therefore $f(x)$ is the mass per meter $480wd$ times the downward acceleration of gravity $-g = -9.81 \text{ m/sec}^2$, or the constant $f(x) = f = -480wdg$. The reader should check that the units match on both sides of (2.27). There is a closed-form solution of (2.27) in the case f is constant, so that the result of your computation can be checked for accuracy.

Following the check of your code for the unloaded beam, you will model two further cases. In the first, a sinusoidal load (or “pile”) will be added to the beam. In this case, there is again a known closed-form solution, but the derivative approximations are not exact, so you will be able to monitor the error of your modeling as a function of the grid size h , and see the effect of conditioning problems for large n . Later, you will put a diver on the beam.

Suggested activities:

1. Write a MATLAB program to define the structure matrix A in (2.34). Then, using the MATLAB \ command or code of your own design, solve the system for the displacements y_i using $n = 10$ grid steps.
2. Plot the solution from Step 1 against the correct solution $y(x) = (f/24EI)x^2(x^2 - 4Lx + 6L^2)$, where $f = f(x)$ is the constant defined above. Check the error at the end of the beam, $x = L$ meters. In this simple case the derivative approximations are exact, so your error should be near machine roundoff.
3. Rerun the calculation in Step 1 for $n = 10 \cdot 2^k$, where $k = 1, \dots, 11$. Make a table of the errors at $x = L$ for each n . For which n is the error smallest? Why does the error begin to increase with n after a certain point? You may want to make an accompanying table of the

condition number of A as a function of n to help answer the last question. To carry out this step for large k , you may need to ask MATLAB to store the matrix A as a sparse matrix to avoid running out of memory. To do this, just initialize A with the command $A = \text{sparse}(n, n)$, and proceed as before. We will discuss sparse matrices in more detail in the next section.

4. Add a sinusoidal pile to the beam. This means adding a function of form $s(x) = -pg \sin \frac{\pi}{L}x$ to the force term $f(x)$. Prove that the solution

$$y(x) = \frac{f}{24EI}x^2(x^2 - 4Lx + 6L^2) - \frac{pgL}{EI\pi} \left(\frac{L^3}{\pi^3} \sin \frac{\pi}{L}x - \frac{x^3}{6} + \frac{L}{2}x^2 - \frac{L^2}{\pi^2}x \right)$$

satisfies the Euler–Bernoulli beam equation and the clamped-free boundary conditions.

5. Rerun the calculation as in Step 3 for the sinusoidal load. (Be sure to include the weight of the beam itself.) Set $p = 100$ kg/m and plot your computed solutions against the correct solution. Answer the questions from Step 3, and in addition the following one: Is the error at $x = L$ proportional to h^2 as claimed above? You may want to plot the error versus h on a log–log graph to investigate this question. Does the condition number come into play?
6. Now remove the sinusoidal load and add a 70 kg diver to the beam, balancing on the last 20 cm of the beam. You must add a force per unit length of $-g$ times 70/0.2 kg/m to $f(x_i)$ for all $1.8 \leq x_i \leq 2$, and solve the problem again with the optimal value of n found in Step 5. Plot the solution and find the deflection of the diving board at the free end.
7. If we also fix the free end of the diving board, we have a “clamped-clamped” beam, obeying identical boundary conditions at each end: $y(0) = y'(0) = y(L) = y'(L) = 0$. This version is used to model the sag in a structure, like a bridge. Begin with the slightly different evenly spaced grid $0 = x_0 < x_1 < \dots < x_n < x_{n+1} = L$, where $h = x_i - x_{i-1}$ for $i = 1, \dots, n$, and find the system of n equations in n unknowns that determine y_1, \dots, y_n . (It should be similar to the clamped-free version, except that the last two rows of the coefficient matrix A should be the first two rows reversed.) Solve for a sinusoidal load and answer the questions of Step 5 for the center $x = L/2$ of the beam. The exact solution for the clamped-clamped beam under a sinusoidal load is

$$y(x) = \frac{f}{24EI}x^2(L-x)^2 - \frac{pgL^2}{\pi^4 EI} \left(L^2 \sin \frac{\pi}{L}x + \pi x(x-L) \right).$$

8. Ideas for further exploration: If the width of the diving board is doubled, how does the displacement of the diver change? Does it change more or less than if the thickness is doubled? (Both beams have the same mass.) How does the maximum displacement change if the cross-section is circular or annular with the same area as the rectangle? (The area moment of inertia for a circular cross-section of radius r is $I = \pi r^4/4$, and for an annular cross-section with inner radius r_1 and outer radius r_2 is $I = \pi(r_2^4 - r_1^4)/4$.) Find out the area moment of inertia for I-beams, for example. The Young’s modulus for different materials are also tabulated and available. For example, the density of steel is about 7850 kg/m³ and its Young’s modulus is about 2×10^{11} Pascals.

The Euler–Bernoulli beam is a relatively simple, classical model. More recent models, such as the Timoshenko beam, take into account more exotic bending, where the beam cross-section may not be perpendicular to the beam’s main axis.



2.5 ITERATIVE METHODS

Gaussian elimination is a finite sequence of $O(n^3)$ floating point operations that result in a solution. For that reason, Gaussian elimination is called a **direct** method for solving systems of linear equations. Direct methods, in theory, give the exact solution within a finite number of steps. (Of course, when carried out by a computer using limited precision, the resulting solution will be only approximate. As we saw earlier, the loss of precision is quantified by the condition number.) Direct methods stand in contrast to the root-finding methods described in Chapter 1, which are iterative in form.

So-called **iterative** methods also can be applied to solving systems of linear equations. Similar to Fixed-Point Iteration, the methods begin with an initial guess and refine the guess at each step, converging to the solution vector.

2.5.1 Jacobi Method

The Jacobi Method is a form of fixed-point iteration for a system of equations. In FPI the first step is to rewrite the equations, solving for the unknown. The first step of the Jacobi Method is to do this in the following standardized way: Solve the i th equation for the i th unknown. Then, iterate as in Fixed-Point Iteration, starting with an initial guess.

► **EXAMPLE 2.19** Apply the Jacobi Method to the system $3u + v = 5, u + 2v = 5$.

Begin by solving the first equation for u and the second equation for v . We will use the initial guess $(u_0, v_0) = (0, 0)$. We have

$$\begin{aligned} u &= \frac{5 - v}{3} \\ v &= \frac{5 - u}{2}. \end{aligned} \quad (2.35)$$

The two equations are iterated:

$$\begin{aligned} \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} u_1 \\ v_1 \end{bmatrix} &= \begin{bmatrix} \frac{5-v_0}{3} \\ \frac{5-u_0}{2} \end{bmatrix} = \begin{bmatrix} \frac{5-0}{3} \\ \frac{5-0}{2} \end{bmatrix} = \begin{bmatrix} \frac{5}{3} \\ \frac{5}{2} \end{bmatrix} \\ \begin{bmatrix} u_2 \\ v_2 \end{bmatrix} &= \begin{bmatrix} \frac{5-v_1}{3} \\ \frac{5-u_1}{2} \end{bmatrix} = \begin{bmatrix} \frac{5-5/2}{3} \\ \frac{5-5/3}{2} \end{bmatrix} = \begin{bmatrix} \frac{5}{6} \\ \frac{5}{3} \end{bmatrix} \\ \begin{bmatrix} u_3 \\ v_3 \end{bmatrix} &= \begin{bmatrix} \frac{5-5/3}{3} \\ \frac{5-5/6}{2} \end{bmatrix} = \begin{bmatrix} \frac{10}{9} \\ \frac{25}{12} \end{bmatrix}. \end{aligned} \quad (2.36)$$

Further steps of Jacobi show convergence toward the solution, which is $[1, 2]$. ◀

Now suppose that the equations are given in the reverse order.

► **EXAMPLE 2.20** Apply the Jacobi Method to the system $u + 2v = 5, 3u + v = 5$.

Solve the first equation for the first variable u and the second equation for v . We begin with


$$\begin{aligned} u &= 5 - 2v \\ v &= 5 - 3u. \end{aligned} \quad (2.37)$$


The two equations are iterated as before, but the results are quite different:

$$\begin{aligned}\begin{bmatrix} u_0 \\ v_0 \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} u_1 \\ v_1 \end{bmatrix} &= \begin{bmatrix} 5 - 2v_0 \\ 5 - 3u_0 \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \end{bmatrix} \\ \begin{bmatrix} u_2 \\ v_2 \end{bmatrix} &= \begin{bmatrix} 5 - 2v_1 \\ 5 - 3u_1 \end{bmatrix} = \begin{bmatrix} -5 \\ -10 \end{bmatrix} \\ \begin{bmatrix} u_3 \\ v_3 \end{bmatrix} &= \begin{bmatrix} 5 - 2(-10) \\ 5 - 3(-5) \end{bmatrix} = \begin{bmatrix} 25 \\ 20 \end{bmatrix}.\end{aligned}\quad (2.38)$$

In this case the Jacobi Method fails, as the iteration diverges. 

Since the Jacobi Method does not always succeed, it is helpful to know conditions under which it does work. One important condition is given in the following definition:

DEFINITION 2.9 The $n \times n$ matrix $A = (a_{ij})$ is **strictly diagonally dominant** if, for each $1 \leq i \leq n$, $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$. In other words, each main diagonal entry dominates its row in the sense that it is greater in magnitude than the sum of magnitudes of the remainder of the entries in its row. 

THEOREM 2.10 If the $n \times n$ matrix A is strictly diagonally dominant, then (1) A is a nonsingular matrix, and (2) for every vector b and every starting guess, the Jacobi Method applied to $Ax = b$ converges to the (unique) solution. 

Theorem 2.10 says that, if A is strictly diagonally dominant, then the Jacobi Method applied to the equation $Ax = b$ converges to a solution for each starting guess. The proof of this fact is given in Section 2.5.3. In Example 2.19, the coefficient matrix is at first

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix},$$

which is strictly diagonally dominant because $3 > 1$ and $2 > 1$. Convergence is guaranteed in this case. On the other hand, in Example 2.20, Jacobi is applied to the matrix


$$A = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix},$$

which is not diagonally dominant, and no such guarantee exists. Note that strict diagonal dominance is only a sufficient condition. The Jacobi Method may still converge in its absence.

► **EXAMPLE 2.21** Determine whether the matrices

$$A = \begin{bmatrix} 3 & 1 & -1 \\ 2 & -5 & 2 \\ 1 & 6 & 8 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 3 & 2 & 6 \\ 1 & 8 & 1 \\ 9 & 2 & -2 \end{bmatrix}$$

are strictly diagonally dominant.

The matrix A is diagonally dominant because $|3| > |1| + |-1|$, $|-5| > |2| + |2|$, and $|8| > |1| + |6|$. B is not, because, for example, $|3| > |2| + |6|$ is not true. However, if the first and third rows of B are exchanged, then B is strictly diagonally dominant and Jacobi is guaranteed to converge. 

The Jacobi Method is a form of fixed-point iteration. Let D denote the main diagonal of A , L denote the lower triangle of A (entries below the main diagonal), and U denote the upper triangle (entries above the main diagonal). Then $A = L + D + U$, and the equation to be solved is $Lx + Dx + Ux = b$. Note that this use of L and U differs from the use in the LU factorization, since all diagonal entries of this L and U are zero. The system of equations $Ax = b$ can be rearranged in a fixed-point iteration of form:

$$\begin{aligned} Ax &= b \\ (D + L + U)x &= b \\ Dx &= b - (L + U)x \\ x &= D^{-1}(b - (L + U)x). \end{aligned} \quad (2.39)$$

Since D is a diagonal matrix, its inverse is the matrix of reciprocals of the diagonal entries of A . The Jacobi Method is just the fixed-point iteration of (2.39):

Jacobi Method

$$\begin{aligned} x_0 &= \text{initial vector} \\ x_{k+1} &= D^{-1}(b - (L + U)x_k) \quad \text{for } k = 0, 1, 2, \dots \end{aligned} \quad (2.40)$$

For Example 2.19,

$$\begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \end{bmatrix},$$

the fixed-point iteration (2.40) with $x_k = \begin{bmatrix} u_k \\ v_k \end{bmatrix}$ is

$$\begin{aligned} \begin{bmatrix} u_{k+1} \\ v_{k+1} \end{bmatrix} &= D^{-1}(b - (L + U)x_k) \\ &= \begin{bmatrix} 1/3 & 0 \\ 0 & 1/2 \end{bmatrix} \left(\begin{bmatrix} 5 \\ 5 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_k \\ v_k \end{bmatrix} \right) \\ &= \begin{bmatrix} (5 - v_k)/3 \\ (5 - u_k)/2 \end{bmatrix}, \end{aligned}$$

which agrees with our original version.

2.5.2 Gauss–Seidel Method and SOR

Closely related to the Jacobi Method is an iteration called the **Gauss–Seidel** Method. The only difference between Gauss–Seidel and Jacobi is that in the former, the most recently updated values of the unknowns are used at each step, even if the updating occurs in the current step. Returning to Example 2.19, we see that Gauss–Seidel looks like this:

$$\begin{aligned} \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} u_1 \\ v_1 \end{bmatrix} &= \begin{bmatrix} \frac{5-v_0}{3} \\ \frac{5-u_1}{2} \end{bmatrix} = \begin{bmatrix} \frac{5-0}{3} \\ \frac{5-5/3}{2} \end{bmatrix} = \begin{bmatrix} \frac{5}{3} \\ \frac{5}{3} \end{bmatrix} \\ \begin{bmatrix} u_2 \\ v_2 \end{bmatrix} &= \begin{bmatrix} \frac{5-v_1}{3} \\ \frac{5-u_2}{2} \end{bmatrix} = \begin{bmatrix} \frac{5-5/3}{3} \\ \frac{5-10/9}{2} \end{bmatrix} = \begin{bmatrix} \frac{10}{9} \\ \frac{35}{18} \end{bmatrix} \\ \begin{bmatrix} u_3 \\ v_3 \end{bmatrix} &= \begin{bmatrix} \frac{5-v_2}{3} \\ \frac{5-u_3}{2} \end{bmatrix} = \begin{bmatrix} \frac{5-35/18}{3} \\ \frac{5-55/54}{2} \end{bmatrix} = \begin{bmatrix} \frac{55}{54} \\ \frac{215}{108} \end{bmatrix}. \end{aligned} \quad (2.41)$$

Note the difference between Gauss–Seidel and Jacobi: The definition of v_1 uses u_1 , not u_0 . We see the approach to the solution $[1, 2]$ as with the Jacobi Method, but somewhat more accurately at the same number of steps. Gauss–Seidel often converges faster than Jacobi if the method is convergent. Theorem 2.11 verifies that the Gauss–Seidel Method, like Jacobi, converges to the solution as long as the coefficient matrix is strictly diagonally dominant.

Gauss–Seidel can be written in matrix form and identified as a fixed-point iteration where we isolate the equation $(L + D + U)x = b$ as

$$(L + D)x_{k+1} = -Ux_k + b.$$

Note that the usage of newly determined entries of x_{k+1} is accommodated by including the lower triangle of A into the left-hand side. Rearranging the equation gives the Gauss–Seidel Method.

Gauss–Seidel Method

$x_0 = \text{initial vector}$

$$x_{k+1} = D^{-1}(b - Ux_k - Lx_{k+1}) \quad \text{for } k = 0, 1, 2, \dots$$

► **EXAMPLE 2.22** Apply the Gauss–Seidel Method to the system

$$\begin{bmatrix} 3 & 1 & -1 \\ 2 & 4 & 1 \\ -1 & 2 & 5 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 1 \end{bmatrix}.$$

The Gauss–Seidel iteration is


$$\begin{aligned} u_{k+1} &= \frac{4 - v_k + w_k}{3} \\ v_{k+1} &= \frac{1 - 2u_{k+1} - w_k}{4} \\ w_{k+1} &= \frac{1 + u_{k+1} - 2v_{k+1}}{5}. \end{aligned}$$

Starting with $x_0 = [u_0, v_0, w_0] = [0, 0, 0]$, we calculate

$$\begin{bmatrix} u_1 \\ v_1 \\ w_1 \end{bmatrix} = \begin{bmatrix} \frac{4-0-0}{3} = \frac{4}{3} \\ \frac{1-8/3-0}{4} = -\frac{5}{12} \\ \frac{1+4/3+5/6}{5} = \frac{19}{30} \end{bmatrix} \approx \begin{bmatrix} 1.3333 \\ -0.4167 \\ 0.6333 \end{bmatrix}$$

and

$$\begin{bmatrix} u_2 \\ v_2 \\ w_2 \end{bmatrix} = \begin{bmatrix} \frac{101}{60} \\ -\frac{3}{4} \\ \frac{251}{300} \end{bmatrix} \approx \begin{bmatrix} 1.6833 \\ -0.7500 \\ 0.8367 \end{bmatrix}.$$

The system is strictly diagonally dominant, and therefore the iteration will converge to the solution $[2, -1, 1]$. 

The method called **Successive Over-Relaxation (SOR)** takes the Gauss–Seidel direction toward the solution and “overshoots” to try to speed convergence. Let ω be a real

number, and define each component of the new guess x_{k+1} as a weighted average of ω times the Gauss–Seidel formula and $1 - \omega$ times the current guess x_k . The number ω is called the **relaxation parameter**, and $\omega > 1$ is referred to as **over-relaxation**.

► **EXAMPLE 2.23** Apply SOR with $\omega = 1.25$ to the system of Example 2.22.

Successive Over-Relaxation yields

$$\begin{aligned} u_{k+1} &= (1 - \omega)u_k + \omega \frac{4 - v_k + w_k}{3} \\ v_{k+1} &= (1 - \omega)v_k + \omega \frac{1 - 2u_{k+1} - w_k}{4} \\ w_{k+1} &= (1 - \omega)w_k + \omega \frac{1 + u_{k+1} - 2v_{k+1}}{5}. \end{aligned}$$

Starting with $[u_0, v_0, w_0] = [0, 0, 0]$, we calculate

$$\begin{bmatrix} u_1 \\ v_1 \\ w_1 \end{bmatrix} \approx \begin{bmatrix} 1.6667 \\ -0.7292 \\ 1.0312 \end{bmatrix}$$

and

$$\begin{bmatrix} u_2 \\ v_2 \\ w_2 \end{bmatrix} \approx \begin{bmatrix} 1.9835 \\ -1.0672 \\ 1.0216 \end{bmatrix}.$$

In this example, the SOR iteration converges faster than Jacobi and Gauss–Seidel to the solution $[2, -1, 1]$. ◀

Just as with Jacobi and Gauss–Seidel, an alternative derivation of SOR follows from treating the system as a fixed-point problem. The problem $Ax = b$ can be written $(L + D + U)x = b$, and, upon multiplication by ω and rearranging,

$$\begin{aligned} (\omega L + \omega D + \omega U)x &= \omega b \\ (\omega L + D)x &= \omega b - \omega Ux + (1 - \omega)Dx \\ x &= (\omega L + D)^{-1}[(1 - \omega)Dx - \omega Ux] + \omega(D + \omega L)^{-1}b. \end{aligned}$$

Successive Over-Relaxation (SOR)

x_0 = initial vector

$$x_{k+1} = (\omega L + D)^{-1}[(1 - \omega)Dx_k - \omega Ux_k] + \omega(D + \omega L)^{-1}b \quad \text{for } k = 0, 1, 2, \dots$$

SOR with $\omega = 1$ is exactly Gauss–Seidel. The parameter ω can also be allowed to be less than 1, in a method called Successive Under-Relaxation.

► **EXAMPLE 2.24** Compare Jacobi, Gauss–Seidel, and SOR on the system of six equations in six unknowns:

$$\begin{bmatrix} 3 & -1 & 0 & 0 & 0 & \frac{1}{2} \\ -1 & 3 & -1 & 0 & \frac{1}{2} & 0 \\ 0 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & 0 \\ 0 & \frac{1}{2} & 0 & -1 & 3 & -1 \\ \frac{1}{2} & 0 & 0 & 0 & -1 & 3 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} \frac{5}{2} \\ \frac{3}{2} \\ 1 \\ 1 \\ \frac{3}{2} \\ \frac{5}{2} \end{bmatrix}. \quad (2.42)$$

The solution is $x = [1, 1, 1, 1, 1, 1]$. The approximate solution vectors x_6 , after running six steps of each of the three methods, are shown in the following table:

Jacobi	Gauss–Seidel	SOR
0.9879	0.9950	0.9989
0.9846	0.9946	0.9993
0.9674	0.9969	1.0004
0.9674	0.9996	1.0009
0.9846	1.0016	1.0009
0.9879	1.0013	1.0004

The parameter ω for Successive Over-Relaxation was set at 1.1. SOR appears to be superior for this problem. ▶

Figure 2.3 compares the infinity norm error in Example 2.24 after six iterations for various ω . Although there is no general theory describing the best choice of ω , clearly there is a best choice in this case. See Ortega [1972] for discussion of the optimal ω in some common special cases.

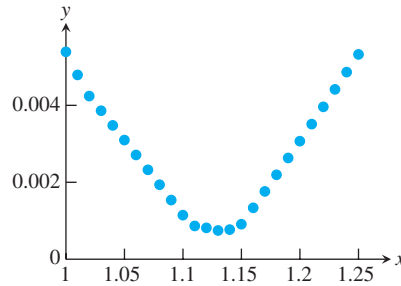


Figure 2.3 Infinity norm error after six steps of SOR in Example 2.24, as a function of over-relaxation parameter ω . Gauss–Seidel corresponds to $\omega = 1$. Minimum error occurs for $\omega \approx 1.13$

2.5.3 Convergence of iterative methods

In this section we prove that the Jacobi and Gauss–Seidel Methods converge for strictly diagonally dominant matrices. This is the content of Theorems 2.10 and 2.11.

The Jacobi Method is written as

$$x_{k+1} = -D^{-1}(L + U)x_k + D^{-1}b. \quad (2.43)$$

Theorem A.7 of Appendix A governs convergence of such an iteration. According to this theorem, we need to know that the spectral radius $\rho(D^{-1}(L + U)) < 1$ in order to guarantee convergence of the Jacobi Method. This is exactly what strict diagonal dominance implies, as shown next.

Proof of Theorem 2.10. Let $R = L + U$ denote the nondiagonal part of the matrix. To check $\rho(D^{-1}R) < 1$, let λ be an eigenvalue of $D^{-1}R$ with corresponding eigenvector v . Choose this v so that $\|v\|_\infty = 1$, so that for some $1 \leq m \leq n$, the component $v_m = 1$ and all other components are no larger than 1. (This can be achieved by starting with any eigenvector and dividing by the largest component. Any constant multiple of an eigenvector is again an eigenvector with the same eigenvalue.) The definition of eigenvalue means that $D^{-1}Rv = \lambda v$, or $Rv = \lambda Dv$.

Since $r_{mm} = 0$, taking absolute values of the m th component of this vector equation implies

$$\begin{aligned} & |r_{m1}v_1 + r_{m2}v_2 + \cdots + r_{m,m-1}v_{m-1} + r_{m,m+1}v_{m+1} + \cdots + r_{mn}v_n| \\ &= |\lambda d_{mm}v_m| = |\lambda||d_{mm}|. \end{aligned}$$

Since all $|v_i| \leq 1$, the left-hand side is at most $\sum_{j \neq m} |r_{mj}|$, which, according to the strict diagonal dominance hypothesis, is less than $|d_{mm}|$. This implies that $|\lambda||d_{mm}| < |d_{mm}|$, which in turn forces $|\lambda| < 1$. Since λ was an arbitrary eigenvalue, we have shown $\rho(D^{-1}R) < 1$, as desired. Now Theorem A.7 from Appendix A implies that Jacobi converges to a solution of $Ax = b$. Finally, since $Ax = b$ has a solution for arbitrary b , A is a nonsingular matrix.

Putting the Gauss–Seidel Method into the form of (2.43) yields

$$x_{k+1} = -(L + D)^{-1}Ux_k + (L + D)^{-1}b.$$

It then becomes clear that convergence of Gauss–Seidel follows if the spectral radius of the matrix

$$(L + D)^{-1}U \tag{2.44}$$

is less than one. The next theorem shows that strict diagonal dominance implies that this requirement is imposed on the eigenvalues.

THEOREM 2.11 If the $n \times n$ matrix A is strictly diagonally dominant, then (1) A is a nonsingular matrix, and (2) for every vector b and every starting guess, the Gauss–Seidel Method applied to $Ax = b$ converges to a solution. ■

Proof. Let λ be an eigenvalue of (2.44), with corresponding eigenvector v . Choose the eigenvector so that $v_m = 1$ and all other components are smaller in magnitude, as in the preceding proof. Note that the entries of L are the a_{ij} for $i > j$, and the entries of U are the a_{ij} for $i < j$. Then viewing row m of the eigenvalue equation of (2.44),

$$\lambda(D + L)v = Uv,$$

yields a string of inequalities similar to the previous proof:

$$\begin{aligned} |\lambda| \left(\sum_{i>m} |a_{mi}| \right) &< |\lambda| \left(|a_{mm}| - \sum_{i<m} |a_{mi}| \right) \\ &\leq |\lambda| \left(|a_{mm}| - \left| \sum_{i<m} a_{mi}v_i \right| \right) \\ &\leq |\lambda| \left| a_{mm} + \sum_{i<m} a_{mi}v_i \right| \\ &= \left| \sum_{i>m} a_{mi}v_i \right| \\ &\leq \sum_{i>m} |a_{mi}|. \end{aligned}$$

It follows that $|\lambda| < 1$, which finishes the proof. ■

2.5.4 Sparse matrix computations

Direct methods based on Gaussian elimination provide the user a finite number of steps that terminate in the solution. What is the reason for pursuing iterative methods, which are only approximate and may require several steps for convergence?

There are two major reasons for using iterative methods like Gauss–Seidel. Both reasons stem from the fact that one step of an iterative method requires only a fraction of the floating point operations of a full LU factorization. As we established earlier in the chapter, Gaussian elimination for an $n \times n$ matrix costs on the order of n^3 operations. A single step of Jacobi’s Method, for example, requires about n^2 multiplications (one for each matrix entry) and about the same number of additions. The question is how many steps will be needed for convergence within the user’s tolerance.

One particular circumstance that argues for an iterative technique is when a good approximation to the solution is already known. For example, suppose that a solution to $Ax = b$ is known, after which A and/or b change by a small amount. We could imagine a dynamic problem where A and b are remeasured constantly as they change, and an accurate updated solution x is constantly required. If the solution to the previous problem is used as a starting guess for the new but similar problem, fast convergence of Jacobi or Gauss–Seidel can be expected.

Suppose the b in problem (2.42) is changed slightly from the original $b = [2.5, 1.5, 1, 1, 1.5, 2.5]$ to a new $b = [2.2, 1.6, 0.9, 1.3, 1.4, 2.45]$. We can check that the true solution of the system is changed from $[1, 1, 1, 1, 1, 1]$ to $[0.9, 1, 1, 1.1, 1, 1]$. Assume that we have in memory the sixth step of the Gauss–Seidel iteration x_6 from the preceding table, to use as a starting guess. Continuing Gauss–Seidel with the new b and with the helpful starting guess x_6 yields a good approximation in only one additional step. The next two steps are as follows:

x_7	x_8
0.8980	0.8994
0.9980	0.9889
0.9659	0.9927
1.0892	1.0966
0.9971	1.0005
0.9993	1.0003

This technique is often called **polishing**, because the method begins with an approximate solution, which could be the solution from a previous, related problem, and then merely refines the approximate solution to make it more accurate. Polishing is common in real-time applications where the same problem needs to be re-solved repeatedly with data that is updated as time passes. If the system is large and time is short, it may be impossible to run an entire Gaussian elimination or even a back substitution in the allotted time. If the solution hasn’t changed too much, a few steps of a relatively cheap iterative method might keep sufficient accuracy as the solution moves through time.

The second major reason to use iterative methods is to solve sparse systems of equations. A coefficient matrix is called **sparse** if many of the matrix entries are known to be zero. Often, of the n^2 eligible entries in a sparse matrix, only $O(n)$ of them are nonzero. A **full** matrix is the opposite, where few entries may be assumed to be zero. Gaussian elimination applied to a sparse matrix usually causes **fill-in**, where the coefficient matrix changes from sparse to full due to the necessary row operations. For this reason, the efficiency of Gaussian elimination and its $PA = LU$ implementation become questionable for sparse matrices, leaving iterative methods as a feasible alternative.

Example 2.24 can be extended to a sparse matrix as follows:

► **EXAMPLE 2.25** Use the Jacobi Method to solve the 100,000-equation version of Example 2.24.

Let n be an even integer, and consider the $n \times n$ matrix A with 3 on the main diagonal, -1 on the super- and subdiagonal, and $1/2$ in the $(i, n+1-i)$ position for all $i = 1, \dots, n$, except for $i = n/2$ and $n/2 + 1$. For $n = 12$,

$$A = \begin{bmatrix} 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 3 & -1 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & -1 & 3 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -1 \\ \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 3 \end{bmatrix}. \quad (2.45)$$

Define the vector $b = (2.5, 1.5, \dots, 1.5, 1.0, 1.0, 1.5, \dots, 1.5, 2.5)$, where there are $n - 4$ repetitions of 1.5 and 2 repetitions of 1.0. Note that if $n = 6$, A and b define the system of Example 2.24. The solution of the system for general n is $[1, \dots, 1]$. No row of A has more than 4 nonzero entries. Since fewer than $4n$ of the n^2 potential entries are nonzero, we may call the matrix A sparse.

If we want to solve this system of equations for $n = 100,000$ or more, what are the options? Treating the coefficient matrix A as a full matrix means storing $n^2 = 10^{10}$ entries, each as a floating point double precision number requiring 8 bytes of storage. Note that 8×10^{10} bytes is approximately 80 gigabytes. Depending on your computational setup, it may be impossible to fit the entire n^2 entries into RAM.

Not only is size an enemy, but so is time. The number of operations required by Gaussian elimination will be on the order of $n^3 \approx 10^{15}$. If your machine runs on the order of a few GHz (10^9 cycles per second), an upper bound on the number of floating point operations per second is around 10^8 . Therefore, $10^{15}/10^8 = 10^7$ is a reasonable guess at the number of seconds required for Gaussian elimination. There are 3×10^7 seconds in a year. Although this is back-of-the-envelope accounting, it is clear that Gaussian elimination for this problem is not an overnight computation.

On the other hand, one step of an iterative method will require approximately $2 \times 4n = 800,000$ operations, two for each nonzero matrix entry. We could do 100 steps of Jacobi iteration and still finish with fewer than 10^8 operations, which should take roughly a second or less on a modern PC. For the system just defined, with $n = 100,000$, the following Jacobi code `jacobi.m` needs only 50 steps to converge from a starting guess of $(0, \dots, 0)$ to the solution $(1, \dots, 1)$ within six correct decimal places. The 50 steps require less than 1 second on a typical PC.

```
% Program 2.1 Sparse matrix setup
% Input: n = size of system
% Outputs: sparse matrix a, r.h.s. b
function [a,b] = sparsesetup(n)
e = ones(n,1); n2=n/2;
a = spdiags([-e 3*e -e],[-1:1,n,n]); % Entries of a
c=spdiags([e/2],0,n,n);c=fliplr(c);a=a+c;
a(n2+1,n2) = -1; a(n2,n2+1) = -1; % Fix up 2 entries
b=zeros(n,1); % Entries of r.h.s. b
b(1)=2.5;b(n)=2.5;b(2:n-1)=1.5;b(n2:n2+1)=1;
```

```


% Program 2.2 Jacobi Method
% Inputs: full or sparse matrix a, r.h.s. b,
%         number of Jacobi iterations, k
% Output: solution x
function x = jacobi(a,b,k)
n=length(b); % find n
d=diag(a); % extract diagonal of a
r=a-diag(d); % r is the remainder
x=zeros(n,1); % initialize vector x
for j=1:k % loop for Jacobi iteration
    x = (b-r*x)./d;
end % End of Jacobi iteration loop

```

Note a few interesting aspects of the preceding code. The program `sparsesetup.m` uses MATLAB's `spdiags` command, which defines the matrix A as a sparse data structure. Essentially, this means that the matrix is represented by a set of triples (i, j, d) , where d is the real number entry in position (i, j) of the matrix. Memory is not reserved for the entire n^2 potential entries, but only on an as-needed basis. The `spdiags` command takes the columns of a matrix and places them along the main diagonal, or a specified sub- or super-diagonal below or above the main diagonal.

MATLAB's matrix manipulation commands are designed to work seamlessly with the sparse matrix data structure. For example, an alternative to the preceding code would be to use MATLAB's `lu` command to solve the system directly. However, for that example, even though A is sparse, the upper-triangular matrix U that follows from Gaussian elimination suffers from fill-in during the process. For example, the upper-triangular U from Gaussian elimination for size $n = 12$ of the preceding matrix A is

$$\begin{bmatrix}
 3 & -1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.500 \\
 0 & 2.7 & -1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.500 & 0.165 \\
 0 & 0 & 2.6 & -1.0 & 0 & 0 & 0 & 0 & 0 & 0.500 & 0.187 & 0.062 \\
 0 & 0 & 0 & 2.6 & -1.000 & 0 & 0 & 0 & 0.500 & 0.191 & 0.071 & 0.024 \\
 0 & 0 & 0 & 0 & 2.618 & -1.000 & 0 & 0.500 & 0.191 & 0.073 & 0.027 & 0.009 \\
 0 & 0 & 0 & 0 & 0 & 2.618 & -1.000 & 0.191 & 0.073 & 0.028 & 0.010 & 0.004 \\
 0 & 0 & 0 & 0 & 0 & 0 & 2.618 & -0.927 & 0.028 & 0.011 & 0.004 & 0.001 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2.562 & -1.032 & -0.012 & -0.005 & -0.001 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2.473 & -1.047 & -0.018 & -0.006 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2.445 & -1.049 & -0.016 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2.440 & -1.044 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2.458
 \end{bmatrix}.$$

Since U turns out to be a relatively full matrix, the memory restrictions previously mentioned again become a limitation. A significant fraction of the n^2 memory locations will be necessary to store U on the way to completing the solution process. It is more efficient, by several orders of magnitude in execution time and storage, to solve this large sparse system by an iterative method. 

2.5 Exercises

1. Compute the first two steps of the Jacobi and the Gauss–Seidel Methods with starting vector $[0, \dots, 0]$.

$$\begin{aligned}
 \text{(a)} \quad & \begin{bmatrix} 3 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \end{bmatrix} & \text{(b)} \quad \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 0 \end{bmatrix} \\
 \text{(c)} \quad & \begin{bmatrix} 3 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 3 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 6 \\ 3 \\ 5 \end{bmatrix}
 \end{aligned}$$

2. Rearrange the equations to form a strictly diagonally dominant system. Apply two steps of the Jacobi and Gauss–Seidel Methods from starting vector $[0, \dots, 0]$.

$$\begin{array}{lll} \text{(a)} & \begin{array}{l} u + 3v = -1 \\ 5u + 4v = 6 \end{array} & \begin{array}{l} u - 8v - 2w = 1 \\ u + v + 5w = 4 \\ 3u - v + w = -2 \end{array} & \text{(c)} & \begin{array}{l} u + 4v = 5 \\ v + 2w = 2 \\ 4u + 3w = 0 \end{array} \end{array}$$

3. Apply two steps of SOR to the systems in Exercise 1. Use starting vector $[0, \dots, 0]$ and $\omega = 1.5$.
4. Apply two steps of SOR to the systems in Exercise 2 after rearranging. Use starting vector $[0, \dots, 0]$ and $\omega = 1$ and 1.2 .
5. Let λ be an eigenvalue of an $n \times n$ matrix A . (a) Prove the Gershgorin Circle Theorem: There is a diagonal entry A_{mm} such that $|A_{mm} - \lambda| \leq \sum_{j \neq m} |A_{mj}|$. (Hint: Begin with an eigenvector v such that $\|v\|_\infty = 1$, as in the proof of Theorem 2.10.) (b) Prove that a strictly diagonally dominant matrix cannot have a zero eigenvalue. This is an alternative proof of part (1) of Theorem 2.10.

2.5 Computer Problems

1. Use the Jacobi Method to solve the sparse system within six correct decimal places (forward error in the infinity norm) for $n = 100$ and $n = 100000$. The correct solution is $[1, \dots, 1]$. Report the number of steps needed and the backward error. The system is

$$\begin{bmatrix} 3 & -1 & & & \\ -1 & 3 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 3 & -1 \\ & & & -1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ \vdots \\ 1 \\ 2 \end{bmatrix}.$$

2. Use the Jacobi Method to solve the sparse system within three correct decimal places (forward error in the infinity norm) for $n = 100$. The correct solution is $[1, -1, 1, -1, \dots, 1, -1]$. Report the number of steps needed and the backward error. The system is

$$\begin{bmatrix} 2 & 1 & & & \\ 1 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 2 & 1 \\ & & & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ -1 \end{bmatrix}.$$

3. Rewrite Program 2.2 to carry out Gauss–Seidel iteration. Solve the problem in Example 2.24 to check your work.
4. Rewrite Program 2.2 to carry out SOR. Use $\omega = 1.1$ to recheck Example 2.24.
5. Carry out the steps of Computer Problem 1 with $n = 100$ for (a) Gauss–Seidel Method and (b) SOR with $\omega = 1.2$.
6. Carry out the steps of Computer Problem 2 for (a) Gauss–Seidel Method and (b) SOR with $\omega = 1.5$.

7. Using your program from Computer Problem 3, decide how large a system of type (2.38) you can solve accurately by the Gauss–Seidel Method in one second of computation. Report the time required and forward error for various values of n .

2.6 METHODS FOR SYMMETRIC POSITIVE-DEFINITE MATRICES

Symmetric matrices hold a favored position in linear systems analysis because of their special structure, and because they have only about half as many independent entries as general matrices. That raises the question whether a factorization like the LU can be realized for half the computational complexity, and using only half the memory locations. For symmetric positive-definite matrices, this goal can be achieved with the Cholesky factorization.

Symmetric positive-definite matrices also allow a quite different approach to solving $Ax = b$, one that does not depend on a matrix factorization. This new approach, called the conjugate gradient method, is especially useful for large, sparse matrices, where it falls into the family of iterative methods.

To begin the section, we define the concept of positive-definiteness for symmetric matrices. Then we show that every symmetric positive-definite matrix A can be factored as $A = R^T R$ for an upper-triangular matrix R , the Cholesky factorization. As a result, the problem $Ax = b$ can be solved using two back substitutions, just as with the LU factorization in the nonsymmetric case. We close the section with the conjugate gradient algorithm and an introduction to preconditioning.

2.6.1 Symmetric positive-definite matrices

DEFINITION 2.12 The $n \times n$ matrix A is **symmetric** if $A^T = A$. The matrix A is **positive-definite** if $x^T Ax > 0$ for all vectors $x \neq 0$. \square

► **EXAMPLE 2.26** Show that the matrix $A = \begin{bmatrix} 2 & 2 \\ 2 & 5 \end{bmatrix}$ is symmetric positive-definite.

Clearly A is symmetric. To show it is positive-definite, one applies the definition:

$$\begin{aligned} x^T Ax &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= 2x_1^2 + 4x_1x_2 + 5x_2^2 \\ &= 2(x_1 + x_2)^2 + 3x_2^2 \end{aligned}$$

This expression is always non-negative, and cannot be zero unless both $x_2 = 0$ and $x_1 + x_2 = 0$, which together imply $x = 0$. \blacktriangleleft

► **EXAMPLE 2.27** Show that the symmetric matrix $A = \begin{bmatrix} 2 & 4 \\ 4 & 5 \end{bmatrix}$ is not positive-definite.

Compute $x^T Ax$ by completing the square:

$$\begin{aligned} x^T Ax &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= 2x_1^2 + 8x_1x_2 + 5x_2^2 \\ &= 2(x_1^2 + 4x_1x_2) + 5x_2^2 \\ &= 2(x_1 + 2x_2)^2 - 8x_2^2 + 5x_2^2 \\ &= 2(x_1 + 2x_2)^2 - 3x_2^2 \end{aligned}$$

Setting $x_1 = -2$ and $x_2 = 1$, for example, causes the result to be less than zero, contradicting the definition of positive-definite. ◀

Note that a symmetric positive-definite matrix must be nonsingular, since it is impossible for a nonzero vector x to satisfy $Ax = 0$. There are three additional important facts about this class of matrices.

Property 1 If the $n \times n$ matrix A is symmetric, then A is positive-definite if and only if all of its eigenvalues are positive.

Proof. Theorem A.5 says that, the set of unit eigenvectors is orthonormal and spans \mathbb{R}^n . If A is positive-definite and $Av = \lambda v$ for a nonzero vector v , then $0 < v^T Av = v^T(\lambda v) = \lambda \|v\|_2^2$, so $\lambda > 0$. On the other hand, if all eigenvalues of A are positive, then write any nonzero $x = c_1 v_1 + \dots + c_n v_n$ where the v_i are orthonormal unit vectors and not all c_i are zero. Then $x^T Ax = (c_1 v_1 + \dots + c_n v_n)^T (\lambda_1 c_1 v_1 + \dots + \lambda_n c_n v_n) = \lambda_1 c_1^2 + \dots + \lambda_n c_n^2 > 0$, so A is positive-definite. ◻

The eigenvalues of A in Example 2.26 are 6 and 1. The eigenvalues of A in Example 2.27 are approximately 7.77 and -0.77 .

Property 2 If A is $n \times n$ symmetric positive-definite and X is an $n \times m$ matrix of full rank with $n \geq m$, then $X^T A X$ is $m \times m$ symmetric positive-definite.

Proof. The matrix is symmetric since $(X^T A X)^T = X^T A X$. To prove positive-definite, consider a nonzero m -vector v . Note that $v^T (X^T A X) v = (Xv)^T A (Xv) \geq 0$, with equality only if $Xv = 0$, due to the positive-definiteness of A . Since X has full rank, its columns are linearly independent, so that $Xv = 0$ implies $v = 0$. ◻

DEFINITION 2.13 A **principal** submatrix of a square matrix A is a square submatrix whose diagonal entries are diagonal entries of A . ◻

Property 3 Any principal submatrix of a symmetric positive-definite matrix is symmetric positive-definite.

Proof. Exercise 12. ◻

For example, if

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

is symmetric positive-definite, then so is

$$\begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix}.$$

2.6.2 Cholesky factorization

To demonstrate the main idea, we start with a 2×2 case. All of the important issues arise there; the extension to the general size is only some extra bookkeeping.

Consider the symmetric positive-definite matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}.$$

By Property 3 of symmetric positive-definite matrices, we know that $a > 0$. In addition, we know that the determinant $ac - b^2$ of A is positive, since the determinant is the product of the eigenvalues, all positive by Property 1. Writing $A = R^T R$ with an upper triangular R implies the form

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sqrt{a} & 0 \\ u & v \end{bmatrix} \begin{bmatrix} \sqrt{a} & u \\ 0 & v \end{bmatrix} = \begin{bmatrix} a & u\sqrt{a} \\ u\sqrt{a} & u^2 + v^2 \end{bmatrix},$$

and we want to check whether this is possible. Comparing left and right sides yields the identities $u = b/\sqrt{a}$ and $v^2 = c - u^2$. Note that $v^2 = c - (b/\sqrt{a})^2 = c - b^2/a > 0$ from our knowledge of the determinant. This verifies that v can be defined as a real number and so the Cholesky factorization

$$A = \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} \sqrt{a} & 0 \\ \frac{b}{\sqrt{a}} & \sqrt{c - b^2/a} \end{bmatrix} \begin{bmatrix} \sqrt{a} & \frac{b}{\sqrt{a}} \\ 0 & \sqrt{c - b^2/a} \end{bmatrix} = R^T R$$

exists for 2×2 symmetric positive-definite matrices. The Cholesky factorization is not unique; clearly we could just as well have chosen v to be the negative square root of $c - b^2/a$.

The next result guarantees that the same idea works for the $n \times n$ case.

THEOREM 2.14 (Cholesky Factorization Theorem) If A is a symmetric positive-definite $n \times n$ matrix, then there exists an upper triangular $n \times n$ matrix R such that $A = R^T R$. ■

Proof. We construct R by induction on the size n . The case $n = 2$ was done above. Consider A partitioned as

$$A = \left[\begin{array}{c|c} a & b^T \\ \hline b & C \end{array} \right]$$

where b is an $(n - 1)$ -vector and C is an $(n - 1) \times (n - 1)$ submatrix. We will use block multiplication (see the Appendix section A.2) to simplify the argument. Set $u = b/\sqrt{a}$ as in the 2×2 case. Setting $A_1 = C - uu^T$ and defining the invertible matrix

$$S = \left[\begin{array}{c|c} \sqrt{a} & u^T \\ \hline 0 & I \end{array} \right]$$

yields

$$\begin{aligned}
 S^T \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{bmatrix} S &= \begin{bmatrix} \sqrt{a} & 0 & \cdots & 0 \\ u & & & I \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{bmatrix} \begin{bmatrix} \sqrt{a} & u^T \\ 0 & \\ \vdots & I \\ 0 & \end{bmatrix} \\
 &= \begin{bmatrix} a & b^T \\ b & uu^T + A_1 \end{bmatrix} = A
 \end{aligned}$$

Notice that A_1 is symmetric positive-definite. This follows from the facts that

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & A_1 & \\ 0 & & & \end{bmatrix} = (S^T)^{-1} A S^{-1}$$

is symmetric positive-definite by Property 2, and therefore so is the $(n-1) \times (n-1)$ principal submatrix A_1 by Property 3. By the induction hypothesis, $A_1 = V^T V$ where V is upper triangular. Finally, define the upper triangular matrix

$$R = \begin{bmatrix} \sqrt{a} & u^T \\ 0 & \\ \vdots & V \\ 0 & \end{bmatrix}$$

and check that

$$R^T R = \begin{bmatrix} \sqrt{a} & 0 & \cdots & 0 \\ u & & & V^T \end{bmatrix} \begin{bmatrix} \sqrt{a} & u^T \\ 0 & \\ \vdots & V \\ 0 & \end{bmatrix} = \begin{bmatrix} a & b^T \\ b & uu^T + V^T V \end{bmatrix} = A,$$

which completes the proof. \square

The construction of the proof can be carried out explicitly, in what has become the standard algorithm for the Cholesky factorization. The matrix R is built from the outside in. First we find $r_{11} = \sqrt{a_{11}}$ and set the rest of the top row of R to $u^T = b^T/r_{11}$. Then uu^T is subtracted from the lower principal $(n-1) \times (n-1)$ submatrix, and the same steps are repeated on it to fill in the second row of R . These steps are continued until all rows of R are determined. According to the theorem, the new principal submatrix is positive-definite at every stage of the construction, so by Property 3, the top left corner entry is positive, and the square root operation succeeds. This approach can be put directly into the following algorithm. We use the “colon notation” where convenient to denote submatrices.

Cholesky factorization

```

for  $k = 1, 2, \dots, n$ 
  if  $A_{kk} < 0$ , stop, end
   $R_{kk} = \sqrt{A_{kk}}$ 
   $u^T = \frac{1}{R_{kk}} A_{k,k+1:n}$ 
   $R_{k,k+1:n} = u^T$ 
   $A_{k+1:n,k+1:n} = A_{k+1:n,k+1:n} - uu^T$ 
end

```

The resulting R is upper triangular and satisfies $A = R^T R$.

► **EXAMPLE 2.28** Find the Cholesky factorization of $\begin{bmatrix} 4 & -2 & 2 \\ -2 & 2 & -4 \\ 2 & -4 & 11 \end{bmatrix}$.

The top row of R is $R_{11} = \sqrt{a_{11}} = 2$, followed by $R_{1,2:3} = [-2, 2]/R_{11} = [-1, 1]$:

$$R = \begin{bmatrix} 2 & -1 & 1 \\ \vdots & & \end{bmatrix}.$$

Subtracting the outer product $uu^T = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 1 \end{bmatrix}$ from the lower principal 2×2 submatrix $A_{2:3,2:3}$ of A leaves

$$\begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & 2 & -4 \\ \vdots & -4 & 11 \end{bmatrix} - \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & 1 & -1 \\ \vdots & -1 & 1 \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & 1 & -3 \\ \vdots & -3 & 10 \end{bmatrix}.$$

Now we repeat the same steps on the 2×2 submatrix to find $R_{22} = 1$ and $R_{23} = -3/1 = -3$:

$$R = \begin{bmatrix} 2 & -1 & 1 \\ \vdots & 1 & -3 \\ \vdots & \vdots & \vdots \end{bmatrix}.$$

The lower 1×1 principal submatrix of A is $10 - (-3)(-3) = 1$, so $R_{33} = \sqrt{1}$. The Cholesky factor of A is

$$R = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix}.$$

Solving $Ax = b$ for symmetric positive-definite A follows the same idea as the LU factorization. Now that $A = R^T R$ is a product of two triangular matrices, we need to solve the lower triangular system $R^T c = b$ and the upper triangular system $Rx = c$ to determine the solution x .

2.6.3 Conjugate Gradient Method

The introduction of the Conjugate Gradient Method (Hestenes and Steifel, 1952) ushered in a new era for iterative methods to solve sparse matrix problems. Although the method was slow to catch on, once effective preconditioners were developed, huge problems that could not be attacked any other way became feasible. The achievement led shortly to much further progress and a new generation of iterative solvers.

SPOTLIGHT ON

Orthogonality

Our first real application of orthogonality in this book uses it in a roundabout way, to solve a problem that has no obvious link to orthogonality. The Conjugate Gradient Method tracks down the solution of a positive-definite $n \times n$ linear system by successively locating and eliminating the n orthogonal components of the error, one by one. The complexity of the algorithm is minimized by using the directions established by pairwise orthogonal residual vectors. We will develop this point of view further in Chapter 4, culminating in the GMRES method, a nonsymmetric counterpart to conjugate gradients.

The ideas behind conjugate gradients rely on the generalization of the usual idea of inner product. The Euclidean inner product $(v, w) = v^T w$ is symmetric and linear in the inputs v and w , since $(v, w) = (w, v)$ and $(\alpha v + \beta w, u) = \alpha(v, u) + \beta(w, u)$ for scalars α and β . The Euclidean inner product is also positive-definite, in that $(v, v) > 0$ if $v \neq 0$.

DEFINITION 2.15 Let A be a symmetric positive-definite $n \times n$ matrix. For two n -vectors v and w , define the **A -inner product**

$$(v, w)_A = v^T A w.$$

The vectors v and w are **A -conjugate** if $(v, w)_A = 0$. □

Note that the new inner product inherits the properties of symmetry, linearity, and positive-definiteness from the matrix A . Because A is symmetric, so is the A -inner product: $(v, w)_A = v^T A w = (v^T A w)^T = w^T A v = (w, v)_A$. The A -inner product is also linear, and positive-definiteness follows from the fact that if A is positive-definite, then

$$(v, v)_A = v^T A v > 0$$

if $v \neq 0$.

Strictly speaking, the Conjugate Gradient Method is a direct method, and arrives at the solution x of the symmetric positive-definite system $Ax = b$ with the following finite loop:

Conjugate Gradient Method

```

 $x_0$  = initial guess
 $d_0 = r_0 = b - Ax_0$ 
for  $k = 0, 1, 2, \dots, n - 1$ 
    if  $r_k = 0$ , stop, end
     $\alpha_k = \frac{r_k^T r_k}{d_k^T A d_k}$ 
     $x_{k+1} = x_k + \alpha_k d_k$ 
     $r_{k+1} = r_k - \alpha_k A d_k$ 
     $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$ 
     $d_{k+1} = r_{k+1} + \beta_k d_k$ 
end

```

An informal description of the iteration is next, to be followed by proof of the necessary facts in Theorem 2.16. The conjugate gradient iteration updates three different vectors on each step. The vector x_k is the approximate solution at step k . The vector r_k represents the

residual of the approximate solution x_k . This is clear for r_0 by definition, and during the iteration, notice that

$$\begin{aligned} Ax_{k+1} + r_{k+1} &= A(x_k + \alpha_k d_k) + r_k - \alpha_k A d_k \\ &= Ax_k + r_k, \end{aligned}$$

and so by induction $r_k = b - Ax_k$ for all k . Finally, the vector d_k represents the new search direction used to update the approximation x_k to the improved version x_{k+1} .

The method succeeds because each residual is arranged to be orthogonal to all previous residuals. If this can be done, the method runs out of orthogonal directions in which to look, and must reach a zero residual and a correct solution in at most n steps. The key to accomplishing the orthogonality among residuals turns out to be choosing the search directions d_k pairwise conjugate. The concept of conjugacy generalizes orthogonality and gives its name to the algorithm.

Now we explain the choices of α_k and β_k . The directions d_k are chosen from the vector space span of the previous residuals, as seen inductively from the last line of the pseudocode. In order to ensure that the next residual is orthogonal to all past residuals, α_k is chosen precisely so that the new residual r_{k+1} is orthogonal to the direction d_k :

$$\begin{aligned} x_{k+1} &= x_k + \alpha_k d_k \\ b - Ax_{k+1} &= b - Ax_k - \alpha_k A d_k \\ r_{k+1} &= r_k - \alpha_k A d_k \\ 0 &= d_k^T r_{k+1} = d_k^T r_k - \alpha_k d_k^T A d_k \\ \alpha_k &= \frac{d_k^T r_k}{d_k^T A d_k}. \end{aligned}$$

This is not exactly how α_k is written in the algorithm, but note that since d_{k-1} is orthogonal to r_k , we have

$$\begin{aligned} d_k - r_k &= \beta_{k-1} d_{k-1} \\ r_k^T d_k - r_k^T r_k &= 0, \end{aligned}$$

which justifies the rewriting $r_k^T d_k = r_k^T r_k$. Secondly, the coefficient β_k is chosen to ensure the pairwise A -conjugacy of the d_k :

$$\begin{aligned} d_{k+1} &= r_{k+1} + \beta_k d_k \\ 0 &= d_k^T A d_{k+1} = d_k^T A r_{k+1} + \beta_k d_k^T A d_k \\ \beta_k &= -\frac{d_k^T A r_{k+1}}{d_k^T A d_k}. \end{aligned}$$

The expression for β_k can be rewritten in the simpler form seen in the algorithm, as shown in (2.47) below.

Theorem 2.16 below verifies that all r_k produced by the conjugate gradient iteration are orthogonal to one another. Since they are n -dimensional vectors, at most n of the r_k can be pairwise orthogonal, so either r_n or a previous r_k must be zero, solving $Ax = b$. Therefore after at most n steps, conjugate gradient arrives at a solution. In theory, the method is a direct, not an iterative, method.

Before turning to the theorem that guarantees the success of the Conjugate Gradient Method, it is instructive to carry out an example in exact arithmetic.

► **EXAMPLE 2.29** Solve $\begin{bmatrix} 2 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 6 \\ 3 \end{bmatrix}$ using the Conjugate Gradient Method.

Following the above algorithm we have

$$\begin{aligned}
 x_0 &= \begin{bmatrix} 0 \\ 0 \end{bmatrix}, r_0 = d_0 = \begin{bmatrix} 6 \\ 3 \end{bmatrix} \\
 \alpha_0 &= \frac{\begin{bmatrix} 6 \\ 3 \end{bmatrix}^T \begin{bmatrix} 6 \\ 3 \end{bmatrix}}{\begin{bmatrix} 6 \\ 3 \end{bmatrix}^T \begin{bmatrix} 2 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} 6 \\ 3 \end{bmatrix}} = \frac{45}{6 \cdot 18 + 3 \cdot 27} = \frac{5}{21} \\
 x_1 &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \frac{5}{21} \begin{bmatrix} 6 \\ 3 \end{bmatrix} = \begin{bmatrix} 10/7 \\ 5/7 \end{bmatrix} \\
 r_1 &= \begin{bmatrix} 6 \\ 3 \end{bmatrix} - \frac{5}{21} \begin{bmatrix} 18 \\ 27 \end{bmatrix} = 12 \begin{bmatrix} 1/7 \\ -2/7 \end{bmatrix} \\
 \beta_0 &= \frac{r_1^T r_1}{r_0^T r_0} = \frac{144 \cdot 5/49}{36 + 9} = \frac{16}{49} \\
 d_1 &= 12 \begin{bmatrix} 1/7 \\ -2/7 \end{bmatrix} + \frac{16}{49} \begin{bmatrix} 6 \\ 3 \end{bmatrix} = \begin{bmatrix} 180/49 \\ -120/49 \end{bmatrix} \\
 \alpha_1 &= \frac{\begin{bmatrix} 12/7 \\ -24/7 \end{bmatrix}^T \begin{bmatrix} 12/7 \\ -24/7 \end{bmatrix}}{\begin{bmatrix} 180/49 \\ -120/49 \end{bmatrix}^T \begin{bmatrix} 2 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} 180/49 \\ -120/49 \end{bmatrix}} = \frac{7}{10} \\
 x_2 &= \begin{bmatrix} 10/7 \\ 5/7 \end{bmatrix} + \frac{7}{10} \begin{bmatrix} 180/49 \\ -120/49 \end{bmatrix} = \begin{bmatrix} 4 \\ -1 \end{bmatrix} \\
 r_2 &= 12 \begin{bmatrix} 1/7 \\ -2/7 \end{bmatrix} - \frac{7}{10} \begin{bmatrix} 2 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} 180/49 \\ -120/49 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}
 \end{aligned}$$

Since $r_2 = b - Ax_2 = 0$, the solution is $x_2 = [4, -1]$. ◀

THEOREM 2.16 Let A be a symmetric positive-definite $n \times n$ matrix and let $b \neq 0$ be a vector. In the Conjugate Gradient Method, assume that $r_k \neq 0$ for $k < n$ (if $r_k = 0$ the equation is solved). Then for each $1 \leq k \leq n$,

(a) The following three subspaces of R^n are equal:

$$\langle x_1, \dots, x_k \rangle = \langle r_0, \dots, r_{k-1} \rangle = \langle d_0, \dots, d_{k-1} \rangle,$$

(b) the residuals r_k are pairwise orthogonal: $r_k^T r_j = 0$ for $j < k$,

(c) the directions d_k are pairwise A -conjugate: $d_k^T A d_j = 0$ for $j < k$. ■

Proof. (a) For $k = 1$, note that $\langle x_1 \rangle = \langle d_0 \rangle = \langle r_0 \rangle$, since $x_0 = 0$. By definition $x_k = x_{k-1} + \alpha_{k-1} d_{k-1}$. This implies by induction that $\langle x_1, \dots, x_k \rangle = \langle d_0, \dots, d_{k-1} \rangle$. A similar argument using $d_k = r_k + \beta_{k-1} d_{k-1}$ shows that $\langle r_0, \dots, r_{k-1} \rangle$ is equal to $\langle d_0, \dots, d_{k-1} \rangle$.

For (b) and (c), proceed by induction. When $k = 0$ there is nothing to prove. Assume (b) and (c) hold for k , and we will prove (b) and (c) for $k + 1$. Multiply the definition of r_{k+1} by r_j^T on the left:

$$r_j^T r_{k+1} = r_j^T r_k - \frac{r_k^T r_k}{d_k^T A d_k} r_j^T A d_k. \quad (2.46)$$

If $j \leq k - 1$, then $r_j^T r_k = 0$ by the induction hypothesis (b). Since r_j can be expressed as a combination of d_0, \dots, d_j , the term $r_j^T A d_k = 0$ from the induction hypothesis (c),

and (b) holds. On the other hand, if $j = k$, then $r_k^T r_{k+1} = 0$ again follows from (2.46) because $d_k^T A d_k = r_k^T A d_k + \beta_{k-1} d_{k-1}^T A d_k = r_k^T A d_k$, using the induction hypothesis (c). This proves (b).

Now that $r_k^T r_{k+1} = 0$, (2.46) with $j = k + 1$ says

$$\frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} = -\frac{r_{k+1}^T A d_k}{d_k^T A d_k}. \quad (2.47)$$

This together with multiplying the definition of d_{k+1} on the left by $d_j^T A$ yields

$$d_j^T A d_{k+1} = d_j^T A r_{k+1} - \frac{r_{k+1}^T A d_k}{d_k^T A d_k} d_j^T A d_k. \quad (2.48)$$

If $j = k$, then $d_k^T A d_{k+1} = 0$ from (2.48), using the symmetry of A . If $j \leq k - 1$, then $A d_j = (r_j - r_{j+1})/\alpha_j$ (from the definition of r_{j+1}) is orthogonal to r_{k+1} , showing the first term on the right-hand side of (2.48) is zero, and the second term is zero by the induction hypothesis, which completes the argument for (c). \square

In Example 2.29, notice that r_1 is orthogonal to r_0 , as guaranteed by Theorem 2.16. This fact is the key to success for the Conjugate Gradient Method: Each new residual r_i is orthogonal to all previous r_i 's. If one of the r_i turns out to be zero, then $A x_i = b$ and x_i is the solution. If not, after n steps through the loop, r_n is orthogonal to a space spanned by the n pairwise orthogonal vectors r_0, \dots, r_{n-1} , which must be all of R^n . So r_n must be the zero vector, and $A x_n = b$.

The Conjugate Gradient Method is in some ways simpler than Gaussian elimination. For example, writing the code appears to be more foolproof—there are no row operations to worry about, and there is no triple loop as in Gaussian elimination. Both are direct methods, and they both arrive at the theoretically correct solution in a finite number of steps. So two questions remain: Why shouldn't conjugate gradient be preferred to Gaussian elimination, and why is Conjugate Gradient often treated as an iterative method?

The answer to both questions begins with an operation count. Moving through the loop requires one matrix-vector product $A d_{n-1}$ and several additional dot products. The matrix-vector product alone requires n^2 multiplications for each step (along with about the same number of additions), for a total of n^3 multiplications after n steps. Compared to the count of $n^3/3$ for Gaussian elimination, this is three times too expensive.

The picture changes if A is sparse. Assume that n is too large for the $n^3/3$ operations of Gaussian elimination to be feasible. Although Gaussian elimination must be run to completion to give a solution x , Conjugate Gradient gives an approximation x_i on each step.

The backward error, the Euclidean length of the residual, decreases on each step, and so at least by that measure, $A x_i$ is getting nearer to b on each step. Therefore by monitoring the r_i , a good enough solution x_i may be found to avoid completing all n steps. In this context, Conjugate Gradient becomes indistinguishable from an iterative method.

The method fell out of favor shortly after its discovery because of its susceptibility to accumulation of round-off errors when A is an ill-conditioned matrix. In fact, its performance on ill-conditioned matrices is inferior to Gaussian elimination with partial pivoting. In modern days, this obstruction is relieved by **preconditioning**, which essentially changes the problem to a better-conditioned matrix system, after which Conjugate Gradient is applied. We will investigate the preconditioned Conjugate Gradient Method in the next section.

The title of the method comes from what the Conjugate Gradient Method is really doing: sliding down the slopes of a quadratic paraboloid in n dimensions. The “gradient” part of

the title means it is finding the direction of fastest decline using calculus, and “conjugate” means not quite that its individual steps are orthogonal to one another, but that at least the residuals r_i are. The geometric details of the method and its motivation are interesting. The original article Hestenes and Steifel [1952] gives a complete description.

► **EXAMPLE 2.30** Apply the Conjugate Gradient Method to system (2.45) with $n = 100,000$.

After 20 steps of the Conjugate Gradient Method, the difference between the computed solution x and the true solution $(1, \dots, 1)$ is less than 10^{-9} in the vector infinity norm. The total time of execution was less than one second on a PC. ◀

2.6.4 Preconditioning

Convergence of iterative methods like the Conjugate Gradient Method can be accelerated by the use of a technique called preconditioning. The convergence rates of iterative methods often depend, directly or indirectly, on the condition number of the coefficient matrix A . The idea of preconditioning is to reduce the effective condition number of the problem.

The preconditioned form of the $n \times n$ linear system $Ax = b$ is

$$M^{-1}Ax = M^{-1}b,$$

where M is an invertible $n \times n$ matrix called the **preconditioner**. All we have done is to left-multiply the equation by a matrix. An effective preconditioner reduces the condition number of the problem by attempting to invert A . Conceptually, it tries to do two things at once: the matrix M should be (1) as close to A as possible and (2) simple to invert. These two goals usually stand in opposition to one another.

The matrix closest to A is A itself. Using $M = A$ would bring the condition number of the problem to 1, but presumably A is not trivial to invert or we would not be using a sophisticated solution method. The easiest matrix to invert is the identity matrix $M = I$, but this does not reduce the condition number. The perfect preconditioner would be a matrix in the middle of the two extremes that combines the best properties of both.

A particularly simple choice is the **Jacobi preconditioner** $M = D$, where D is the diagonal of A . The inverse of D is the diagonal matrix of reciprocals of the entries of D . In a strictly diagonally dominant matrix, for example, the Jacobi preconditioner holds a close resemblance to A while being simple to invert. Note that each diagonal entry of a symmetric positive-definite matrix is strictly positive by Property 3 of section 2.6.1, so finding reciprocals is not a problem.

When A is a symmetric positive-definite $n \times n$ matrix, we will choose a symmetric positive-definite matrix M for use as a preconditioner. Recall the M -inner product $(v, w)_M = v^T M w$ as defined in Section 2.6.3. The Preconditioned Conjugate Gradient Method is now easy to describe: Replace $Ax = b$ with the preconditioned equation $M^{-1}Ax = M^{-1}b$, and replace the Euclidean inner product with $(v, w)_M$. The reasoning used for the original conjugate gradient method still applies because the matrix $M^{-1}A$ remains symmetric positive-definite in the new inner product.

For example,

$$(M^{-1}Av, w)_M = v^T A M^{-1} M w = v^T A w = v^T M M^{-1} A w = (v, M^{-1}Aw)_M.$$

To convert the algorithm from Section 2.6.3 to the preconditioned version, let $z_k = M^{-1}b - M^{-1}Ax_k = M^{-1}r_k$ be the residual of the preconditioned system. Then

$$\begin{aligned}
\alpha_k &= \frac{(z_k, z_k)_M}{(d_k, M^{-1} A d_k)_M} \\
x_{k+1} &= x_k + \alpha d_k \\
z_{k+1} &= z_k - \alpha M^{-1} A d_k \\
\beta_k &= \frac{(z_{k+1}, z_{k+1})_M}{(z_k, z_k)_M} \\
d_{k+1} &= z_{k+1} + \beta_k d_k.
\end{aligned}$$

Multiplications by M can be reduced by noting that

$$\begin{aligned}
(z_k, z_k)_M &= z_k^T M z_k = z_k^T r_k \\
(d_k, M^{-1} A d_k)_M &= d_k^T A d_k \\
(z_{k+1}, z_{k+1})_M &= z_{k+1}^T M z_{k+1} = z_{k+1}^T r_{k+1}.
\end{aligned}$$

With these simplifications, the pseudocode for the preconditioned version goes as follows.

Preconditioned Conjugate Gradient Method

```

 $x_0$  = initial guess
 $r_0 = b - Ax_0$ 
 $d_0 = z_0 = M^{-1} r_0$ 
for  $k = 0, 1, 2, \dots, n - 1$ 
  if  $r_k = 0$ , stop, end
   $\alpha_k = r_k^T z_k / d_k^T A d_k$ 
   $x_{k+1} = x_k + \alpha_k d_k$ 
   $r_{k+1} = r_k - \alpha_k A d_k$ 
   $z_{k+1} = M^{-1} r_{k+1}$ 
   $\beta_k = r_{k+1}^T z_{k+1} / r_k^T z_k$ 
   $d_{k+1} = z_{k+1} + \beta_k d_k$ 
end

```

The approximation to the solution of $Ax = b$ after k steps is x_k . Note that no explicit multiplications by M^{-1} should be carried out. They should be replaced with appropriate back substitutions due to the relative simplicity of M .

The Jacobi preconditioner is the simplest of an extensive and growing library of possible choices. We will describe one further family of examples, and direct the reader to the literature for more sophisticated alternatives.

The **symmetric successive over-relaxation (SSOR)** preconditioner is defined by

$$M = (D + \omega L) D^{-1} (D + \omega U)$$

where $A = L + D + U$ is divided into its lower triangular part, diagonal, and upper triangular part. As in the SOR method, ω is a constant between 0 and 2. The special case $\omega = 1$ is called the **Gauss–Seidel preconditioner**.

A preconditioner is of little use if it is difficult to invert. Notice that the SSOR preconditioner is defined as a product $M = (I + \omega L D^{-1})(D + \omega U)$ of a lower triangular and an upper triangular matrix, so that the equation $z = M^{-1}v$ can be solved by two back substitutions:

$$\begin{aligned}
(I + \omega L D^{-1})c &= v \\
(D + \omega U)z &= c
\end{aligned}$$

For a sparse matrix, the two back substitutions can be done in time proportional to the number of nonzero entries. In other words, multiplication by M^{-1} is not significantly higher in complexity than multiplication by M .

► **EXAMPLE 2.31** Let A denote the matrix with diagonal entries $A_{ii} = \sqrt{i}$ for $i = 1, \dots, n$ and $A_{i,i+10} = A_{i+10,i} = \cos i$ for $i = 1, \dots, n - 10$, with all other entries zero. Set x to be the vector of n ones, and define $b = Ax$. For $n = 500$, solve $Ax = b$ with the Conjugate Gradient Method in three ways: using no preconditioner, using the Jacobi preconditioner, and using the Gauss–Seidel preconditioner.

The matrix can be defined in MATLAB by

```
A=diag(sqrt(1:n))+ diag(cos(1:(n-10)),10)
      + diag(cos(1:(n-10)),-10).
```

Figure 2.4 shows the three different results. Even with this simply defined matrix, the Conjugate Gradient Method is fairly slow to converge without preconditioning. The Jacobi preconditioner, which is quite easy to apply, makes a significant improvement, while the Gauss–Seidel preconditioner requires only about 10 steps to reach machine accuracy. ◀

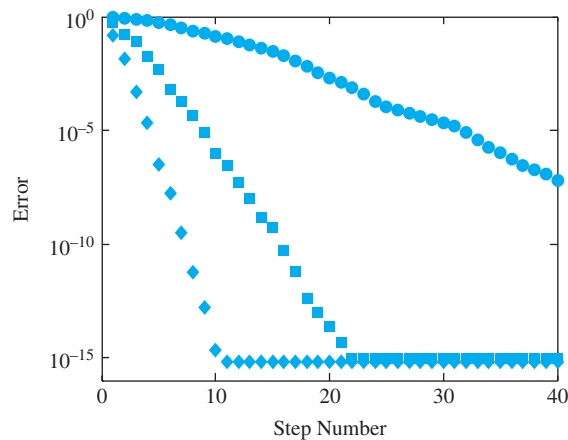


Figure 2.4 Efficiency of Preconditioned Conjugate Gradient Method for the solution of Example 2.31. Error is plotted by step number. Circles: no preconditioner. Squares: Jacobi preconditioner. Diamonds: Gauss–Seidel preconditioner.

2.6 Exercises

1. Show that the following matrices are symmetric positive-definite by expressing $x^T Ax$ as a sum of squares.

$$(a) \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 3 \\ 3 & 10 \end{bmatrix} \quad (c) \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

2. Show that the following symmetric matrices are not positive-definite by finding a vector $x \neq 0$ such that $x^T Ax < 0$.

$$(a) \begin{bmatrix} 1 & 0 \\ 0 & -3 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix} \quad (c) \begin{bmatrix} 1 & -1 \\ -1 & 0 \end{bmatrix} \quad (d) \begin{bmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

3. Use the Cholesky factorization procedure to express the matrices in Exercise 1 in the form $A = R^T R$.
4. Show that the Cholesky factorization procedure fails for the matrices in Exercise 2.
5. Find the Cholesky factorization $A = R^T R$ of each matrix.

$$(a) \begin{bmatrix} 1 & 2 \\ 2 & 8 \end{bmatrix} (b) \begin{bmatrix} 4 & -2 \\ -2 & 5/4 \end{bmatrix} (c) \begin{bmatrix} 25 & 5 \\ 5 & 26 \end{bmatrix} (d) \begin{bmatrix} 1 & -2 \\ -2 & 5 \end{bmatrix}$$

6. Find the Cholesky factorization $A = R^T R$ of each matrix.

$$(a) \begin{bmatrix} 4 & -2 & 0 \\ -2 & 2 & -3 \\ 0 & -3 & 10 \end{bmatrix} (b) \begin{bmatrix} 1 & 2 & 0 \\ 2 & 5 & 2 \\ 0 & 2 & 5 \end{bmatrix} (c) \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} (d) \begin{bmatrix} 1 & -1 & -1 \\ -1 & 2 & 1 \\ -1 & 1 & 2 \end{bmatrix}$$

7. Solve the system of equations by finding the Cholesky factorization of A followed by two back substitutions.

$$(a) \begin{bmatrix} 1 & -1 \\ -1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ -7 \end{bmatrix} (b) \begin{bmatrix} 4 & -2 \\ -2 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \end{bmatrix}$$

8. Solve the system of equations by finding the Cholesky factorization of A followed by two back substitutions.

$$(a) \begin{bmatrix} 4 & 0 & -2 \\ 0 & 1 & 1 \\ -2 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 0 \end{bmatrix} (b) \begin{bmatrix} 4 & -2 & 0 \\ -2 & 2 & -1 \\ 0 & -1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ -7 \end{bmatrix}$$

9. Prove that if $d > 4$, the matrix $A = \begin{bmatrix} 1 & 2 \\ 2 & d \end{bmatrix}$ is positive-definite.
10. Find all numbers d such that $A = \begin{bmatrix} 1 & -2 \\ -2 & d \end{bmatrix}$ is positive-definite.
11. Find all numbers d such that $A = \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & 1 \\ 0 & 1 & d \end{bmatrix}$ is positive-definite.
12. Prove that a principal submatrix of a symmetric positive-definite matrix is symmetric positive-definite. (Hint: Consider an appropriate X and use Property 2.)
13. Solve the problems by carrying out the Conjugate Gradient Method by hand.

$$(a) \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} (b) \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

14. Solve the problems by carrying out the Conjugate Gradient Method by hand.

$$(a) \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} (b) \begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} -3 \\ 3 \end{bmatrix}$$

15. Carry out the conjugate gradient iteration in the general scalar case $Ax = b$ where A is a 1×1 matrix. Find α_1, x_1 , and confirm that $r_1 = 0$ and $Ax_1 = b$.

2.6 Computer Problems

1. Write a MATLAB version of the Conjugate Gradient Method and use it to solve the systems

$$(a) \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

2. Use a MATLAB version of conjugate gradient to solve the following problems:

$$(a) \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 3 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & 1 \\ 0 & 1 & 5 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ 4 \end{bmatrix}$$

3. Solve the system $Hx = b$ by the Conjugate Gradient Method, where H is the $n \times n$ Hilbert matrix and b is the vector of all ones, for (a) $n = 4$ (b) $n = 8$.
4. Solve the sparse problem of (2.45) by the Conjugate Gradient Method for (a) $n = 6$ (b) $n = 12$.
5. Use the Conjugate Gradient Method to solve (2.45) for $n = 100, 1000$, and $10,000$. Report the size of the final residual, and the number of steps required.
6. Let A be the $n \times n$ matrix with $n = 1000$ and entries $A(i, i) = i$, $A(i, i + 1) = A(i + 1, i) = 1/2$, $A(i, i + 2) = A(i + 2, i) = 1/2$ for all i that fit within the matrix. (a) Print the nonzero structure `spy(A)`. (b) Let x_e be the vector of n ones. Set $b = Ax_e$, and apply the Conjugate Gradient Method, without preconditioner, with the Jacobi preconditioner, and with the Gauss–Seidel preconditioner. Compare errors of the three runs in a plot versus step number.
7. Let $n = 1000$. Start with the $n \times n$ matrix A from Computer Problem 6, and add the nonzero entries $A(i, 2i) = A(2i, i) = 1/2$ for $1 \leq i \leq n/2$. Carry out steps (a) and (b) as in that problem.
8. Let $n = 500$, and let A be the $n \times n$ matrix with entries $A(i, i) = 2$, $A(i, i + 2) = A(i + 2, i) = 1/2$, $A(i, i + 4) = A(i + 4, i) = 1/2$ for all i , and $A(500, i) = A(i, 500) = -0.1$ for $1 \leq i \leq 495$. Carry out steps (a) and (b) as in Computer Problem 6.
9. Let A be the matrix from Computer Problem 8, but with the diagonal elements replaced by $A(i, i) = \sqrt[3]{i}$. Carry out parts (a) and (b) as in that problem.
10. Let C be the 195×195 matrix block with $C(i, i) = 2$, $C(i, i + 3) = C(i + 3, i) = 0.1$, $C(i, i + 39) = C(i + 39, i) = 1/2$, $C(i, i + 42) = C(i + 42, i) = 1/2$ for all i . Define A to be the $n \times n$ matrix with $n = 780$ formed by four diagonally arranged blocks C , and with blocks $\frac{1}{2}C$ on the super- and subdiagonal. Carry out steps (a) and (b) as in Computer Problem 6 to solve $Ax = b$.

2.7 NONLINEAR SYSTEMS OF EQUATIONS

Chapter 1 contains methods for solving one equation in one unknown, usually nonlinear. In this Chapter, we have studied solution methods for systems of equations, but required the equations to be linear. The combination of nonlinear and “more than one equation” raises the degree of difficulty considerably. This section describes Newton’s Method and variants for the solution of systems of nonlinear equations.

2.7.1 Multivariate Newton's Method

The one-variable Newton's Method

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

provides the main outline of the multivariate Newton's Method. Both are derived from the linear approximation afforded by the Taylor expansion. For example, let

$$\begin{aligned} f_1(u, v, w) &= 0 \\ f_2(u, v, w) &= 0 \\ f_3(u, v, w) &= 0 \end{aligned} \tag{2.49}$$

be three nonlinear equations in three unknowns u, v, w . Define the vector-valued function $F(u, v, w) = (f_1, f_2, f_3)$, and denote the problem (2.49) by $F(x) = 0$, where $x = (u, v, w)$.

The analogue of the derivative f' in the one-variable case is the **Jacobian matrix** defined by

$$DF(x) = \begin{bmatrix} \frac{\partial f_1}{\partial u} & \frac{\partial f_1}{\partial v} & \frac{\partial f_1}{\partial w} \\ \frac{\partial f_2}{\partial u} & \frac{\partial f_2}{\partial v} & \frac{\partial f_2}{\partial w} \\ \frac{\partial f_3}{\partial u} & \frac{\partial f_3}{\partial v} & \frac{\partial f_3}{\partial w} \end{bmatrix}.$$

The Taylor expansion for vector-valued functions around x_0 is

$$F(x) = F(x_0) + DF(x_0) \cdot (x - x_0) + O(x - x_0)^2.$$

For example, the linear expansion of $F(u, v) = (e^{u+v}, \sin u)$ around $x_0 = (0, 0)$ is

$$\begin{aligned} F(x) &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} e^0 & e^0 \\ \cos 0 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + O(x^2) \\ &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} u + v \\ u \end{bmatrix} + O(x^2). \end{aligned}$$

Newton's Method is based on a linear approximation, ignoring the $O(x^2)$ terms. As in the one-dimensional case, let $x = r$ be the root, and let x_0 be the current guess. Then

$$0 = F(r) \approx F(x_0) + DF(x_0) \cdot (r - x_0),$$

or

$$-DF(x_0)^{-1}F(x_0) \approx r - x_0. \tag{2.50}$$

Therefore, a better approximation for the root is derived by solving (2.50) for r .

Multivariate Newton's Method

x_0 = initial vector

$$x_{k+1} = x_k - (DF(x_k))^{-1}F(x_k) \quad \text{for } k = 0, 1, 2, \dots$$

Since computing inverses is computationally burdensome, we use a trick to avoid it. On each step, instead of following the preceding definition literally, set $x_{k+1} = x_k + s$, where s is the solution of $DF(x_k)s = -F(x_k)$. Now, only Gaussian elimination ($n^3/3$ multiplications) is needed to carry out a step, instead of computing an inverse (about three times as many). Therefore, the iteration step for multivariate Newton's Method is

$$\begin{cases} DF(x_k)s = -F(x_k) \\ x_{k+1} = x_k + s. \end{cases} \quad (2.51)$$

► **EXAMPLE 2.32** Use Newton's Method with starting guess $(1, 2)$ to find a solution of the system

$$\begin{aligned} v - u^3 &= 0 \\ u^2 + v^2 - 1 &= 0. \end{aligned}$$

Figure 2.5 shows the sets on which $f_1(u, v) = v - u^3$ and $f_2(u, v) = u^2 + v^2 - 1$ are zero and their two intersection points, which are the solutions to the system of equations. The Jacobian matrix is

$$DF(u, v) = \begin{bmatrix} -3u^2 & 1 \\ 2u & 2v \end{bmatrix}.$$

Using starting point $x_0 = (1, 2)$, on the first step we must solve the matrix equation (2.51):

$$\begin{bmatrix} -3 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = - \begin{bmatrix} 1 \\ 4 \end{bmatrix}.$$

The solution is $s = (0, -1)$, so the first iteration produces $x_1 = x_0 + s = (1, 1)$. The second step requires solving

$$\begin{bmatrix} -3 & 1 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = - \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

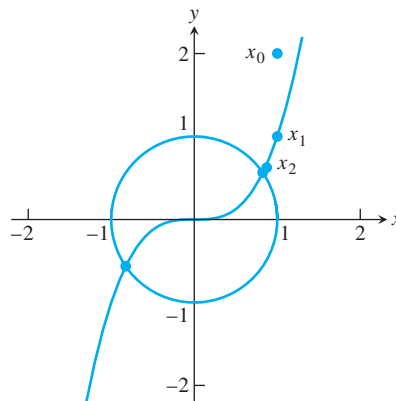


Figure 2.5 Newton's Method for Example 2.32. The two roots are the dots on the circle. Newton's Method produces the dots that are converging to the solution at approximately $(0.8260, 0.5636)$.

The solution is $s = (-1/8, -3/8)$ and $x_2 = x_1 + s = (7/8, 5/8)$. Both iterates are shown in Figure 2.5. Further steps yield the following table:

step	u	v
0	1.00000000000000	2.00000000000000
1	1.00000000000000	1.00000000000000
2	0.87500000000000	0.62500000000000
3	0.82903634826712	0.56434911242604
4	0.82604010817065	0.56361977350284
5	0.82603135773241	0.56362416213163
6	0.82603135765419	0.56362416216126
7	0.82603135765419	0.56362416216126

The familiar doubling of correct decimal places characteristic of quadratic convergence is evident in the output sequence. The symmetry of the equations shows that if (u, v) is a solution, then so is $(-u, -v)$, as is visible in Figure 2.5. The second solution can also be found by applying Newton's Method with a nearby starting guess. ◀

► **EXAMPLE 2.33** Use Newton's Method to find the solutions of the system

$$\begin{aligned} f_1(u, v) &= 6u^3 + uv - 3v^3 - 4 = 0 \\ f_2(u, v) &= u^2 - 18uv^2 + 16v^3 + 1 = 0. \end{aligned}$$

Notice that $(u, v) = (1, 1)$ is one solution. It turns out that there are two others. The Jacobian matrix is

$$DF(u, v) = \begin{bmatrix} 18u^2 + v & u - 9v^2 \\ 2u - 18v^2 & -36uv + 48v^2 \end{bmatrix}.$$

Which solution is found by Newton's Method depends on the starting guess, just as in the one-dimensional case. Using starting point $(u_0, v_0) = (2, 2)$, iterating the preceding formula yields the following table:

step	u	v
0	2.00000000000000	2.00000000000000
1	1.37258064516129	1.34032258064516
2	1.07838681200443	1.05380123264984
3	1.00534968896520	1.00269261871539
4	1.00003367866506	1.00002243772010
5	1.00000000111957	1.00000000057894
6	1.00000000000000	1.00000000000000
7	1.00000000000000	1.00000000000000

Other initial vectors lead to the other two roots, which are approximately $(0.865939, 0.462168)$ and $(0.886809, -0.294007)$. See Computer Problem 2. ◀

Newton's Method is a good choice if the Jacobian can be calculated. If not, the best alternative is Broyden's Method, the subject of the next section.

2.7.2 Broyden's Method

Newton's Method for solving one equation in one unknown requires knowledge of the derivative. The development of this method in Chapter 1 was followed by the discussion of the Secant Method, for use when the derivative is not available or is too expensive to evaluate.

Now that we have a version of Newton's Method for systems of nonlinear equations $F(x) = 0$, we are faced with the same question: What if the Jacobian matrix DF is not

available? Although there is no simple extension of Newton's Method to a Secant Method for systems, Broyden [1965] suggested a method that is generally considered the next best thing.

Suppose A_i is the best approximation available at step i to the Jacobian matrix, and that it has been used to create

$$x_{i+1} = x_i - A_i^{-1} F(x_i). \quad (2.52)$$

To update A_i to A_{i+1} for the next step, we would like to respect the derivative aspect of the Jacobian DF , and satisfy

$$A_{i+1} \delta_{i+1} = \Delta_{i+1}, \quad (2.53)$$

where $\delta_{i+1} = x_{i+1} - x_i$ and $\Delta_{i+1} = F(x_{i+1}) - F(x_i)$. On the other hand, for the orthogonal complement of δ_{i+1} , we have no new information. Therefore, we ask that

$$A_{i+1} w = A_i w \quad (2.54)$$

for every w satisfying $\delta_{i+1}^T w = 0$. One checks that a matrix that satisfies both (2.53) and (2.54) is

$$A_{i+1} = A_i + \frac{(\Delta_{i+1} - A_i \delta_{i+1}) \delta_{i+1}^T}{\delta_{i+1}^T \delta_{i+1}}. \quad (2.55)$$

Broyden's Method uses the Newton's Method step (2.52) to advance the current guess, while updating the approximate Jacobian by (2.55). Summarizing, the algorithm starts with an initial guess x_0 and an initial approximate Jacobian A_0 , which can be chosen to be the identity matrix if there is no better choice.

Broyden's Method I

x_0 = initial vector

A_0 = initial matrix

for $i = 0, 1, 2, \dots$

$$x_{i+1} = x_i - A_i^{-1} F(x_i)$$

$$A_{i+1} = A_i + \frac{(\Delta_{i+1} - A_i \delta_{i+1}) \delta_{i+1}^T}{\delta_{i+1}^T \delta_{i+1}}$$

end

where $\delta_{i+1} = x_{i+1} - x_i$ and $\Delta_{i+1} = F(x_{i+1}) - F(x_i)$.

Note that the Newton-type step is carried out by solving $A_i \delta_{i+1} = F(x_i)$, just as for Newton's Method. Also like Newton's Method, Broyden's Method is not guaranteed to converge to a solution.

A second approach to Broyden's Method avoids the relatively expensive matrix solver step $A_i \delta_{i+1} = F(x_i)$. Since we are at best only approximating the derivative DF during the iteration, we may as well be approximating the inverse of DF instead, which is what is needed in the Newton step.

We redo the derivation of Broyden from the point of view of $B_i = A_i^{-1}$. We would like to have

$$\delta_{i+1} = B_{i+1} \Delta_{i+1}, \quad (2.56)$$

where $\delta_{i+1} = x_{i+1} - x_i$ and $\Delta_{i+1} = F(x_{i+1}) - F(x_i)$, and for every w satisfying $\delta_{i+1}^T w = 0$, still satisfy $A_{i+1} w = A_i w$, or

$$B_{i+1} A_i w = w. \quad (2.57)$$

A matrix that satisfies both (2.56) and (2.57) is

$$B_{i+1} = B_i + \frac{(\delta_{i+1} - B_i \Delta_{i+1}) \delta_{i+1}^T B_i}{\delta_{i+1}^T B_i \Delta_{i+1}}. \quad (2.58)$$

The new version of the iteration, which needs no matrix solve, is

$$x_{i+1} = x_i - B_i F(x_i). \quad (2.59)$$

The resulting algorithm is called Broyden's Method II.

Broyden's Method II

x_0 = initial vector

B_0 = initial matrix

for $i = 0, 1, 2, \dots$

$$x_{i+1} = x_i - B_i F(x_i)$$

$$B_{i+1} = B_i + \frac{(\delta_{i+1} - B_i \Delta_{i+1}) \delta_{i+1}^T B_i}{\delta_{i+1}^T B_i \Delta_{i+1}}$$

end

where $\delta_i = x_i - x_{i-1}$ and $\Delta_i = F(x_i) - F(x_{i-1})$.

To begin, an initial vector x_0 and an initial guess for B_0 are needed. If it is impossible to compute derivatives, the choice $B_0 = I$ can be used.

A perceived disadvantage of Broyden II is that estimates for the Jacobian, needed for some applications, are not easily available. The matrix B_i is an estimate for the matrix inverse of the Jacobian. Broyden I, on the other hand, keeps track of A_i , which estimates the Jacobian. For this reason, in some circles Broyden I and II are referred to as "Good Broyden" and "Bad Broyden," respectively.

Both versions of Broyden's Method converge superlinearly (to simple roots), slightly slower than the quadratic convergence of Newton's Method. If a formula for the Jacobian is available, it usually speeds convergence to use the inverse of $DF(x_0)$ for the initial matrix B_0 .

MATLAB code for Broyden's Method II is as follows:

```
% Program 2.3 Broyden's Method II
% Input: initial vector x0, max steps k
% Output: solution x
% Example usage: broyden2(f, [1;1], 10)
function x=broyden2(f, x0, k)
[n,m]=size(x0);
b=eye(n,n); % initial b
for i=1:k
    x=x0-b*f(x0);
    del=x-x0; delta=f(x)-f(x0);
    b=b+(del-b*delta)*del'*(del'*b*delta);
    x0=x;
end
```

For example, a solution of the system in Example 2.32 is found by defining a function

```
>> f=@(x) [x(2)-x(1)^3; x(1)^2+x(2)^2-1];
```

and calling Broyden's Method II as

```
>> x=broyden2(f, [1;1], 10)
```

Broyden's Method, in either implementation, is very useful in cases where the Jacobian is unavailable. A typical instance of this situation is illustrated in the model of pipe buckling in Reality Check 7.

2.7 Exercises

- Find the Jacobian of the functions (a) $F(u, v) = (u^3, uv^3)$
(b) $F(u, v) = (\sin uv, e^{uv})$ (c) $F(u, v) = (u^2 + v^2 - 1, (u - 1)^2 + v^2 - 1)$
(d) $F(u, v, w) = (u^2 + v - w^2, \sin uvw, uvw^4)$.
- Use the Taylor expansion to find the linear approximation $L(x)$ to $F(x)$ near x_0 .
(a) $F(u, v) = (1 + e^{u+2v}, \sin(u + v))$, $x_0 = (0, 0)$
(b) $F(u, v) = (u + e^{u-v}, 2u + v)$, $x_0 = (1, 1)$
- Sketch the two curves in the uv -plane, and find all solutions exactly by simple algebra.
(a) $\begin{cases} u^2 + v^2 = 1 \\ (u - 1)^2 + v^2 = 1 \end{cases}$ (b) $\begin{cases} u^2 + 4v^2 = 4 \\ 4u^2 + v^2 = 4 \end{cases}$ (c) $\begin{cases} u^2 - 4v^2 = 4 \\ (u - 1)^2 + v^2 = 4 \end{cases}$
- Apply two steps of Newton's Method to the systems in Exercise 3, with starting point $(1, 1)$.
- Apply two steps of Broyden I to the systems in Exercise 3, with starting point $(1, 1)$, using $A_0 = I$.
- Apply two steps of Broyden II to the systems in Exercise 3, with starting point $(1, 1)$, using $B_0 = I$.
- Prove that (2.55) satisfies (2.53) and (2.54).
- Prove that (2.58) satisfies (2.56) and (2.57).

2.7 Computer Problems

- Implement Newton's Method with appropriate starting points to find all solutions. Check with Exercise 3 to make sure your answers are correct.

$$(a) \begin{cases} u^2 + v^2 = 1 \\ (u - 1)^2 + v^2 = 1 \end{cases} \quad (b) \begin{cases} u^2 + 4v^2 = 4 \\ 4u^2 + v^2 = 4 \end{cases} \quad (c) \begin{cases} u^2 - 4v^2 = 4 \\ (u - 1)^2 + v^2 = 4 \end{cases}$$

- Use Newton's Method to find the three solutions of Example 2.31.
- Use Newton's Method to find the two solutions of the system $u^3 - v^3 + u = 0$ and $u^2 + v^2 = 1$.
- (a) Apply Newton's Method to find both solutions of the system of three equations.

$$\begin{aligned} 2u^2 - 4u + v^2 + 3w^2 + 6w + 2 &= 0 \\ u^2 + v^2 - 2v + 2w^2 - 5 &= 0 \\ 3u^2 - 12u + v^2 + 3w^2 + 8 &= 0 \end{aligned}$$

- Use Multivariate Newton's Method to find the two points in common of the three given spheres in three-dimensional space. (a) Each sphere has radius 1, with centers $(1, 1, 0)$, $(1, 0, 1)$, and $(0, 1, 1)$. (Ans. $(1, 1, 1)$ and $(1/3, 1/3, 1/3)$) (b) Each sphere has radius 5, with centers $(1, -2, 0)$, $(-2, 2, -1)$, and $(4, -2, 3)$.

6. Although a generic intersection of three spheres in three-dimensional space is two points, it can be a single point. Apply Multivariate Newton's Method to find the single point of intersection of the spheres with center $(1, 0, 1)$ and radius $\sqrt{8}$, center $(0, 2, 2)$ and radius $\sqrt{2}$, and center $(0, 3, 3)$ and radius $\sqrt{2}$. Does the iteration still converge quadratically? Explain.
7. Apply Broyden I with starting guesses $x_0 = (1, 1)$ and $A_0 = I$ to the systems in Exercise 3. Report the solutions to as much accuracy as possible and the number of steps required.
8. Apply Broyden II with starting guesses $(1, 1)$ and $B_0 = I$ to the systems in Exercise 3. Report the solutions to as much accuracy as possible and the number of steps required.
9. Apply Broyden I to find the sets of two intersection points in Computer Problem 5.
10. Apply Broyden I to find the intersection point in Computer Problem 6. What can you observe about the convergence rate?
11. Apply Broyden II to find the sets of two intersection points in Computer Problem 5.
12. Apply Broyden II to find the intersection point in Computer Problem 6. What can you observe about the convergence rate?

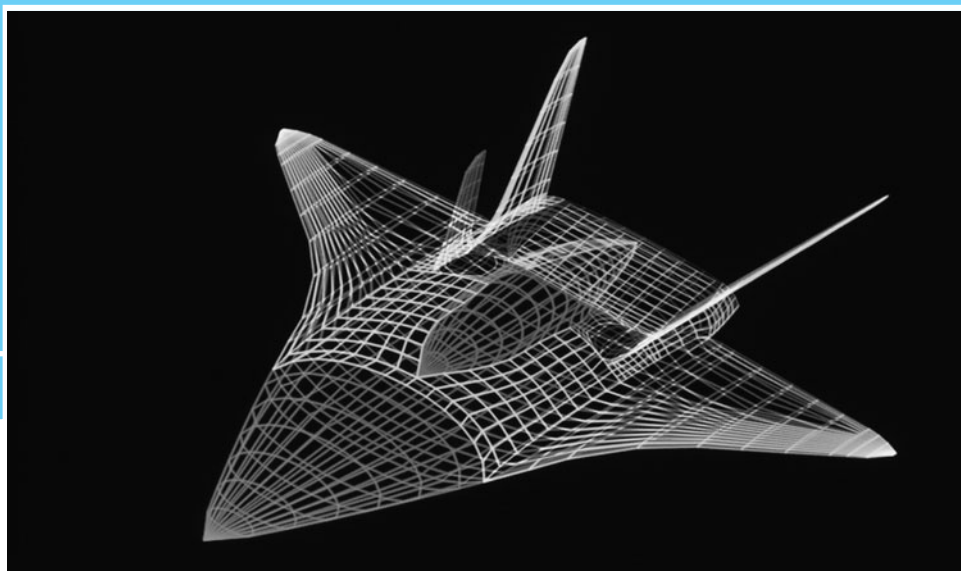
Software and Further Reading

Many excellent texts have appeared on numerical linear algebra, including Stewart [1973] and the comprehensive reference Golub and Van Loan [1996]. Two excellent books with a modern approach to numerical linear algebra are Demmel [1997] and Trefethen and Bau [1997]. Books to consult on iterative methods include Axelsson [1994], Hackbush [1994], Kelley [1995], Saad [1996], Traub [1964], Varga [2000], Young [1971], and Dennis and Schnabel [1983].

LAPACK is a comprehensive, public domain software package containing high-quality routines for matrix algebra computations, including methods for solving $Ax = b$, matrix factorizations, and condition number estimation. It is carefully written to be portable to modern computer architectures, including shared memory vector and parallel processors. See Anderson et al. [1990].

The portability of LAPACK depends on the fact that its algorithms are written in such a way as to maximize use of the Basic Linear Algebra Subprograms (BLAS), a set of primitive matrix/vector computations that can be tuned to optimize performance on particular machines and architectures. BLAS is divided roughly into three parts: Level 1, requiring $O(n)$ operations like dot products; Level 2, operations such as matrix/vector multiplication, that are $O(n^2)$; and Level 3, including full matrix/matrix multiplication, which has complexity $O(n^3)$.

The general dense matrix routine in LAPACK for solving $Ax = b$ in double precision, using the $PA = LU$ factorization, is called DGESV, and there are other versions for sparse and banded matrices. See www.netlib.org/lapack for more details. Implementations of LAPACK routines also form the basis for MATLAB's matrix algebra computations, and those of the IMSL and NAG packages.



Interpolation

Polynomial interpolation is an ancient practice, but the heavy industrial use of interpolation began with cubic splines in the 20th century. Motivated by practices in the shipbuilding and aircraft industries, engineers Paul de Casteljau and Pierre Bézier at rival European car manufacturers Citroen and Renault, followed by others at General Motors in the United States, spurred the development of what are now called cubic splines and Bézier splines.

Although developed for aerodynamic studies of automobiles, splines have been used for many applications, including computer typesetting. A revolution in printing was caused by two Xerox engineers who formed a company named Adobe and released the

PostScript™ language in 1984. It came to the attention of Steve Jobs at Apple Corporation, who was looking for a way to control a newly invented laser printer. Bézier splines were a simple way to adapt the same mathematical curves to fonts with multiple printer resolutions. Later, Adobe used many of the fundamental ideas of PostScript as the basis of a more flexible format called PDF (Portable Document Format), which became a ubiquitous document file type by the early 21st century.

Reality Check

Check Reality Check 3 on page 183 explores how PDF files use Bézier splines to represent printed characters in arbitrary fonts.

Efficient ways of representing data are fundamental to advancing the understanding of scientific problems. At its most fundamental, approximating data by a polynomial is an act of data compression. Suppose that points (x, y) are taken from a given function $y = f(x)$, or perhaps from an experiment where x denotes temperature and y denotes reaction rate. A function on the real numbers represents an infinite amount of information. Finding a polynomial through the set of data means replacing the information with a rule that can be evaluated in a finite number of steps. Although it is unrealistic to expect the polynomial to represent the function exactly at new inputs x , it may be close enough to solve practical problems.

This chapter introduces polynomial interpolation and spline interpolation as convenient tools for finding functions that pass through given data points.

3.1 DATA AND INTERPOLATING FUNCTIONS

A function is said to interpolate a set of data points if it passes through those points. Suppose that a set of (x, y) data points has been collected, such as $(0, 1)$, $(2, 2)$, and $(3, 4)$. There is a parabola that passes through the three points, shown in Figure 3.1. This parabola is called the degree 2 interpolating polynomial passing through the three points.

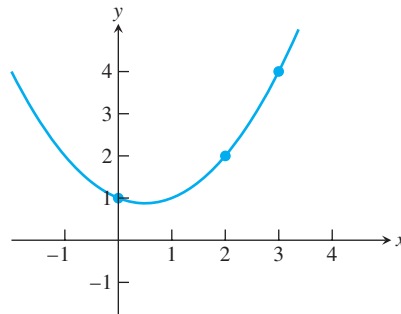


Figure 3.1 Interpolation by parabola. The points $(0,1)$, $(2,2)$, and $(3,4)$ are interpolated by the function $P(x) = \frac{1}{2}x^2 - \frac{1}{2}x + 1$.

DEFINITION 3.1 The function $y = P(x)$ **interpolates** the data points $(x_1, y_1), \dots, (x_n, y_n)$ if $P(x_i) = y_i$ for each $1 \leq i \leq n$. □

Note that P is required to be a function; that is, each value x corresponds to a single y . This puts a restriction on the set of data points $\{(x_i, y_i)\}$ that can be interpolated—the x_i 's must be all distinct in order for a function to pass through them. There is no such restriction on the y_i 's.

To begin, we will look for an interpolating polynomial. Does such a polynomial always exist? Assuming that the x -coordinates of the points are distinct, the answer is yes. No matter how many points are given, there is some polynomial $y = P(x)$ that runs through all the points. This and several other facts about interpolating polynomials are proved in this section.

Interpolation is the reverse of evaluation. In polynomial evaluation (such as the nested multiplication of Chapter 0), we are given a polynomial and asked to evaluate a y -value for a given x -value—that is, compute points lying on the curve. Polynomial interpolation asks for the opposite process: Given these points, compute a polynomial that can generate them.

SPOTLIGHT ON

Complexity

Why do we use polynomials? Polynomials are very often used for interpolation because of their straightforward mathematical properties. There is a simple theory about when an interpolating polynomial of a given degree exists for a given set of points. More important, in a real sense, polynomials are the most fundamental of functions for digital computers. Central processing units usually have fast methods in hardware for adding and multiplying floating point numbers, which are the only operations needed to evaluate a polynomial. Complicated functions can be approximated by interpolating polynomials in order to make them computable with these two hardware operations.

3.1.1 Lagrange interpolation

Assume that n data points $(x_1, y_1), \dots, (x_n, y_n)$ are given, and that we would like to find an interpolating polynomial. There is an explicit formula, called the Lagrange interpolating formula, for writing down a polynomial of degree $d = n - 1$ that interpolates the points. For example, suppose that we are given three points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. Then the polynomial

$$P_2(x) = y_1 \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + y_2 \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + y_3 \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} \quad (3.1)$$

is the **Lagrange interpolating polynomial** for these points. First notice why the points each lie on the polynomial curve. When x_1 is substituted for x , the terms evaluate to $y_1 + 0 + 0 = y_1$. The second and third numerators are chosen to disappear when x_1 is substituted, and the first denominator is chosen just so to balance the first denominator so that y_1 pops out. It is similar when x_2 and x_3 are substituted. When any other number is substituted for x , we have little control over the result. But then, the job was only to interpolate at the three points—that is the extent of our concern. Second, notice that the polynomial (3.1) is of degree 2 in the variable x .

► **EXAMPLE 3.1** Find an interpolating polynomial for the data points $(0, 1)$, $(2, 2)$, and $(3, 4)$ in Figure 3.1.

Substituting into Lagrange's formula (3.1) yields

$$\begin{aligned} P_2(x) &= 1 \frac{(x - 2)(x - 3)}{(0 - 2)(0 - 3)} + 2 \frac{(x - 0)(x - 3)}{(2 - 0)(2 - 3)} + 4 \frac{(x - 0)(x - 2)}{(3 - 0)(3 - 2)} \\ &= \frac{1}{6}(x^2 - 5x + 6) + 2 \left(-\frac{1}{2} \right) (x^2 - 3x) + 4 \left(\frac{1}{3} \right) (x^2 - 2x) \\ &= \frac{1}{2}x^2 - \frac{1}{2}x + 1. \end{aligned}$$

Check that $P_2(0) = 1$, $P_2(2) = 2$, and $P_2(3) = 4$. ◀

In general, suppose that we are presented with n points $(x_1, y_1), \dots, (x_n, y_n)$. For each k between 1 and n , define the degree $n - 1$ polynomial

$$L_k(x) = \frac{(x - x_1) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_1) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}.$$

The interesting property of L_k is that $L_k(x_k) = 1$, while $L_k(x_j) = 0$, where x_j is any of the other data points. Then define the degree $n - 1$ polynomial

$$P_{n-1}(x) = y_1 L_1(x) + \cdots + y_n L_n(x).$$

This is a straightforward generalization of the polynomial in (3.1) and works the same way. Substituting x_k for x yields

$$P_{n-1}(x_k) = y_1 L_1(x_k) + \cdots + y_n L_n(x_k) = 0 + \cdots + 0 + y_k L_k(x_k) + 0 + \cdots + 0 = y_k,$$

so it works as designed.

We have constructed a polynomial of degree at most $n - 1$ that passes through any set of n points with distinct x_i 's. Interestingly, it is the only one.

THEOREM 3.2 Main Theorem of Polynomial Interpolation. Let $(x_1, y_1), \dots, (x_n, y_n)$ be n points in the plane with distinct x_i . Then there exists one and only one polynomial P of degree $n - 1$ or less that satisfies $P(x_i) = y_i$ for $i = 1, \dots, n$. ■

Proof. The existence is proved by the explicit formula for Lagrange interpolation. To show there is only one, assume for the sake of argument that there are two, say, $P(x)$ and $Q(x)$, that have degree at most $n - 1$ and that both interpolate all n points. That is, we are assuming that $P(x_1) = Q(x_1) = y_1$, $P(x_2) = Q(x_2) = y_2$, \dots , $P(x_n) = Q(x_n) = y_n$. Now define the new polynomial $H(x) = P(x) - Q(x)$. Clearly, the degree of H is also at most $n - 1$, and note that $0 = H(x_1) = H(x_2) = \dots = H(x_n)$; that is, H has n distinct zeros. According to the Fundamental Theorem of Algebra, a degree d polynomial can have at most d zeros, unless it is the identically zero polynomial. Therefore, H is the identically zero polynomial, and $P(x) \equiv Q(x)$. We conclude that there is a unique $P(x)$ of degree $\leq n - 1$ interpolating the n points (x_i, y_i) . □

► **EXAMPLE 3.2** Find the polynomial of degree 3 or less that interpolates the points $(0, 2)$, $(1, 1)$, $(2, 0)$, and $(3, -1)$.

The Lagrange form is as follows:

$$\begin{aligned} P(x) &= 2 \frac{(x-1)(x-2)(x-3)}{(0-1)(0-2)(0-3)} + 1 \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)} \\ &\quad + 0 \frac{(x-0)(x-1)(x-3)}{(2-0)(2-1)(2-3)} - 1 \frac{(x-0)(x-1)(x-2)}{(3-0)(3-1)(3-2)} \\ &= -\frac{1}{3}(x^3 - 6x^2 + 11x - 6) + \frac{1}{2}(x^3 - 5x^2 + 6x) - \frac{1}{6}(x^3 - 3x^2 + 2x) \\ &= -x + 2. \end{aligned}$$

Theorem 3.2 says that there exists exactly one interpolating polynomial of degree 3 or less, but it may or may not be exactly degree 3. In Example 3.2, the data points are collinear, so the interpolating polynomial has degree 1. Theorem 3.2 implies that there are no interpolating polynomials of degree 2 or 3. It may be already intuitively obvious to you that no parabola or cubic curve can pass through four collinear points, but here is the reason. ◀

3.1.2 Newton's divided differences

The Lagrange interpolation method, as described in the previous section, is a constructive way to write the unique polynomial promised by Theorem 3.2. It is also intuitive; one glance explains why it works. However, it is seldom used for calculation because alternative methods result in more manageable and less computationally complex forms.

Newton's divided differences give a particularly simple way to write the interpolating polynomial. Given n data points, the result will be a polynomial of degree at most $n - 1$, just as Lagrange form does. Theorem 3.2 says that it can be none other than the same as the Lagrange interpolating polynomial, written in a disguised form.

The idea of divided differences is fairly simple, but some notation needs to be mastered first. Assume that the data points come from a function $f(x)$, so that our goal is to interpolate $(x_1, f(x_1)), \dots, (x_n, f(x_n))$.

DEFINITION 3.3 Denote by $f[x_1 \dots x_n]$ the coefficient of the x^{n-1} term in the (unique) polynomial that interpolates $(x_1, f(x_1)), \dots, (x_n, f(x_n))$. □

Example 3.1 shows that $f[0\ 2\ 3] = 1/2$, where we assume $f(0) = 1$, $f(2) = 2$, and $f(3) = 4$. Of course, by uniqueness, all permutations of 0, 2, 3 give the same value: $1/2 = f[0\ 3\ 2] = f[3\ 0\ 2]$ etc. Using this definition, the following somewhat remarkable alternative formula for the interpolating polynomial holds, called the **Newton's divided difference formula**

$$\begin{aligned} P(x) = & f[x_1] + f[x_1\ x_2](x - x_1) \\ & + f[x_1\ x_2\ x_3](x - x_1)(x - x_2) \\ & + f[x_1\ x_2\ x_3\ x_4](x - x_1)(x - x_2)(x - x_3) \\ & + \dots \\ & + f[x_1 \dots x_n](x - x_1) \dots (x - x_{n-1}). \end{aligned} \quad (3.2)$$

Moreover, the coefficients $f[x_1 \dots x_k]$ from the above definition can be recursively calculated as follows. List the data points in a table:

x_1	$f(x_1)$
x_2	$f(x_2)$
\vdots	\vdots
x_n	$f(x_n)$

Now define the divided differences, which are the real numbers

$$\begin{aligned} f[x_k] &= f(x_k) \\ f[x_k\ x_{k+1}] &= \frac{f[x_{k+1}] - f[x_k]}{x_{k+1} - x_k} \\ f[x_k\ x_{k+1}\ x_{k+2}] &= \frac{f[x_{k+1}\ x_{k+2}] - f[x_k\ x_{k+1}]}{x_{k+2} - x_k} \\ f[x_k\ x_{k+1}\ x_{k+2}\ x_{k+3}] &= \frac{f[x_{k+1}\ x_{k+2}\ x_{k+3}] - f[x_k\ x_{k+1}\ x_{k+2}]}{x_{k+3} - x_k}, \end{aligned} \quad (3.3)$$

and so on. Both important facts, that (1) the unique polynomial interpolating $(x_1, f(x_1)), \dots, (x_n, f(x_n))$ is given by (3.2) and (2) the coefficients can be calculated as (3.3), are not immediately obvious, and proofs will be provided in Section 3.2.2. Notice that the divided difference formula gives the interpolating polynomial as a nested polynomial. It is automatically ready to be evaluated in an efficient way.

Newton's divided differences

Given $x = [x_1, \dots, x_n]$, $y = [y_1, \dots, y_n]$

for $j = 1, \dots, n$

$f[x_j] = y_j$

end

for $i = 2, \dots, n$

for $j = 1, \dots, n + 1 - i$

$f[x_j \dots x_{j+i-1}] = (f[x_{j+1} \dots x_{j+i-1}] - f[x_j \dots x_{j+i-2}]) / (x_{j+i-1} - x_j)$

end

end

The interpolating polynomial is

$$P(x) = \sum_{i=1}^n f[x_1 \dots x_i](x - x_1) \dots (x - x_{i-1})$$

The recursive definition of the Newton's divided differences allows arrangement into a convenient table. For three points the table has the form

$$\begin{array}{c|cc} x_1 & f[x_1] & \\ & & f[x_1 \ x_2] \\ x_2 & f[x_2] & & f[x_1 \ x_2 \ x_3] \\ & & f[x_2 \ x_3] & \\ x_3 & f[x_3] & & \end{array}$$

The coefficients of the polynomial (3.2) can be read from the top edge of the triangle.

► **EXAMPLE 3.3** Use divided differences to find the interpolating polynomial passing through the points (0, 1), (2, 2), (3, 4).

Applying the definitions of divided differences leads to the following table:

$$\begin{array}{c|cc} 0 & 1 & \\ & & \frac{1}{2} \\ 2 & 2 & & \frac{1}{2} \\ & & 2 & \\ 3 & 4 & & \end{array}$$

This table is computed as follows: After writing down the x and y coordinates in separate columns, calculate the next columns, left to right, as divided differences, as in (3.3). For example,

$$\begin{aligned} \frac{2-1}{2-0} &= \frac{1}{2} \\ \frac{2-\frac{1}{2}}{3-0} &= \frac{1}{2} \\ \frac{4-2}{3-2} &= 2. \end{aligned}$$

After completing the divided difference triangle, the coefficients of the polynomial $1, 1/2, 1/2$ can be read from the top edge of the table. The interpolating polynomial can be written as

$$P(x) = 1 + \frac{1}{2}(x-0) + \frac{1}{2}(x-0)(x-2),$$

or, in nested form,

$$P(x) = 1 + (x-0) \left(\frac{1}{2} + (x-2) \cdot \frac{1}{2} \right).$$

The base points for the nested form (see Chapter 0) are $r_1 = 0$ and $r_2 = 2$. Alternatively, we could do more algebra and write the interpolating polynomial as

$$P(x) = 1 + \frac{1}{2}x + \frac{1}{2}x(x-2) = \frac{1}{2}x^2 - \frac{1}{2}x + 1,$$

matching the Lagrange interpolation version shown previously. ◀

Using the divided difference approach, new data points that arrive after computing the original interpolating polynomial can be easily added.


► **EXAMPLE 3.4** Add the fourth data point (1, 0) to the list in Example 3.3.

We can keep the calculations that were already done and just add a new bottom row to the triangle:

$$\begin{array}{c|cccc}
0 & 1 & & & \\
& & \frac{1}{2} & & \\
2 & 2 & & \frac{1}{2} & \\
& & 2 & & -\frac{1}{2} \\
3 & 4 & & 0 & \\
& & 2 & & \\
1 & 0 & & &
\end{array}$$

The result is one new term to add to the original polynomial $P_2(x)$. Reading from the top edge of the triangle, we see that the new degree 3 interpolating polynomial is

$$P_3(x) = 1 + \frac{1}{2}(x - 0) + \frac{1}{2}(x - 0)(x - 2) - \frac{1}{2}(x - 0)(x - 2)(x - 3).$$

Note that $P_3(x) = P_2(x) - \frac{1}{2}(x - 0)(x - 2)(x - 3)$, so the previous polynomial can be reused as part of the new one. 

It is interesting to compare the extra work necessary to add a new point to the Lagrange formulation versus the divided difference formulation. The Lagrange polynomial must be restarted from the beginning when a new point is added; none of the previous calculation can be used. On the other hand, in divided difference form, we keep the earlier work and add one new term to the polynomial. Therefore, the divided difference approach has a “real-time updating” property that the Lagrange form lacks.


► **EXAMPLE 3.5** Use Newton’s divided differences to find the interpolating polynomial passing through $(0, 2)$, $(1, 1)$, $(2, 0)$, $(3, -1)$.

The divided difference triangle is

$$\begin{array}{c|cccc}
0 & 2 & & & \\
& & -1 & & \\
1 & 1 & & 0 & \\
& & -1 & & 0 \\
2 & 0 & & 0 & \\
& & -1 & & \\
3 & -1 & & &
\end{array}$$

Reading off the coefficients, we find that the interpolating polynomial of degree 3 or less is

$$P(x) = 2 + (-1)(x - 0) = 2 - x,$$

agreeing with Example 3.2, but arrived at with much less work. 

3.1.3 How many degree d polynomials pass through n points?

Theorem 3.2, the Main Theorem of Polynomial Interpolation, answers this question if $0 \leq d \leq n - 1$. Given $n = 3$ points $(0, 1)$, $(2, 2)$, $(3, 4)$, there is one interpolating polynomial of degree 2 or less. Example 3.1 shows that it is degree 2, so there are no degree 0 or 1 interpolating polynomials through the three data points.

How many degree 3 polynomials interpolate the same three points? One way to construct such a polynomial is clear from the previous discussion: Add a fourth point. Extending the Newton’s divided difference triangle gives a new top coefficient. In Example 3.4, the point $(1, 0)$ was added. The resulting polynomial,

$$P_3(x) = P_2(x) - \frac{1}{2}(x - 0)(x - 2)(x - 3), \quad (3.4)$$

passes through the three points in question, in addition to the new point $(1, 0)$. So there is at least one degree 3 polynomial passing through our three original points $(0, 1)$, $(2, 2)$, $(3, 4)$.

Of course, there are many different ways we could have chosen the fourth point. For example, if we keep the same $x_4 = 1$ and simply change y_4 from 0, we *must* get a different degree 3 interpolating polynomial, since a function can only go through one y -value at x_4 . Now we know there are infinitely many polynomials that interpolate the three points (x_1, y_1) , (x_2, y_2) , (x_3, y_3) , since for any fixed x_4 there are infinitely many ways y_4 can be chosen, each giving a different polynomial. This line of thinking shows that given n data points (x_i, y_i) with distinct x_i , there are infinitely many degree n polynomials passing through them.

A second look at (3.4) suggests a more direct way to produce interpolating polynomials of degree 3 through three points. Instead of adding a fourth point to generate a new degree 3 coefficient, why not just pencil in an arbitrary degree 3 coefficient? Does the result interpolate the original three points? Yes, because $P_2(x)$ does, and the new term evaluates to zero at x_1, x_2 , and x_3 . So there is really no need to construct the extra Newton's divided differences for this purpose. Any degree 3 polynomial of the form

$$P_3(x) = P_2(x) + cx(x-2)(x-3)$$

with $c \neq 0$ will pass through $(0, 1)$, $(2, 2)$, and $(3, 4)$. This technique will also easily construct (infinitely many) polynomials of degree $\geq n$ for n given data points, as illustrated in the next example.

► **EXAMPLE 3.6** How many polynomials of each degree $0 \leq d \leq 5$ pass through the points $(-1, -5)$, $(0, -1)$, $(2, 1)$, and $(3, 11)$?

The Newton's divided difference triangle is

$$\begin{array}{c|ccc} -1 & -5 & & \\ & & 4 & \\ 0 & -1 & & -1 \\ & & 1 & & 1 \\ 2 & 1 & & 3 & \\ & & 10 & \\ 3 & 11 & & \end{array}$$

So there are no interpolating polynomials of degree 0, 1, or 2, and the single degree 3 is

$$P_3(x) = -5 + 4(x+1) - (x+1)x + (x+1)x(x-2).$$

There are infinitely many degree 4 interpolating polynomials

$$P_4(x) = P_3(x) + c_1(x+1)x(x-2)(x-3)$$

for arbitrary $c_1 \neq 0$, and infinitely many degree 5 interpolating polynomials

$$P_5(x) = P_3(x) + c_2(x+1)x^2(x-2)(x-3)$$

for arbitrary $c_2 \neq 0$. ◀

3.1.4 Code for interpolation

The MATLAB program `newtd.m` for computing the coefficients follows:


```

%Program 3.1 Newton Divided Difference Interpolation Method
%Computes coefficients of interpolating polynomial
%Input: x and y are vectors containing the x and y coordinates
%      of the n data points
%Output: coefficients c of interpolating polynomial in nested form
%Use with nest.m to evaluate interpolating polynomial
function c=newtdd(x,y,n)
for j=1:n
    v(j,1)=y(j);           % Fill in y column of Newton triangle
end
for i=2:n
    % For column i,
    for j=1:n+1-i
        % fill in column from top to bottom
        v(j,i)=(v(j+1,i-1)-v(j,i-1))/(x(j+i-1)-x(j));
    end
end
for i=1:n
    c(i)=v(1,i);           % Read along top of triangle
end
% for output coefficients

```

This program can be applied to the data points of Example 3.3 to return the coefficients $1, 1/2, 1/2$ found above. These coefficients can be used in the nested multiplication program to evaluate the interpolating polynomial at various x -values.

For example, the MATLAB code segment

```

x0=[0 2 3];
y0=[1 2 4];
c=newtdd(x0,y0,3);
x=0:.01:4;
y=nest(2,c,x,x0);
plot(x0,y0,'o',x,y)

```

will result in the plot of the polynomial shown in Figure 3.1.

SPOTLIGHT ON

Compression

This is our first encounter with the concept of compression in numerical analysis. At first, interpolation may not seem like compression. After all, we take n points as input and deliver n coefficients (of the interpolating polynomial) as output. What has been compressed?

Think of the data points as coming from somewhere, say as representatives chosen from the multitude of points on a curve $y = f(x)$. The degree $n - 1$ polynomial, characterized by n coefficients, is a “compressed version” of $f(x)$, and may in some cases be used as a fairly simple representative of $f(x)$ for computational purposes.

For example, what happens when the sin key is pushed on a calculator? The calculator has hardware to add and multiply, but how does it compute the sin of a number? Somehow the operation must reduce to the evaluation of a polynomial, which requires exactly those operations. By choosing data points lying on the sine curve, an interpolating polynomial can be calculated and stored in the calculator as a compressed version of the sine function.

This type of compression is “lossy compression,” meaning that there will be error involved, since the sine function is not actually a polynomial. How much error is made when a function $f(x)$ is replaced by an interpolating polynomial is the subject of the next section.

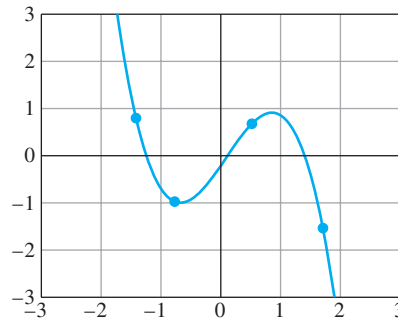


Figure 3.2 Interpolation program 3.2 using mouse input. Screenshot of MATLAB code `clickinterp.m` with four input data points.

Now that we have MATLAB code for finding the coefficients of the interpolating polynomial (`newtdd.m`) and for evaluating the polynomial (`nest.m`), we can put them together to build a polynomial interpolation routine. The program `clickinterp.m` uses MATLAB's graphics capability to plot the interpolation polynomial as it is being created. See Figure 3.2. MATLAB's mouse input command `ginput` is used to facilitate data entry.

```
%Program 3.2. Polynomial Interpolation Program
%Click in MATLAB figure window to locate data point.
%    Continue, to add more points.
%    Press return to terminate program.
function clickinterp
xl=-3;xr=3;yb=-3;yt=3;
plot([xl xr],[0 0],'k',[0 0],[yb yt],'k');grid on;
xlist=[];ylist=[];
k=0;      % initialize counter k
while(0==0)
    [xnew,ynew] = ginput(1); % get mouse click
    if length(xnew) < 1
        break % if return pressed, terminate
    end
    k=k+1; % k counts clicks
    xlist(k)=xnew; ylist(k)=ynew; % add new point to the list
    c=newtdd(xlist,ylist,k); % get interpolation coeffs
    x=xl:.01:xr; % define x coordinates of curve
    y=nest(k-1,c,x,xlist); % get y coordinates of curve
    plot(xlist,ylist,'o',x,y,[xl xr],[0 0],'k',[0 0],[yb yt],'k');
    axis([xl xr yb yt]);grid on;
end
```

3.1.5 Representing functions by approximating polynomials

A major use of polynomial interpolation is to replace evaluation of a complicated function by evaluation of a polynomial, which involves only elementary computer operations like addition, subtraction, and multiplication. Think of this as a form of compression: Something complex is replaced with something simpler and computable, with perhaps some loss in accuracy that we will have to analyze. We begin with an example from trigonometry.

► **EXAMPLE 3.7** Interpolate the function $f(x) = \sin x$ at 4 equally spaced points on $[0, \pi/2]$.

Let's compress the sine function on the interval $[0, \pi/2]$. Take four data points at equally spaced points and form the divided difference triangle. We list the values to four correct places:

0	0.0000			
		0.9549		
$\pi/6$	0.5000		-0.2443	
		0.6990		-0.1139
$2\pi/6$	0.8660		-0.4232	
		0.2559		
$3\pi/6$	1.0000			

The degree 3 interpolating polynomial is therefore

$$\begin{aligned}
 P_3(x) &= 0 + 0.9549x - 0.2443x(x - \pi/6) - 0.1139x(x - \pi/6)(x - \pi/3) \\
 &= 0 + x(0.9549 + (x - \pi/6)(-0.2443 + (x - \pi/3)(-0.1139))). \quad (3.5)
 \end{aligned}$$

This polynomial is graphed together with the sine function in Figure 3.3. At this level of resolution, $P_3(x)$ and $\sin x$ are virtually indistinguishable on the interval $[0, \pi/2]$. We have compressed the infinite amount of information held by the sine curve into a few stored coefficients and the ability to perform the 3 adds and 3 multiplies in (3.5). ◀

How close are we to designing the `sin` key on a calculator? Certainly we need to handle inputs from the entire real line. But due to the symmetries of the sine function, we have done the hard part. The interval $[0, \pi/2]$ is a so-called **fundamental domain** for sine, meaning that an input from any other interval can be referred back to it. Given an input x from $[\pi/2, \pi]$, say, we can compute $\sin x$ as $\sin(\pi - x)$, since \sin is symmetric about $x = \pi/2$. Given an input x from $[\pi, 2\pi]$, $\sin x = -\sin(2\pi - x)$ due to antisymmetry about $x = \pi$. Finally, because \sin repeats its behavior on the interval $[0, 2\pi]$ across the entire real line, we can calculate for any input by first reducing modulo 2π . This leads to a straightforward design for the `sin` key:

```

%Program 3.3 Building a sin calculator key, attempt #1
%Approximates sin curve with degree 3 polynomial
%      (Caution: do not use to build bridges,
%      at least until we have discussed accuracy.)
%Input:  x
%Output: approximation for sin(x)
function y=sin1(x)
%First calculate the interpolating polynomial and
% store coefficients
b=pi*(0:3)/6;yb=sin(b);    % b holds base points
c=newtdd(b,yb,4);
%For each input x, move x to the fundamental domain and evaluate
%      the interpolating polynomial
s=1;                        % Correct the sign of sin
x1=mod(x,2*pi);
if x1>pi
    x1 = 2*pi-x1;
    s = -1;
end
if x1 > pi/2
    x1 = pi-x1;
end
y = s*nest(3,c,x1,b);

```

Most of the work in Program 3.3 is to place x into the fundamental domain. Then we evaluate the degree 3 polynomial by nested multiplication. Here is some typical output from Program 3.3:

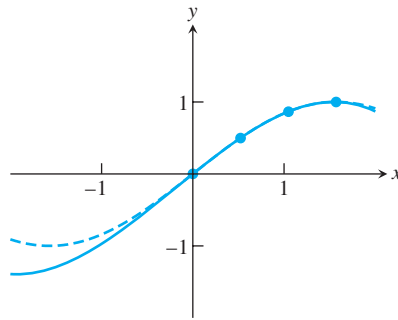


Figure 3.3 Degree 3 interpolation of $\sin x$. The interpolation polynomial (solid curve) is plotted along with $y = \sin x$. Equally spaced interpolation nodes are at $0, \pi/6, 2\pi/6$, and $3\pi/6$. The approximation is very close between 0 and $\pi/2$.

x	$\sin x$	$\sin 1(x)$	error
1	0.8415	0.8411	0.0004
2	0.9093	0.9102	0.0009
3	0.1411	0.1428	0.0017
4	-0.7568	-0.7557	0.0011
14	0.9906	0.9928	0.0022
1000	0.8269	0.8263	0.0006

This is not bad for the first try. The error is usually under 1 percent. In order to get enough correct digits to fill the calculator readout, we'll need to know a little more about interpolation error, the topic of the next section.

3.1 Exercises

- Use Lagrange interpolation to find a polynomial that passes through the points.
 - $(0, 1), (2, 3), (3, 0)$
 - $(-1, 0), (2, 1), (3, 1), (5, 2)$
 - $(0, -2), (2, 1), (4, 4)$
- Use Newton's divided differences to find the interpolating polynomials of the points in Exercise 1, and verify agreement with the Lagrange interpolating polynomial.
- How many degree d polynomials pass through the four points $(-1, 3), (1, 1), (2, 3), (3, 7)$? Write one down if possible. (a) $d = 2$ (b) $d = 3$ (c) $d = 6$.
- (a) Find a polynomial $P(x)$ of degree 3 or less whose graph passes through the points $(0, 0), (1, 1), (2, 2), (3, 7)$. (b) Find two other polynomials (of any degree) that pass through these four points. (c) Decide whether there exists a polynomial $P(x)$ of degree 3 or less whose graph passes through the points $(0, 0), (1, 1), (2, 2), (3, 7)$, and $(4, 2)$.
- (a) Find a polynomial $P(x)$ of degree 3 or less whose graph passes through the four data points $(-2, 8), (0, 4), (1, 2), (3, -2)$. (b) Describe any other polynomials of degree 4 or less which pass through the four points in part (a).
- Write down a polynomial of degree exactly 5 that interpolates the four points $(1, 1), (2, 3), (3, 3), (4, 4)$.

7. Find $P(0)$, where $P(x)$ is the degree 10 polynomial that is zero at $x = 1, \dots, 10$ and satisfies $P(12) = 44$.
8. Let $P(x)$ be the degree 9 polynomial that takes the value 112 at $x = 1$, takes the value 2 at $x = 10$, and equals zero for $x = 2, \dots, 9$. Calculate $P(0)$.
9. Give an example of the following, or explain why no such example exists. (a) A degree 6 polynomial $L(x)$ that is zero at $x = 1, 2, 3, 4, 5, 6$ and equal to 10 at $x = 7$. (b) A degree 6 polynomial $L(x)$ that is zero at $x = 1, 2, 3, 4, 5, 6$, equal to 10 at $x = 7$, and equal to 70 at $x = 8$.
10. Let $P(x)$ be the degree 5 polynomial that takes the value 10 at $x = 1, 2, 3, 4, 5$ and the value 15 at $x = 6$. Find $P(7)$.
11. Let P_1, P_2, P_3 , and P_4 be four different points lying on a parabola $y = ax^2 + bx + c$. How many cubic (degree 3) polynomials pass through those four points? Explain your answer.
12. Can a degree 3 polynomial intersect a degree 4 polynomial in exactly five points? Explain.
13. Let $P(x)$ be the degree 10 polynomial through the 11 points $(-5, 5), (-4, 5), (-3, 5), (-2, 5), (-1, 5), (0, 5), (1, 5), (2, 5), (3, 5), (4, 5), (5, 42)$. Calculate $P(6)$.
14. Write down 4 noncollinear points $(1, y_1), (2, y_2), (3, y_3), (4, y_4)$ that do not lie on any polynomial $y = P_3(x)$ of degree exactly three.
15. Write down the degree 25 polynomial that passes through the points $(1, -1), (2, -2), \dots, (25, -25)$ and has constant term equal to 25.
16. List all degree 42 polynomials that pass through the eleven points $(-5, 5), (-4, 4), \dots, (4, -4), (5, -5)$ and have constant term equal to 42.
17. The estimated mean atmospheric concentration of carbon dioxide in earth's atmosphere is given in the table that follows, in parts per million by volume. Find the degree 3 interpolating polynomial of the data and use it to estimate the CO_2 concentration in (a) 1950 and (b) 2050. (The actual concentration in 1950 was 310 ppm.)

year	CO_2 (ppm)
1800	280
1850	283
1900	291
2000	370

18. The expected lifetime of an industrial fan when operated at the listed temperature is shown in the table that follows. Estimate the lifetime at 70°C by using (a) the parabola from the last three data points (b) the degree 3 curve using all four points.

temp ($^\circ\text{C}$)	hrs ($\times 1000$)
25	95
40	75
50	63
60	54

3.1 Computer Problems

1. Apply the following world population figures to estimate the 1980 population, using (a) the straight line through the 1970 and 1990 estimates; (b) the parabola through the 1960, 1970, and 1990 estimates; and (c) the cubic curve through all four data points. Compare with the 1980 estimate of 4452584592.

year	population
1960	3039585530
1970	3707475887
1990	5281653820
2000	6079603571

2. Write a version of Program 3.2 that is a MATLAB function, whose inputs x and y are equal length vectors of data points, and whose output is a plot of the interpolating polynomial. In this way, the points can be entered more accurately than by mouse input. Check your program by replicating Figure 3.2.
3. Write a MATLAB function `polyinterp.m` that takes as input a set of (x, y) interpolating points and another x_0 , and outputs y_0 , the value of the interpolating polynomial at x_0 . The first line of the file should be `function y0 = polyinterp(x, y, x0)`, where x and y are input vectors of data points. Your function may call `newtdd` from Program 3.1 and `nest` from Chapter 0, and may be structured similarly to Program 3.2, but without the graphics. Demonstrate that your function works.
4. Remodel the `sin1` calculator key in Program 3.3 to build `cos1`, a cosine key that follows the same principles. First decide on the fundamental domain for cosine.
5. (a) Use the addition formulas for \sin and \cos to prove that $\tan(\pi/2 - x) = 1/\tan x$. (b) Show that $[0, \pi/4]$ can be used as a fundamental domain for $\tan x$. (c) Design a tangent key, following the principles of Program 3.3, using degree 3 polynomial interpolation on this fundamental domain. (d) Empirically calculate the maximum error of the tangent key in $[0, \pi/4]$.

3.2 INTERPOLATION ERROR

The accuracy of our `sin` calculator key depends on the approximation in Figure 3.3. How close is it? We presented a table indicating that, for a few examples, the first two digits are fairly reliable, but after that the digits are not always correct. In this section, we investigate ways to measure this error and determine how to make it smaller.

3.2.1 Interpolation error formula

Assume that we start with a function $y = f(x)$ and take data points from it to build an interpolating polynomial $P(x)$, as we did with $f(x) = \sin x$ in Example 3.7. The **interpolation error** at x is $f(x) - P(x)$, the difference between the original function that provided the data points and the interpolating polynomial, evaluated at x . The interpolation error is the vertical distance between the curves in Figure 3.3. The next theorem gives a formula for the interpolation error that is usually impossible to evaluate exactly, but often can at least lead to an error bound.

THEOREM 3.4 Assume that $P(x)$ is the (degree $n - 1$ or less) interpolating polynomial fitting the n points $(x_1, y_1), \dots, (x_n, y_n)$. The interpolation error is

$$f(x) - P(x) = \frac{(x - x_1)(x - x_2) \cdots (x - x_n)}{n!} f^{(n)}(c), \quad (3.6)$$

where c lies between the smallest and largest of the numbers x, x_1, \dots, x_n . ■

See Section 3.2.2 for a proof of Theorem 3.3. We can use the theorem to assess the accuracy of the \sin key we built in Example 3.7. Equation (3.6) yields

$$\sin x - P(x) = \frac{(x - 0)(x - \frac{\pi}{6})(x - \frac{\pi}{3})(x - \frac{\pi}{2})}{4!} f^{(4)}(c),$$

where $0 < c < \pi/2$. The fourth derivative $f^{(4)}(c) = \sin c$ varies from 0 to 1 in this range. At worst, $|\sin c|$ is no more than 1, so we can be assured of an upper bound on interpolation error:

$$|\sin x - P(x)| \leq \frac{|(x - 0)(x - \frac{\pi}{6})(x - \frac{\pi}{3})(x - \frac{\pi}{2})|}{24} |1|.$$

At $x = 1$, the worst-case error is

$$|\sin 1 - P(1)| \leq \frac{|(1 - 0)(1 - \frac{\pi}{6})(1 - \frac{\pi}{3})(1 - \frac{\pi}{2})|}{24} |1| \approx 0.0005348. \quad (3.7)$$

This is an upper bound for the error, since we used a “worst case” bound for the fourth derivative. Note that the actual error at $x = 1$ was .0004, which is within the error bound given by (3.7). We can make some conclusions on the basis of the form of the interpolation error formula. We expect smaller errors when x is closer to the middle of the interval of x_i 's than when it is near one of the ends, because there will be more small terms in the product. For example, we compare the preceding error bound to the case $x = 0.2$, which is near the left end of the range of data points. In this case, the error formula is

$$|\sin 0.2 - P(0.2)| \leq \frac{|(.2 - 0)(.2 - \frac{\pi}{6})(.2 - \frac{\pi}{3})(.2 - \frac{\pi}{2})|}{24} |1| \approx 0.00313,$$

about six times larger. Correspondingly, the actual error is larger, specifically,

$$|\sin 0.2 - P(0.2)| = |0.19867 - 0.20056| = 0.00189.$$

► **EXAMPLE 3.8** Find an upper bound for the difference at $x = 0.25$ and $x = 0.75$ between $f(x) = e^x$ and the polynomial that interpolates it at the points $-1, -0.5, 0, 0.5, 1$.

Construction of the interpolating polynomial, shown in Figure 3.4, is not necessary to find the bound. The interpolation error formula (3.6) gives

$$f(x) - P_4(x) = \frac{(x + 1)(x + \frac{1}{2})x(x - \frac{1}{2})(x - 1)}{5!} f^{(5)}(c),$$

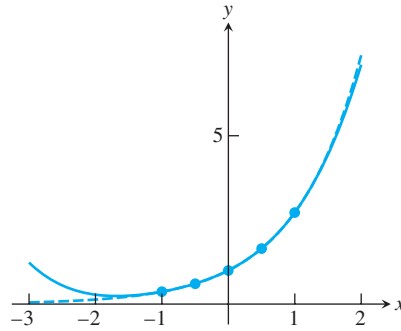


Figure 3.4 Interpolating Polynomial for Approximating $f(x) = e^x$. Equally spaced base points $-1, -0.5, 0, 0.5, 1$. The solid curve is the interpolating polynomial.

where $-1 < c < 1$. The fifth derivative is $f^{(5)}(c) = e^c$. Since e^x is increasing with x , its maximum is at the right-hand end of the interval, so $|f^{(5)}| \leq e^1$ on $[-1, 1]$. For $-1 \leq x \leq 1$, the error formula becomes

$$|e^x - P_4(x)| \leq \frac{(x+1)\left(x+\frac{1}{2}\right)x\left(x-\frac{1}{2}\right)(x-1)}{5!}e.$$

At $x = 0.25$, the interpolation error has the upper bound

$$|e^{0.25} - P_4(0.25)| \leq \frac{(1.25)(0.75)(0.25)(-0.25)(-0.75)}{120}e \approx .000995.$$

At $x = 0.75$, the interpolation error is potentially larger:

$$|e^{0.75} - P_4(0.75)| \leq \frac{(1.75)(1.25)(0.75)(0.25)(0.25)}{120}e \approx .002323.$$

Note again that the interpolation error will tend to be smaller close to the center of the interpolation interval. ▶

3.2.2 Proof of Newton form and error formula

In this section, we explain the reasoning behind two important facts used earlier. First we establish the Newton's divided difference form of the interpolating polynomial, and then we prove the interpolation error formula.

Recall what we know so far. If x_1, \dots, x_n are n distinct points on the real line and y_1, \dots, y_n are arbitrary, we know by Theorem 3.2 that there is exactly one (degree at most $n - 1$) interpolating polynomial $P_{n-1}(x)$ for these points. We also know that the Lagrange interpolating formula gives such a polynomial.

We are missing the proof that the Newton's divided difference formula also gives an interpolating polynomial. Once we prove that it does in Theorem 3.5, we will know it must agree with the Lagrange version.

Let $P(x)$ denote the (unique) polynomial that interpolates $(x_1, f(x_1)), \dots, (x_n, f(x_n))$, and as in Definition 3.3, denote by $f[x_1 \dots x_n]$ the degree $n - 1$ coefficient of $P(x)$. Thus $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$, where $a_{n-1} = f[x_1 \dots x_n]$, and two facts are readily apparent.

FACT 1 $f[x_1 \dots x_n] = f[\sigma(x_1) \dots \sigma(x_n)]$ for any permutation σ of the x_i . □

Proof. Clear by uniqueness of the interpolating polynomial, proved in Theorem 3.2. □

FACT 2 $P(x)$ can be written in the form

$$P(x) = c_0 + c_1(x - x_1) + c_2(x - x_1)(x - x_2) + \dots + c_{n-1}(x - x_1) \cdots (x - x_{n-1}).$$

□

Proof. Clearly we should choose $c_{n-1} = a_{n-1}$. The remaining $c_{n-2}, c_{n-3}, \dots, c_0$ are defined recursively by setting c_k to be the degree k coefficient of the (degree at most k) polynomial

$$P(x) - c_{n-1}(x - x_1) \cdots (x - x_{n-1}) - c_{n-2}(x - x_1) \cdots (x - x_{n-2}) \\ - \dots - c_{k+1}(x - x_1) \cdots (x - x_{k+1}).$$

(This is a degree at most k polynomial due to the choice of c_{k+1} .) □

THEOREM 3.5 Let $P(x)$ be the interpolating polynomial of $(x_1, f(x_1)), \dots, (x_n, f(x_n))$ where the x_i are distinct. Then

$$(a) P(x) = f[x_1] + f[x_1 x_2](x - x_1) + f[x_1 x_2 x_3](x - x_1)(x - x_2) + \dots \\ + f[x_1 x_2 \dots x_n](x - x_1)(x - x_2) \cdots (x - x_{n-1}), \text{ and}$$

$$(b) \text{ for } k > 1, f[x_1 \dots x_k] = \frac{f[x_2 \dots x_k] - f[x_1 \dots x_{k-1}]}{x_k - x_1}.$$

■

Proof. (a) We must prove that $c_{k-1} = f[x_1 \dots x_k]$ for $k = 1, \dots, n$. It is already clear for $k = n$ by definition. In general, successively substitute x_1, \dots, x_k into the form of $P(x)$ in Fact 2. Only the first k terms are nonzero. We conclude that the polynomial consisting of the first k terms of $P(x)$ suffice to interpolate x_1, \dots, x_k , and so by Definition 3.2 and the uniqueness of interpolating polynomial, $c_{k-1} = f[x_1 \dots x_k]$. (b) According to (a), the interpolating polynomial of $x_2, x_3, \dots, x_{k-1}, x_1, x_k$ is

$$P_1(x) = f[x_2] + f[x_2 x_3](x - x_2) + \dots + f[x_2 x_3 \dots x_{k-1} x_1](x - x_2) \cdots (x - x_{k-1}) \\ + f[x_2 x_3 \dots x_{k-1} x_1 x_k](x - x_2) \cdots (x - x_{k-1})(x - x_1)$$

and the interpolating polynomial of $x_2, x_3, \dots, x_{k-1}, x_k, x_1$ is

$$P_2(x) = f[x_2] + f[x_2 x_3](x - x_2) + \dots + f[x_2 x_3 \dots x_{k-1} x_k](x - x_2) \cdots (x - x_{k-1}) \\ + f[x_2 x_3 \dots x_{k-1} x_k x_1](x - x_2) \cdots (x - x_{k-1})(x - x_k).$$

By uniqueness, $P_1 = P_2$. Setting $P_1(x_k) = P_2(x_k)$ and canceling terms yields

$$f[x_2 \dots x_{k-1} x_1](x_k - x_2) \cdots (x_k - x_{k-1}) + f[x_2 \dots x_{k-1} x_1 x_k](x_k - x_2) \\ \cdots (x_k - x_{k-1})(x_k - x_1) = f[x_2 \dots x_k](x_k - x_2) \cdots (x_k - x_{k-1})$$

or

$$f[x_2 \dots x_{k-1} x_1] + f[x_2 \dots x_{k-1} x_1 x_k](x_k - x_1) = f[x_2 \dots x_k].$$

Using Fact 1, this can be rearranged to

$$f[x_1 \dots x_k] = \frac{f[x_2 \dots x_k] - f[x_1 \dots x_{k-1}]}{x_k - x_1}.$$

□

Next we prove the Interpolation Error Theorem 3.4. Consider adding one more point x to the set of interpolation points. The new interpolation polynomial would be

$$P_n(t) = P_{n-1}(t) + f[x_1 \dots x_n x](t - x_1) \cdots (t - x_n).$$

Evaluated at the extra point x , $P_n(x) = f(x)$, so

$$f(x) = P_{n-1}(x) + f[x_1 \dots x_n x](x - x_1) \cdots (x - x_n). \quad (3.8)$$

This formula is true for all x . Now define

$$h(t) = f(t) - P_{n-1}(t) - f[x_1 \dots x_n x](t - x_1) \cdots (t - x_n).$$

Note that $h(x) = 0$ by (3.8) and $0 = h(x_1) = \cdots = h(x_n)$ because P_{n-1} interpolates f at these points. Between each neighboring pair of the $n + 1$ points x, x_1, \dots, x_n , there must be a new point where $h' = 0$, by Rolle's Theorem (see Chapter 0). There are n of these points. Between each pair of these, there must be a new point where $h'' = 0$; there are $n - 1$ of these. Continuing in this way, there must be one point c for which $h^{(n)}(c) = 0$, where c lies between the smallest and largest of x, x_1, \dots, x_n . Note that

$$h^{(n)}(t) = f^{(n)}(t) - n! f[x_1 \dots x_n x],$$

because the n th derivative of the polynomial $P_{n-1}(t)$ is zero. Substituting c gives

$$f[x_1 \dots x_n x] = \frac{f^{(n)}(c)}{n!},$$

which leads to

$$f(x) = P_{n-1}(x) + \frac{f^{(n)}(c)}{n!} (x - x_1) \cdots (x - x_n),$$

using (3.8).

3.2.3 Runge phenomenon

Polynomials can fit any set of data points, as Theorem 3.2 shows. However, there are some shapes that polynomials prefer over others. You can achieve a better understanding of this point by playing with Program 3.2. Try data points that cause the function to be zero at equally spaced points $x = -3, -2.5, -2, -1.5, \dots, 2.5, 3$, except for $x = 0$, where we set a value of 1. The data points are flat along the x -axis, except for a triangular “bump” at $x = 0$, as shown in Figure 3.5.

The polynomial that goes through points situated like this refuses to stay between 0 and 1, unlike the data points. This is an illustration of the so-called **Runge phenomenon**. It is usually used to describe extreme “polynomial wiggle” associated with high-degree polynomial interpolation at evenly spaced points.

► **EXAMPLE 3.9** Interpolate $f(x) = 1/(1 + 12x^2)$ at evenly spaced points in $[-1, 1]$.

This is called the **Runge example**. The function has the same general shape as the triangular bump in Figure 3.5. Figure 3.6 shows the result of the interpolation, behavior

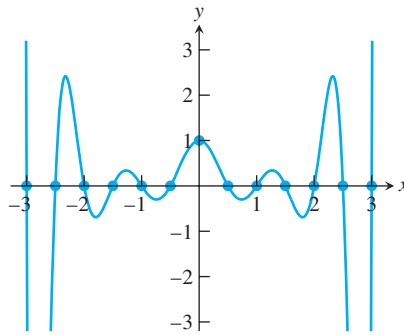


Figure 3.5 Interpolation of Triangular Bump Function. The interpolating polynomial wiggles much more than the input data points.

that is characteristic of the Runge phenomenon: polynomial wiggle near the ends of the interpolation interval. ▶

As we have seen, examples with the Runge phenomenon characteristically have large error near the outside of the interval of data points. The cure for this problem is intuitive: Move some of the interpolation points toward the outside of the interval, where the function producing the data can be better fit. We will see how to accomplish this in the next section on Chebyshev interpolation.

3.2 Exercises

- Find the degree 2 interpolating polynomial $P_2(x)$ through the points $(0, 0)$, $(\pi/2, 1)$, and $(\pi, 0)$.
 - Calculate $P_2(\pi/4)$, an approximation for $\sin(\pi/4)$.
 - Use Theorem 3.3 to give an error bound for the approximation in part (b).
 - Using a calculator or MATLAB, compare the actual error to your error bound.
- Given the data points $(1, 0)$, $(2, \ln 2)$, $(4, \ln 4)$, find the degree 2 interpolating polynomial.
 - Use the result of (a) to approximate $\ln 3$.
 - Use Theorem 3.3 to give an error bound for the approximation in part (b).
 - Compare the actual error to your error bound.
- Assume that the polynomial $P_9(x)$ interpolates the function $f(x) = e^{-2x}$ at the 10 evenly spaced points $x = 0, 1/9, 2/9, 3/9, \dots, 8/9, 1$.

 - Find an upper bound for the error $|f(1/2) - P_9(1/2)|$.
 - How many decimal places can you guarantee to be correct if $P_9(1/2)$ is used to approximate e ?
- Consider the interpolating polynomial for $f(x) = 1/(x + 5)$ with interpolation nodes $x = 0, 2, 4, 6, 8, 10$. Find an upper bound for the interpolation error at

 - $x = 1$ and
 - $x = 5$.
- Assume that a function $f(x)$ has been approximated by the degree 5 interpolating polynomial $P(x)$, using the data points $(x_i, f(x_i))$, where $x_1 = .1$, $x_2 = .2$, $x_3 = .3$, $x_4 = .4$, $x_5 = .5$, $x_6 = .6$. Do you expect the interpolation error $|f(x) - P(x)|$ to be smaller for $x = .35$ or for $x = .55$? Quantify your answer.
- Assume that the polynomial $P_5(x)$ interpolates a function $f(x)$ at the six data points $(x_i, f(x_i))$ with x -coordinates $x_1 = 0$, $x_2 = .2$, $x_3 = .4$, $x_4 = .6$, $x_5 = .8$, and $x_6 = 1$. Assume that the interpolation error at $x = .3$ is $|f(.3) - P_5(.3)| = .01$. Estimate the new interpolation error

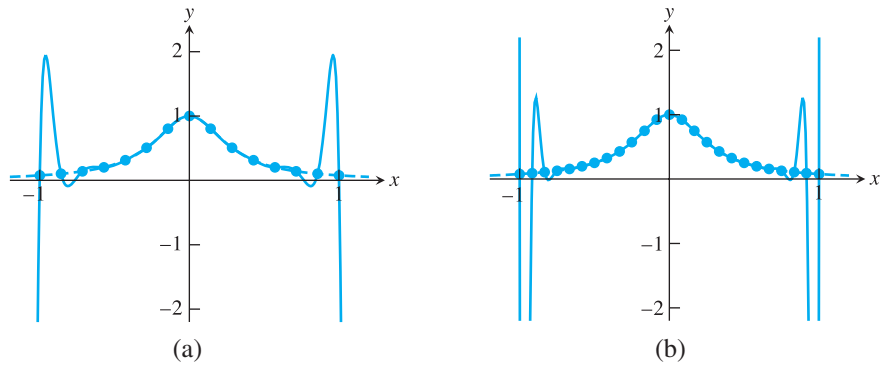


Figure 3.6 Runge Example. Polynomial interpolation of the Runge function of Example 3.9 at evenly spaced base points causes extreme variation near the ends of the interval, similar to Figure 3.5 (a) 15 base points (b) 25 base points.

$|f(.3) - P_7(.3)|$ that would result if two additional interpolation points $(x_6, y_6) = (.1, f(.1))$ and $(x_7, y_7) = (.5, f(.5))$ are added. What assumptions have you made to produce this estimate?

3.2 Computer Problems

- (a) Use the method of divided differences to find the degree 4 interpolating polynomial $P_4(x)$ for the data $(0.6, 1.433329)$, $(0.7, 1.632316)$, $(0.8, 1.896481)$, $(0.9, 2.247908)$, and $(1.0, 2.718282)$. (b) Calculate $P_4(0.82)$ and $P_4(0.98)$. (c) The preceding data come from the function $f(x) = e^{x^2}$. Use the interpolation error formula to find upper bounds for the error at $x = 0.82$ and $x = 0.98$, and compare the bounds with the actual error. (d) Plot the actual interpolation error $P(x) - e^{x^2}$ on the intervals $[.5, 1]$ and $[0, 2]$.
- Plot the interpolation error of the `sin1` key from Program 3.3 on the interval $[-2\pi, 2\pi]$.
- The total world oil production in millions of barrels per day is shown in the table that follows. Determine and plot the degree 9 polynomial through the data. Use it to estimate 2010 oil production. Does the Runge phenomenon occur in this example? In your opinion, is the interpolating polynomial a good model of the data? Explain.

year	bbl/day ($\times 10^6$)
1994	67.052
1995	68.008
1996	69.803
1997	72.024
1998	73.400
1999	72.063
2000	74.669
2001	74.487
2002	74.065
2003	76.777

- Use the degree 3 polynomial through the first four data points in Computer Problem 3 to estimate the 1998 world oil production. Is the Runge phenomenon present?

3.3 CHEBYSHEV INTERPOLATION

It is common to choose the base points x_i for interpolation to be evenly spaced. In many cases, the data to be interpolated are available only in that form—for example, when the data consist of instrument readings separated by a constant time interval. In other cases—for instance, the sine key—we are free to choose the base points as we see fit. It turns out that the choice of base point spacing can have a significant effect on the interpolation error. Chebyshev interpolation refers to a particular optimal way of spacing the points.

3.3.1 Chebyshev's theorem

The motivation for Chebyshev interpolation is to improve control of the maximum value of the interpolation error

$$\frac{(x - x_1)(x - x_2) \cdots (x - x_n)}{n!} f^{(n)}(c)$$

on the interpolation interval. Let's fix the interval to be $[-1, 1]$ for now.

The numerator

$$(x - x_1)(x - x_2) \cdots (x - x_n) \quad (3.9)$$

of the interpolation error formula is itself a degree n polynomial in x and has some maximum value on $[-1, 1]$. Is it possible to find particular x_1, \dots, x_n in $[-1, 1]$ that cause the maximum value of (3.9) to be as small as possible? This is called the minimax problem of interpolation.

For example, Figure 3.7(a) shows a plot of the degree 9 polynomial (3.9) when x_1, \dots, x_9 are evenly spaced. The tendency for this polynomial to be large near the ends of the interval $[-1, 1]$ is a manifestation of the Runge phenomenon. Figure 3.7(b) shows the same polynomial (3.9), but where the points x_1, \dots, x_9 have been chosen in a way that equalizes the size of the polynomial throughout $[-1, 1]$. The points have been chosen according to Theorem 3.8, presented shortly.

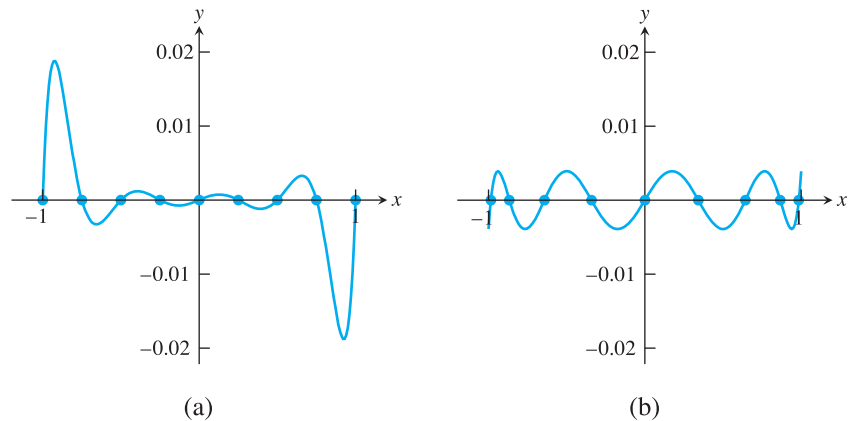


Figure 3.7 Part of the Interpolation Error Formula. Plots of $(x - x_1) \cdots (x - x_9)$ for (a) nine evenly spaced base points x_i (b) nine Chebyshev roots x_i .

In fact, this precise positioning, in which the base points x_i are chosen to be $\cos \frac{\pi}{18}, \cos \frac{3\pi}{18}, \dots, \cos \frac{17\pi}{18}$, makes the maximum absolute value of (3.9) equal to $1/256$, the minimum possible for nine points on the interval $[-1, 1]$. Such positioning, due to Chebyshev, is summarized in the following theorem:

THEOREM 3.6 The choice of real numbers $-1 \leq x_1, \dots, x_n \leq 1$ that makes the value of

$$\max_{-1 \leq x \leq 1} |(x - x_1) \cdots (x - x_n)|$$

as small as possible is

$$x_i = \cos \frac{(2i - 1)\pi}{2n} \quad \text{for } i = 1, \dots, n,$$

and the minimum value is $1/2^{n-1}$. In fact, the minimum is achieved by

$$(x - x_1) \cdots (x - x_n) = \frac{1}{2^{n-1}} T_n(x),$$

where $T_n(x)$ denotes the degree n Chebyshev polynomial. ■

The proof of this theorem is given later, after we establish a few properties of Chebyshev polynomials. We conclude from the theorem that interpolation error can be minimized if the n interpolation base points in $[-1, 1]$ are chosen to be the roots of the degree n Chebyshev interpolating polynomial $T_n(x)$. These roots are

$$x_i = \cos \frac{\text{odd } \pi}{2n} \tag{3.10}$$

where “odd” stands for the odd numbers from 1 to $2n - 1$. Then we are guaranteed that the absolute value of (3.9) is less than $1/2^{n-1}$ for all x in $[-1, 1]$.

Choosing the Chebyshev roots as the base points for interpolation distributes the interpolation error as evenly as possible across the interval $[-1, 1]$. We will call the interpolating polynomial that uses the Chebyshev roots as base points the **Chebyshev interpolating polynomial**.

► **EXAMPLE 3.10** Find a worst-case error bound for the difference on $[-1, 1]$ between $f(x) = e^x$ and the degree 4 Chebyshev interpolating polynomial.

The interpolation error formula (3.6) gives

$$f(x) - P_4(x) = \frac{(x - x_1)(x - x_2)(x - x_3)(x - x_4)(x - x_5)}{5!} f^{(5)}(c),$$

where

$$x_1 = \cos \frac{\pi}{10}, \quad x_2 = \cos \frac{3\pi}{10}, \quad x_3 = \cos \frac{5\pi}{10}, \quad x_4 = \cos \frac{7\pi}{10}, \quad x_5 = \cos \frac{9\pi}{10}$$

are the Chebyshev roots and where $-1 < c < 1$. According to the Chebyshev Theorem 3.6, for $-1 \leq x \leq 1$,

$$|(x - x_1) \cdots (x - x_5)| \leq \frac{1}{2^4}.$$

In addition, $|f^{(5)}| \leq e^1$ on $[-1, 1]$. The interpolation error is

$$|e^x - P_4(x)| \leq \frac{e}{2^4 5!} \approx 0.00142$$

for all x in the interval $[-1, 1]$.

Compare this result with Example 3.8. The error bound for Chebyshev interpolation for the entire interval is only slightly larger than the bound for a point near the center of the interval, when evenly spaced interpolation is used. Near the ends of the interval, the Chebyshev error is much smaller. ◀

Returning to the Runge Example 3.9, we can eliminate the Runge phenomenon by choosing the interpolation points according to Chebyshev’s idea. Figure 3.8 shows that the interpolation error is made small throughout the interval $[-1, 1]$.

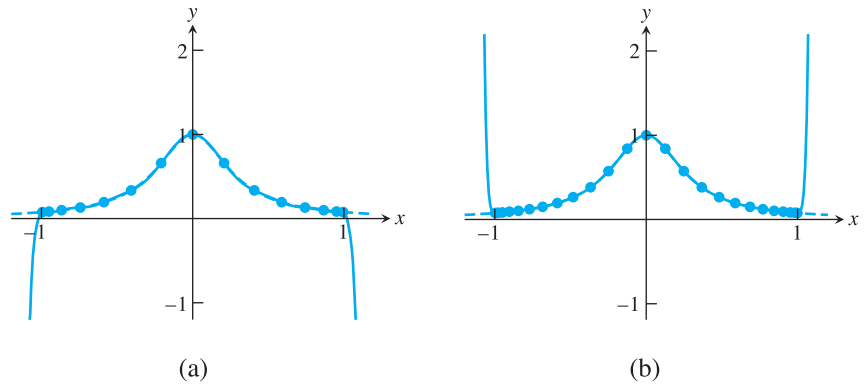


Figure 3.8 Interpolation of Runge Example with Chebyshev nodes. The Runge function $f(x)=1/(1+12x^2)$ is graphed along with its Chebyshev interpolation polynomial for (a) 15 points (b) 25 points. The error on $[-1, 1]$ is negligible at this resolution. The polynomial wiggle of Figure 3.6 has vanished, at least between -1 and 1 .

3.3.2 Chebyshev polynomials

Define the n th **Chebyshev polynomial** by $T_n(x) = \cos(n \arccos x)$. Despite its appearance, it is a polynomial in the variable x for each n . For example, for $n = 0$ it gives the degree 0 polynomial 1, and for $n = 1$ we get $T_1(x) = \cos(\arccos x) = x$. For $n = 2$, recall the cosine addition formula $\cos(a + b) = \cos a \cos b - \sin a \sin b$. Set $y = \arccos x$, so that $\cos y = x$. Then $T_2(x) = \cos 2y = \cos^2 y - \sin^2 y = 2\cos^2 y - 1 = 2x^2 - 1$, a degree 2 polynomial. In general, note that

$$\begin{aligned} T_{n+1}(x) &= \cos(n+1)y = \cos(ny + y) = \cos ny \cos y - \sin ny \sin y \\ T_{n-1}(x) &= \cos(n-1)y = \cos(ny - y) = \cos ny \cos y - \sin ny \sin(-y). \end{aligned} \quad (3.11)$$

Because $\sin(-y) = -\sin y$, we can add the preceding equations to get

$$T_{n+1}(x) + T_{n-1}(x) = 2\cos ny \cos y = 2x T_n(x). \quad (3.12)$$

The resulting relation,

$$T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x), \quad (3.13)$$

is called the **recursion relation** for the Chebyshev polynomials. Several facts follow from (3.13):

FACT 1 The T_n 's are polynomials. We showed this explicitly for T_0, T_1 , and T_2 . Since T_3 is a polynomial combination of T_1 and T_2 , T_3 is also a polynomial. The same argument goes for all T_n . The first few Chebyshev polynomials (see Figure 3.9) are

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x. \end{aligned} \quad \square$$

FACT 2 $\deg(T_n) = n$, and the leading coefficient is 2^{n-1} . This is clear for $n = 1$ and 2 , and the recursion relation extends the fact to all n . \square

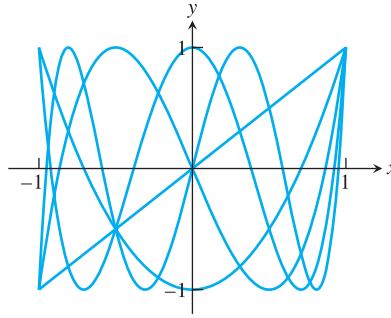


Figure 3.9 Plot of the Degree 1 through 5 Chebyshev Polynomials. Note that $T_n(1) = 1$ and the maximum absolute value taken on by $T_n(x)$ inside $[-1, 1]$ is 1.

FACT 3 $T_n(1) = 1$ and $T_n(-1) = (-1)^n$. Both are clear for $n = 1$ and 2. In general,

$$T_{n+1}(1) = 2(1)T_n(1) - T_{n-1}(1) = 2(1) - 1 = 1$$

and

$$\begin{aligned} T_{n+1}(-1) &= 2(-1)T_n(-1) - T_{n-1}(-1) \\ &= -2(-1)^n - (-1)^{n-1} \\ &= (-1)^{n-1}(2 - 1) = (-1)^{n-1} = (-1)^{n+1}. \end{aligned}$$

□

FACT 4 The maximum absolute value of $T_n(x)$ for $-1 \leq x \leq 1$ is 1. This follows immediately from the fact that $T_n(x) = \cos y$ for some y . □

FACT 5 All zeros of $T_n(x)$ are located between -1 and 1 . See Figure 3.10. In fact, the zeros are the solution of $0 = \cos(n \arccos x)$. Since $\cos y = 0$ if and only if $y = \text{odd integer} \cdot (\pi/2)$, we find that

$$\begin{aligned} n \arccos x &= \text{odd} \cdot \pi/2 \\ x &= \cos \frac{\text{odd} \cdot \pi}{2n}. \end{aligned}$$

□

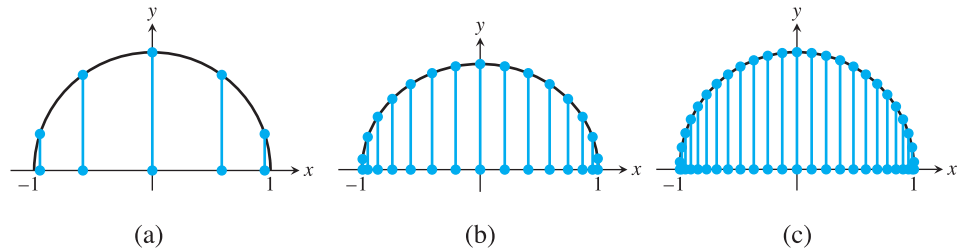


Figure 3.10 Location of Zeros of the Chebyshev Polynomial. The roots are the x -coordinates of evenly spaced points around the circle (a) degree 5 (b) degree 15 (c) degree 25.

FACT 6 $T_n(x)$ alternates between -1 and 1 a total of $n + 1$ times. In fact, this happens at $\cos 0, \cos \pi/n, \dots, \cos(n-1)\pi/n, \cos \pi$. □

It follows from Fact 2 that the polynomial $T_n(x)/2^{n-1}$ is monic (has leading coefficient 1). Since, according to Fact 5, all roots of $T_n(x)$ are real, we can write $T_n(x)/2^{n-1}$ in

factored form as $(x - x_1) \cdots (x - x_n)$ where the x_i are the Chebyshev nodes as described in Theorem 3.8.

Chebyshev's theorem follows directly from these facts.

Proof of Theorem 3.6. Let $P_n(x)$ be a monic polynomial with an even smaller absolute maximum on $[-1, 1]$; in other words, $|P_n(x)| < 1/2^{n-1}$ for $-1 \leq x \leq 1$. This assumption leads to a contradiction. Since $T_n(x)$ alternates between -1 and 1 a total of $n + 1$ times (Fact 6), at these $n + 1$ points the difference $P_n - T_n/2^{n-1}$ is alternately positive and negative. Therefore, $P_n - T_n/2^{n-1}$ must cross zero at least n times; that is, it must have at least n roots. This contradicts the fact that, because P_n and $T_n/2^{n-1}$ are monic, their difference is of degree $\leq n - 1$.

3.3.3 Change of interval

So far our discussion of Chebyshev interpolation has been restricted to the interval $[-1, 1]$, because Theorem 3.6 is most easily stated for this interval. Next, we will move the whole methodology to a general interval $[a, b]$.

The base points are moved so that they have the same relative positions in $[a, b]$ that they had in $[-1, 1]$. It is best to think of doing this in two steps: (1) *Stretch* the points by the factor $(b - a)/2$ (the ratio of the two interval lengths), and (2) *Translate* the points by $(b + a)/2$ to move the center of mass from 0 to the midpoint of $[a, b]$. In other words, move from the original points

$$\cos \frac{\text{odd } \pi}{2n}$$

to

$$\frac{b - a}{2} \cos \frac{\text{odd } \pi}{2n} + \frac{b + a}{2}.$$

With the new Chebyshev base points x_1, \dots, x_n in $[a, b]$, the corresponding upper bound on the numerator of the interpolation error formula is changed due to the stretch by $(b - a)/2$ on each factor $x - x_i$. As a result, the minimax value $1/2^{n-1}$ must be replaced by $[(b - a)/2]^n / 2^{n-1}$.

Chebyshev interpolation nodes

On the interval $[a, b]$,

$$x_i = \frac{b + a}{2} + \frac{b - a}{2} \cos \frac{(2i - 1)\pi}{2n}$$

for $i = 1, \dots, n$. The inequality

$$|(x - x_1) \cdots (x - x_n)| \leq \frac{\left(\frac{b-a}{2}\right)^n}{2^{n-1}} \quad (3.14)$$

holds on $[a, b]$.

The next example illustrates the use of Chebyshev interpolation in a general interval.

► **EXAMPLE 3.11** Find the four Chebyshev base points for interpolation on the interval $[0, \pi/2]$, and find an upper bound for the Chebyshev interpolation error for $f(x) = \sin x$ on the interval.

SPOTLIGHT ON

Compression

As shown in this section, Chebyshev interpolation is a good way to turn general functions into a small number of floating point operations, for ease of computation. An upper bound for the error made is easily available, is usually smaller than for evenly spaced interpolation, and can be made as small as desired.

Although we have used the sine function to demonstrate this process, a different approach is taken to construct the actual “sine key” on most calculators and canned software. Special properties of the sine function allow it to be approximated by a simple Taylor expansion, slightly altered to take rounding effects into account. Because sine is an odd function, the even-numbered terms in its Taylor series around zero are missing, making it especially efficient to calculate.

This is a second attempt. We used evenly spaced base points in Example 3.7. The Chebyshev base points are

$$\frac{\frac{\pi}{2} - 0}{2} \cos\left(\frac{\text{odd } \pi}{2(4)}\right) + \frac{\frac{\pi}{2} + 0}{2},$$

or


$$x_1 = \frac{\pi}{4} + \frac{\pi}{4} \cos \frac{\pi}{8}, x_2 = \frac{\pi}{4} + \frac{\pi}{4} \cos \frac{3\pi}{8}, x_3 = \frac{\pi}{4} + \frac{\pi}{4} \cos \frac{5\pi}{8}, x_4 = \frac{\pi}{4} + \frac{\pi}{4} \cos \frac{7\pi}{8}.$$

From (3.14), the worst-case interpolation error for $0 \leq x \leq \pi/2$ is

$$\begin{aligned} |\sin x - P_3(x)| &= \frac{|(x - x_1)(x - x_2)(x - x_3)(x - x_4)|}{4!} |f^{(4)}(c)| \\ &\leq \frac{\left(\frac{\frac{\pi}{2} - 0}{2}\right)^4}{4! 2^3} 1 \approx 0.00198. \end{aligned}$$

The Chebyshev interpolating polynomial for this example is evaluated at several points in the following table:

x	$\sin x$	$P_3(x)$	error
1	0.8415	0.8408	0.0007
2	0.9093	0.9097	0.0004
3	0.1411	0.1420	0.0009
4	-0.7568	-0.7555	0.0013
14	0.9906	0.9917	0.0011
1000	0.8269	0.8261	0.0008

The interpolation errors are well below the worst-case estimate. Figure 3.11 plots the interpolation error as a function of x on the interval $[0, \pi/2]$, compared with the same for evenly spaced interpolation. The Chebyshev error (dashed curve) is a bit smaller and is distributed more evenly throughout the interpolation interval. 

► EXAMPLE 3.12 Design a sine key that will give output correct to 10 decimal places.

Thanks to our work earlier on setting up a fundamental domain for the sine function, we can continue to concentrate on the interval $[0, \pi/2]$. Repeat the previous calculation, but leave n , the number of base points, as an unknown to be determined. The maximum interpolation error for the polynomial $P_{n-1}(x)$ on the interval $[0, \pi/2]$ is

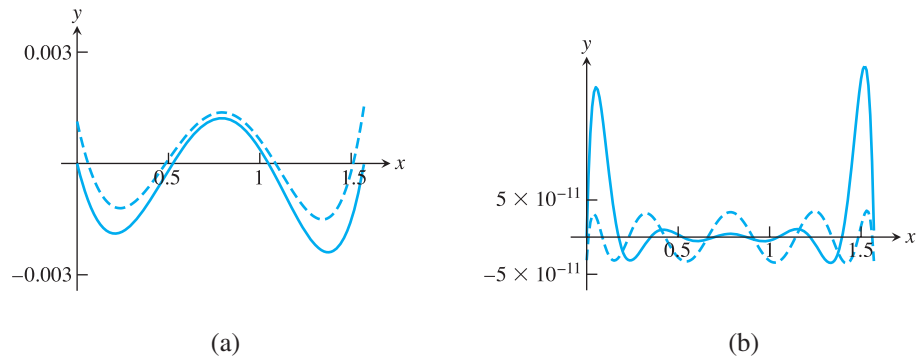


Figure 3.11 Interpolation error for approximating $f(x) = \sin x$. (a) Interpolation error for degree 3 interpolating polynomial with evenly spaced base points (solid curve) and Chebyshev base points (dashed curve). (b) Same as (a), but degree 9.

$$|\sin x - P_{n-1}(x)| = \frac{|(x - x_1) \cdots (x - x_n)|}{n!} |f^{(n)}(c)|$$

$$\leq \frac{\left(\frac{\pi/2 - 0}{2}\right)^n}{n! 2^{n-1}} 1.$$

This equation is not simple to solve for n , but a little trial and error finds that for $n = 9$ the error bound is $\approx 0.1224 \times 10^{-8}$, and for $n = 10$ it is $\approx 0.4807 \times 10^{-10}$. The latter meets our criterion for 10 correct decimal places. Figure 3.11(b) compares the actual error of the Chebyshev interpolation polynomial with the error of the evenly spaced interpolation polynomial.

The 10 Chebyshev base points on $[0, \pi/2]$ are $\pi/4 + (\pi/4)\cos(\text{odd } \pi/20)$. The key can be designed by storing the 10 y -values for sine at the base points and doing a nested multiplication evaluation for each key press. ◀

The following MATLAB code `sin2.m` carries out the preceding task. The code is a bit awkward as written: We have to do 10 sin evaluations, at the 10 Chebyshev nodes, in order to set up the interpolating polynomial to approximate sin at one point. Of course, in a real implementation, these numbers would be computed once and stored.

```
%Program 3.4 Building a sin calculator key, attempt #2
%Approximates sin curve with degree 9 polynomial
%Input: x
%Output: approximation for sin(x), correct to 10 decimal places
function y=sin2(x)
%First calculate the interpolating polynomial and
% store coefficients
n=10;
b=pi/4+(pi/4)*cos((1:2:2*n-1)*pi/(2*n));
yb=sin(b); % b holds Chebyshev base points
c=newtdd(b,yb,n);
%For each input x, move x to the fundamental domain and evaluate
% the interpolating polynomial
s=1; % Correct the sign of sin
x1=mod(x,2*pi);
if x1>pi
    x1 = 2*pi-x1;
    s = -1;
end
```

```

end
if x1 > pi/2
    x1 = pi-x1;
end
y = s*nest(n-1,c,x1,b);

```

In this chapter, we have often illustrated polynomial interpolation, either evenly spaced or using Chebyshev nodes, for the purpose of approximating the trigonometric functions. Although polynomial interpolation can be used to approximate sine and cosine to arbitrarily high accuracy, most calculators use a slightly more efficient approach called the CORDIC (Coordinate Rotation Digital Computer) algorithm (Volder [1959]). CORDIC is an elegant iterative method, based on complex arithmetic, that can be applied to several special functions. Polynomial interpolation remains a simple and useful technique for approximating general functions and for representing and compressing data.

3.3 Exercises

- List the Chebyshev interpolation nodes x_1, \dots, x_n in the given interval. (a) $[-1, 1]$, $n = 6$ (b) $[-2, 2]$, $n = 4$ (c) $[4, 12]$, $n = 6$ (d) $[-0.3, 0.7]$, $n = 5$
- Find the upper bound for $|(x - x_1) \dots (x - x_n)|$ on the intervals and Chebyshev nodes in Exercise 1.
- Assume that Chebyshev interpolation is used to find a fifth degree interpolating polynomial $Q_5(x)$ on the interval $[-1, 1]$ for the function $f(x) = e^x$. Use the interpolation error formula to find a worst-case estimate for the error $|e^x - Q_5(x)|$ that is valid for x throughout the interval $[-1, 1]$. How many digits after the decimal point will be correct when $Q_5(x)$ is used to approximate e^x ?
- Answer the same questions as in Exercise 3, but for the interval $[0.6, 1.0]$.
- Find an upper bound for the error on $[0, 2]$ when the degree 3 Chebyshev interpolating polynomial is used to approximate $f(x) = \sin x$.
- Assume that you are to use Chebyshev interpolation to find a degree 3 interpolating polynomial $Q_3(x)$ that approximates the function $f(x) = x^{-3}$ on the interval $[3, 4]$. (a) Write down the (x, y) points that will serve as interpolation nodes for Q_3 . (b) Find a worst-case estimate for the error $|x^{-3} - Q_3(x)|$ that is valid for all x in the interval $[3, 4]$. How many digits after the decimal point will be correct when $Q_3(x)$ is used to approximate x^{-3} ?
- Suppose you are designing the **ln** key for a calculator whose display shows six digits to the right of the decimal point. Find the least degree d for which Chebyshev interpolation on the interval $[1, e]$ will approximate within this accuracy.
- Let $T_n(x)$ denote the degree n Chebyshev polynomial. Find a formula for $T_n(0)$.
- Determine the following values: (a) $T_{999}(-1)$ (b) $T_{1000}(-1)$ (c) $T_{999}(0)$ (d) $T_{1000}(0)$ (e) $T_{999}(-1/2)$ (f) $T_{1000}(-1/2)$.

3.3 Computer Problems

- Rebuild Program 3.3 to implement the Chebyshev interpolating polynomial with four nodes on the interval $[0, \pi/2]$. (Only one line of code needs to be changed.) Then plot the polynomial and the sine function on the interval $[-2, 2]$.

2. Build a MATLAB program to evaluate the cosine function correct to 10 decimal places using Chebyshev interpolation. Start by interpolating on a fundamental domain $[0, \pi/2]$, and extend your answer to inputs between -10^4 and 10^4 . You may want to use some of the MATLAB code written in this chapter.
3. Carry out the steps of Computer Problem 2 for $\ln x$, for inputs x between 10^{-4} and 10^4 . Use $[1, e]$ as the fundamental domain. What is the degree of the interpolation polynomial that guarantees 10 correct digits? Your program should begin by finding the integer k such that $e^k \leq x < e^{k+1}$. Then xe^{-k} lies in the fundamental domain. Demonstrate the accuracy of your program by comparing it with MATLAB's `log` command.
4. Let $f(x) = e^{|x|}$. Compare evenly spaced interpolation with Chebyshev interpolation by plotting degree n polynomials of both types on the interval $[-1, 1]$, for $n = 10$ and 20 . For evenly spaced interpolation, the left and right interpolation base points should be -1 and 1 . By sampling at a 0.01 step size, create the empirical interpolation errors for each type, and plot a comparison. Can the Runge phenomenon be observed in this problem?
5. Carry out the steps of Computer Problem 4 for $f(x) = e^{-x^2}$.

3.4 CUBIC SPLINES

Splines represent an alternative approach to data interpolation. In polynomial interpolation, a single formula, given by a polynomial, is used to meet all data points. The idea of splines is to use several formulas, each a low-degree polynomial, to pass through the data points.

The simplest example of a spline is a linear spline, in which one “connects the dots” with straight-line segments. Assume that we are given a set of data points $(x_1, y_1), \dots, (x_n, y_n)$ with $x_1 < \dots < x_n$. A linear spline consists of the $n - 1$ line segments that are drawn between neighboring pairs of points. Figure 3.12(a) shows a linear spline where, between each neighboring pair of points $(x_i, y_i), (x_{i+1}, y_{i+1})$, the linear function $y = a_i + b_i x$ is drawn through the two points. The given data points in the figure are $(1, 2)$, $(2, 1)$, $(4, 4)$, and $(5, 3)$, and the linear spline is given by

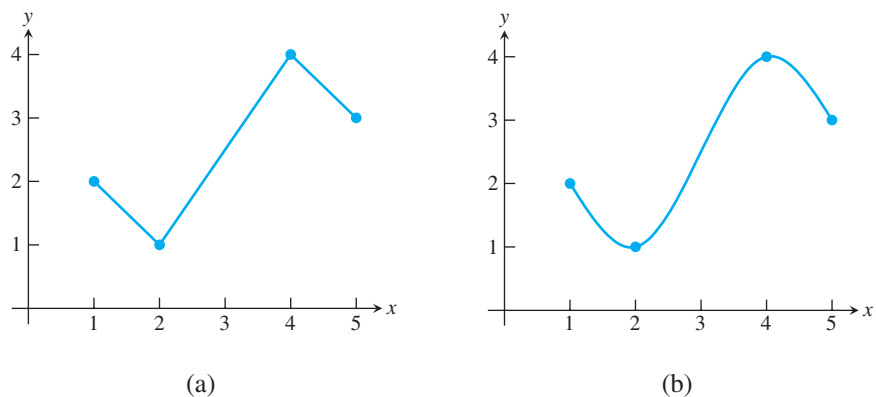


Figure 3.12 Splines through four data points. (a) Linear spline through $(1, 2)$, $(2, 1)$, $(4, 4)$, and $(5, 3)$ consists of three linear polynomials given by (3.15). (b) Cubic spline through the same points, given by (3.16).

$$\begin{aligned}
S_1(x) &= 2 - (x - 1) \text{ on } [1, 2] \\
S_2(x) &= 1 + \frac{3}{2}(x - 2) \text{ on } [2, 4] \\
S_3(x) &= 4 - (x - 4) \text{ on } [4, 5].
\end{aligned} \tag{3.15}$$

The linear spline successfully interpolates an arbitrary set of n data points. However, linear splines lack smoothness. Cubic splines are meant to address this shortcoming of linear splines. A cubic spline replaces linear functions between the data points by degree 3 (cubic) polynomials.

An example of a cubic spline that interpolates the same points $(1, 2)$, $(2, 1)$, $(4, 4)$, and $(5, 3)$ is shown in Figure 3.12(b). The equations defining the spline are

$$\begin{aligned}
S_1(x) &= 2 - \frac{13}{8}(x - 1) + 0(x - 1)^2 + \frac{5}{8}(x - 1)^3 \text{ on } [1, 2] \\
S_2(x) &= 1 + \frac{1}{4}(x - 2) + \frac{15}{8}(x - 2)^2 - \frac{5}{8}(x - 2)^3 \text{ on } [2, 4] \\
S_3(x) &= 4 + \frac{1}{4}(x - 4) - \frac{15}{8}(x - 4)^2 + \frac{5}{8}(x - 4)^3 \text{ on } [4, 5].
\end{aligned} \tag{3.16}$$

Note in particular the smooth transition from one S_i to the next at the base points, or “knots,” $x = 2$ and $x = 4$. This is achieved by arranging for the neighboring pieces S_i and S_{i+1} of the spline to have the same zeroth, first, and second derivatives when evaluated at the knots. Just how to do this is the topic of the next section.

Given n points $(x_1, y_1), \dots, (x_n, y_n)$, there is obviously one and only one linear spline through the data points. This will not be true for cubic splines. We will find that there are infinitely many through any set of data points. Extra conditions will be added when it is necessary to nail down a particular spline of interest.

3.4.1 Properties of splines

To be a little more precise about the properties of a cubic spline, we make the following definition: Assume that we are given the n data points $(x_1, y_1), \dots, (x_n, y_n)$, where the x_i are distinct and in increasing order. A **cubic spline** $S(x)$ through the data points $(x_1, y_1), \dots, (x_n, y_n)$ is a set of cubic polynomials

$$\begin{aligned}
S_1(x) &= y_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 \text{ on } [x_1, x_2] \\
S_2(x) &= y_2 + b_2(x - x_2) + c_2(x - x_2)^2 + d_2(x - x_2)^3 \text{ on } [x_2, x_3] \\
&\vdots \\
S_{n-1}(x) &= y_{n-1} + b_{n-1}(x - x_{n-1}) + c_{n-1}(x - x_{n-1})^2 + d_{n-1}(x - x_{n-1})^3 \text{ on } [x_{n-1}, x_n]
\end{aligned} \tag{3.17}$$

with the following properties:

Property 1 $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$ for $i = 1, \dots, n - 1$.

Property 2 $S'_{i-1}(x_i) = S'_i(x_i)$ for $i = 2, \dots, n - 1$.

Property 3 $S''_{i-1}(x_i) = S''_i(x_i)$ for $i = 2, \dots, n - 1$.

Property 1 guarantees that the spline $S(x)$ interpolates the data points. Property 2 forces the slopes of neighboring parts of the spline to agree where they meet, and Property 3 does the same for the curvature, represented by the second derivative.

► **EXAMPLE 3.13** Check that $\{S_1, S_2, S_3\}$ in (3.16) satisfies all cubic spline properties for the data points $(1, 2)$, $(2, 1)$, $(4, 4)$, and $(5, 3)$.

We will check all three properties.

Property 1. There are $n = 4$ data points. We must check

$$\begin{aligned} S_1(1) &= 2 \quad \text{and} \quad S_1(2) = 1 \\ S_2(2) &= 1 \quad \text{and} \quad S_2(4) = 4 \\ S_3(4) &= 4 \quad \text{and} \quad S_3(5) = 3. \end{aligned}$$

These follow easily from the defining equations (3.16).

Property 2. The first derivatives of the spline functions are

$$\begin{aligned} S'_1(x) &= -\frac{13}{8} + \frac{15}{8}(x-1)^2 \\ S'_2(x) &= \frac{1}{4} + \frac{15}{4}(x-2) - \frac{15}{8}(x-2)^2 \\ S'_3(x) &= \frac{1}{4} - \frac{15}{4}(x-4) + \frac{15}{8}(x-4)^2. \end{aligned}$$

We must check $S'_1(2) = S'_2(2)$ and $S'_2(4) = S'_3(4)$. The first is

$$-\frac{13}{8} + \frac{15}{8} = \frac{1}{4},$$


and the second is

$$\frac{1}{4} + \frac{15}{4}(4-2) - \frac{15}{8}(4-2)^2 = \frac{1}{4},$$

both of which check out.

Property 3. The second derivatives are

$$\begin{aligned} S''_1(x) &= \frac{15}{4}(x-1) \\ S''_2(x) &= \frac{15}{4} - \frac{15}{4}(x-2) \\ S''_3(x) &= -\frac{15}{4} + \frac{15}{4}(x-4). \end{aligned} \tag{3.18}$$

We must check $S''_1(2) = S''_2(2)$ and $S''_2(4) = S''_3(4)$, both of which are true. Therefore, (3.16) is a cubic spline. 

Constructing a spline from a set of data points means finding the coefficients b_i, c_i, d_i that make Properties 1–3 hold. Before we discuss how to determine the unknown coefficients b_i, c_i, d_i of the spline, let us count the number of conditions imposed by the definition. The first half of Property 1 is already reflected in the form (3.17); it says that the constant term of the cubic S_i must be y_i . The second half of Property 1 consists of $n - 1$ separate equations that must be satisfied by the coefficients, which we consider as unknowns. Each of Properties 2 and 3 add $n - 2$ additional equations, for a total of $n - 1 + 2(n - 2) = 3n - 5$ independent equations to be satisfied.

How many unknown coefficients are there? For each part S_i of the spline, three coefficients b_i, c_i, d_i are needed, for a total of $3(n - 1) = 3n - 3$. Therefore, solving for the coefficients is a problem of solving $3n - 5$ linear equations in $3n - 3$ unknowns. Unless there are inconsistent equations in the system (and there are not), the system of equations is underdetermined and so has infinitely many solutions. In other words, there are infinitely many cubic splines passing through the arbitrary set of data points $(x_1, y_1), \dots, (x_n, y_n)$.

Users of splines normally exploit the shortage of equations by adding two extra to the $3n - 5$ equations to arrive at a system of m equations in m unknowns, where $m = 3n - 3$. Aside from allowing the user to constrain the spline to given specifications, narrowing the field to a single solution simplifies computing and describing the result.

The simplest way of adding two more constraints is to require, in addition to the previous $3n - 5$ constraints, that the spline $S(x)$ have an inflection point at each end of the defining interval $[x_1, x_n]$. The constraints added to Properties 1–3 are

Property 4a Natural spline. $S''_1(x_1) = 0$ and $S''_{n-1}(x_n) = 0$.

A cubic spline that satisfies these two additional conditions is called a **natural** cubic spline. Note that (3.16) is a natural cubic spline, since it is easily verified from (3.18) that $S''_1(1) = 0$ and $S''_3(5) = 0$.

There are several other ways to add two more conditions. Usually, as in the case of the natural spline, they determine extra properties of the left and right ends of the spline, so they are called **end conditions**. We will take up this topic in the next section, but for now we concentrate on natural cubic splines.

Now that we have the right number of equations, $3n - 3$ equations in $3n - 3$ unknowns, we can write a MATLAB function to solve them for the spline coefficients. First we write out the equations in the unknowns b_i, c_i, d_i . Part 2 of Property 1 then implies the $n - 1$ equations:

$$\begin{aligned} y_2 &= S_1(x_2) = y_1 + b_1(x_2 - x_1) + c_1(x_2 - x_1)^2 + d_1(x_2 - x_1)^3 \\ &\vdots \\ y_n &= S_{n-1}(x_n) = y_{n-1} + b_{n-1}(x_n - x_{n-1}) + c_{n-1}(x_n - x_{n-1})^2 \\ &\quad + d_{n-1}(x_n - x_{n-1})^3. \end{aligned} \quad (3.19)$$

Property 2 generates the $n - 2$ equations,

$$\begin{aligned} 0 &= S'_1(x_2) - S'_2(x_2) = b_1 + 2c_1(x_2 - x_1) + 3d_1(x_2 - x_1)^2 - b_2 \\ &\vdots \\ 0 &= S'_{n-2}(x_{n-1}) - S'_{n-1}(x_{n-1}) = b_{n-2} + 2c_{n-2}(x_{n-1} - x_{n-2}) \\ &\quad + 3d_{n-2}(x_{n-1} - x_{n-2})^2 - b_{n-1}, \end{aligned} \quad (3.20)$$

and Property 3 implies the $n - 2$ equations:

$$\begin{aligned} 0 &= S''_1(x_2) - S''_2(x_2) = 2c_1 + 6d_1(x_2 - x_1) - 2c_2 \\ &\vdots \\ 0 &= S''_{n-2}(x_{n-1}) - S''_{n-1}(x_{n-1}) = 2c_{n-2} + 6d_{n-2}(x_{n-1} - x_{n-2}) - 2c_{n-1}. \end{aligned} \quad (3.21)$$

Instead of solving the equations in this form, the system can be simplified drastically by decoupling the equations. With a little algebra, a much smaller system of equations in the c_i can be solved first, followed by explicit formulas for the b_i and d_i in terms of the known c_i .

It is conceptually simpler if an extra unknown $c_n = S''_{n-1}(x_n)/2$ is introduced. In addition, we introduce the shorthand notation $\delta_i = x_{i+1} - x_i$ and $\Delta_i = y_{i+1} - y_i$. Then (3.21) can be solved for the coefficients

$$d_i = \frac{c_{i+1} - c_i}{3\delta_i} \quad \text{for } i = 1, \dots, n-1. \quad (3.22)$$

Solving (3.19) for b_i yields

$$\begin{aligned} b_i &= \frac{\Delta_i}{\delta_i} - c_i \delta_i - d_i \delta_i^2 \\ &= \frac{\Delta_i}{\delta_i} - c_i \delta_i - \frac{\delta_i}{3}(c_{i+1} - c_i) \\ &= \frac{\Delta_i}{\delta_i} - \frac{\delta_i}{3}(2c_i + c_{i+1}) \end{aligned} \quad (3.23)$$

for $i = 1, \dots, n-1$.

Substituting (3.22) and (3.23) into (3.20) results in the following $n-2$ equations in c_1, \dots, c_n :

$$\begin{aligned} \delta_1 c_1 + 2(\delta_1 + \delta_2)c_2 + \delta_2 c_3 &= 3 \left(\frac{\Delta_2}{\delta_2} - \frac{\Delta_1}{\delta_1} \right) \\ &\vdots \\ \delta_{n-2} c_{n-2} + 2(\delta_{n-2} + \delta_{n-1})c_{n-1} + \delta_{n-1} c_n &= 3 \left(\frac{\Delta_{n-1}}{\delta_{n-1}} - \frac{\Delta_{n-2}}{\delta_{n-2}} \right). \end{aligned}$$

Two more equations are given by the natural spline conditions (Property 4a):

$$\begin{aligned} S''_1(x_1) &= 0 \rightarrow 2c_1 = 0 \\ S''_{n-1}(x_n) &= 0 \rightarrow 2c_n = 0. \end{aligned}$$

This gives a total of n equations in n unknowns c_i , which can be written in the matrix form

$$\begin{aligned} &\begin{bmatrix} 1 & 0 & 0 & & & & \\ \delta_1 & 2\delta_1 + 2\delta_2 & \delta_2 & \ddots & & & \\ 0 & \delta_2 & 2\delta_2 + 2\delta_3 & \delta_3 & & & \\ & \ddots & & \ddots & \ddots & & \\ & & & & \delta_{n-2} & 2\delta_{n-2} + 2\delta_{n-1} & \delta_{n-1} \\ & & & & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 3 \left(\frac{\Delta_2}{\delta_2} - \frac{\Delta_1}{\delta_1} \right) \\ \vdots \\ 3 \left(\frac{\Delta_{n-1}}{\delta_{n-1}} - \frac{\Delta_{n-2}}{\delta_{n-2}} \right) \\ 0 \end{bmatrix}. \end{aligned} \quad (3.24)$$

After c_1, \dots, c_n are obtained from (3.24), b_1, \dots, b_{n-1} and d_1, \dots, d_{n-1} are found from (3.22) and (3.23).

Note that (3.24) is always solvable for the c_i . The coefficient matrix is strictly diagonally dominant, so by Theorem 2.10, there is a unique solution for the c_i and therefore also for the b_i and d_i . We have thus proved the following theorem:

THEOREM 3.7 Let $n \geq 2$. For a set of data points $(x_1, y_1), \dots, (x_n, y_n)$ with distinct x_i , there is a unique natural cubic spline fitting the points. ■

Natural cubic spline

Given $x = [x_1, \dots, x_n]$ where $x_1 < \dots < x_n$, $y = [y_1, \dots, y_n]$

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

$$a_i = y_i$$

$$\delta_i = x_{i+1} - x_i$$

$$\Delta_i = y_{i+1} - y_i$$

end

Solve (3.24) for c_1, \dots, c_n

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

$$d_i = \frac{c_{i+1} - c_i}{3\delta_i}$$

$$b_i = \frac{\Delta_i}{\delta_i} - \frac{\delta_i}{3}(2c_i + c_{i+1})$$

end

The natural cubic spline is

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \text{ on } [x_i, x_{i+1}] \text{ for } i = 1, \dots, n - 1.$$

► **EXAMPLE 3.14** Find the natural cubic spline through $(0, 3)$, $(1, -2)$, and $(2, 1)$.

The x -coordinates are $x_1 = 0$, $x_2 = 1$, and $x_3 = 2$. The y -coordinates are $a_1 = y_1 = 3$, $a_2 = y_2 = -2$, and $a_3 = y_3 = 1$, and the differences are $\delta_1 = \delta_2 = 1$, $\Delta_1 = -5$, and $\Delta_2 = 3$. The tridiagonal matrix equation (3.24) is

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 4 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 24 \\ 0 \end{bmatrix}.$$

The solution is $[c_1, c_2, c_3] = [0, 6, 0]$. Now, (3.22) and (3.23) yield

$$d_1 = \frac{c_2 - c_1}{3\delta_1} = \frac{6}{3} = 2$$

$$d_2 = \frac{c_3 - c_2}{3\delta_2} = \frac{-6}{3} = -2$$

$$b_1 = \frac{\Delta_1}{\delta_1} - \frac{\delta_1}{3}(2c_1 + c_2) = -5 - \frac{1}{3}(6) = -7$$

$$b_2 = \frac{\Delta_2}{\delta_2} - \frac{\delta_2}{3}(2c_2 + c_3) = 3 - \frac{1}{3}(12) = -1.$$

Therefore, the cubic spline is

$$S_1(x) = 3 - 7x + 0x^2 + 2x^3 \text{ on } [0, 1]$$

$$S_2(x) = -2 - 1(x - 1) + 6(x - 1)^2 - 2(x - 1)^3 \text{ on } [1, 2].$$

MATLAB code for this calculation follows. For different (not natural) endpoint conditions, discussed in the next section, the top and bottom rows of (3.24) are replaced by other appropriate rows.

```
%Program 3.5 Calculation of spline coefficients
%Calculates coefficients of cubic spline
%Input: x,y vectors of data points
%   plus two optional extra data v1, vn
%Output: matrix of coefficients b1,c1,d1;b2,c2,d2;...
function coeff=splinecoeff(x,y)
n=length(x);v1=0;vn=0;
A=zeros(n,n);           % matrix A is nxn
r=zeros(n,1);
for i=1:n-1              % define the deltas
    dx(i)= x(i+1)-x(i); dy(i)=y(i+1)-y(i);
end
for i=2:n-1              % load the A matrix
    A(i,i-1:i+1)=[dx(i-1) 2*(dx(i-1)+dx(i)) dx(i)];
    r(i)=3*(dy(i)/dx(i)-dy(i-1)/dx(i-1)); % right-hand side
end
% Set endpoint conditions
% Use only one of following 5 pairs:
A(1,1) = 1;             % natural spline conditions
A(n,n) = 1;
%A(1,1)=2;r(1)=v1;      % curvature-adj conditions
%A(n,n)=2;r(n)=vn;
%A(1,1:2)=[2*dx(1) dx(1)];r(1)=3*(dy(1)/dx(1)-v1); %clamped
%A(n,n-1:n)=[dx(n-1) 2*dx(n-1)];r(n)=3*(vn-dy(n-1)/dx(n-1));
%A(1,1:2)=[1 -1];       % parabol-term conditions, for n>=3
%A(n,n-1:n)=[1 -1];
%A(1,1:3)=[dx(2) -(dx(1)+dx(2)) dx(1)]; % not-a-knot, for n>=4
%A(n,n-2:n)=[dx(n-1) -(dx(n-2)+dx(n-1)) dx(n-2)];
coeff=zeros(n,3);
coeff(:,2)=A\r;         % solve for c coefficients
for i=1:n-1             % solve for b and d
    coeff(i,3)=(coeff(i+1,2)-coeff(i,2))/(3*dx(i));
    coeff(i,1)=dy(i)/dx(i)-dx(i)*(2*coeff(i,2)+coeff(i+1,2))/3;
end
coeff=coeff(1:n-1,1:3);
```

We have taken the liberty of listing other choices for end conditions, although they are commented out for now. The alternative conditions will be discussed in the next section. Another MATLAB function, titled **splineplot.m**, calls **splinecoeff.m** to get the coefficients and then plots the cubic spline:

```
%Program 3.6 Cubic spline plot
%Computes and plots spline from data points
%Input: x,y vectors of data points, number k of plotted points
%   per segment
%Output: x1, y1 spline values at plotted points
function [x1,y1]=splineplot(x,y,k)
n=length(x);
coeff=splinecoeff(x,y);
x1=[]; y1=[];
for i=1:n-1
    xs=linspace(x(i),x(i+1),k+1);
    dx=xs-x(i);
```

```

ys=coeff(i,3)*dx; % evaluate using nested multiplication
ys=(ys+coeff(i,2)).*dx;
ys=(ys+coeff(i,1)).*dx+y(i);
x1=[x1; xs(1:k)']; y1=[y1;ys(1:k)'];
end
x1=[x1; x(end)];y1=[y1;y(end)];
plot(x,y,'o',x1,y1)

```

Figure 3.13(a) shows a natural cubic spline generated by `splineplot.m`.

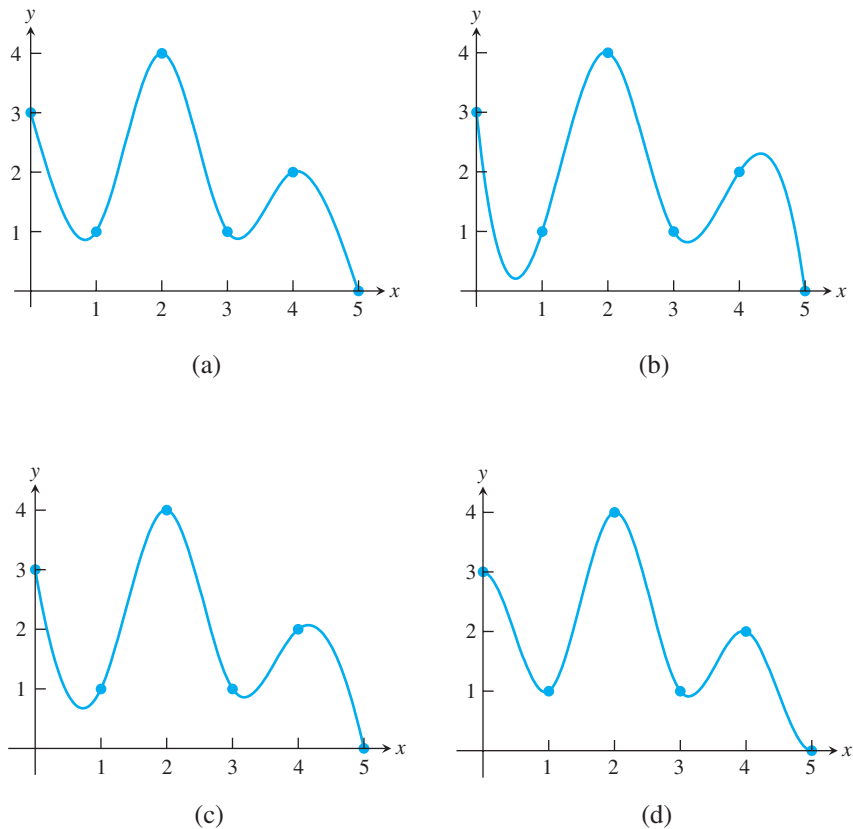


Figure 3.13 Cubic splines through six points. The plots are generated by `splineplot(x,y,10)` with input vectors $x=[0 \ 1 \ 2 \ 3 \ 4 \ 5]$ and $y=[3 \ 1 \ 4 \ 1 \ 2 \ 0]$. (a) Natural cubic spline (notice inflection points at ends) (b) Not-a-knot cubic spline (single cubic equation on $[0,2]$ and on $[3,5]$) (c) Parabolically terminated spline (d) Clamped cubic spline (clamped at slope 0 at both ends).

3.4.2 Endpoint conditions

The two extra conditions specified in Property 4a are called the “endpoint conditions” for a natural spline. Requiring that these be satisfied along with Properties 1 through 3 narrows the field to exactly one cubic spline, according to Theorem 3.9. It turns out that there are many different versions of Property 4, meaning many other pairs of endpoint conditions, for which an analogous theorem holds. In this section, we present a few of the more popular ones.

Property 4b Curvature-adjusted cubic spline. The first alternative to a natural cubic spline requires setting $S_1''(x_1)$ and $S_{n-1}''(x_n)$ to arbitrary values, chosen by the user, instead of zero. This

choice corresponds to setting the desired curvatures at the left and right endpoints of the spline. In terms of (3.23), it translates to the two extra conditions

$$\begin{aligned} 2c_1 &= v_1 \\ 2c_n &= v_n, \end{aligned}$$

where v_1, v_n denote the desired values. The equations turn into the two tableau rows

$$\left[\begin{array}{ccccccccc|c} 2 & 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 & v_1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 2 & v_n \end{array} \right]$$

to replace the top and bottom rows of (3.24), which were added for the natural spline. Notice that the new coefficient matrix is again strictly diagonally dominant, so that a generalized form of Theorem 3.9 holds for curvature-adjusted splines. (See Theorem 3.10, presented shortly.) In `splinecoeff.m`, the two lines

```
A(1,1)=2;r(1)=v1;           % curvature-adj conditions
A(n,n)=2;r(n)=vn;
```

must be substituted in place of the two existing lines for the natural spline.

The next alternative set of end conditions is

Property 4c Clamped cubic spline. This alternative is similar to the preceding one, but it is the first derivatives $S'_1(x_1)$ and $S'_{n-1}(x_n)$ that are set to user-specified values v_1 and v_n , respectively. Thus, the slope at the beginning and end of the spline are under the user's control. Using (3.22) and (3.23), we can write the extra condition $S'_1(x_1) = v_1$ as

$$2\delta_1 c_1 + \delta_1 c_2 = 3 \left(\frac{\Delta_1}{\delta_1} - v_1 \right)$$

and $S'_{n-1}(x_n) = v_n$ as

$$\delta_{n-1} c_{n-1} + 2\delta_{n-1} c_n = 3 \left(v_n - \frac{\Delta_{n-1}}{\delta_{n-1}} \right).$$

The two corresponding tableau rows are

$$\left[\begin{array}{cccccccccc|c} 2\delta_1 & \delta_1 & 0 & 0 & \cdots & \cdots & 0 & 0 & 0 & 3(\Delta_1/\delta_1 - v_1) \\ 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & \delta_{n-1} & 2\delta_{n-1} & 3(v_n - \Delta_{n-1}/\delta_{n-1}) \end{array} \right].$$

Note that strict diagonal dominance holds also for the revised coefficient matrix in (3.24), so Theorem 3.9 also holds with the natural spline replaced with the clamped spline. In `splinecoeff.m`, the two lines

```
A(1,1:2)=[2*dx(1) dx(1)];r(1)=3*(dy(1)/dx(1)-v1);
A(n,n-1:n)=[dx(n-1) 2*dx(n-1)];r(n)=3*(vn-dy(n-1)/dx(n-1));
```

must be substituted. See Figure 3.13 for a clamped spline with $v_1 = v_n = 0$.

Property 4d Parabolically terminated cubic spline. The first and last parts of the spline, S_1 and S_{n-1} , are forced to be at most degree 2, by specifying that $d_1 = 0 = d_{n-1}$. Equivalently, according to (3.22), we can require that $c_1 = c_2$ and $c_{n-1} = c_n$. The equations form the two tableau rows

$$\left[\begin{array}{cccccccccc|c} 1 & -1 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 1 & -1 & 0 \end{array} \right]$$

to be used as the top and bottom rows of (3.24). Assume that the number n of data points satisfies $n \geq 3$. (See Exercise 19 for the case $n = 2$.) In this case, upon replacing c_1 by c_2 and c_n by c_{n-1} , we find that the matrix equation reduces to a strictly diagonally dominant $n - 2 \times n - 2$ matrix equation in c_2, \dots, c_{n-1} . Therefore, a version of Theorem 3.9 holds for parabolically terminated splines, assuming that $n \geq 3$.

In `splinecoeff.m`, the two lines

```
A(1,1:2)=[1 -1]; % parabol-term conditions
A(n,n-1:n)=[1 -1];
```

must be substituted.

Property 4e Not-a-knot cubic spline. The two added equations are $d_1 = d_2$ and $d_{n-2} = d_{n-1}$, or equivalently, $S_1'''(x_2) = S_2'''(x_2)$ and $S_{n-2}'''(x_{n-1}) = S_{n-1}'''(x_{n-1})$. Since S_1 and S_2 are polynomials of degree 3 or less, requiring their third derivatives to agree at x_2 , while their zeroth, first, and second derivatives already agree there, causes S_1 and S_2 to be identical cubic polynomials. (Cubics are defined by four coefficients, and four conditions are specified.) Thus, x_2 is not needed as a base point: The spline is given by the same formula $S_1 = S_2$ on the entire interval $[x_1, x_3]$. The same reasoning shows that $S_{n-2} = S_{n-1}$, so not only x_2 , but also x_{n-1} , is “no longer a knot.”

Note that $d_1 = d_2$ implies that $(c_2 - c_1)/\delta_1 = (c_3 - c_2)/\delta_2$, or

$$\delta_2 c_1 - (\delta_1 + \delta_2) c_2 + \delta_1 c_3 = 0,$$

and similarly, $d_{n-2} = d_{n-1}$ implies that

$$\delta_{n-1} c_{n-2} - (\delta_{n-2} + \delta_{n-1}) c_{n-1} + \delta_{n-2} c_n = 0.$$

It follows that the two tableau rows are

$$\left(\begin{array}{cccccccccccc|c} \delta_2 & -(\delta_1 + \delta_2) & \delta_1 & 0 & \cdots & \cdots & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & \delta_{n-1} & -(\delta_{n-2} + \delta_{n-1}) & \delta_{n-2} & 0 & 0 \end{array} \right).$$

In `splinecoeff.m`, the two lines

```
A(1,1:3)=[dx(2) - (dx(1)+dx(2)) dx(1)]; % not-a-knot conditions
A(n,n-2:n)=[dx(n-1) - (dx(n-2)+dx(n-1)) dx(n-2)];
```

are used. Figure 3.13(b) shows an example of a not-a-knot cubic spline, compared with the natural spline through the same data points in part (a) of the figure.

As mentioned earlier, a theorem analogous to Theorem 3.7 exists for each of the preceding choices of end conditions:

THEOREM 3.8 Assume that $n \geq 2$. Then, for a set of data points $(x_1, y_1), \dots, (x_n, y_n)$ and for any one of the end conditions given by Properties 4a–4c, there is a unique cubic spline satisfying the end conditions and fitting the points. The same is true assuming that $n \geq 3$ for Property 4d and $n \geq 4$ for Property 4e. ■

MATLAB’s default `spline` command constructs a not-a-knot spline when given four or more points. Let x and y be vectors containing the x_i and y_i data values, respectively. Then the y -coordinate of the not-a-knot spline at another input x_0 is calculated by the MATLAB command

```
>> y0 = spline(x,y,x0);
```

If \mathbf{x}_0 is a vector of x -coordinates, then the output \mathbf{y}_0 will be a corresponding vector of y -coordinates, suitable for plotting, etc. Alternatively, if the vector input \mathbf{y} has exactly two more inputs than \mathbf{x} , the clamped cubic spline is calculated, with clamps v_1 and v_n equal to the first and last entries of \mathbf{y} .

3.4 Exercises

1. Decide whether the equations form a cubic spline.

$$(a) \quad S(x) = \begin{cases} x^3 + x - 1 & \text{on } [0,1] \\ -(x-1)^3 + 3(x-1)^2 + 3(x-1) + 1 & \text{on } [1,2] \end{cases}$$

$$(b) \quad S(x) = \begin{cases} 2x^3 + x^2 + 4x + 5 & \text{on } [0,1] \\ (x-1)^3 + 7(x-1)^2 + 12(x-1) + 12 & \text{on } [1,2] \end{cases}$$

2. (a) Check the spline conditions for

$$\begin{cases} S_1(x) = 1 + 2x + 3x^2 + 4x^3 & \text{on } [0, 1] \\ S_2(x) = 10 + 20(x-1) + 15(x-1)^2 + 4(x-1)^3 & \text{on } [1, 2] \end{cases}.$$

(b) Regardless of your answer to (a), decide whether any of the following extra conditions are satisfied for this example: natural, parabolically terminated, not-a-knot.

3. Find c in the following cubic splines. Which of the three end conditions—natural, parabolically terminated, or not-a-knot—if any, are satisfied?

$$(a) \quad S(x) = \begin{cases} 4 - \frac{11}{4}x + \frac{3}{4}x^3 & \text{on } [0,1] \\ 2 - \frac{1}{2}(x-1) + c(x-1)^2 - \frac{3}{4}(x-1)^3 & \text{on } [1,2] \end{cases}$$

$$(b) \quad S(x) = \begin{cases} 3 - 9x + 4x^2 & \text{on } [0,1] \\ -2 - (x-1) + c(x-1)^2 & \text{on } [1,2] \end{cases}$$

$$(c) \quad S(x) = \begin{cases} -2 - \frac{3}{2}x + \frac{7}{2}x^2 - x^3 & \text{on } [0,1] \\ -1 + c(x-1) + \frac{1}{2}(x-1)^2 - (x-1)^3 & \text{on } [1,2] \\ 1 + \frac{1}{2}(x-2) - \frac{5}{2}(x-2)^2 - (x-2)^3 & \text{on } [2,3] \end{cases}$$

4. Find k_1, k_2, k_3 in the following cubic spline. Which of the three end conditions—natural, parabolically terminated, or not-a-knot—if any, are satisfied?

$$S(x) = \begin{cases} 4 + k_1x + 2x^2 - \frac{1}{6}x^3 & \text{on } [0, 1] \\ 1 - \frac{4}{3}(x-1) + k_2(x-1)^2 - \frac{1}{6}(x-1)^3 & \text{on } [1, 2]. \\ 1 + k_3(x-2) + (x-2)^2 - \frac{1}{6}(x-2)^3 & \text{on } [2, 3] \end{cases}$$

5. How many natural cubic splines on $[0, 2]$ are there for the given data $(0, 0), (1, 1), (2, 2)$? Exhibit one such spline.
6. Find the parabolically terminated cubic spline through the data points $(0,1), (1,1), (2,1), (3,1), (4,1)$. Is this spline also not-a-knot? natural?
7. Solve equations (3.24) to find the natural cubic spline through the three points (a) $(0,0), (1,1), (2,4)$ (b) $(-1,1), (1,1), (2,4)$.
8. Solve equations (3.24) to find the natural cubic spline through the three points (a) $(0,1), (2,3), (3,2)$ (b) $(0,0), (1,1), (2,6)$.

9. Find $S'(0)$ and $S'(3)$ for the cubic spline

$$\begin{cases} S_1(x) = 3 + b_1x + x^3 & \text{on } [0, 1] \\ S_2(x) = 1 + b_2(x - 1) + 3(x - 1)^2 - 2(x - 1)^3 & \text{on } [1, 3] \end{cases}.$$

10. True or false: Given $n = 3$ data points, the parabolically terminated cubic spline through the points must be not-a-knot.
11. (a) How many parabolically terminated cubic splines on $[0, 2]$ are there for the given data $(0, 2)$, $(1, 0)$, $(2, 2)$? Exhibit one such spline. (b) Answer the same question for not-a-knot.
12. How many not-a-knot cubic splines are there for the given data $(1, 3)$, $(3, 3)$, $(4, 2)$, $(5, 0)$? Exhibit one such spline.
13. (a) Find b_1 and c_3 in the cubic spline

$$S(x) = \begin{cases} -1 + b_1x - \frac{5}{9}x^2 + \frac{5}{9}x^3 & \text{on } [0, 1] \\ \frac{14}{9}(x - 1) + \frac{10}{9}(x - 1)^2 - \frac{2}{3}(x - 1)^3 & \text{on } [1, 2] \\ 2 + \frac{16}{9}(x - 2) + c_3(x - 2)^2 - \frac{1}{9}(x - 2)^3 & \text{on } [2, 3] \end{cases}$$

(b) Is this spline natural? (c) This spline satisfies “clamped” endpoint conditions. What are the values of the two clamps?

14. Consider the cubic spline

$$\begin{cases} S_1(x) = 6 - 2x + \frac{1}{2}x^3 & \text{on } [0, 2] \\ S_2(x) = 6 + 4(x - 2) + c(x - 2)^2 + d(x - 2)^3 & \text{on } [2, 3] \end{cases}$$

(a) Find c . (b) Does there exist a number d such that the spline is natural? If so, find d .

15. Can a cubic spline be both natural and parabolically terminated? If so, what else can you say about such a spline?
16. Does there exist a (simultaneously) natural, parabolically terminated, not-a-knot cubic spline through each set of data points $(x_1, y_1), \dots, (x_{100}, y_{100})$ with distinct x_i ? If so, give a reason. If not, explain what conditions must hold on the 100 points in order for such a spline to exist.
17. Assume that the leftmost piece of a given natural cubic spline is the constant function $S_1(x) = 1$ on the interval $[-1, 0]$. Find three different possibilities for the neighboring piece $S_2(x)$ of the spline on $[0, 1]$.
18. Assume that a car travels along a straight road from one point to another from a standing start at time $t = 0$ to a standing stop at time $t = 1$. The distance along the road is sampled at certain times between 0 and 1. Which cubic spline (in terms of end conditions) will be most appropriate for describing distance versus time?
19. The case $n = 2$ for parabolically terminated cubic splines is not covered by Theorem 3.8. Discuss existence and uniqueness for the cubic spline in this case.
20. Discuss the existence and uniqueness of a not-a-knot cubic spline when $n = 2$ and $n = 3$.
21. Theorem 3.8 says that there is exactly one not-a-knot spline through any given four points with distinct x_i . (a) How many not-a-knot splines go through any given 3 points with distinct x_i ? (b) Find a not-a-knot spline through $(0, 0)$, $(1, 1)$, $(2, 4)$ that is not parabolically terminated.

3.4 Computer Problems

- Find the equations and plot the natural cubic spline that interpolates the data points (a) $(0, 3), (1, 5), (2, 4), (3, 1)$ (b) $(-1, 3), (0, 5), (3, 1), (4, 1), (5, 1)$.
- Find and plot the not-a-knot cubic spline that interpolates the data points (a) $(0, 3), (1, 5), (2, 4), (3, 1)$ (b) $(-1, 3), (0, 5), (3, 1), (4, 1), (5, 1)$.
- Find and plot the cubic spline S satisfying $S(0) = 1, S(1) = 3, S(2) = 3, S(3) = 4, S(4) = 2$ and with $S''(0) = S''(4) = 0$.
- Find and plot the cubic spline S satisfying $S(0) = 1, S(1) = 3, S(2) = 3, S(3) = 4, S(4) = 2$ and with $S''(0) = 3$ and $S''(4) = 2$.
- Find and plot the cubic spline S satisfying $S(0) = 1, S(1) = 3, S(2) = 3, S(3) = 4, S(4) = 2$ and with $S'(0) = 0$ and $S'(4) = 1$.
- Find and plot the cubic spline S satisfying $S(0) = 1, S(1) = 3, S(2) = 3, S(3) = 4, S(4) = 2$ and with $S'(0) = -2$ and $S'(4) = 1$.
- Find the clamped cubic spline that interpolates $f(x) = \cos x$ at five evenly spaced points in $[0, \pi/2]$, including the endpoints. What is the best choice for $S'(0)$ and $S'(\pi/2)$ to minimize interpolation error? Plot the spline and $\cos x$ on $[0, 2]$.
- Carry out the steps of Computer Problem 7 for the function $f(x) = \sin x$.
- Find the clamped cubic spline that interpolates $f(x) = \ln x$ at five evenly spaced points in $[1, 3]$, including the endpoints. Empirically find the maximum interpolation error on $[1, 3]$.
- Find the number of interpolation nodes in Computer Problem 9 required to make the maximum interpolation error at most 0.5×10^{-7} .
- (a) Consider the natural cubic spline through the world population data points in Computer Problem 3.1.1. Evaluate the year 1980 and compare with the correct population. (b) Using a linear spline, estimate the slopes at 1960 and 2000, and use these slopes to find the clamped cubic spline through the data. Plot the spline and estimate the 1980 population. Which estimates better, natural or clamped?
- Recall the carbon dioxide data of Exercise 3.1.17. (a) Find and plot the natural cubic spline through the data, and compute the spline estimate for the CO_2 concentration in 1950. (b) Carry out the same analysis for the parabolically terminated spline. (c) How does the not-a-knot spline differ from the solution to Exercise 3.1.17?
- In a single plot, show the natural, not-a-knot, and parabolically terminated cubic splines through the world oil production data from Computer Problem 3.2.3.
- Compile a list of 101 consecutive daily close prices of an exchange-traded stock from a financial data website. (a) Plot the interpolating polynomial through every fifth point. That is, let $x_0 = 0 : 5 : 100$ and y_0 denote the stock prices on days $0, 5, 10, \dots, 100$. Plot the degree 20 interpolating polynomial at points $x = 0 : 1 : 100$ and compare with the daily price data. What is the maximum interpolation error? Is the Runge phenomenon evident in your plot? (b) Plot the natural cubic spline with interpolating nodes $0 : 5 : 100$ instead of the interpolating polynomial, along with the daily data. Answer the same two questions. (c) Compare the two approaches of representing the data.

15. Compile a list of 121 hourly temperatures over five consecutive days from a weather data website. Let $x_0 = 0 : 6 : 120$ denote hours, and y_0 denote the temperatures at hours 0, 6, 12, ..., 120. Carry out steps (a)–(c) of Computer Problem 14, suitably adapted.

3.5 BÉZIER CURVES

Bézier curves are splines that allow the user to control the slopes at the knots. In return for the extra freedom, the smoothness of the first and second derivatives across the knot, which are automatic features of the cubic splines of the previous section, are no longer guaranteed. Bézier splines are appropriate for cases where corners (discontinuous first derivatives) and abrupt changes in curvature (discontinuous second derivatives) are occasionally needed.

Pierre Bézier developed the idea during his work for the Renault automobile company. The same idea was discovered independently by Paul de Casteljau, working for Citroen, a rival automobile company. It was considered an industrial secret by both companies, and the fact that both had developed the idea came to light only after Bézier published his research. Today the Bézier curve is a cornerstone of computer-aided design and manufacturing.

Each piece of a planar Bézier spline is determined by four points (x_1, y_1) , (x_2, y_2) , (x_3, y_3) , (x_4, y_4) . The first and last of the points are endpoints of the spline curve, and the middle two are **control points**, as shown in Figure 3.14. The curve leaves (x_1, y_1) along the tangent direction $(x_2 - x_1, y_2 - y_1)$ and ends at (x_4, y_4) along the tangent direction $(x_4 - x_3, y_4 - y_3)$. The equations that accomplish this are expressed as a parametric curve $(x(t), y(t))$ for $0 \leq t \leq 1$.

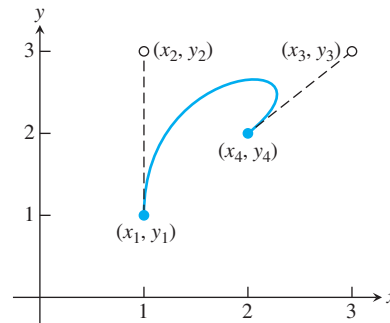


Figure 3.14 Bézier curve of Example 3.15. The points (x_1, y_1) and (x_4, y_4) are spline points, while (x_2, y_2) and (x_3, y_3) are control points.

Bézier curve

Given endpoints (x_1, y_1) , (x_4, y_4)
control points (x_2, y_2) , (x_3, y_3)

Set

$$b_x = 3(x_2 - x_1)$$

$$c_x = 3(x_3 - x_2) - b_x$$

$$d_x = x_4 - x_1 - b_x - c_x$$

$$b_y = 3(y_2 - y_1)$$

$$c_y = 3(y_3 - y_2) - b_y$$

$$d_y = y_4 - y_1 - b_y - c_y.$$

The Bézier curve is defined for $0 \leq t \leq 1$ by

$$\begin{aligned}x(t) &= x_1 + b_x t + c_x t^2 + d_x t^3 \\y(t) &= y_1 + b_y t + c_y t^2 + d_y t^3.\end{aligned}$$

It is easy to check the claims of the previous paragraph from the equations. In fact, according to Exercise 11,

$$\begin{aligned}x(0) &= x_1 \\x'(0) &= 3(x_2 - x_1) \\x(1) &= x_4 \\x'(1) &= 3(x_4 - x_3),\end{aligned}\tag{3.25}$$

and the analogous facts hold for $y(t)$.

► **EXAMPLE 3.15** Find the Bézier curve $(x(t), y(t))$ through the points $(x, y) = (1, 1)$ and $(2, 2)$ with control points $(1, 3)$ and $(3, 3)$.

The four points are $(x_1, y_1) = (1, 1)$, $(x_2, y_2) = (1, 3)$, $(x_3, y_3) = (3, 3)$, and $(x_4, y_4) = (2, 2)$. The Bézier formulas yield $b_x = 0$, $c_x = 6$, $d_x = -5$ and $b_y = 6$, $c_y = -6$, $d_y = 1$. The Bézier spline

$$\begin{aligned}x(t) &= 1 + 6t^2 - 5t^3 \\y(t) &= 1 + 6t - 6t^2 + t^3\end{aligned}$$

is shown in Figure 3.14 along with the control points. ◀

Bézier curves are building blocks that can be stacked to fit arbitrary function values and slopes. They are an improvement over cubic splines, in the sense that the slopes at the nodes can be specified as the user wants them. However, this freedom comes at the expense of smoothness: The second derivatives from the two different directions generally disagree at the nodes. In some applications, this disagreement is an advantage.

As a special case, when the control points equal the endpoints, the spline is a simple line segment, as shown next.

► **EXAMPLE 3.16** Prove that the Bézier spline with $(x_1, y_1) = (x_2, y_2)$ and $(x_3, y_3) = (x_4, y_4)$ is a line segment.

The Bézier formulas show that the equations are

$$\begin{aligned}x(t) &= x_1 + 3(x_4 - x_1)t^2 - 2(x_4 - x_1)t^3 = x_1 + (x_4 - x_1)t^2(3 - 2t) \\y(t) &= y_1 + 3(y_4 - y_1)t^2 - 2(y_4 - y_1)t^3 = y_1 + (y_4 - y_1)t^2(3 - 2t)\end{aligned}$$

for $0 \leq t \leq 1$. Every point in the spline has the form

$$\begin{aligned}(x(t), y(t)) &= (x_1 + r(x_4 - x_1), y_1 + r(y_4 - y_1)) \\&= ((1 - r)x_1 + rx_4, (1 - r)y_1 + ry_4),\end{aligned}$$

where $r = t^2(3 - 2t)$. Since $0 \leq r \leq 1$, each point lies on the line segment connecting (x_1, y_1) and (x_4, y_4) . ◀

Bézier curves are simple to program and are often used in drawing software. A freehand curve in the plane can be viewed as a parametric curve $(x(t), y(t))$ and represented by a Bézier spline. The equations are implemented in the following MATLAB freehand drawing program. The user clicks the mouse once to fix a starting point (x_0, y_0) in the plane, and

three more clicks to mark the first control point, second control point, and endpoint. A Bézier spline is drawn between the start and end points. Each subsequent triple of mouse clicks extends the curve further, using the previous endpoint as the starting point for the next piece. The MATLAB command `ginput` is used to read the mouse location. Figure 3.15 shows a screenshot of `bezierdraw.m`.

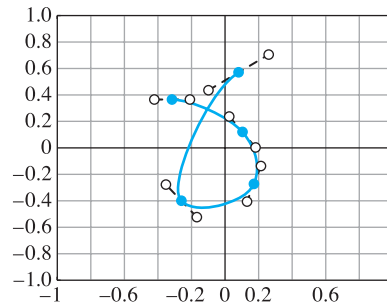


Figure 3.15 Program 3.7 built from Bézier curves. Screenshot of MATLAB code `bezierdraw.m`, including direction vectors drawn at each control point.

```
%Program 3.7 Freehand Draw Program Using Bezier Splines
%Click in Matlab figure window to locate first point, and click
%      three more times to specify 2 control points and the next
%      spline point. Continue with groups of 3 points to add more
%      to the curve. Press return to terminate program.
function bezierdraw
plot([-1 1],[0,0],'k',[0 0],[-1 1],'k');hold on
t=0:.02:1;
[x,y]=ginput(1);           % get one mouse click
while(0 == 0)
    [xnew,ynew] = ginput(3); % get three mouse clicks
    if length(xnew) < 3
        break               % if return pressed, terminate
    end
    x=[x;xnew];y=[y;ynew];   % plot spline points and control pts
    plot([x(1) x(2)],[y(1) y(2)],'r:',x(2),y(2),'rs');
    plot([x(3) x(4)],[y(3) y(4)],'r:',x(3),y(3),'rs');
    plot(x(1),y(1),'bo',x(4),y(4),'bo');
    bx=3*(x(2)-x(1)); by=3*(y(2)-y(1)); % spline equations ...
    cx=3*(x(3)-x(2))-bx;cy=3*(y(3)-y(2))-by;
    dx=x(4)-x(1)-bx-cx;dy=y(4)-y(1)-by-cy;
    xp=x(1)+t.*(bx+t.*(cx+t*dx));      % Horner's method
    yp=y(1)+t.*(by+t.*(cy+t*dy));
    plot(xp,yp)                        % plot spline curve
    x=x(4);y=y(4);                    % promote last to first and repeat
end
hold off
```

Although our discussion has been restricted to two-dimensional Bézier curves, the defining equations are easily extended to three dimensions, in which they are called Bézier space curves. Each piece of the spline requires four (x, y, z) points—two endpoints and two control points—just as in the two-dimensional case. Examples of Bézier space curves are explored in the exercises.

3.5 Exercises

- Find the one-piece Bézier curve $(x(t), y(t))$ defined by the given four points.
(a) (0,0), (0,2), (2,0), (1,0) (b) (1,1), (0,0), (-2,0), (-2,1) (c) (1,2), (1,3), (2,3), (2,2)
- Find the first endpoint, two control points, and last endpoint for the following one-piece Bézier curves.

$$(a) \begin{cases} x(t) = 1 + 6t^2 + 2t^3 \\ y(t) = 1 - t + t^3 \end{cases} \quad (b) \begin{cases} x(t) = 3 + 4t - t^2 + 2t^3 \\ y(t) = 2 - t + t^2 + 3t^3 \end{cases}$$

$$(c) \begin{cases} x(t) = 2 + t^2 - t^3 \\ y(t) = 1 - t + 2t^3 \end{cases}$$

- Find the three-piece Bézier curve forming the triangle with vertices (1, 2), (3, 4), and (5, 1).
- Build a four-piece Bézier spline that forms a square with sides of length 5.
- Describe the character drawn by the following two-piece Bezier curve:
(0,2) (1,2) (1,1) (0,1)
(0,1) (1,1) (1,0) (0,0)
- Describe the character drawn by the following three-piece Bezier curve:
(0,1) (0,1) (0,0) (0,0)
(0,0) (0,1) (1,1) (1,0)
(1,0) (1,1) (2,1) (2,0)
- Find a one-piece Bézier spline that has vertical tangents at its endpoints $(-1, 0)$ and $(1, 0)$ and that passes through $(0, 1)$.
- Find a one-piece Bézier spline that has a horizontal tangent at endpoint $(0, 1)$ and a vertical tangent at endpoint $(1, 0)$ and that passes through $(1/3, 2/3)$ at $t = 1/3$.
- Find the one-piece Bézier space curve $(x(t), y(t), z(t))$ defined by the four points.
(a) (1, 0, 0), (2, 0, 0), (0, 2, 1), (0, 1, 0) (b) (1, 1, 2), (1, 2, 3), $(-1, 0, 0)$, (1, 1, 1)
(c) (2, 1, 1), (3, 1, 1), (0, 1, 3), (3, 1, 3)
- Find the knots and control points for the following Bézier space curves.

$$(a) \begin{cases} x(t) = 1 + 6t^2 + 2t^3 \\ y(t) = 1 - t + t^3 \\ z(t) = 1 + t + 6t^2 \end{cases} \quad (b) \begin{cases} x(t) = 3 + 4t - t^2 + 2t^3 \\ y(t) = 2 - t + t^2 + 3t^3 \\ z(t) = 3 + t + t^2 - t^3 \end{cases}$$

$$(c) \begin{cases} x(t) = 2 + t^2 - t^3 \\ y(t) = 1 - t + 2t^3 \\ z(t) = 2t^3 \end{cases}$$

- Prove the facts in (3.25), and explain how they justify the Bézier formulas.
- Given (x_1, y_1) , (x_2, y_2) , (x_3, y_3) , and (x_4, y_4) , show that the equations

$$\begin{aligned} x(t) &= x_1(1-t)^3 + 3x_2(1-t)^2t + 3x_3(1-t)t^2 + x_4t^3 \\ y(t) &= y_1(1-t)^3 + 3y_2(1-t)^2t + 3y_3(1-t)t^2 + y_4t^3 \end{aligned}$$

give the Bézier curve with endpoints (x_1, y_1) , (x_4, y_4) and control points (x_2, y_2) , (x_3, y_3) .

3.5 Computer Problems

1. Plot the curve in Exercise 7.
2. Plot the curve in Exercise 8.
3. Plot the letter from Bézier curves. (a) W (b) B (c) C (d) D.

Reality ✓
Check

3 Fonts from Bézier curves

In this project, we explain how to draw letters and numerals by using two-dimensional Bézier curves. They can be implemented by modifying the MATLAB code in Program 3.7 or by writing a PDF file.

Modern fonts are built directly from Bézier curves, in order to be independent of the printer or imaging device. Bézier curves were a fundamental part of the PostScript language from its start in the 1980s, and the PostScript commands for drawing curves have migrated in slightly altered form to the PDF format. Here is a complete PDF file that illustrates the curve we discussed in Example 3.15.

```
%PDF-1.7
1 0 obj
<<
/Length 2 0 R
>>
stream
100 100 m
100 300 300 300 200 200 c
S
endstream
endobj
2 0 obj
1000
endobj
4 0 obj
<<
/Type /Page
/Parent 5 0 R
/Contents 1 0 R
>>
endobj
5 0 obj
<<
/Kids [4 0 R]
/Count 1
/Type /Pages
/MediaBox [0 0 612 792]
>>
endobj
3 0 obj
<<
/Pages 5 0 R
/Type /Catalog
>>
endobj
xref
0 6
0000000000 65535 f
0000000100 00000 n
0000000200 00000 n
0000000500 00000 n
0000000300 00000 n
0000000400 00000 n
trailer
<<
/Size 6
/Root 3 0 R
>>
startxref
1000
%%EOF
```

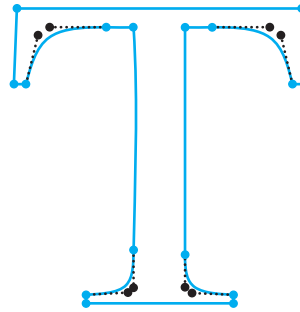


Figure 3.16 Times-Roman T made with Bézier splines. Blue circles are spline endpoints, and black circles are control points.

Most of the lines in this template file do various housekeeping chores. For example, the first line identifies the file as a PDF. We will focus on the lines between `stream` and `endstream`, which are the ones that identify the Bézier curve. The move command (`m`) sets the current plot point to be the (x, y) point specified by the two preceding numbers—in this case, the point $(100, 100)$. The curve command (`c`) accepts three (x, y) points and constructs the Bézier spline starting at the current plot point, treating the three (x, y) pairs as the two control points and the endpoint, respectively. The stroke command (`S`) draws the curve.

This text file `sample.pdf` can be downloaded from the textbook website. If it is opened with a PDF viewer, the Bézier curve of Figure 3.14 will be displayed. The coordinates have been multiplied by 100 to match the default conventions of PDF, which are 72 units to the inch. A sheet of letter-sized paper is 612 units wide and 792 high.

At present, characters from hundreds of fonts are drawn on computer screens and printers using Bézier curves. Of course, since PDF files often contain many characters, there are shortcuts for predefined fonts. The Bézier curve information for common fonts is usually stored in the PDF reader rather than the PDF file. We will choose to ignore this fact for now in order to see what we can do on our own.

Let's begin with a typical example. The upper case T character in the Times Roman font is constructed out of the following 16 Bézier curves. Each line consists of the numbers $x_1 y_1 x_2 y_2 x_3 y_3 x_4 y_4$ that define one piece of the Bézier spline.

```
237 620 237 620 237 120 237 120;
237 120 237 35 226 24 143 19;
143 19 143 19 143 0 143 0;
143 0 143 0 435 0 435 0;
435 0 435 0 435 19 435 19;
435 19 353 23 339 36 339 109;
339 109 339 108 339 620 339 620;
339 620 339 620 393 620 393 620;
393 620 507 620 529 602 552 492;
552 492 552 492 576 492 576 492;
576 492 576 492 570 662 570 662;
570 662 570 662 6 662 6 662;
6 662 6 662 0 492 0 492;
0 492 0 492 24 492 24 492;
24 492 48 602 71 620 183 620;
183 620 183 620 237 620 237 620;
```

To create a PDF file that writes the letter T, one needs to add commands within the `stream/endstream` area of the above template file. First, move to the initial endpoint $(237, 620)$

```
237 620 m
```

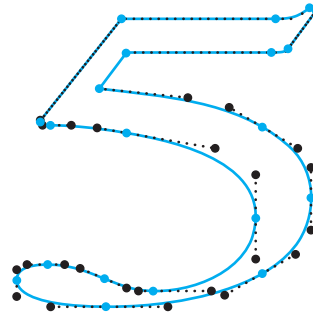


Figure 3.17 Times-Roman 5 made with Bézier splines. Blue circles are spline endpoints, and black circles are control points.

after which the first curve is drawn by the command

```
237 620 237 120 237 120 c
```

followed by fifteen more `c` commands, and the stroke command (`S`) to finish the letter `T`, shown in Figure 3.16. Note that the move command is necessary only at the first step; after that the next curve command takes the current plot point as the first point in the next Bézier curve, and needs only three more points to complete the curve command. The next curve command is completed in the same way, and so on. As an alternative to the stroke command `S`, the `f` command will fill in the outline if the figure is closed. The command `b` will both stroke and fill.

The number 5 is drawn by the following 21-piece Bézier curve and is shown in Figure 3.17:

```
149 597 149 597 149 597 345 597;
345 597 361 597 365 599 368 606;
368 606 406 695 368 606 406 695;
406 695 397 702 406 695 397 702;
397 702 382 681 372 676 351 676;
351 676 351 676 351 676 142 676;
142 676 33 439 142 676 33 439;
33 439 32 438 32 436 32 434;
32 434 32 428 35 426 44 426;
44 426 74 426 109 420 149 408;
149 408 269 372 324 310 324 208;
324 208 324 112 264 37 185 37;
185 37 165 37 149 44 119 66;
119 66 86 90 65 99 42 99;
42 99 14 99 0 87 0 62;
0 62 0 24 46 0 121 0;
121 0 205 0 282 27 333 78;
333 78 378 123 399 180 399 256;
399 256 399 327 381 372 333 422;
333 422 288 468 232 491 112 512;
112 512 112 512 149 597 149 597;
```

Suggested activities:

1. Use the `bezierdraw.m` program of Section 3.5 to sketch the upper case initial of your first name.
2. Revise the draw program to accept an $n \times 8$ matrix of numbers, each row representing a piece of a Bézier spline. Have the program draw the lower case letter `f` in the Times-Roman font, using the following 21-piece Bézier curve:


```

289 452 289 452 166 452 166 452;
166 452 166 452 166 568 166 568;
166 568 166 627 185 657 223 657;
223 657 245 657 258 647 276 618;
276 618 292 589 304 580 321 580;
321 580 345 580 363 598 363 621;
363 621 363 657 319 683 259 683;
259 683 196 683 144 656 118 611;
118 611 92 566 84 530 83 450;
83 450 83 450 1 450 1 450;
1 450 1 450 1 418 1 418;
1 418 1 418 83 418 83 418;
83 418 83 418 83 104 83 104;
83 104 83 31 72 19 0 15;
0 15 0 15 0 0 0 0;
0 0 0 0 260 0 260 0;
260 0 260 0 260 15 260 15;
260 15 178 18 167 29 167 104;
167 104 167 104 167 418 167 418;
167 418 167 418 289 418 289 418;
289 418 289 418 289 452 289 452;

```

3. Using the template above and your favorite text editor, write a PDF file that draws the lower case letter *f*. The program should begin with an *m* command to move to the first point, followed by 21 *c* commands and a stroke or fill command. These commands should lie between the *stream* and *endstream* commands. Test your file by opening it in a PDF viewer.
4. Here are some other PDF commands:

```

1.0 0.0 0.0 RG % set stroke color to red
0.0 1.0 0.0 rg % set fill color to green
2 w           % set stroke width to 2
b             % both stroke and fill (S is stroke, f is fill,
                b both)

```

Colors are represented according to the RGB convention, by three numbers between 0 and 1 embodying the relative contributions of red, green, and blue. Linear transformations may be used to change the size of the Bézier curves, and rotate and skew the results. Such coordinate changes are accomplished with the *cm* command. Preceding the curve commands with

```
a b c d e f cm
```

for real numbers a, b, c, d, e, f will transform the underlying planar coordinate system by

$$x' = ax + by + e$$

$$y' = cx + dy + f.$$

For example, using the *cm* command with $a = d = 0.5, b = c = e = f = 0$ reduces the size by a factor of 2, and $a = d = -0.5, b = c = 0$, and $e = f = 400$ turns the result upside down and translates by 400 units in the x and y directions. Other choices can perform rotations, reflections, or skews of the original Bézier curves. Coordinate changes are cumulative. In this step, use the coordinate system commands to present a resized, colored, and skewed version of the lower case *f* or other characters.

5. Although font information was a closely guarded secret for many years, much of it is now freely available on the Web. Search for other fonts, and find Bézier curve data that will draw letters of your choice in PDF or with `bezierdraw.m`.
6. Design your own letter or numeral. You should begin by drawing the figure on graph paper, respecting any symmetries that might be present. Estimate control points, and be prepared to revise them later as needed. ✓

Software and Further Reading

Interpolation software usually consists of separate codes for determining and evaluating the interpolating polynomial. MATLAB provides the `polyfit` and `polyval` commands for this purpose. The MATLAB `spline` command calculates not-a-knot splines by default, but has options for several other common end conditions. The command `interp1` combines several one-dimensional interpolation options. The NAG library contains subroutines `e01aef` and `e01baf` for polynomial and spline interpolation, and the IMSL has a number of spline routines based on various end conditions.

A classical reference for basic interpolation facts is Davis [1975], and the references Rivlin [1981] and Rivlin [1990] cover function approximation and Chebyshev interpolation. DeBoor [2001] on splines is also a classic; see also Schultz [1973] and Schumaker [1981]. Applications to computer-aided modelling and design are treated in Farin [1990] and Yamaguchi [1988]. The CORDIC Method for approximation of special functions was introduced in Volder [1959]. For more information on PDF files, see the *PDF Reference*, 6th Ed., published by Adobe Systems Inc. [2006].



Least Squares

The global positioning system (GPS) is a satellite-based location technology that provides accurate positioning at any time, from any point on earth. In just a few years, GPS has gone from a special-purpose navigation technology used by pilots, ship captains, and hikers to everyday use in automobiles, cellphones, and PDAs.

The system consists of 24 satellites following precisely regulated orbits, emitting synchronized signals.

An earth-based receiver picks up the satellite signals, finds its distance from all visible satellites, and uses the data to triangulate its position.

Reality Check ✓

Reality Check 4 on page 238 shows the use of equation solvers and least squares calculations to do the location estimation.

The concept of least squares dates from the pioneering work of Gauss and Legendre in the early 19th century. Its use permeates modern statistics and mathematical modeling. The key techniques of regression and parameter estimation have become fundamental tools in the sciences and engineering.

In this chapter, the normal equations are introduced and applied to a variety of data-fitting problems. Later, a more sophisticated approach, using the QR factorization, is explored, followed by a discussion of nonlinear least squares problems.

4.1 LEAST SQUARES AND THE NORMAL EQUATIONS

The need for least squares methods comes from two different directions, one each from our studies of Chapters 2 and 3. In Chapter 2, we learned how to find the solution of $Ax = b$ when a solution exists. In this chapter, we find out what to do when there is no solution. When the equations are inconsistent, which is likely if the number of equations exceeds the number of unknowns, the answer is to find the next best thing: the least squares approximation.

Chapter 3 addressed finding polynomials that exactly fit data points. However, if the data points are numerous, or the data points are collected only within some margin of error, fitting a high-degree polynomial exactly is rarely the best approach. In such cases, it is more reasonable to fit a simpler model that may only approximate the data points. Both problems, solving inconsistent systems of equations and fitting data approximately, are driving forces behind least squares.

4.1.1 Inconsistent systems of equations

It is not hard to write down a system of equations that has no solutions. Consider the following three equations in two unknowns:

$$\begin{aligned}x_1 + x_2 &= 2 \\x_1 - x_2 &= 1 \\x_1 + x_2 &= 3.\end{aligned}\tag{4.1}$$

Any solution must satisfy the first and third equations, which cannot both be true. A system of equations with no solution is called **inconsistent**.

What is the meaning of a system with no solutions? Perhaps the coefficients are slightly inaccurate. In many cases, the number of equations is greater than the number of unknown variables, making it unlikely that a solution can satisfy all the equations. In fact, m equations in n unknowns typically have no solution when $m > n$. Even though Gaussian elimination will not give us a solution to an inconsistent system $Ax = b$, we should not completely give up. An alternative in this situation is to find a vector x that comes the closest to being a solution.

If we choose this “closeness” to mean close in Euclidean distance, there is a straightforward algorithm for finding the closest x . This special x will be called the **least squares solution**.

We can get a better picture of the failure of system (4.1) to have a solution by writing it in a different way. The matrix form of the system is $Ax = b$, or

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}.\tag{4.2}$$

The alternative view of matrix/vector multiplication is to write the equivalent equation

$$x_1 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + x_2 \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}.\tag{4.3}$$

In fact, any $m \times n$ system $Ax = b$ can be viewed as a vector equation

$$x_1 v_1 + x_2 v_2 + \cdots + x_n v_n = b,\tag{4.4}$$

which expresses b as a linear combination of the columns v_i of A , with coefficients x_1, \dots, x_n . In our case, we are trying to hit the target vector b as a linear combination of two other three-dimensional vectors. Since the combinations of two three-dimensional vectors form a plane inside R^3 , equation (4.3) has a solution only if the vector b lies in that plane. This will always be the situation when we are trying to solve m equations in n unknowns, with $m > n$. Too many equations make the problem overspecified and the equations inconsistent.

Figure 4.1(b) shows a direction for us to go when a solution does not exist. There is no pair x_1, x_2 that solves (4.1), but there is a point in the plane Ax of all possible candidates that

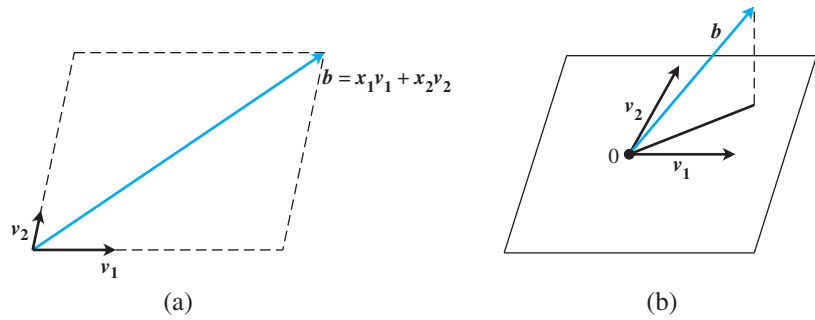


Figure 4.1 Geometric solution of a system of three equations in two unknowns.

(a) Equation (4.3) requires that the vector b , the right-hand side of the equation, is a linear combination of the columns vectors v_1 and v_2 . (b) If b lies outside of the plane defined by v_1 and v_2 , there will be no solution. The least squares solution \bar{x} makes the combination vector $A\bar{x}$ the one in the plane Ax that is nearest to b in the sense of Euclidean distance.

is closest to b . This special vector $A\bar{x}$ is distinguished by the following fact: The residual vector $b - A\bar{x}$ is perpendicular to the plane $\{Ax | x \in R^n\}$. We will exploit this fact to find a formula for \bar{x} , the least squares “solution.”

First we establish some notation. Recall the concept of the **transpose** A^T of the $m \times n$ matrix A , which is the $n \times m$ matrix whose rows are the columns of A and whose columns are the rows of A , in the same order. The transpose of the sum of two matrices is the sum of the transposes, $(A + B)^T = A^T + B^T$. The transpose of a product of two matrices is the product of the transposes in the reverse order—that is, $(AB)^T = B^T A^T$.

To work with perpendicularity, recall that two vectors are at right angles to one another if their dot product is zero. For two m -dimensional column vectors u and v , we can write the dot product solely in terms of matrix multiplication by

$$u^T v = [u_1, \dots, u_m] \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix}. \quad (4.5)$$

The vectors u and v are perpendicular, or **orthogonal**, if $u^T \cdot v = 0$, using ordinary matrix multiplication.

Now we return to our search for a formula for \bar{x} . We have established that

$$(b - A\bar{x}) \perp \{Ax | x \in R^n\}.$$

Expressing the perpendicularity in terms of matrix multiplication, we find that

$$(Ax)^T (b - A\bar{x}) = 0 \text{ for all } x \text{ in } R^n.$$

Using the preceding fact about transposes, we can rewrite this expression as

$$x^T A^T (b - A\bar{x}) = 0 \text{ for all } x \text{ in } R^n,$$

SPOTLIGHT ON

Orthogonality

Least squares is based on orthogonality. The shortest distance from a point to a plane is carried by a line segment orthogonal to the plane. The normal equations are a computational way to locate the line segment, which represents the least squares error.

meaning that the n -dimensional vector $A^T(b - A\bar{x})$ is perpendicular to every vector x in R^n , including itself. There is only one way for that to happen:

$$A^T(b - A\bar{x}) = 0.$$

This gives a system of equations that defines the least squares solution,

$$A^T A\bar{x} = A^T b. \quad (4.6)$$

The system of equations (4.6) is known as the **normal equations**. Its solution \bar{x} is the so-called least squares solution of the system $Ax = b$.

Normal equations for least squares

Given the inconsistent system

$$Ax = b,$$

solve

$$A^T A\bar{x} = A^T b$$

for the least squares solution \bar{x} that minimizes the Euclidean length of the residual $r = b - Ax$.

► **EXAMPLE 4.1** Use the normal equations to find the least squares solution of the inconsistent system (4.1).

The problem in matrix form $Ax = b$ has

$$A = \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}.$$

The components of the normal equations are

$$A^T A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

and

$$A^T b = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}.$$

The normal equations

$$\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \end{bmatrix}$$

can now be solved by Gaussian elimination. The tableau form is

$$\left[\begin{array}{cc|c} 3 & 1 & 6 \\ 1 & 3 & 4 \end{array} \right] \longrightarrow \left[\begin{array}{cc|c} 3 & 1 & 6 \\ 0 & 8/3 & 2 \end{array} \right],$$

which can be solved to get $\bar{x} = (\bar{x}_1, \bar{x}_2) = (7/4, 3/4)$.



Substituting the least squares solution into the original problem yields

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \frac{7}{4} \\ \frac{3}{4} \end{bmatrix} = \begin{bmatrix} 2.5 \\ 1 \\ 2.5 \end{bmatrix} \neq \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}.$$

To measure our success at fitting the data, we calculate the residual of the least squares solution \bar{x} as

$$r = b - A\bar{x} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 2.5 \\ 1 \\ 2.5 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 0.0 \\ 0.5 \end{bmatrix}.$$

If the residual is the zero vector, then we have solved the original system $Ax = b$ exactly. If not, the Euclidean length of the residual vector is a backward error measure of how far \bar{x} is from being a solution.

There are at least three ways to express the size of the residual. The Euclidean length of a vector,

$$\|r\|_2 = \sqrt{r_1^2 + \cdots + r_m^2}, \quad (4.7)$$

is a norm in the sense of Chapter 2, called the **2-norm**. The **squared error**

$$SE = r_1^2 + \cdots + r_m^2,$$

and the **root mean squared error** (the root of the mean of the squared error)

$$RMSE = \sqrt{SE/m} = \sqrt{(r_1^2 + \cdots + r_m^2)/m}, \quad (4.8)$$

are also used to measure the error of the least squares solution. The three expressions are closely related; namely

$$RMSE = \frac{\sqrt{SE}}{\sqrt{m}} = \frac{\|r\|_2}{\sqrt{m}},$$

so finding the \bar{x} that minimizes one, minimizes all. For Example 4.1, the $SE = (.5)^2 + 0^2 + (-.5)^2 = 0.5$, the 2-norm of the error is $\|r\|_2 = \sqrt{0.5} \approx 0.707$, and the $RMSE = \sqrt{0.5/3} = 1/\sqrt{6} \approx 0.408$.

► EXAMPLE 4.2


Solve the least squares problem $\begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -3 \\ 15 \\ 9 \end{bmatrix}$.

The normal equations $A^T Ax = A^T b$ are

$$\begin{bmatrix} 9 & 6 \\ 6 & 29 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 45 \\ 75 \end{bmatrix}.$$

The solution of the normal equations are $\bar{x}_1 = 3.8$ and $\bar{x}_2 = 1.8$. The residual vector is

$$\begin{aligned} r = b - A\bar{x} &= \begin{bmatrix} -3 \\ 15 \\ 9 \end{bmatrix} - \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} 3.8 \\ 1.8 \end{bmatrix} \\ &= \begin{bmatrix} -3 \\ 15 \\ 9 \end{bmatrix} - \begin{bmatrix} -3.4 \\ 13 \\ 11.2 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 2 \\ -2.2 \end{bmatrix}, \end{aligned}$$

which has Euclidean norm $\|e\|_2 = \sqrt{(0.4)^2 + 2^2 + (-2.2)^2} = 3$. This problem is solved in an alternative way in Example 4.14. 

4.1.2 Fitting models to data

Let $(t_1, y_1), \dots, (t_m, y_m)$ be a set of points in the plane, which we will often refer to as the “data points.” Given a fixed class of models, such as all lines $y = c_1 + c_2t$, we can seek to locate the specific instance of the model that best fits the data points in the 2-norm. The core of the least squares idea consists of measuring the residual of the fit by the squared errors of the model at the data points and finding the model parameters that minimize this quantity. This criterion is displayed in Figure 4.2.

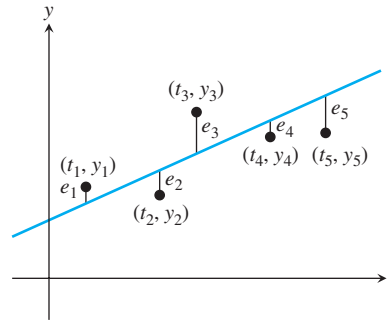


Figure 4.2 Least squares fitting of a line to data. The best line is the one for which the squared error $e_1^2 + e_2^2 + \dots + e_5^2$ is as small as possible among all lines $y = c_1 + c_2t$.

► **EXAMPLE 4.3** Find the line that best fits the three data points $(t, y) = (1, 2), (-1, 1)$, and $(1, 3)$ in Figure 4.3.

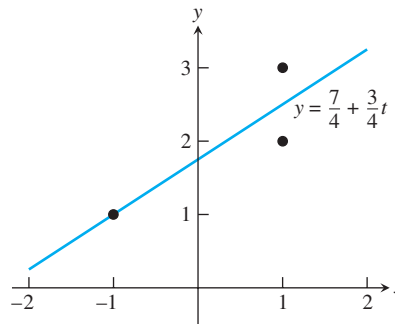


Figure 4.3 Best line in Example 4.3. One each of the data points lies above, on, and below the best line.

The model is $y = c_1 + c_2t$, and the goal is to find the best c_1 and c_2 . Substitution of the data points into the model yields

$$\begin{aligned} c_1 + c_2(1) &= 2 \\ c_1 + c_2(-1) &= 1 \\ c_1 + c_2(1) &= 3, \end{aligned}$$

or, in matrix form,

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}.$$

We know this system has no solution (c_1, c_2) for two separate reasons. First, if there is a solution, then the $y = c_1 + c_2t$ would be a line containing the three data points. However, it is easily seen that the points are not collinear. Second, this is the system of equation (4.2) that we discussed at the beginning of this chapter. We noticed then that the first and third equations are inconsistent, and we found that the best solution in terms of least squares is $(c_1, c_2) = (7/4, 3/4)$. Therefore, the best line is $y = 7/4 + 3/4t$. ◀

We can evaluate the fit by using the statistics defined earlier. The residuals at the data points are

t	y	line	error
1	2	2.5	-0.5
-1	1	1.0	0.0
1	3	2.5	0.5

and the RMSE is $1/\sqrt{6}$, as seen earlier.

The previous example suggests a three-step program for solving least squares data-fitting problems.

Fitting data by least squares

Given a set of m data points $(t_1, y_1), \dots, (t_m, y_m)$.

STEP 1. Choose a model. Identify a parameterized model, such as $y = c_1 + c_2t$, which will be used to fit the data.

STEP 2. Force the model to fit the data. Substitute the data points into the model. Each data point creates an equation whose unknowns are the parameters, such as c_1 and c_2 in the line model. This results in a system $Ax = b$, where the unknown x represents the unknown parameters.

STEP 3. Solve the normal equations. The least squares solution for the parameters will be found as the solution to the system of normal equations $A^T Ax = A^T b$.

These steps are demonstrated in the following example:

► **EXAMPLE 4.4** Find the best line and best parabola for the four data points $(-1, 1)$, $(0, 0)$, $(1, 0)$, $(2, -2)$ in Figure 4.4.

In accordance with the preceding program, we will follow three steps:

(1) Choose the model $y = c_1 + c_2t$ as before. (2) Forcing the model to fit the data yields

SPOTLIGHT ON

Compression

Least squares is a classic example of data compression. The input consists of a set of data points, and the output is a model that, with a relatively few parameters, fits the data as well as possible. Usually, the reason for using least squares is to replace noisy data with a plausible underlying model. The model is then often used for signal prediction or classification purposes.

In Section 4.2, various models are used to fit data, including polynomials, exponentials, and trigonometric functions. The trigonometric approach will be pursued further in Chapters 10 and 11, where elementary Fourier analysis is discussed as an introduction to signal processing.

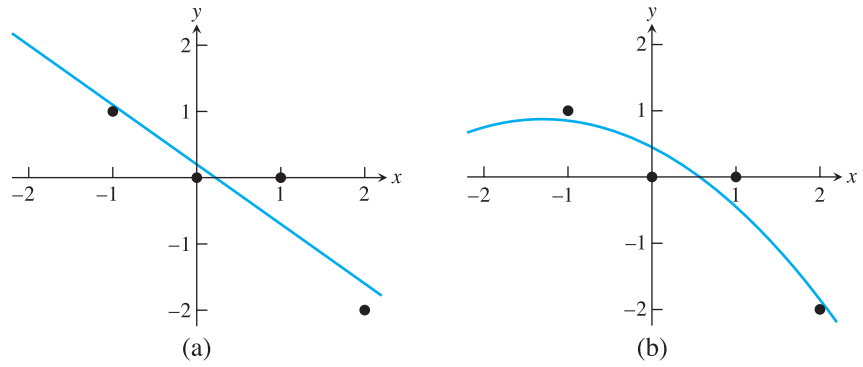


Figure 4.4 Least Squares Fits to Data Points in Example 4.4. (a) Best line $y = 0.2 - 0.9t$. RMSE is 0.418. (b) Best parabola $y = 0.45 - 0.65t - 0.25t^2$. RMSE is 0.335.

$$\begin{aligned} c_1 + c_2(-1) &= 1 \\ c_1 + c_2(0) &= 0 \\ c_1 + c_2(1) &= 0 \\ c_1 + c_2(2) &= -2, \end{aligned}$$

or, in matrix form,

$$\begin{bmatrix} 1 & -1 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -2 \end{bmatrix}.$$

(3) The normal equations are

$$\begin{bmatrix} 4 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} -1 \\ -5 \end{bmatrix}.$$

Solving for the coefficients c_1 and c_2 results in the best line $y = c_1 + c_2t = 0.2 - 0.9t$.

The residuals are

t	y	line	error
-1	1	1.1	-0.1
0	0	0.2	-0.2
1	0	-0.7	0.7
2	-2	-1.6	-0.4

The error statistics are squared error $SE = (-.1)^2 + (-.2)^2 + (.7)^2 + (-.4)^2 = 0.7$ and $RMSE = \sqrt{.7}/\sqrt{4} = 0.418$.

Next, we extend this example by keeping the same four data points, but changing the model. Set $y = c_1 + c_2t + c_3t^2$ and substitute the data points to yield

$$\begin{aligned} c_1 + c_2(-1) + c_3(-1)^2 &= 1 \\ c_1 + c_2(0) + c_3(0)^2 &= 0 \\ c_1 + c_2(1) + c_3(1)^2 &= 0 \\ c_1 + c_2(2) + c_3(2)^2 &= -2, \end{aligned}$$

SPOTLIGHT ON

Conditioning

Since input data is assumed to be subject to errors in least squares problems, it is especially important to reduce error magnification. We have presented the normal equations as the most straightforward approach to solving the least squares problem, and it is fine for small problems. However, the condition number $\text{cond}(A^T A)$ is approximately the square of the original $\text{cond}(A)$, which will greatly increase the possibility that the problem is ill-conditioned. More sophisticated methods allow computing the least squares solution directly from A without forming $A^T A$. These methods are based on the QR-factorization, introduced in Section 4.3, and the singular value decomposition of Chapter 12.

or, in matrix form,


$$\begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ -2 \end{bmatrix}.$$

This time, the normal equations are three equations in three unknowns:

$$\begin{bmatrix} 4 & 2 & 6 \\ 2 & 6 & 8 \\ 6 & 8 & 18 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -5 \\ -7 \end{bmatrix}.$$

Solving for the coefficients results in the best parabola $y = c_1 + c_2 t + c_3 t^2 = 0.45 - 0.65t - 0.25t^2$. The residual errors are given in the following table:

t	y	parabola	error
-1	1	0.85	0.15
0	0	0.45	-0.45
1	0	-0.45	0.45
2	-2	-1.85	-0.15

The error statistics are squared error $\text{SE} = (.15)^2 + (-.45)^2 + (.45)^2 + (-.15)^2 = 0.45$ and $\text{RMSE} = \sqrt{.45}/\sqrt{4} \approx 0.335$. 

The MATLAB commands `polyfit` and `polyval` are designed not only to interpolate data, but also to fit data with polynomial models. For n input data points, `polyfit` used with input degree $n - 1$ returns the coefficients of the interpolating polynomial of degree $n - 1$. If the input degree is less than $n - 1$, `polyfit` will instead find the best least squares polynomial of that degree. For example, the commands

```
>> x0 = [-1 0 1 2];
>> y0 = [1 0 0 -2];
>> c = polyfit(x0, y0, 2);
>> x = -1:.01:2;
>> y = polyval(c, x);
>> plot(x0, y0, 'o', x, y)
```

find the coefficients of the least squares degree-two polynomial and plot it along with the given data from Example 4.4.

Example 4.4 shows that least squares modeling need not be restricted to finding best lines. By expanding the definition of the model, we can fit coefficients for any model as long as the coefficients enter the model in a linear way.

4.1.3 Conditioning of least squares

We have seen that the least squares problem reduces to solving the normal equations $A^T A \bar{x} = A^T b$. How accurately can the least squares solution \bar{x} be determined? This is a question about the forward error of the normal equations. We carry out a double precision numerical experiment to test this question, by solving the normal equations in a case where the correct answer is known.

► **EXAMPLE 4.5** Let $x_1 = 2.0, x_2 = 2.2, x_3 = 2.4, \dots, x_{11} = 4.0$ be equally spaced points in $[2, 4]$, and set $y_i = 1 + x_i + x_i^2 + x_i^3 + x_i^4 + x_i^5 + x_i^6 + x_i^7$ for $1 \leq i \leq 11$. Use the normal equations to find the least squares polynomial $P(x) = c_1 + c_2x + \dots + c_8x^7$ fitting the (x_i, y_i) .

A degree 7 polynomial is being fit to 11 data points lying on the degree 7 polynomial $P(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7$. Obviously, the correct least squares solution is $c_1 = c_2 = \dots = c_8 = 1$. Substituting the data points into the model $P(x)$ yields the system $Ac = b$:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^7 \\ 1 & x_2 & x_2^2 & \cdots & x_2^7 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{11} & x_{11}^2 & \cdots & x_{11}^7 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_8 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{11} \end{bmatrix}.$$

The coefficient matrix A is a **Van der Monde matrix**, a matrix whose j th column consists of the elements of the second column raised to the $(j - 1)$ st power. We use MATLAB to solve the normal equations:

```
>> x = (2+(0:10)/5)';
>> y = 1+x+x.^2+x.^3+x.^4+x.^5+x.^6+x.^7;
>> A = [x.^0 x x.^2 x.^3 x.^4 x.^5 x.^6 x.^7];
>> c = (A'*A) \ (A'*y)
```

```
c=
    1.5134
   -0.2644
    2.3211
    0.2408
    1.2592
    0.9474
    1.0059
    0.9997

>> cond(A'*A)

ans=
    1.4359e+019
```

Solving the normal equations in double precision cannot deliver an accurate value for the least squares solution. The condition number of $A^T A$ is too large to deal with in double precision arithmetic, and the normal equations are ill-conditioned, even though the original least squares problem is moderately conditioned. There is clearly room for improvement in the normal equations approach to least squares. In Example 4.15, we revisit this problem after developing an alternative that avoids forming $A^T A$. ◀

4.1 Exercises

1. Solve the normal equations to find the least squares solution and 2-norm error for the following inconsistent systems:

$$(a) \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} \quad (c) \begin{bmatrix} 1 & 2 \\ 1 & 1 \\ 2 & 1 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ 3 \\ 2 \end{bmatrix}$$

2. Find the least squares solutions and RMSE of the following systems:

$$(a) \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 2 \\ 1 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 2 \end{bmatrix}$$

3. Find the least squares solution of the inconsistent system

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix}.$$

4. Let $m \geq n$, let A be the $m \times n$ identity matrix (the principal submatrix of the $m \times m$ identity matrix), and let $b = [b_1, \dots, b_m]$ be a vector. Find the least squares solution of $Ax = b$ and the 2-norm error.
5. Prove that the 2-norm is a vector norm. You will need to use the Cauchy–Schwarz inequality $|u \cdot v| \leq \|u\|_2 \|v\|_2$.
6. Let A be an $n \times n$ nonsingular matrix. (a) Prove that $(A^T)^{-1} = (A^{-1})^T$. (b) Let b be an n -vector; then $Ax = b$ has exactly one solution. Prove that this solution satisfies the normal equations.
7. Find the best line through the set of data points, and find the RMSE:
(a) $(-3, 3), (-1, 2), (0, 1), (1, -1), (3, -4)$ (b) $(1, 1), (1, 2), (2, 2), (2, 3), (4, 3)$.
8. Find the best line through each set of data points, and find the RMSE:
(a) $(0, 0), (1, 3), (2, 3), (5, 6)$ (b) $(1, 2), (3, 2), (4, 1), (6, 3)$ (c) $(0, 5), (1, 3), (2, 3), (3, 1)$.
9. Find the best parabola through each data point set in Exercise 8, and compare the RMSE with the best-line fit.
10. Find the best degree 3 polynomial through each set in Exercise 8. Also, find the degree 3 interpolating polynomial, and compare.
11. Assume that the height of a model rocket is measured at four times, and the measured times and heights are $(t, h) = (1, 135), (2, 265), (3, 385), (4, 485)$, in seconds and meters. Fit the model $h = a + bt - 4.905t^2$ to estimate the eventual maximum height of the object and when it will return to earth.
12. Given data points $(x, y, z) = (0, 0, 3), (0, 1, 2), (1, 0, 3), (1, 1, 5), (1, 2, 6)$, find the plane in three dimensions (model $z = c_0 + c_1x + c_2y$) that best fits the data.

4.1 Computer Problems

- Form the normal equations, and compute the least squares solution and 2-norm error for the following inconsistent systems:

$$(a) \begin{bmatrix} 3 & -1 & 2 \\ 4 & 1 & 0 \\ -3 & 2 & 1 \\ 1 & 1 & 5 \\ -2 & 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ -5 \\ 15 \\ 0 \end{bmatrix} \quad (b) \begin{bmatrix} 4 & 2 & 3 & 0 \\ -2 & 3 & -1 & 1 \\ 1 & 3 & -4 & 2 \\ 1 & 0 & 1 & -1 \\ 3 & 1 & 3 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 10 \\ 0 \\ 2 \\ 0 \\ 5 \end{bmatrix}$$

- Consider the world oil production data of Computer Problem 3.2.3. Find the best least squares (a) line, (b) parabola, and (c) cubic curve through the 10 data points and the RMSE of the fits. Use each to estimate the 2010 production level. Which fit best represents the data in terms of RMSE?
- Consider the world population data of Computer Problem 3.1.1. Find the best least squares (a) line, (b) parabola through the data points, and the RMSE of the fit. In each case, estimate the 1980 population. Which fit gives the best estimate?
- Consider the carbon dioxide concentration data of Exercise 3.1.13. Find the best least squares (a) line, (b) parabola, and (c) cubic curve through the data points and the RMSE of the fit. In each case, estimate the 1950 CO₂ concentration.
- A company test-markets a new soft drink in 22 cities of approximately equal size. The selling price (in dollars) and the number sold per week in the cities are listed as follows:

city	price	sales/week
1	0.59	3980
2	0.80	2200
3	0.95	1850
4	0.45	6100
5	0.79	2100
6	0.99	1700
7	0.90	2000
8	0.65	4200
9	0.79	2440
10	0.69	3300
11	0.79	2300

city	price	sales/week
12	0.49	6000
13	1.09	1190
14	0.95	1960
15	0.79	2760
16	0.65	4330
17	0.45	6960
18	0.60	4160
19	0.89	1990
20	0.79	2860
21	0.99	1920
22	0.85	2160

- First, the company wants to find the “demand curve”: how many it will sell at each potential price. Let P denote price and S denote sales per week. Find the line $S = c_1 + c_2 P$ that best fits the data from the table in the sense of least squares. Find the normal equations and the coefficients c_1 and c_2 of the least squares line. Plot the least squares line along with the data, and calculate the root mean square error.
 - After studying the results of the test marketing, the company will set a single selling price P throughout the country. Given a manufacturing cost of \$0.23 per unit, the total profit (per city, per week) is $S(P - 0.23)$ dollars. Use the results of the preceding least squares approximation to find the selling price for which the company’s profit will be maximized.
- What is the “slope” of the parabola $y = x^2$ on $[0, 1]$? Find the best least squares line that fits the parabola at n evenly spaced points in the interval for (a) $n = 10$ and (b) $n = 20$. Plot the

- parabola and the lines. What do you expect the result to be as $n \rightarrow \infty$? (c) Find the minimum of the function $F(c_1, c_2) = \int_0^1 (x^2 - c_1 - c_2x)^2 dx$, and explain its relation to the problem.
- Find the least squares (a) line (b) parabola through the 13 data points of Figure 3.5 and the RMSE of each fit.
 - Let A be the $10 \times n$ matrix formed by the first n columns of the 10×10 Hilbert matrix. Let c be the n -vector $[1, \dots, 1]$, and set $b = Ac$. Use the normal equations to solve the least squares problem $Ax = b$ for (a) $n = 6$ (b) $n = 8$, and compare with the correct least squares solution $\bar{x} = c$. How many correct decimal places can be computed? Use condition number to explain the results. (This least squares problem is revisited in Computer Problem 4.3.7.)
 - Let x_1, \dots, x_{11} be 11 evenly spaced points in $[2, 4]$ and $y_i = 1 + x_i + x_i^2 + \dots + x_i^d$. Use the normal equations to compute the best degree d polynomial, where (a) $d = 5$ (b) $d = 6$ (c) $d = 8$. Compare with Example 4.5. How many correct decimal places of the coefficients can be computed? Use condition number to explain the results. (This least squares problem is revisited in Computer Problem 4.3.8.)
 - The following data, collected by US Bureau of Economic Analysis, lists the year-over-year percent change in mean disposable personal income in the United States during 15 election years. Also, the proportion of the U.S. electorate that voted for the incumbent party's presidential candidate is listed. The first line of the table says that income increased by 1.49% from 1951 to 1952, and that 44.6% of the electorate voted for Adlai Stevenson, the incumbent Democratic party's candidate for president. Find the best least squares linear model for incumbent party vote as a function of income change. Plot this line along with the 15 data points. How many percentage points of vote can the incumbent party expect for each additional percent of change in personal income?

year	% income change	% incumbent vote
1952	1.49	44.6
1956	3.03	57.8
1960	0.57	49.9
1964	5.74	61.3
1968	3.51	49.6
1972	3.73	61.8
1976	2.98	49.0
1980	-0.18	44.7
1984	6.23	59.2
1988	3.38	53.9
1992	2.15	46.5
1996	2.10	54.7
2000	3.93	50.3
2004	2.47	51.2
2008	-0.41	45.7

4.2 A SURVEY OF MODELS

The previous linear and polynomial models illustrate the use of least squares to fit data. The art of data modeling includes a wide variety of models, some derived from physical principles underlying the source of the data and others based on empirical factors.

4.2.1 Periodic data

Periodic data calls for periodic models. Outside air temperatures, for example, obey cycles on numerous timescales, including daily and yearly cycles governed by the rotation of the earth and the revolution of the earth around the sun. As a first example, hourly temperature data are fit to sines and cosines.

► **EXAMPLE 4.6** Fit the recorded temperatures in Washington, D.C., on January 1, 2001, as listed in the following table, to a periodic model:

time of day	t	temp (C)
12 mid.	0	-2.2
3 am	$\frac{1}{8}$	-2.8
6 am	$\frac{1}{4}$	-6.1
9 am	$\frac{3}{8}$	-3.9
12 noon	$\frac{1}{2}$	0.0
3 pm	$\frac{5}{8}$	1.1
6 pm	$\frac{3}{4}$	-0.6
9 pm	$\frac{7}{8}$	-1.1

We choose the model $y = c_1 + c_2 \cos 2\pi t + c_3 \sin 2\pi t$ to match the fact that temperature is roughly periodic with a period of 24 hours, at least in the absence of longer-term temperature movements. The model uses this information by fixing the period to be exactly one day, where we are using days for the t units. The variable t is listed in these units in the table.

Substituting the data into the model results in the following overdetermined system of linear equations:

$$\begin{aligned}
 c_1 + c_2 \cos 2\pi(0) + c_3 \sin 2\pi(0) &= -2.2 \\
 c_1 + c_2 \cos 2\pi\left(\frac{1}{8}\right) + c_3 \sin 2\pi\left(\frac{1}{8}\right) &= -2.8 \\
 c_1 + c_2 \cos 2\pi\left(\frac{1}{4}\right) + c_3 \sin 2\pi\left(\frac{1}{4}\right) &= -6.1 \\
 c_1 + c_2 \cos 2\pi\left(\frac{3}{8}\right) + c_3 \sin 2\pi\left(\frac{3}{8}\right) &= -3.9 \\
 c_1 + c_2 \cos 2\pi\left(\frac{1}{2}\right) + c_3 \sin 2\pi\left(\frac{1}{2}\right) &= 0.0 \\
 c_1 + c_2 \cos 2\pi\left(\frac{5}{8}\right) + c_3 \sin 2\pi\left(\frac{5}{8}\right) &= 1.1 \\
 c_1 + c_2 \cos 2\pi\left(\frac{3}{4}\right) + c_3 \sin 2\pi\left(\frac{3}{4}\right) &= -0.6 \\
 c_1 + c_2 \cos 2\pi\left(\frac{7}{8}\right) + c_3 \sin 2\pi\left(\frac{7}{8}\right) &= -1.1
 \end{aligned}$$

SPOTLIGHT ON

Orthogonality

The least squares problem can be simplified considerably by special choices of basis functions. The choices in Examples 4.6 and 4.7, for instance, yield normal equations already in diagonal form. This property of orthogonal basis functions is explored in detail in Chapter 10. Model (4.9) is a Fourier expansion.

The corresponding inconsistent matrix equation is $Ax = b$, where

$$A = \begin{bmatrix} 1 & \cos 0 & \sin 0 \\ 1 & \cos \frac{\pi}{4} & \sin \frac{\pi}{4} \\ 1 & \cos \frac{\pi}{2} & \sin \frac{\pi}{2} \\ 1 & \cos \frac{3\pi}{4} & \sin \frac{3\pi}{4} \\ 1 & \cos \pi & \sin \pi \\ 1 & \cos \frac{5\pi}{4} & \sin \frac{5\pi}{4} \\ 1 & \cos \frac{3\pi}{2} & \sin \frac{3\pi}{2} \\ 1 & \cos \frac{7\pi}{4} & \sin \frac{7\pi}{4} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & \sqrt{2}/2 & \sqrt{2}/2 \\ 1 & 0 & 1 \\ 1 & -\sqrt{2}/2 & \sqrt{2}/2 \\ 1 & -1 & 0 \\ 1 & -\sqrt{2}/2 & -\sqrt{2}/2 \\ 1 & 0 & -1 \\ 1 & \sqrt{2}/2 & -\sqrt{2}/2 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} -2.2 \\ -2.8 \\ -6.1 \\ -3.9 \\ 0.0 \\ 1.1 \\ -0.6 \\ -1.1 \end{bmatrix}.$$

The normal equations $A^T A c = A^T b$ are

$$\begin{bmatrix} 8 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} -15.6 \\ -2.9778 \\ -10.2376 \end{bmatrix},$$

which are easily solved as $c_1 = -1.95$, $c_2 = -0.7445$, and $c_3 = -2.5594$. The best version of the model, in the sense of least squares, is $y = -1.9500 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t$, with $\text{RMSE} \approx 1.063$. Figure 4.5(a) compares the least squares fit model with the actual hourly recorded temperatures.

► **EXAMPLE 4.7** Fit the temperature data to the improved model

$$y = c_1 + c_2 \cos 2\pi t + c_3 \sin 2\pi t + c_4 \cos 4\pi t. \quad (4.9)$$

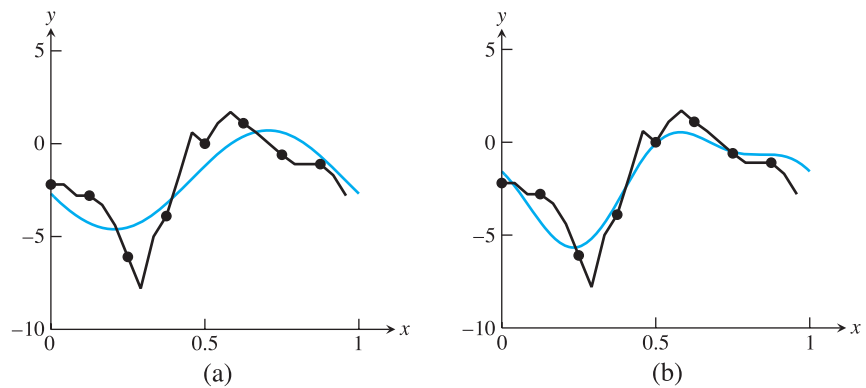


Figure 4.5 Least Squares Fits to Periodic Data in Example 4.6. (a) Sinusoid model $y = -1.95 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t$ shown in bold, along with recorded temperature trace on Jan 1, 2001. (b) Improved sinusoid $y = -1.95 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t + 1.125 \cos 4\pi t$ fits the data more closely.

The system of equations is now

$$\begin{aligned}
 c_1 + c_2 \cos 2\pi(0) + c_3 \sin 2\pi(0) + c_4 \cos 4\pi(0) &= -2.2 \\
 c_1 + c_2 \cos 2\pi\left(\frac{1}{8}\right) + c_3 \sin 2\pi\left(\frac{1}{8}\right) + c_4 \cos 4\pi\left(\frac{1}{8}\right) &= -2.8 \\
 c_1 + c_2 \cos 2\pi\left(\frac{1}{4}\right) + c_3 \sin 2\pi\left(\frac{1}{4}\right) + c_4 \cos 4\pi\left(\frac{1}{4}\right) &= -6.1 \\
 c_1 + c_2 \cos 2\pi\left(\frac{3}{8}\right) + c_3 \sin 2\pi\left(\frac{3}{8}\right) + c_4 \cos 4\pi\left(\frac{3}{8}\right) &= -3.9 \\
 c_1 + c_2 \cos 2\pi\left(\frac{1}{2}\right) + c_3 \sin 2\pi\left(\frac{1}{2}\right) + c_4 \cos 4\pi\left(\frac{1}{2}\right) &= 0.0 \\
 c_1 + c_2 \cos 2\pi\left(\frac{5}{8}\right) + c_3 \sin 2\pi\left(\frac{5}{8}\right) + c_4 \cos 4\pi\left(\frac{5}{8}\right) &= 1.1 \\
 c_1 + c_2 \cos 2\pi\left(\frac{3}{4}\right) + c_3 \sin 2\pi\left(\frac{3}{4}\right) + c_4 \cos 4\pi\left(\frac{3}{4}\right) &= -0.6 \\
 c_1 + c_2 \cos 2\pi\left(\frac{7}{8}\right) + c_3 \sin 2\pi\left(\frac{7}{8}\right) + c_4 \cos 4\pi\left(\frac{7}{8}\right) &= -1.1,
 \end{aligned}$$

leading to the following normal equations:

$$\begin{bmatrix} 8 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} -15.6 \\ -2.9778 \\ -10.2376 \\ 4.5 \end{bmatrix}.$$

The solutions are $c_1 = -1.95$, $c_2 = -0.7445$, $c_3 = -2.5594$, and $c_4 = 1.125$, with $\text{RMSE} \approx 0.705$. Figure 4.5(b) shows that the extended model $y = -1.95 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t + 1.125 \cos 4\pi t$ substantially improves the fit. ◀

4.2.2 Data linearization

Exponential growth of a population is implied when its rate of change is proportional to its size. Under perfect conditions, when the growth environment is unchanging and when the population is well below the carrying capacity of the environment, the model is a good representation.

The **exponential model**

$$y = c_1 e^{c_2 t} \quad (4.10)$$

cannot be directly fit by least squares because c_2 does not appear linearly in the model equation. Once the data points are substituted into the model, the difficulty is clear: The set of equations to solve for the coefficients are nonlinear and cannot be expressed as a linear system $Ax = b$. Therefore, our derivation of the normal equations is irrelevant.

There are two ways to deal with the problem of nonlinear coefficients. The more difficult way is to directly minimize the least square error, that is, solve the nonlinear least squares problem. We return to this problem in Section 4.5. The simpler way is to change the problem. Instead of solving the original least squares problem, we can solve a different problem, which is related to the original, by “linearizing” the model.

In the case of the exponential model (4.10), the model is linearized by applying the natural logarithm:

$$\ln y = \ln(c_1 e^{c_2 t}) = \ln c_1 + c_2 t. \quad (4.11)$$

Note that for an exponential model, the graph of $\ln y$ is a linear plot in t . At first glance, it appears that we have only traded one problem for another. The c_2 coefficient is now linear in the model, but c_1 no longer is. However, by renaming $k = \ln c_1$, we can write

$$\ln y = k + c_2 t. \quad (4.12)$$

Now both coefficients k and c_2 are linear in the model. After solving the normal equations for the best k and c_2 , we can find the corresponding $c_1 = e^k$ if we wish.

It should be noted that our way out of the difficulty of nonlinear coefficients was to change the problem. The original least squares problem we posed was to fit the data to (4.10)—that is, to find c_1, c_2 that minimize

$$(c_1 e^{c_2 t_1} - y_1)^2 + \cdots + (c_1 e^{c_2 t_m} - y_m)^2, \quad (4.13)$$

the sum of squares of the residuals of the equations $c_1 e^{c_2 t_i} = y_i$ for $i = 1, \dots, m$. For now, we solve the revised problem minimizing least squares error in “log space”—that is, by finding c_1, c_2 that minimizes

$$(\ln c_1 + c_2 t_1 - \ln y_1)^2 + \cdots + (\ln c_1 + c_2 t_m - \ln y_m)^2, \quad (4.14)$$

the sum of squares of the residuals of the equations $\ln c_1 + c_2 t_i = \ln y_i$ for $i = 1, \dots, m$. These are two different minimizations and have different solutions, meaning that they generally result in different values of the coefficients c_1, c_2 .

Which method is correct for this problem, the nonlinear least squares of (4.13) or the model-linearized version (4.14)? The former is least squares, as we have defined it. The latter is not. However, depending on the context of the data, either may be the more natural choice. To answer the question, the user needs to decide which errors are most important to minimize, the errors in the original sense or the errors in “log space.” In fact, the log model is linear, and it may be argued that only after log-transforming the data to a linear relation is it natural to evaluate the fitness of the model.

► **EXAMPLE 4.8** Use model linearization to find the best least squares exponential fit $y = c_1 e^{c_2 t}$ to the following world automobile supply data:

year	cars ($\times 10^6$)
1950	53.05
1955	73.04
1960	98.31
1965	139.78
1970	193.48
1975	260.20
1980	320.39

The data describe the number of automobiles operating throughout the world in the given year. Define the time variable t in terms of years since 1950. Solving the linear least squares problem yields $k_1 \approx 3.9896$, $c_2 \approx 0.06152$. Since $c_1 \approx e^{3.9896} \approx 54.03$, the model

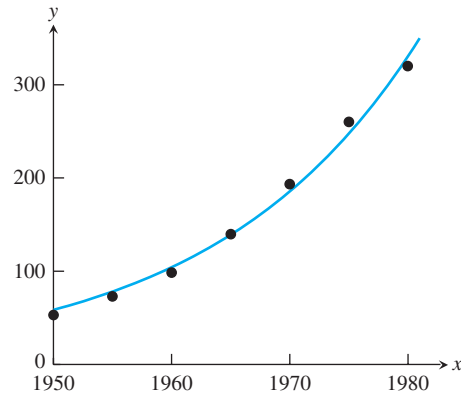


Figure 4.6 Exponential fit of world automobile supply data, using linearization.

The best least squares fit is $y = 54.03e^{0.06152t}$. Compare with Figure 4.14.

is $y = 54.03e^{0.06152t}$. The RMSE of the log-linearized model in log space is ≈ 0.0357 , while RMSE of the original exponential model is ≈ 9.56 . The best model and data are plotted in Figure 4.6. ▶

▶ EXAMPLE 4.9 The number of transistors on Intel central processing units since the early 1970s is given in the table that follows. Fit the model $y = c_1e^{c_2t}$ to the data.

CPU	year	transistors
4004	1971	2,250
8008	1972	2,500
8080	1974	5,000
8086	1978	29,000
286	1982	120,000
386	1985	275,000
486	1989	1,180,000
Pentium	1993	3,100,000
Pentium II	1997	7,500,000
Pentium III	1999	24,000,000
Pentium 4	2000	42,000,000
Itanium	2002	220,000,000
Itanium 2	2003	410,000,000

Parameters will be fit by using model linearization (4.11). Linearizing the model gives

$$\ln y = k + c_2t.$$

We will let $t = 0$ correspond to the year 1970. Substituting the data into the linearized model yields

$$\begin{aligned} k + c_2(1) &= \ln 2250 \\ k + c_2(2) &= \ln 2500 \\ k + c_2(4) &= \ln 5000 \\ k + c_2(8) &= \ln 29000, \end{aligned} \tag{4.15}$$

and so forth. The matrix equation is $Ax = b$, where $x = (k, c_2)$,

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 4 \\ 1 & 8 \\ \vdots & \vdots \\ 1 & 33 \end{bmatrix}, \text{ and } b = \begin{bmatrix} \ln 2250 \\ \ln 2500 \\ \ln 5000 \\ \ln 29000 \\ \vdots \\ \ln 410000000 \end{bmatrix}. \quad (4.16)$$

The normal equations $A^T A x = A^T b$ are

$$\begin{bmatrix} 13 & 235 \\ 235 & 5927 \end{bmatrix} \begin{bmatrix} k \\ c_2 \end{bmatrix} = \begin{bmatrix} 176.90 \\ 3793.23 \end{bmatrix},$$

which has solution $k \approx 7.197$ and $c_2 \approx 0.3546$, leading to $c_1 = e^k \approx 1335.3$. The exponential curve $y = 1335.3e^{0.3546t}$ is shown in Figure 4.7 along with the data. The doubling time for the law is $\ln 2/c_2 \approx 1.95$ years. Gordon C. Moore, cofounder of Intel, predicted in 1965 that over the ensuing decade, computing power would double every 2 years. Astoundingly, that exponential rate has continued for 40 years. There is some evidence in Figure 4.7 that this rate has accelerated since 2000.

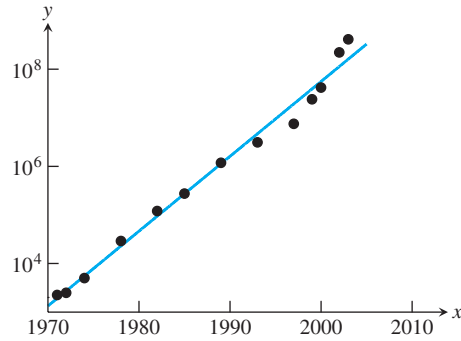


Figure 4.7 Semilog Plot of Moore's Law. Number of transistors on CPU chip versus year.

Another important example with nonlinear coefficients is the **power law** model $y = c_1 t^{c_2}$. This model also can be simplified with linearization by taking logs of both sides:

$$\begin{aligned} \ln y &= \ln c_1 + c_2 \ln t \\ &= k + c_2 \ln t. \end{aligned} \quad (4.17)$$

Substitution of data into the model will give

$$k + c_2 \ln t_1 = \ln y_1 \quad (4.18)$$

$$\vdots$$

$$k + c_2 \ln t_n = \ln y_n, \quad (4.19)$$

resulting in the matrix form

$$A = \begin{bmatrix} 1 & \ln t_1 \\ \vdots & \vdots \\ 1 & \ln t_n \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} \ln y_1 \\ \vdots \\ \ln y_n \end{bmatrix}. \quad (4.20)$$

The normal equations allow determination of k and c_2 , and $c_1 = e^k$.

► **EXAMPLE 4.10** Use linearization to fit the given height–weight data with a power law model.

The mean height and weight of boys ages 2–11 were collected in the U.S. National Health and Nutrition Examination Survey by the Centers for Disease Control (CDC) in 2002, resulting in the following table:

age (yrs.)	height (m)	weight (kg)
2	0.9120	13.7
3	0.9860	15.9
4	1.0600	18.5
5	1.1300	21.3
6	1.1900	23.5
7	1.2600	27.2
8	1.3200	32.7
9	1.3800	36.0
10	1.4100	38.6
11	1.4900	43.7

Following the preceding strategy, the resulting power law for weight versus height is $W = 16.3H^{2.42}$. The relationship is graphed in Figure 4.8. Since weight is a proxy for volume, the coefficient $c_2 \approx 2.42$ can be viewed as the “effective dimension” of the human body.

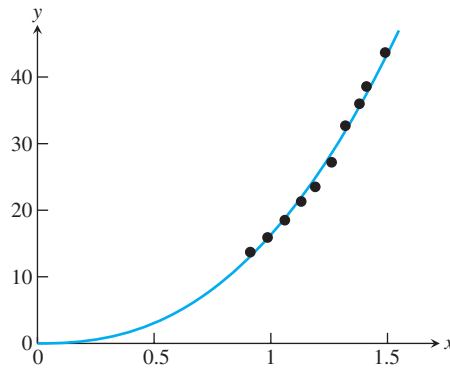


Figure 4.8 Power law of weight versus height for 2–11-year-olds. The best fit formula is $W = 16.3H^{2.42}$.

The time course of drug concentration y in the bloodstream is well described by

$$y = c_1 t e^{c_2 t}, \quad (4.21)$$

where t denotes time after the drug was administered. The characteristics of the model are a quick rise as the drug enters the bloodstream, followed by slow exponential decay. The **half-life** of the drug is the time from the peak concentration to the time it drops to half that level. The model can be linearized by applying the natural logarithm to both sides, producing

$$\begin{aligned} \ln y &= \ln c_1 + \ln t + c_2 t \\ k + c_2 t &= \ln y - \ln t, \end{aligned}$$

where we have set $k = \ln c_1$. This leads to the matrix equation $Ax = b$, where

$$A = \begin{bmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_m \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} \ln y_1 - \ln t_1 \\ \vdots \\ \ln y_m - \ln t_m \end{bmatrix}. \quad (4.22)$$

The normal equations are solved for k and c_2 , and $c_1 = e^k$.

► **EXAMPLE 4.11** Fit the model (4.21) with the measured level of the drug norfluoxetine in a patient's bloodstream, given in the following table:

hour	concentration (ng/ml)
1	8.0
2	12.3
3	15.5
4	16.8
5	17.1
6	15.8
7	15.2
8	14.0

Solving the normal equations yields $k \approx 2.28$ and $c_2 \approx -0.215$, and $c_1 \approx e^{2.28} \approx 9.77$. The best version of the model is $y = 9.77te^{-0.215t}$, plotted in Figure 4.9. From the model, the timing of the peak concentration and the half-life can be estimated. (See Computer Problem 5.)

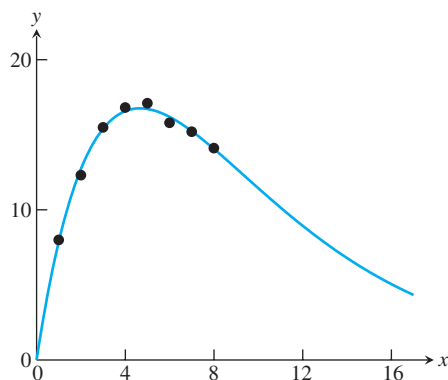


Figure 4.9 Plot of drug concentration in blood. Model (4.21) shows exponential decay after initial peak.

It is important to realize that model linearization changes the least squares problem. The solution obtained will minimize the RMSE with respect to the linearized problem, not necessarily the original problem, which in general will have a different set of optimal parameters. If they enter the model nonlinearly, they cannot be computed from the normal equations, and we need nonlinear techniques to solve the original least squares problem. This is done in the Gauss–Newton Method in Section 4.5, where we revisit the automobile supply data and compare fitting the exponential model in linearized and nonlinearized forms.

4.2 Exercises

1. Fit data to the periodic model $y = F_3(t) = c_1 + c_2 \cos 2\pi t + c_3 \sin 2\pi t$. Find the 2-norm error and the RMSE.

t	y	t	y	t	y
0	1	0	1	0	3
(a) 1/4	3	(b) 1/4	3	(c) 1/2	1
1/2	2	1/2	2	1	3
3/4	0	3/4	1	3/2	2

2. Fit the data to the periodic models $F_3(t) = c_1 + c_2 \cos 2\pi t + c_3 \sin 2\pi t$ and $F_4(t) = c_1 + c_2 \cos 2\pi t + c_3 \sin 2\pi t + c_4 \cos 4\pi t$. Find the 2-norm errors $\|e\|_2$ and compare the fits of F_3 and F_4 .

t	y	t	y
0	0	0	4
1/6	2	1/6	2
(a) 1/3	0	(b) 1/3	0
1/2	-1	1/2	-5
2/3	1	2/3	-1
5/6	1	5/6	3

3. Fit data to the exponential model by using linearization. Find the 2-norm of the difference between the data points y_i and the best model $c_1 e^{c_2 t_i}$.

t	y	t	y
-2	1	0	1
(a) 0	2	(b) 1	1
1	2	1	2
2	5	2	4

4. Fit data to the exponential model by using linearization. Find the 2-norm of the difference between the data points y_i and the best model $c_1 e^{c_2 t_i}$.

t	y	t	y
-2	4	0	10
(a) -1	2	(b) 1	5
1	1	2	2
2	1/2	3	1

5. Fit data to the power law model by using linearization. Find the RMSE of the fit.

t	y	t	y
1	6	1	2
(a) 2	2	(b) 1	4
3	1	2	5
4	1	3	6
		5	10

6. Fit data to the drug concentration model (4.21). Find the RMSE of the fit.

	t	y		t	y
(a)	1	3	(b)	1	2
	2	4		2	4
	3	5		3	3
	4	5		4	2

4.2 Computer Problems

1. Fit the monthly data for Japan 2003 oil consumption, shown in the following table, with the periodic model (4.9), and calculate the RMSE:

month	oil use (10^6 bbl/day)
Jan	6.224
Feb	6.665
Mar	6.241
Apr	5.302
May	5.073
Jun	5.127
Jul	4.994
Aug	5.012
Sep	5.108
Oct	5.377
Nov	5.510
Dec	6.372

- The temperature data in Example 4.6 was taken from the Weather Underground website www.wunderground.com. Find a similar selection of hourly temperature data from a location and date of your choice, and fit it with the two sinusoidal models of the example.
- Consider the world population data of Computer Problem 3.1.1. Find the best exponential fit of the data points by using linearization. Estimate the 1980 population, and find the estimation error.
- Consider the carbon dioxide concentration data of Exercise 3.1.17. Find the best exponential fit of the difference between the CO_2 level and the background (279 ppm) by using linearization. Estimate the 1950 CO_2 concentration, and find the estimation error.
- (a) Find the time at which the maximum concentration is reached in model (4.21). (b) Use an equation solver to estimate the half-life from the model in Example 4.11.
- The bloodstream concentration of a drug, measured hourly after administration, is given in the accompanying table. Fit the model (4.21). Find the estimated maximum and the half-life. Suppose that the therapeutic range for the drug is 4–15 ng/ml. Use the equation solver of your choice to estimate the time the drug concentration stays within therapeutic levels.

hour	concentration (ng/ml)
1	6.2
2	9.5
3	12.3
4	13.9
5	14.6
6	13.5
7	13.3
8	12.7
9	12.4
10	11.9

7. The file `windmill.txt`, available from the textbook website, is a list of 60 numbers which represent the monthly megawatt-hours generated from Jan. 2005 to Dec. 2009 by a wind turbine owned by the Minnkota Power Cooperative near Valley City, ND. The data is currently available at <http://www.minnkota.com>. For reference, a typical home uses around 1 MWh per month.

(a) Find a rough model of power output as a yearly periodic function. Fit the data to equation (4.9),

$$f(t) = c_1 + c_2 \cos 2\pi t + c_3 \sin 2\pi t + c_4 \cos 4\pi t$$

where the units of t are years, that is $0 \leq t \leq 5$, and write down the resulting function.

(b) Plot the data and the model function for years $0 \leq t \leq 5$. What features of the data are captured by the model?

8. The file `scrippsy.txt`, available from the textbook website, is a list of 50 numbers which represent the concentration of atmospheric carbon dioxide, in parts per million by volume (ppv), recorded at Mauna Loa, Hawaii, each May 15 of the years 1961 to 2010. The data is part of a data collection effort initiated by Charles Keeling of the Scripps Oceanographic Institute (Keeling et al. [2001]). Subtract the background level 279 ppm as in Computer Problem 4, and fit the data to an exponential model. Plot the data along with the best fit exponential function, and report the RMSE.

9. The file `scrippsm.txt`, available from the textbook website, is a list of 180 numbers which represent the concentration of atmospheric carbon dioxide, in parts per million by volume (ppv), recorded monthly at Mauna Loa from Jan. 1996 to Dec. 2010, taken from the same Scripps study as Computer Problem 8.

(a) Carry out a least squares fit of the CO_2 data using the model

$$f(t) = c_1 + c_2 t + c_3 \cos 2\pi t + c_4 \sin 2\pi t$$

where t is measured in months. Report the best fit coefficients c_i and the RMSE of the fit. Plot the continuous curve from Jan. 1989 to the end of this year, including the 180 data points in the plot.

(b) Use your model to predict the CO_2 concentration in May 2004, Sept. 2004, May 2005, and Sept. 2005. These months tend to contain the yearly maxima and minima of the CO_2 cycle. The actual recorded values are 380.63, 374.06, 382.45, and 376.73 ppv, respectively. Report the model error at these four points.

(c) Add the extra term $c_5 \cos 4\pi t$ and redo parts (a) and (b). Compare the new RMSE and four model errors.

- (d) Repeat part (c) using the extra term cs^2 . Which term leads to more improvement in the model, part (c) or (d)?
- (e) Add both terms from (c) and (d) and redo parts (a) and (b). Prepare a table summarizing your results from all parts of the problem, and try to provide an explanation for the results.
- See the website <http://scrippsco2.ucsd.edu> for much more data and analysis of the Scripps carbon dioxide study.

4.3 QR FACTORIZATION

In Chapter 2, the LU factorization was used to solve matrix equations. The factorization is useful because it encodes the steps of Gaussian elimination. In this section, we develop the QR factorization as a way to solve least squares calculations that is superior to the normal equations.

After introducing the factorization by way of Gram–Schmidt orthogonalization, we return to Example 4.5, for which the normal equations turned out to be inadequate. Later in this section, Householder reflections are introduced as a more efficient method of computing Q and R .

4.3.1 Gram–Schmidt orthogonalization and least squares

The Gram–Schmidt method orthogonalizes a set of vectors. Given an input set of m -dimensional vectors, the goal is to find an orthogonal coordinate system for the subspace spanned by the set. More precisely, given n linearly independent input vectors, it computes n mutually perpendicular unit vectors spanning the same subspace as the input vectors. The unit length is with respect to the Euclidean or 2-norm (4.7), which is used throughout Chapter 4.

Let A_1, \dots, A_n be linearly independent vectors from R^m . Thus $n \leq m$. The Gram–Schmidt method begins by dividing A_1 by its length to make it a unit vector. Define

$$y_1 = A_1 \quad \text{and} \quad q_1 = \frac{y_1}{\|y_1\|_2}. \quad (4.23)$$

To find the second unit vector, subtract away the projection of A_2 in the direction of q_1 , and normalize the result:

$$y_2 = A_2 - q_1(q_1^T A_2), \quad \text{and} \quad q_2 = \frac{y_2}{\|y_2\|_2}. \quad (4.24)$$

Then $q_1^T y_2 = q_1^T (A_2 - q_1(q_1^T A_2)) = q_1^T A_2 - q_1^T A_2 = 0$, so q_1 and q_2 are pairwise orthogonal, as shown in Figure 4.10.

At the j th step, define

$$y_j = A_j - q_1(q_1^T A_j) - q_2(q_2^T A_j) - \dots - q_{j-1}(q_{j-1}^T A_j) \quad \text{and} \quad q_j = \frac{y_j}{\|y_j\|_2}. \quad (4.25)$$

It is clear that q_j is orthogonal to each of the previously produced q_i for $i = 1, \dots, j-1$, since (4.25) implies

$$\begin{aligned} q_i^T y_j &= q_i^T A_j - q_i^T q_1 q_1^T A_j - \dots - q_i^T q_{j-1} q_{j-1}^T A_j \\ &= q_i^T A_j - q_i^T q_i q_i^T A_j = 0, \end{aligned}$$

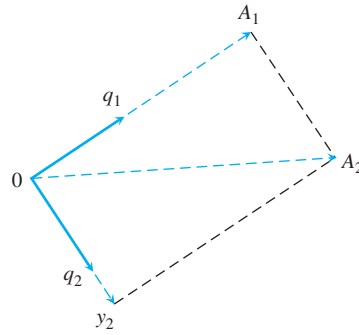


Figure 4.10 Gram–Schmidt orthogonalization. The input vectors are A_1 and A_2 , and the output is the orthonormal set consisting of q_1 and q_2 . The second orthogonal vector q_2 is formed by subtracting the projection of A_2 in the direction of q_1 from A_2 , followed by normalizing.

where by induction hypothesis, the q_i are pairwise orthogonal for $i < j$. Geometrically, (4.25) corresponds to subtracting from A_j the projections of A_j onto the previously determined orthogonal vectors $q_i, i = 1, \dots, j - 1$. What remains is orthogonal to the q_i and, after dividing by its length to become a unit vector, is used as q_j . Therefore, the set $\{q_1, \dots, q_n\}$ consists of mutually orthogonal vectors spanning the same subspace of R^m as $\{A_1, \dots, A_n\}$.

The result of Gram–Schmidt orthogonalization can be put into matrix form by introducing new notation for the dot products in the above calculation. Define $r_{jj} = \|y_j\|_2$ and $r_{ij} = q_i^T A_j$. Then (4.23) and (4.24) can be written

$$\begin{aligned} A_1 &= r_{11}q_1 \\ A_2 &= r_{12}q_1 + r_{22}q_2, \end{aligned}$$

and the general case (4.25) translates to

$$A_j = r_{1j}q_1 + \dots + r_{j-1,j}q_{j-1} + r_{jj}q_j.$$

Therefore, the result of Gram–Schmidt orthogonalization can be written in matrix form as

$$(A_1 | \dots | A_n) = (q_1 | \dots | q_n) \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{bmatrix}, \quad (4.26)$$

or $A = QR$, where we consider A to be the matrix consisting of the columns A_j . We call this the **reduced QR factorization**; the full version is just ahead. The assumption that the vectors A_j are linearly independent guarantees that the main diagonal coefficients r_{jj} are nonzero. Conversely, if A_j lies in the span of A_1, \dots, A_{j-1} , then the projections onto the latter vectors make up the entire vector, and $r_{jj} = \|y_j\|_2 = 0$.

► **EXAMPLE 4.12** Find the reduced QR factorization by applying Gram–Schmidt orthogonalization to the

columns of $A = \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix}$.

Set $y_1 = A_1 = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$. Then $r_{11} = \|y_1\|_2 = \sqrt{1^2 + 2^2 + 2^2} = 3$, and the first unit vector is

$$q_1 = \frac{y_1}{\|y_1\|_2} = \begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \\ \frac{2}{3} \end{bmatrix}.$$

To find the second unit vector, set

$$y_2 = A_2 - q_1 q_1^T A_2 = \begin{bmatrix} -4 \\ 3 \\ 2 \end{bmatrix} - \begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \\ \frac{2}{3} \end{bmatrix} 2 = \begin{bmatrix} -\frac{14}{3} \\ \frac{5}{3} \\ \frac{2}{3} \end{bmatrix}$$

and

$$q_2 = \frac{y_2}{\|y_2\|_2} = \frac{1}{5} \begin{bmatrix} -\frac{14}{3} \\ \frac{5}{3} \\ \frac{2}{3} \end{bmatrix} = \begin{bmatrix} -\frac{14}{15} \\ \frac{1}{3} \\ \frac{2}{15} \end{bmatrix}.$$

Since $r_{12} = q_1^T A_2 = 2$ and $r_{22} = \|y_2\|_2 = 5$, the result written in matrix form (4.26) is

$$A = \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 1/3 & -14/15 \\ 2/3 & 1/3 \\ 2/3 & 2/15 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 0 & 5 \end{bmatrix} = QR.$$

We use the term “classical” for this version of Gram–Schmidt, since we will provide an upgraded, or “modified,” version at the end of this section.

Classical Gram–Schmidt orthogonalization

```

    Let  $A_j, j = 1, \dots, n$  be linearly independent vectors.
for  $j = 1, 2, \dots, n$ 
     $y = A_j$ 
    for  $i = 1, 2, \dots, j - 1$ 
         $r_{ij} = q_i^T A_j$ 
         $y = y - r_{ij} q_i$ 
    end
     $r_{jj} = \|y\|_2$ 
     $q_j = y/r_{jj}$ 
end
```

When the method is successful, it is customary to fill out the matrix of orthogonal unit vectors to a complete basis of R^m , to achieve the “full” QR factorization. This can be done, for example, by adding $m - n$ extra vectors to the A_j , so that the m vectors span R^m , and carrying out the Gram–Schmidt method. In terms of the basis of R^m formed by q_1, \dots, q_m , the original vectors can be expressed as

$$(A_1 | \cdots | A_n) = (q_1 | \cdots | q_m) \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \\ 0 & \cdots & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix}. \quad (4.27)$$

This matrix equation is the **full QR factorization** of the matrix $A = (A_1 | \cdots | A_n)$, formed by the original input vectors. Note the matrix sizes in the full QR factorization: A is $m \times n$, Q is a square $m \times m$ matrix, and the upper triangular matrix R is $m \times n$, the same size as A . The matrix Q in the full QR factorization has a special place in numerical analysis and is given a special definition.

DEFINITION 4.1 A square matrix Q is **orthogonal** if $Q^T = Q^{-1}$. □

Note that a square matrix is orthogonal if and only if its columns are pairwise orthogonal unit vectors (Exercise 9). Therefore, a full QR factorization is the equation $A = QR$, where Q is an orthogonal square matrix and R is an upper triangular matrix the same size as A .

The key property of an orthogonal matrix is that it preserves the Euclidean norm of a vector.

LEMMA 4.2 If Q is an orthogonal $m \times m$ matrix and x is an m -dimensional vector, then $\|Qx\|_2 = \|x\|_2$. ■

Proof. $\|Qx\|_2^2 = (Qx)^T Qx = x^T Q^T Qx = x^T x = \|x\|_2^2$. □

The product of two orthogonal $m \times m$ matrices is again orthogonal (Exercise 10). The QR factorization of an $m \times m$ matrix by the Gram–Schmidt method requires approximately m^3 multiplication/divisions, three times more than the LU factorization, plus about the same number of additions (Exercise 11).

► **EXAMPLE 4.13** Find the full QR factorization of $A = \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix}$.

SPOTLIGHT ON

Orthogonality

In Chapter 2, we found that the LU factorization is an efficient means of encoding the information of Gaussian elimination. In the same way, the QR factorization records the orthogonalization of a matrix, namely, the construction of an orthogonal set that spans the space of column vectors of A . Doing calculations with orthogonal matrices is preferable because (1) they are easy to invert by definition, and (2) by Lemma 4.2, they do not magnify errors.


In Example 4.12, we found the orthogonal unit vectors $q_1 = \begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \\ \frac{2}{3} \end{bmatrix}$ and

$$q_2 = \begin{bmatrix} -\frac{14}{15} \\ \frac{1}{3} \\ \frac{2}{15} \end{bmatrix}. \text{ Adding a third vector } A_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ leads to}$$

$$\begin{aligned} y_3 &= A_3 - q_1 q_1^T A_3 - q_2 q_2^T A_3 \\ &= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} \frac{1}{3} \\ \frac{2}{3} \\ \frac{2}{3} \end{bmatrix} \frac{1}{3} - \begin{bmatrix} -\frac{14}{15} \\ \frac{1}{3} \\ -\frac{2}{15} \end{bmatrix} \left(-\frac{14}{15}\right) = \frac{2}{225} \begin{bmatrix} 2 \\ 10 \\ -11 \end{bmatrix} \end{aligned}$$

and $q_3 = y_3 / \|y_3\| = \begin{bmatrix} \frac{2}{15} \\ \frac{10}{15} \\ -\frac{11}{15} \end{bmatrix}$. Putting the parts together, we obtain the full QR factorization

$$A = \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 1/3 & -14/15 & 2/15 \\ 2/3 & 1/3 & 2/3 \\ 2/3 & 2/15 & -11/15 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 0 & 5 \\ 0 & 0 \end{bmatrix} = QR.$$

Note that the choice of A_3 was arbitrary. Any third column vector linearly independent of the first two columns could be used. Compare this result with the reduced QR factorization in Example 4.12. 

The MATLAB command `qr` carries out the QR factorization on an $m \times n$ matrix. It does not use Gram–Schmidt orthogonalization, but uses more efficient and stable methods that will be introduced in a later subsection. The command

```
>> [Q, R] = qr(A, 0)
```

returns the reduced QR factorization, and

```
>> [Q, R] = qr(A)
```

returns the full QR factorization.

There are three major applications of the QR factorization. We will describe two of them here; the third is the QR algorithm for eigenvalue calculations, introduced in Chapter 12.

First, the QR factorization can be used to solve a system of n equations in n unknowns $Ax = b$. Just factor $A = QR$, and the equation $Ax = b$ becomes $QRx = b$ and $Rx = Q^T b$. Assuming that A is nonsingular, the diagonal entries of the upper triangular matrix R are nonzero, so that R is nonsingular. A triangular back substitution yields the solution x . As mentioned before, this approach is about three times more expensive in terms of complexity when compared with the LU approach.

The second application is to least squares. Let A be an $m \times n$ matrix with $m \geq n$. To minimize $\|Ax - b\|_2$, rewrite as $\|QRx - b\|_2 = \|Rx - Q^T b\|_2$ by Lemma 4.2.

The vector inside the Euclidean norm is

$$\begin{bmatrix} e_1 \\ \vdots \\ e_n \\ \hline e_{n+1} \\ \vdots \\ e_m \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \\ \hline 0 & \cdots & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} d_1 \\ \vdots \\ d_n \\ \hline d_{n+1} \\ \vdots \\ d_m \end{bmatrix} \quad (4.28)$$

where $d = Q^T b$. Assume that $r_{ii} \neq 0$. Then the upper part (e_1, \dots, e_n) of the error vector e can be made zero by back substitution. The choice of the x_i makes no difference for the lower part of the error vector; clearly, $(e_{n+1}, \dots, e_m) = (-d_{n+1}, \dots, -d_m)$. Therefore, the least squares solution is minimized by using the x from back-solving the upper part, and the least squares error is $\|e\|_2^2 = d_{n+1}^2 + \cdots + d_m^2$.

Least squares by QR factorization

Given the $m \times n$ inconsistent system

$$Ax = b,$$

find the full QR factorization $A = QR$ and set

$$\begin{aligned} \hat{R} &= \text{upper } n \times n \text{ submatrix of } R \\ \hat{d} &= \text{upper } n \text{ entries of } d = Q^T b \end{aligned}$$

Solve $\hat{R}\bar{x} = \hat{d}$ for least squares solution \bar{x} .


► **EXAMPLE 4.14** Use the full QR factorization to solve the least squares problem $\begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -3 \\ 15 \\ 9 \end{bmatrix}$.

We need to solve $Rx = Q^T b$, or

$$\begin{bmatrix} 3 & 2 \\ 0 & 5 \\ \hline 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \frac{1}{15} \begin{bmatrix} 5 & 10 & 10 \\ -14 & 5 & 2 \\ 2 & 10 & -11 \end{bmatrix} \begin{bmatrix} -3 \\ 15 \\ 9 \end{bmatrix} = \begin{bmatrix} 15 \\ 9 \\ \hline 3 \end{bmatrix}.$$

The least squares error will be $\|e\|_2 = \|(0, 0, 3)\|_2 = 3$. Equating the upper parts yields

$$\begin{bmatrix} 3 & 2 \\ 0 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 15 \\ 9 \end{bmatrix},$$

whose solution is $\bar{x}_1 = 3.8, \bar{x}_2 = 1.8$. This least squares problem was solved by the normal equations in Example 4.2. 

Finally, we return to the problem in Example 4.5 that led to an ill-conditioned system of normal equations.

SPOTLIGHT ON

Conditioning

In Chapter 2, we found that the best way to handle ill-conditioned problems is to avoid them. Example 4.15 is a classic case of that advice. While the normal equations of Example 4.5 are ill-conditioned, the QR approach solves least squares without constructing $A^T A$.

► **EXAMPLE 4.15** Use the full QR factorization to solve the least squares problem of Example 4.5.

The normal equations were notably unsuccessful in solving this least squares problem of 11 equations in 8 variables. We use the MATLAB `qr` command to carry out an alternative approach:

```
>> x=(2+(0:10)/5)';
>> y=1+x+x.^2+x.^3+x.^4+x.^5+x.^6+x.^7;
>> A=[x.^0 x.^1 x.^2 x.^3 x.^4 x.^5 x.^6 x.^7];
>> [Q,R]=qr(A);
>> b=Q'*y;
>> c=R(1:8,1:8)\b(1:8)

c=
    0.99999991014308
    1.00000021004107
    0.99999979186557
    1.00000011342980
    0.99999996325039
    1.00000000708455
    0.9999999924685
    1.00000000003409
```

Six decimal places of the correct solution $c = [1, \dots, 1]$ are found by using QR factorization. This approach finds the least squares solution without forming the normal equations, which have a condition number of about 10^{19} . ◀

4.3.2 Modified Gram–Schmidt orthogonalization

A slight modification to Gram–Schmidt turns out to enhance its accuracy in machine calculations. The new algorithm called modified Gram–Schmidt is mathematically equivalent to the original, or “classical” Gram–Schmidt algorithm.

Modified Gram–Schmidt orthogonalization

Let $A_j, j = 1, \dots, n$ be linearly independent vectors.

```
for j = 1, 2, ..., n
    y = A_j
    for i = 1, 2, ..., j - 1
        rij = qiT y
        y = y - rij qi
    end
    rjj = ||y||2
    qj = y/rjj
end
```

The only difference from classical Gram–Schmidt is that A_j is replaced by y in the innermost loop. Geometrically speaking, when projecting away the part of vector A_j in the direction of q_2 , for example, one should subtract away the projection of the remainder y of A_j with the q_1 part already removed, instead of the projection of A_j itself on q_2 . Modified Gram–Schmidt is the version that will be used in the GMRES algorithm in Section 4.4.

► **EXAMPLE 4.16** Compare the results of classical Gram–Schmidt and modified Gram–Schmidt, computed in double precision, on the matrix of almost-parallel vectors

$$\begin{bmatrix} 1 & 1 & 1 \\ \delta & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & \delta \end{bmatrix}$$

where $\delta = 10^{-10}$.

First, we apply classical Gram–Schmidt.

$$y_1 = A_1 = \begin{bmatrix} 1 \\ \delta \\ 0 \\ 0 \end{bmatrix} \quad \text{and} \quad q_1 = \frac{1}{\sqrt{1+\delta^2}} \begin{bmatrix} 1 \\ \delta \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ \delta \\ 0 \\ 0 \end{bmatrix}.$$

Note that $\delta^2 = 10^{-20}$ is a perfectly acceptable double precision number, but $1 + \delta^2 = 1$ after rounding. Then

$$y_2 = \begin{bmatrix} 1 \\ 0 \\ \delta \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ \delta \\ 0 \\ 0 \end{bmatrix} q_1^T A_2 = \begin{bmatrix} 1 \\ 0 \\ \delta \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ \delta \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -\delta \\ \delta \\ 0 \end{bmatrix} \quad \text{and} \quad q_2 = \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}$$

after dividing by $\|y_2\|_2 = \sqrt{\delta^2 + \delta^2} = \sqrt{2}\delta$. Completing classical Gram–Schmidt,


$$y_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \delta \end{bmatrix} - \begin{bmatrix} 1 \\ \delta \\ 0 \\ 0 \end{bmatrix} q_1^T A_3 - \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} q_2^T A_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \delta \end{bmatrix} - \begin{bmatrix} 1 \\ \delta \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -\delta \\ 0 \\ \delta \end{bmatrix} \quad \text{and} \quad q_3 = \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}.$$

Unfortunately, due to the double precision rounding done in the first step, q_2 and q_3 turn out to be not orthogonal:

$$q_2^T q_3 = \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}^T \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{2}.$$

On the other hand, modified Gram–Schmidt does much better. While q_1 and q_2 are calculated the same way, q_3 is found as

$$\begin{aligned} y_3^1 &= \begin{bmatrix} 1 \\ 0 \\ 0 \\ \delta \end{bmatrix} - \begin{bmatrix} 1 \\ \delta \\ 0 \\ 0 \end{bmatrix} q_1^T A_3 = \begin{bmatrix} 0 \\ -\delta \\ 0 \\ \delta \end{bmatrix}, \\ y_3 &= y_3^1 - \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} q_2^T y_3^1 = \begin{bmatrix} 0 \\ -\delta \\ 0 \\ \delta \end{bmatrix} - \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} \frac{\delta}{\sqrt{2}} \\ &= \begin{bmatrix} 0 \\ -\frac{\delta}{2} \\ -\frac{\delta}{2} \\ \delta \end{bmatrix} \quad \text{and} \quad q_3 = \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{6}} \\ -\frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \end{bmatrix}. \end{aligned}$$


Now $q_2^T q_3 = 0$ as desired. Note that for both classical and modified Gram–Schmidt, $q_1^T q_2$ is on the order of δ , so even modified Gram–Schmidt leaves room for improvement. Orthogonalization by Householder reflectors, described in the next section, is widely considered to be more computationally stable. 


4.3.3 Householder reflectors

Although the modified Gram–Schmidt orthogonalization method is an improved way to calculate the QR factorization of a matrix, it is not the best way. An alternative method using Householder reflectors requires fewer operations and is more stable, in the sense of amplification of rounding errors. In this section, we will define the reflectors and show how they are used to factorize a matrix.

A Householder reflector is an orthogonal matrix that reflects all m -vectors through an $m - 1$ dimensional plane. This means that the length of each vector is unchanged when multiplied by the matrix, making Householder reflectors ideal for moving vectors. Given a vector x that we would like to relocate to a vector w of equal length, the recipe for Householder reflectors gives a matrix H such that $Hx = w$.

The origin of the recipe is clear in Figure 4.11. Draw the $m - 1$ dimensional plane bisecting x and w , and perpendicular to the vector connecting them. Then reflect all vectors through the plane.

LEMMA 4.3 Assume that x and w are vectors of the same Euclidean length, $\|x\|_2 = \|w\|_2$. Then $w - x$ and $w + x$ are perpendicular. 

Proof. $(w - x)^T (w + x) = w^T w - x^T w + w^T x - x^T x = \|w\|^2 - \|x\|^2 = 0$. 

Define the vector $v = w - x$, and consider the projection matrix

$$P = \frac{vv^T}{v^T v}. \quad (4.29)$$

A **projection matrix** is a matrix that satisfies $P^2 = P$. Exercise 13 asks the reader to verify that P in (4.29) is a symmetric projection matrix and that $Pv = v$. Geometrically, for any vector u , Pu is the projection of u onto v . Figure 4.11 hints that if we subtract twice the projection Px from x , we should get w . To verify this, set $H = I - 2P$. Then

$$\begin{aligned} Hx &= x - 2Px \\ &= w - v - \frac{2vv^T x}{v^T v} \\ &= w - v - \frac{vv^T x}{v^T v} - \frac{vv^T (w - v)}{v^T v} \\ &= w - \frac{vv^T (w + x)}{v^T v} \\ &= w, \end{aligned} \quad (4.30)$$

the latter equality following from Lemma 4.3, since $w + x$ is orthogonal to $v = w - x$.

The matrix H is called a **Householder reflector**. Note that H is a symmetric (Exercise 14) and orthogonal matrix, since

$$\begin{aligned} H^T H &= HH = (I - 2P)(I - 2P) \\ &= I - 4P + 4P^2 \\ &= I. \end{aligned}$$

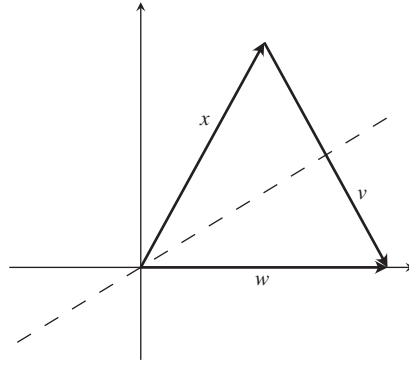


Figure 4.11 Householder reflector. Given equal length vectors x and w , reflection through the bisector of the angle between them (dotted line) exchanges them.

These facts are summarized in the following theorem:

THEOREM 4.4 Householder reflectors. Let x and w be vectors with $\|x\|_2 = \|w\|_2$ and define $v = w - x$. Then $H = I - 2vv^T/v^Tv$ is a symmetric orthogonal matrix and $Hx = w$. ■

► **EXAMPLE 4.17** Let $x = [3, 4]$ and $w = [5, 0]$. Find a Householder reflector H that satisfies $Hx = w$.

Set

$$v = w - x = \begin{bmatrix} 5 \\ 0 \end{bmatrix} - \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ -4 \end{bmatrix},$$

and define the projection matrix

$$P = \frac{vv^T}{v^Tv} = \frac{1}{20} \begin{bmatrix} 4 & -8 \\ -8 & 16 \end{bmatrix} = \begin{bmatrix} 0.2 & -0.4 \\ -0.4 & 0.8 \end{bmatrix}.$$

Then

$$H = I - 2P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0.4 & -0.8 \\ -0.8 & 1.6 \end{bmatrix} = \begin{bmatrix} 0.6 & 0.8 \\ 0.8 & -0.6 \end{bmatrix}.$$

Check that H moves x to w and vice versa:

$$Hx = \begin{bmatrix} 0.6 & 0.8 \\ 0.8 & -0.6 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \end{bmatrix} = w$$

and

$$Hw = \begin{bmatrix} 0.6 & 0.8 \\ 0.8 & -0.6 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix} = x. \quad \blacktriangleleft$$

As a first application of Householder reflectors, we will develop a new way to do the QR factorization. In Chapter 12, we apply Householder to the eigenvalue problem, to put matrices into upper Hessenberg form. In both applications, we will use reflectors for a single purpose: to move a column vector x to a coordinate axis as a way of putting zeros into a matrix.

We start with a matrix A that we want to write in the form $A = QR$. Let x_1 be the first column of A . Let $w = \pm(\|x_1\|_2, 0, \dots, 0)$ be a vector along the first coordinate axis of identical Euclidean length. (Either sign works in theory. For numerical stability, the sign is often chosen to be the opposite of the sign of the first component of x to avoid the possibility of subtracting nearly equal numbers when forming v .) Create the Householder reflector H_1 such that $H_1 x = w$. In the 4×3 case, multiplying H_1 by A results in

$$H_1 A = H_1 \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix}.$$

We have introduced some zeros into A . We want to continue in this way until A becomes upper triangular; then we will have R of the QR factorization. Find the Householder reflector \hat{H}_2 that moves the $(m-1)$ -vector x_2 consisting of the lower $m-1$ entries in column 2 of $H_1 A$ to $\pm(\|x_2\|_2, 0, \dots, 0)$. Since \hat{H}_2 is an $(m-1) \times (m-1)$ -matrix, define H_2 to be the $m \times m$ matrix formed by putting \hat{H}_2 into the lower part of the identity matrix. Then

$$\left(\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & & & \\ 0 & \hat{H}_2 & & \\ 0 & & & \end{array} \right) \left(\begin{array}{c|ccc} \times & \times & \times & \\ \hline 0 & \times & \times & \\ 0 & \times & \times & \\ 0 & \times & \times & \end{array} \right) = \left(\begin{array}{c|ccc} \times & \times & \times & \\ \hline 0 & \times & \times & \\ 0 & 0 & \times & \\ 0 & 0 & \times & \end{array} \right)$$

The result $H_2 H_1 A$ is one step from upper triangularity. One more step gives

$$\left(\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & & \\ 0 & 0 & \hat{H}_3 & \end{array} \right) \left(\begin{array}{cc|cc} \times & \times & \times & \\ \hline 0 & \times & \times & \\ 0 & 0 & \times & \\ 0 & 0 & \times & \end{array} \right) = \left(\begin{array}{cc|cc} \times & \times & \times & \\ \hline 0 & \times & \times & \\ 0 & 0 & \times & \\ 0 & 0 & 0 & \end{array} \right)$$

and the result

$$H_3 H_2 H_1 A = R,$$

an upper triangular matrix. Multiplying on the left by the inverses of the Householder reflectors allows us to rewrite the result as

$$A = H_1 H_2 H_3 R = QR,$$

where $Q = H_1 H_2 H_3$. Note that $H_i^{-1} = H_i$ since H_i is symmetric orthogonal. Computer Problem 3 asks the reader to write code for the factorization via Householder reflectors.

► **EXAMPLE 4.18** Use Householder reflectors to find the QR factorization of

$$A = \begin{bmatrix} 3 & 1 \\ 4 & 3 \end{bmatrix}.$$

We need to find a Householder reflector that moves the first column $[3, 4]$ onto the x -axis. We found such a reflector H_1 in Example 4.17, and

$$H_1 A = \begin{bmatrix} 0.6 & 0.8 \\ 0.8 & -0.6 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 0 & -1 \end{bmatrix}.$$

Multiplying both sides on the left by $H_1^{-1} = H_1$ yields

$$A = \begin{bmatrix} 3 & 1 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 0.6 & 0.8 \\ 0.8 & -0.6 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 0 & -1 \end{bmatrix} = QR,$$

where $Q = H_1^T = H_1$.



► EXAMPLE 4.19

Use Householder reflectors to find the QR factorization of $A = \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix}$.

We need to find a Householder reflector that moves the first column $x = [1, 2, 2]$ to the vector $w = [|x|_2, 0, 0]$. Set $v = w - x = [3, 0, 0] - [1, 2, 2] = [2, -2, -2]$. Referring to Theorem 4.4, we have

$$H_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \frac{2}{12} \begin{bmatrix} 4 & -4 & -4 \\ -4 & 4 & 4 \\ -4 & 4 & 4 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & -\frac{2}{3} \\ \frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

and

$$H_1 A = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & -\frac{2}{3} \\ \frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 0 & -3 \\ 0 & -4 \end{bmatrix}.$$

The remaining step is to move the vector $\hat{x} = [-3, -4]$ to $\hat{w} = [5, 0]$. Calculating \hat{H}_2 from Theorem 4.4 yields


$$\begin{bmatrix} -0.6 & -0.8 \\ -0.8 & 0.6 \end{bmatrix} \begin{bmatrix} -3 \\ -4 \end{bmatrix} = \begin{bmatrix} 5 \\ 0 \end{bmatrix},$$

leading to

$$H_2 H_1 A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.6 & -0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix} \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & -\frac{2}{3} \\ \frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 0 & 5 \\ 0 & 0 \end{bmatrix} = R.$$

Multiplying both sides on the left by $H_1^{-1} H_2^{-1} = H_1 H_2$ yields the QR factorization:

$$\begin{aligned} \begin{bmatrix} 1 & -4 \\ 2 & 3 \\ 2 & 2 \end{bmatrix} &= H_1 H_2 R = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{1}{3} & -\frac{2}{3} \\ \frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.6 & -0.8 \\ 0 & -0.8 & 0.6 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 0 & 5 \\ 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1/3 & -14/15 & -2/15 \\ 2/3 & 1/3 & -2/3 \\ 2/3 & 2/15 & 11/15 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 0 & 5 \\ 0 & 0 \end{bmatrix} = QR. \end{aligned}$$

Compare this result with the factorization from Gram–Schmidt orthogonalization in Example 4.13. 

The QR factorization is not unique for a given $m \times n$ matrix A . For example, define $D = \text{diag}(d_1, \dots, d_m)$, where each d_i is either $+1$ or -1 . Then $A = QR = QDDR$, and we check that QD is orthogonal and DR is upper triangular.

Exercise 12 asks for an operation count of QR factorization by Householder reflections, which comes out to $(2/3)m^3$ multiplications and the same number of additions—lower complexity than Gram–Schmidt orthogonalization. Moreover, the Householder method is known to deliver better orthogonality in the unit vectors and has lower memory requirements. For these reasons, it is the method of choice for factoring typical matrices into QR .

10



Trigonometric Interpolation and the FFT

The digital signal processing (DSP) chip is the backbone of advanced consumer electronics. Cellular phones, CD and DVD controllers, automobile electronics, personal digital assistants, digital modems, cameras, and televisions all make use of these ubiquitous devices. The hallmark of the DSP chip is its ability to do rapid digital calculations, including the fast Fourier transform (FFT).

One of the most basic functions of DSP is to separate desired input information from unwanted noise by

filtering. The ability to extract signals from a cluttered background is an important part of the ongoing quest to build reliable speech recognition software. It is also a key element of pattern recognition devices, used by soccer-playing robot dogs to turn sensory inputs into usable data.

Reality Check ✓

Reality Check 10 on page 492 describes the Wiener filter, a fundamental building block of noise reduction via DSP.

Not even the most optimistic trigonometry teacher of a half-century ago could have envisioned the impact sines and cosines have had on modern technology. As we learned in Chapter 4, trig functions of multiple frequencies are natural interpolating functions for periodic data. The Fourier transform is almost unreasonably efficient at carrying out the interpolation and is irreplaceable in the data-intensive applications of modern signal processing.

The efficiency of trigonometric interpolation is bound up with the concept of orthogonality. We will see that orthogonal basis functions make interpolation and least squares fitting

of data much simpler and more accurate. The Fourier transform exploits this orthogonality and provides an efficient means of interpolation with sines and cosines. The computational breakthrough of Cooley and Tukey called the Fast Fourier Transform (FFT) means that the DFT can be computed very cheaply.

This chapter covers the basic ideas of the Discrete Fourier Transform (DFT), including a short introduction to complex numbers. The role of the DFT in trigonometric interpolation and least squares approximation is featured and viewed as a special case of approximation by orthogonal basis functions. This is the essence of digital filtering and signal processing.

10.1 THE FOURIER TRANSFORM

The French mathematician Jean Baptiste Joseph Fourier, after escaping the guillotine during the French Revolution and going to war alongside Napoleon, found time to develop a theory of heat conduction. To make the theory work, he needed to expand functions—not in terms of polynomials, as Taylor series, but in a revolutionary way first developed by Euler and Bernoulli—in terms of sine and cosine functions. Although rejected by the leading mathematicians of the time due to a perceived lack of rigor, today Fourier's methods pervade many areas of applied mathematics, physics, and engineering. In this section, we introduce the Discrete Fourier Transform and describe an efficient algorithm to compute it, the Fast Fourier Transform.

10.1.1 Complex arithmetic

The bookkeeping requirements of trigonometric functions can be greatly simplified by adopting the language of complex numbers. Every complex number has form $z = a + bi$, where $i = \sqrt{-1}$. Each z is represented geometrically as a two-dimensional vector of size a along the real (horizontal) axis, and size b along the imaginary (vertical) axis, as shown in Figure 10.1. The **complex magnitude** of the number $z = a + bi$ is defined to be $|z| = \sqrt{a^2 + b^2}$ and is exactly the distance of the complex number from the origin in the complex plane. The **complex conjugate** of a complex number $z = a + bi$ is $\bar{z} = a - bi$.

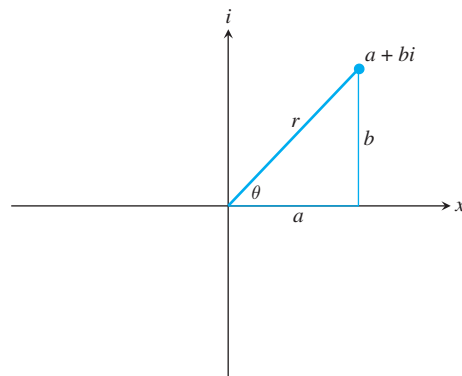


Figure 10.1 Representation of a complex number. The real and imaginary parts are a and bi , respectively. The polar representation is $a + bi = re^{i\theta}$.

The celebrated **Euler formula** for complex arithmetic says $e^{i\theta} = \cos \theta + i \sin \theta$. The complex magnitude of $z = e^{i\theta}$ is 1, so complex numbers of this form lie on the unit circle in the complex plane, as shown in Figure 10.2. Any complex number $a + bi$ can be written in its **polar representation**

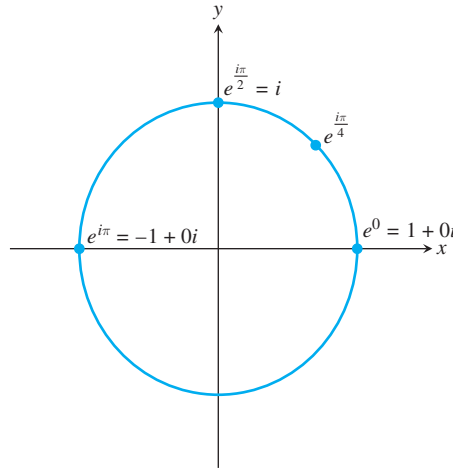


Figure 10.2 Unit circle in the complex plane. Complex numbers of the form $e^{i\theta}$ for some angle θ have magnitude one and lie on the unit circle.

$$z = a + bi = re^{i\theta}, \quad (10.1)$$

where r is the complex magnitude $|z| = \sqrt{a^2 + b^2}$ and $\theta = \arctan b/a$.

The unit circle in the complex plane corresponds to complex numbers of magnitude $r = 1$. To multiply together the two numbers $e^{i\theta}$ and $e^{i\gamma}$ on the unit circle, we could convert to trigonometric functions and then multiply:

$$\begin{aligned} e^{i\theta} e^{i\gamma} &= (\cos \theta + i \sin \theta)(\cos \gamma + i \sin \gamma) \\ &= \cos \theta \cos \gamma - \sin \theta \sin \gamma + i(\sin \theta \cos \gamma + \sin \gamma \cos \theta). \end{aligned}$$

Recognizing the cos addition formula and the sin addition formula, we can rewrite this as

$$\cos(\theta + \gamma) + i \sin(\theta + \gamma) = e^{i(\theta + \gamma)}.$$

Equivalently, just add the exponents:

$$e^{i\theta} e^{i\gamma} = e^{i(\theta + \gamma)}. \quad (10.2)$$

Equation (10.2) shows that the product of two numbers on the unit circle gives a new point on the unit circle whose angle is the sum of the two angles. The Euler formula hides the trigonometry details, like the sine and cosine addition formulas, and makes the bookkeeping much easier. This is the reason we introduce complex arithmetic into the study of trigonometric interpolation. Although it can be done entirely in the real numbers, the Euler formula has a profound simplifying effect.

We single out a special subset of magnitude 1 complex numbers. A complex number z is an n th **root of unity** if $z^n = 1$. On the real number line, there are only two roots of unity, -1 and 1 . In the complex plane, however, there are many. For example, i itself is a 4th root of unity, because $i^4 = (-1)^2 = 1$.

An n th root of unity is called **primitive** if it is not a k th root of unity for any $k < n$. By this definition, -1 is a primitive second root of unity and a nonprimitive fourth root of unity. It is easy to check that for any integer n , the complex number $\omega_n = e^{-i2\pi/n}$ is a primitive n th root of unity. The number $e^{i2\pi/n}$ is also a primitive n th root of unity, but we will follow the usual convention of using the former for the basis of the Fourier transform. Figure 10.3 shows a primitive eighth root of unity $\omega_8 = e^{-i2\pi/8}$ and the other seven roots of unity, which are powers of ω_8 .

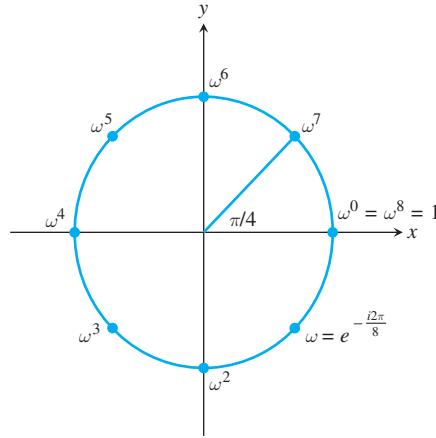


Figure 10.3 Roots of unity. The eight 8th roots of unity are shown. They are generated by $\omega = e^{-2\pi/8}$, meaning that each is ω^k for some integer k . Although ω and ω^3 are primitive 8th roots of unity, ω^2 is not, because it is also a 4th root of unity.

Here is a key identity that we will need later to simplify our computations of the Discrete Fourier Transform. Let ω denote the n th root of unity $\omega = e^{-i2\pi/n}$ where $n > 1$. Then

$$1 + \omega + \omega^2 + \omega^3 + \cdots + \omega^{n-1} = 0. \quad (10.3)$$

The proof of this identity follows from the telescoping sum

$$(1 - \omega)(1 + \omega + \omega^2 + \omega^3 + \cdots + \omega^{n-1}) = 1 - \omega^n = 0. \quad (10.4)$$

Since the first term on the left is not zero, the second must be. A similar method of proof shows that

$$\begin{aligned} 1 + \omega^2 + \omega^4 + \omega^6 + \cdots + \omega^{2(n-1)} &= 0, \\ 1 + \omega^3 + \omega^6 + \omega^9 + \cdots + \omega^{3(n-1)} &= 0, \\ &\vdots \\ 1 + \omega^{n-1} + \omega^{(n-1)2} + \omega^{(n-1)3} + \cdots + \omega^{(n-1)(n-1)} &= 0. \end{aligned} \quad (10.5)$$

The next one is different:

$$\begin{aligned} 1 + \omega^n + \omega^{2n} + \omega^{3n} + \cdots + \omega^{n(n-1)} &= 1 + 1 + 1 + 1 + \cdots + 1 \\ &= n. \end{aligned} \quad (10.6)$$

This information is collected into the following lemma.

LEMMA 10.1 Primitive roots of unity. Let ω be a primitive n th root of unity and k be an integer. Then

$$\sum_{j=0}^{n-1} \omega^{jk} = \begin{cases} n & \text{if } k/n \text{ is an integer} \\ 0 & \text{otherwise} \end{cases}.$$

■

Exercise 6 asks the reader to fill in the details of the proof.

10.1.2 Discrete Fourier Transform

Let $x = [x_0, \dots, x_{n-1}]^T$ be a (real-valued) n -dimensional vector, and denote $\omega = e^{-i2\pi/n}$. Here is the fundamental definition of this chapter.

DEFINITION 10.2 The **Discrete Fourier Transform** (DFT) of $x = [x_0, \dots, x_{n-1}]^T$ is the n -dimensional vector $y = [y_0, \dots, y_{n-1}]$, where $\omega = e^{-i2\pi/n}$ and

$$y_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \omega^{jk}. \quad (10.7)$$

□

For example, Lemma 10.1 shows that the DFT of $x = [1, 1, \dots, 1]$ is $y = [\sqrt{n}, 0, \dots, 0]$. In matrix terms, this definition says

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_0 + ib_0 \\ a_1 + ib_1 \\ a_2 + ib_2 \\ \vdots \\ a_{n-1} + ib_{n-1} \end{bmatrix} = \frac{1}{\sqrt{n}} \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \omega^0 & \omega^3 & \omega^6 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix}. \quad (10.8)$$

Each $y_k = a_k + ib_k$ is a complex number. The $n \times n$ matrix in (10.8) is called the **Fourier matrix**

$$F_n = \frac{1}{\sqrt{n}} \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \omega^0 & \omega^3 & \omega^6 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix}. \quad (10.9)$$

Except for the top row, each row of the Fourier matrix adds to zero, and the same is true for the columns, since F_n is a symmetric matrix. The Fourier matrix has an explicit inverse

$$F_n^{-1} = \frac{1}{\sqrt{n}} \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ \omega^0 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \omega^0 & \omega^{-3} & \omega^{-6} & \cdots & \omega^{-3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2} \end{bmatrix}, \quad (10.10)$$

and the **inverse Discrete Fourier Transform** of the vector y is $x = F_n^{-1}y$. Checking that (10.10) is the inverse of the matrix F_n requires Lemma 11.1 about n th roots of unity. See Exercise 8.

Let $z = e^{i\theta} = \cos \theta + i \sin \theta$ be a point on the unit circle. Then its reciprocal $e^{-i\theta} = \cos \theta - i \sin \theta$ is its complex conjugate. Therefore, the inverse DFT is the matrix of complex conjugates of the entries of F_n :

$$F_n^{-1} = \overline{F_n}. \quad (10.11)$$

DEFINITION 10.3 The **magnitude** of a complex vector v is the real number $\|v\| = \sqrt{v^T v}$. A square complex matrix F is **unitary** if $\overline{F}^T F = I$. □

A unitary matrix, like the Fourier matrix, is the complex version of a real orthogonal matrix. If F is unitary, then $\|Fv\|^2 = \bar{v}^T \overline{F}^T F v = \bar{v}^T v = \|v\|^2$. Thus, the magnitude of a vector is unchanged upon multiplication on the left by F —or F^{-1} for that matter.

Applying the Discrete Fourier Transform is a matter of multiplying by the $n \times n$ matrix F_n , and therefore requires $O(n^2)$ operations (specifically n^2 multiplications and $n(n-1)$ additions). The inverse Discrete Fourier Transform, which is applied by multiplication by F_n^{-1} , is also an $O(n^2)$ process. In Section 10.1.3, we develop a version of the DFT that requires significantly fewer operations, called the Fast Fourier transform.

► **EXAMPLE 10.1** Find the DFT of the vector $x = [1, 0, -1, 0]^T$.

Let ω be the 4th root of unity, or $\omega = e^{-i\pi/2} = \cos(\pi/2) - i \sin(\pi/2) = -i$. Applying the DFT, we get

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ -1 \\ 0 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}. \quad (10.12)$$

The MATLAB command `fft` carries out the DFT with a slightly different normalization, so that $F_n x$ is computed by `fft(x)/sqrt(n)`. The inverse command `ifft` is the inverse of `fft`. Therefore, $F_n^{-1} y$ is computed by the MATLAB command `ifft(y)*sqrt(n)`. In other words, MATLAB's `fft` and `ifft` commands are inverses of one another, although their normalization differs from the definition given here, which has the advantage that F_n and F_n^{-1} are unitary matrices.

Even if the vector x has components that are real numbers, there is no reason for the components of y to be real numbers. But if the x_j are real, the complex numbers y_k have a special property:

LEMMA 10.4 Let $\{y_k\}$ be the DFT of $\{x_j\}$, where the x_j are real numbers. Then (a) y_0 is real, and (b) $y_{n-k} = \overline{y_k}$ for $k = 1, \dots, n-1$. ■

Proof. The reason for (a) is clear from (10.7), since y_0 is the sum of the x_j 's divided by \sqrt{n} . Part (b) follows from the fact that

$$\omega^{n-k} = e^{-i2\pi(n-k)/n} = e^{-i2\pi} e^{i2\pi k/n} = \cos(2\pi k/n) + i \sin(2\pi k/n)$$

while

$$\omega^k = e^{-i2\pi k/n} = \cos(2\pi k/n) - i \sin(2\pi k/n),$$

implying that $\omega^{n-k} = \overline{\omega^k}$. From the definition of Fourier transform,

$$\begin{aligned} y_{n-k} &= \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j (\omega^{n-k})^j \\ &= \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j (\overline{\omega^k})^j \\ &= \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} \overline{x_j (\omega^k)^j} = \overline{y_k}. \end{aligned}$$

Here we have used the fact that the product of complex conjugates is the conjugate of the product. \square

Lemma 10.4 has an interesting consequence. Let n be even and the x_0, \dots, x_{n-1} be real numbers. Then the DFT replaces them with exactly n other real numbers $a_0, a_1, b_1, a_2, b_2, \dots, a_{n/2}$, the real and imaginary parts of the Fourier transform y_0, \dots, y_{n-1} . For example, the $n = 8$ DFT has the form

$$F_8 \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 + ib_1 \\ a_2 + ib_2 \\ a_3 + ib_3 \\ a_4 \\ a_3 - ib_3 \\ a_2 - ib_2 \\ a_1 - ib_1 \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_{\frac{n}{2}-1} \\ \overline{y_{\frac{n}{2}-1}} \\ \vdots \\ \overline{y_1} \end{bmatrix}. \quad (10.13)$$

10.1.3 The Fast Fourier Transform

As mentioned in the last section, the Discrete Fourier Transform applied to an n -vector in the traditional way requires $O(n^2)$ operations. Cooley and Tukey [1965] found a way to accomplish the DFT in $O(n \log n)$ operations in an algorithm called the **Fast Fourier Transform** (FFT). The popularity of the FFT for data analysis followed almost immediately. The field of signal processing converted from primarily analog to digital largely due to this algorithm. We will explain their method and show its superiority to the naive DFT (10.8) through an operation count.

We can write the DFT $F_n x$ as

$$\begin{bmatrix} y_0 \\ \vdots \\ y_{n-1} \end{bmatrix} = \frac{1}{\sqrt{n}} M_n \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix},$$

where

$$M_n = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \dots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \dots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \omega^0 & \omega^3 & \omega^6 & \dots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)^2} \end{bmatrix}.$$

SPOTLIGHT ON

Complexity

The achievement of Cooley and Tukey to reduce the complexity of the DFT from $O(n^2)$ operations to $O(n \log n)$ operations opened up a world of possibilities for Fourier transform methods. A method that scales “almost linearly” with the size of the problem is very valuable. For example, there is a possibility of using it for real-time data, since analysis can occur approximately at the same timescale that data are acquired. The development of the FFT was followed a short time later with specialized circuitry for implementing it, now represented by DSP chips for digital signal processing that are ubiquitous in electronic systems for analysis and control.

We will show how to compute $z = M_n x$ recursively. To complete the DFT requires dividing by \sqrt{n} , or $y = F_n x = z/\sqrt{n}$.

We start by showing how the $n = 4$ case works, to get the main idea across. The general case will then be clear. Let $\omega = e^{-i2\pi/4} = -i$. The Discrete Fourier Transform is

$$\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}. \quad (10.14)$$

Write out the matrix product, but rearrange the order of the terms so that the even-numbered terms come first:

$$\begin{aligned} z_0 &= \omega^0 x_0 + \omega^0 x_2 + \omega^0 (\omega^0 x_1 + \omega^0 x_3) \\ z_1 &= \omega^0 x_0 + \omega^2 x_2 + \omega^1 (\omega^0 x_1 + \omega^2 x_3) \\ z_2 &= \omega^0 x_0 + \omega^4 x_2 + \omega^2 (\omega^0 x_1 + \omega^4 x_3) \\ z_3 &= \omega^0 x_0 + \omega^6 x_2 + \omega^3 (\omega^0 x_1 + \omega^6 x_3) \end{aligned}$$

Using the fact that $\omega^4 = 1$, we can rewrite these equations as

$$\begin{aligned} z_0 &= (\omega^0 x_0 + \omega^0 x_2) + \omega^0 (\omega^0 x_1 + \omega^0 x_3) \\ z_1 &= (\omega^0 x_0 + \omega^2 x_2) + \omega^1 (\omega^0 x_1 + \omega^2 x_3) \\ z_2 &= (\omega^0 x_0 + \omega^0 x_2) + \omega^2 (\omega^0 x_1 + \omega^0 x_3) \\ z_3 &= (\omega^0 x_0 + \omega^2 x_2) + \omega^3 (\omega^0 x_1 + \omega^2 x_3) \end{aligned}$$

Notice that each term in parentheses in the top two lines is repeated verbatim in the bottom two lines. Define

$$\begin{aligned} u_0 &= \mu^0 x_0 + \mu^0 x_2 \\ u_1 &= \mu^0 x_0 + \mu^1 x_2 \end{aligned}$$

and

$$\begin{aligned} v_0 &= \mu^0 x_1 + \mu^0 x_3 \\ v_1 &= \mu^0 x_1 + \mu^1 x_3, \end{aligned}$$

where $\mu = \omega^2$ is a 2nd root of unity. Both $u = (u_0, u_1)^T$ and $v = (v_0, v_1)^T$ are essentially DFTs with $n = 2$; more precisely,

$$\begin{aligned} u &= M_2 \begin{bmatrix} x_0 \\ x_2 \end{bmatrix} \\ v &= M_2 \begin{bmatrix} x_1 \\ x_3 \end{bmatrix}. \end{aligned}$$

We can write the original $M_4 x$ as

$$\begin{aligned} z_0 &= u_0 + \omega^0 v_0 \\ z_1 &= u_1 + \omega^1 v_1 \\ z_2 &= u_0 + \omega^2 v_0 \\ z_3 &= u_1 + \omega^3 v_1. \end{aligned}$$

In summary, the calculation of the DFT(4) has been reduced to a pair of DFT(2)s plus some extra multiplications and additions.

Ignoring the $1/\sqrt{n}$ for a moment, DFT(n) can be reduced to computing two DFT($n/2$)s plus $2n - 1$ extra operations ($n - 1$ multiplications and n additions). A careful count of the additions and multiplications necessary yields Theorem 10.5.

THEOREM 10.5 Operation Count for FFT. Let n be a power of 2. Then the Fast Fourier Transform of size n can be completed in $n(2\log_2 n - 1) + 1$ additions and multiplications, plus a division by \sqrt{n} . ■

Proof. Ignore the square root, which is applied at the end. The result is equivalent to saying that the DFT(2^m) can be completed in $2^m(2m - 1) + 1$ additions and multiplications. In fact, we saw above how a DFT(n), where n is even, can be reduced to a pair of DFT($n/2$)s. If n is a power of two—say, $n = 2^m$ —then we can recursively break down the problem until we get to DFT(1), which is multiplication by the 1×1 identity matrix, taking zero operations. Starting from the bottom up, DFT(1) takes no operations, and DFT(2) requires two additions and a multiplication: $y_0 = u_0 + 1v_0$, $y_1 = u_0 + \omega v_0$, where u_0 and v_0 are DFT(1)s (that is, $u_0 = y_0$ and $v_0 = y_1$).

DFT(4) requires two DFT(2)s plus $2 \cdot 4 - 1 = 7$ further operations, for a total of $2(3) + 7 = 2^m(2m - 1) + 1$ operations, where $m = 2$. We proceed by induction: Assume that this formula is correct for a given m . Then DFT(2^{m+1}) takes two DFT(2^m)s, which take $2(2^m(2m - 1) + 1)$ operations, plus $2 \cdot 2^{m+1} - 1$ extras (to complete equations similar to (10.15)), for a total of

$$\begin{aligned} 2(2^m(2m - 1) + 1) + 2^{m+2} - 1 &= 2^{m+1}(2m - 1 + 2) + 2 - 1 \\ &= 2^{m+1}(2(m + 1) - 1) + 1. \end{aligned}$$

Therefore, the formula $2^m(2m - 1) + 1$ operations is proved for the fast version of DFT(2^m), from which the result follows. □

The fast algorithm for the DFT can be exploited to make a fast algorithm for the inverse DFT without further work. The inverse DFT is the complex conjugate matrix \overline{F}_n . To carry out the inverse DFT of a complex vector y , just conjugate, apply the FFT, and conjugate the result, because

$$F_n^{-1}y = \overline{F}_n y = \overline{F_n \overline{y}}. \quad (10.15)$$

10.1 Exercises

- Find the DFT of the following vectors: (a) $[0, 1, 0, -1]$ (b) $[1, 1, 1, 1]$ (c) $[0, -1, 0, 1]$ (d) $[0, 1, 0, -1, 0, 1, 0, -1]$
- Find the DFT of the following vectors: (a) $[3/4, 1/4, -1/4, 1/4]$ (b) $[9/4, 1/4, -3/4, 1/4]$ (c) $[1, 0, -1/2, 0]$ (d) $[1, 0, -1/2, 0, 1, 0, -1/2, 0]$
- Find the inverse DFT of the following vectors: (a) $[1, 0, 0, 0]$ (b) $[1, 1, -1, 1]$ (c) $[1, -i, 1, i]$ (d) $[1, 0, 0, 0, 3, 0, 0, 0]$
- Find the inverse DFT of the following vectors: (a) $[0, -i, 0, i]$ (b) $[2, 0, 0, 0]$ (c) $[1/2, 1/2, 0, 1/2]$ (d) $[1, 3/2, 1/2, 3/2]$
- (a) Write down all fourth roots of unity and all primitive fourth roots of unity. (b) Write down all primitive seventh roots of unity. (c) How many primitive p th roots of unity exist for a prime number p ?

6. Prove Lemma 10.1.
7. Find the real numbers $a_0, a_1, b_1, a_2, b_2, \dots, a_{n/2}$ as in (10.13) for the Fourier transforms in Exercise 1.
8. Prove that the matrix in (10.10) is the inverse of the Fourier matrix F_n .

10.2 TRIGONOMETRIC INTERPOLATION

What does the Discrete Fourier transform actually do? In this section, we present an interpretation of the output vector y of the Fourier transform as interpolating coefficients for evenly spaced data in order to make its workings more understandable.

10.2.1 The DFT Interpolation Theorem

Let $[c, d]$ be an interval and let n be a positive integer. Define $\Delta t = (d - c)/n$ and $t_j = c + j\Delta t$ for $j = 0, \dots, n - 1$ to be evenly spaced points in the interval. For a given input vector x to the Fourier transform, we will interpret the component x_j as the j th component of a measured signal. For example, we could think of the components of x as a series of measurements, measured at the discrete, evenly spaced times t_j , as shown in Figure 10.4.

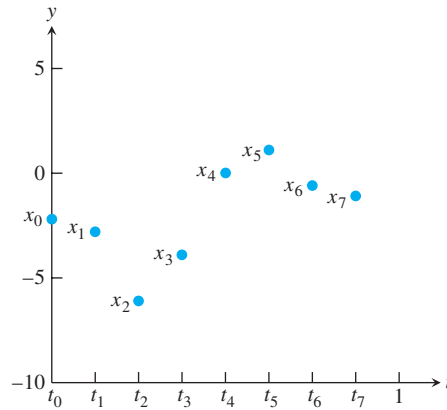


Figure 10.4 The components of x viewed as a time series. The Fourier transform is a way to compute the trigonometric polynomial that interpolates this data.

Let $y = F_n x$ be the DFT of x . Since x is the inverse DFT of y , we can write an explicit formula for the components of x from (10.10), remembering that $\omega = e^{-i2\pi/n}$:

$$x_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} y_k (\omega^{-k})^j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} y_k e^{i2\pi kj/n} = \sum_{k=0}^{n-1} y_k \frac{e^{\frac{i2\pi k(t_j - c)}{d - c}}}{\sqrt{n}}. \quad (10.16)$$

We can view this as interpolation of the points (t_j, x_j) by trigonometric basis functions where the coefficients are y_k . Theorem 10.6 is a simple restatement of (10.16), saying that data points (t_j, x_j) are interpolated by basis functions $e^{i2\pi k(t - c)/(d - c)}/\sqrt{n}$ for $k = 0, \dots, n - 1$, with interpolation coefficients given by $F_n x$.

THEOREM 10.6 DFT Interpolation Theorem. Given an interval $[c, d]$ and positive integer n , let $t_j = c + j(d - c)/n$ for $j = 0, \dots, n - 1$, and let $x = (x_0, \dots, x_{n-1})$ denote a vector of n numbers.

Define $\vec{a} + \vec{b}i = F_n x$, where F_n is the Discrete Fourier Transform matrix. Then the complex function

$$Q(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} (a_k + ib_k) e^{i2\pi k(t-c)/(d-c)}$$

satisfies $Q(t_j) = x_j$ for $j = 0, \dots, n-1$. Furthermore, if the x_j are real, the real function

$$P(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \left(a_k \cos \frac{2\pi k(t-c)}{d-c} - b_k \sin \frac{2\pi k(t-c)}{d-c} \right)$$

satisfies $P(t_j) = x_j$ for $j = 0, \dots, n-1$. ■

In other words, the Fourier transform F_n transforms data $\{x_j\}$ into interpolation coefficients.

The explanation for the last part of the theorem is that, using the Euler formula, we can rewrite the interpolation function in (10.16) as

$$Q(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} (a_k + ib_k) \left(\cos \frac{2\pi k(t-c)}{d-c} + i \sin \frac{2\pi k(t-c)}{d-c} \right).$$

Separate the interpolating function $Q(t) = P(t) + iI(t)$ into its real and imaginary parts. Since the x_j are real numbers, only the real part of $Q(t)$ is needed to interpolate the x_j . The real part is

$$P(t) = P_n(t) = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \left(a_k \cos \frac{2\pi k(t-c)}{d-c} - b_k \sin \frac{2\pi k(t-c)}{d-c} \right). \quad (10.17)$$

A subscript n identifies the number of terms in the trigonometric model. We will sometimes call P_n an **order n trigonometric function**. Lemma 10.4 and the following Lemma 10.7 can be used to simplify the interpolating function $P_n(t)$ further:

LEMMA 10.7 Let $t = j/n$, where j and n are integers. Let k be an integer. Then

$$\cos 2(n-k)\pi t = \cos 2k\pi t \text{ and } \sin 2(n-k)\pi t = -\sin 2k\pi t. \quad (10.18) \quad \text{■}$$

In fact, the cosine addition formula yields $\cos 2(n-k)\pi j/n = \cos(2\pi j - 2jk\pi/n) = \cos(-2jk\pi/n)$ and similarly for sine.

Lemma 10.7, together with Lemma 10.4, implies that the latter half of the trigonometric expansion (10.17) is redundant. We can interpolate at the t_j 's by using only the first half of the terms (except for a change of sign for the sine terms). By Lemma 10.4, the coefficients from the latter half of the expansion are the same as those from the first half (except for a change of sign for the sin terms). Thus, the changes of sign cancel one another out, and we have shown that the simplified version of P_n is

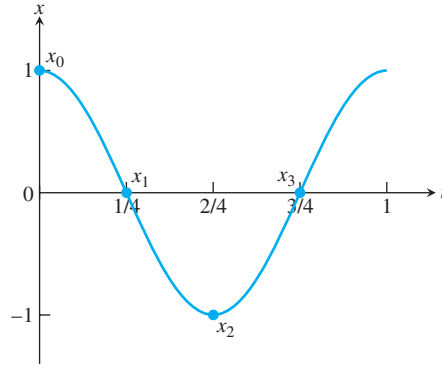


Figure 10.5 Trigonometric interpolation. The input vector x is $[1, 0, -1, 0]^T$. Formula (10.19) gives the interpolating function to be $P_4(t) = \cos 2\pi t$.

$$P_n(t) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} \left(a_k \cos \frac{2k\pi(t-c)}{d-c} - b_k \sin \frac{2k\pi(t-c)}{d-c} \right) + \frac{a_{n/2}}{\sqrt{n}} \cos \frac{n\pi(t-c)}{d-c}.$$

To write this expression, we have assumed that n is even. The formula is slightly different for n odd. See Exercise 5.

COROLLARY 10.8 For an even integer n , let $t_j = c + j(d-c)/n$ for $j = 0, \dots, n-1$, and let $x = (x_0, \dots, x_{n-1})$ denote a vector of n real numbers. Define $\vec{a} + \vec{b}i = F_n x$, where F_n is the Discrete Fourier Transform. Then the function

$$P_n(t) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} \left(a_k \cos \frac{2k\pi(t-c)}{d-c} - b_k \sin \frac{2k\pi(t-c)}{d-c} \right) + \frac{a_{n/2}}{\sqrt{n}} \cos \frac{n\pi(t-c)}{d-c} \quad (10.19)$$

satisfies $P_n(t_j) = x_j$ for $j = 0, \dots, n-1$. ■

► **EXAMPLE 10.2** Find the trigonometric interpolant for Example 10.1.

The interval is $[c, d] = [0, 1]$. Let $x = [1, 0, -1, 0]^T$ and compute its DFT to be $y = [0, 1, 0, 1]^T$. The interpolating coefficients are $a_k + ib_k = y_k$. Therefore, $a_0 = a_2 = 0$, $a_1 = a_3 = 1$, and $b_0 = b_1 = b_2 = b_3 = 0$. According to (10.19), we only need a_0 , a_1 , a_2 , and b_1 . A trigonometric interpolating function for x is given by

$$P_4(t) = \frac{a_0}{2} + (a_1 \cos 2\pi t - b_1 \sin 2\pi t) + \frac{a_2}{2} \cos 4\pi t = \cos 2\pi t.$$

The interpolation of the points (t, x) , where $t = [0, 1/4, 1/2, 3/4]$ and $x = [1, 0, -1, 0]$, is shown in Figure 10.5. ◀

► **EXAMPLE 10.3** Find the trigonometric interpolant for the temperature data from Example 4.6: $x = [-2.2, -2.8, -6.1, -3.9, 0.0, 1.1, -0.6, -1.1]$ on the interval $[0, 1]$.

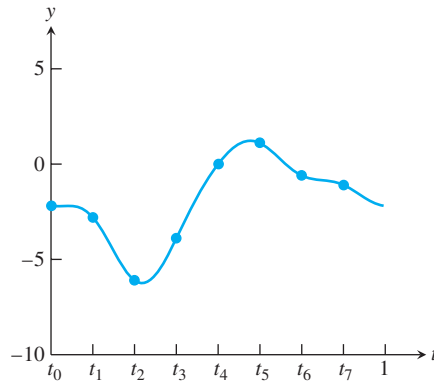


Figure 10.6 Trigonometric interpolation of data from Example 4.6. The data $t = [0, 1/8, 2/8, 3/8, 4/8, 5/8, 6/8, 7/8]$, $x = [-2.2, -2.8, -6.1, -3.9, 0.0, 1.1, -0.6, -1.1]$ are interpolated with the use of the Fourier transform with $n = 8$. The plot is made by Program 10.1 with $p = 100$.

The Fourier transform output, accurate to four decimal places, is

$$y = \begin{bmatrix} -5.5154 \\ -1.0528 + 3.6195i \\ 1.5910 - 1.1667i \\ -0.5028 - 0.2695i \\ -0.7778 \\ -0.5028 + 0.2695i \\ 1.5910 + 1.1667i \\ -1.0528 - 3.6195i \end{bmatrix}.$$

According to formula (10.19), the interpolating function is

$$\begin{aligned} P_8(t) &= \frac{-5.5154}{\sqrt{8}} - \frac{1.0528}{\sqrt{2}} \cos 2\pi t - \frac{3.6195}{\sqrt{2}} \sin 2\pi t \\ &\quad + \frac{1.5910}{\sqrt{2}} \cos 4\pi t + \frac{1.1667}{\sqrt{2}} \sin 4\pi t \\ &\quad - \frac{0.5028}{\sqrt{2}} \cos 6\pi t + \frac{0.2695}{\sqrt{2}} \sin 6\pi t \\ &\quad - \frac{0.7778}{\sqrt{8}} \cos 8\pi t \\ &= -1.95 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t \\ &\quad + 1.125 \cos 4\pi t + 0.825 \sin 4\pi t \\ &\quad - 0.3555 \cos 6\pi t + 0.1906 \sin 6\pi t \\ &\quad - 0.2750 \cos 8\pi t. \end{aligned} \tag{10.20}$$

Figure 10.6 shows the data points and the trigonometric interpolating function. ◀

10.2.2 Efficient evaluation of trigonometric functions

Corollary 10.8 is a powerful statement about interpolation. Although it appears complicated at first, there is another way to evaluate and plot the trigonometric interpolating polynomial

in Figures 10.5 and 10.6, using the DFT to do all the work instead of plotting the sines and cosines of (10.19). After all, we know from Theorem 10.6 that multiplying the vector x of data points by F_n changes data to interpolation coefficients. Conversely, we can turn interpolation coefficients into data points. Instead of evaluating (10.19), just invert the DFT: Multiply the vector of interpolation coefficients $\{a_k + ib_k\}$ by F_n^{-1} .

Of course, if we follow the operation F_n by its inverse, F_n^{-1} , we just get the original data points back and gain nothing. Instead, we will let $p \geq n$ be a larger number. We plan to view (10.19) as an order p trigonometric function and then invert the Fourier transform to evaluate the curve at the p equally spaced points. We can take p large enough to get a continuous-looking plot.

To view the coefficients of $P_n(t)$ as the coefficients of an order p trigonometric polynomial, notice that we can rewrite (10.19) as

$$P_p(t) = \frac{\sqrt{\frac{p}{n}}a_0}{\sqrt{p}} + \frac{2}{\sqrt{p}} \sum_{k=1}^{p/2-1} \left(\sqrt{\frac{p}{n}}a_k \cos \frac{2k\pi(t-c)}{d-c} - \sqrt{\frac{p}{n}}b_k \sin \frac{2k\pi(t-c)}{d-c} \right) + \frac{\sqrt{\frac{p}{n}}a_{n/2}}{\sqrt{p}} \cos n\pi t \quad (10.21)$$

where we set $a_k = b_k = 0$ for $k = \frac{n}{2} + 1, \dots, \frac{p}{2}$. We conclude from (10.21) that the way to produce p points lying on the curve (10.19) at $t_j = c + j(d-c)/n$ for $j = 0, \dots, n-1$ is to multiply the Fourier coefficients by $\sqrt{p/n}$ and then invert the DFT.

We write MATLAB code to implement this idea. Roughly speaking, we want to implement

$$F_p^{-1} \sqrt{\frac{p}{n}} F_n x$$

using MATLAB's commands `fft` and `ifft`, where

$$F_p^{-1} = \sqrt{p} \cdot \text{ifft} \quad \text{and} \quad F_n = \frac{1}{\sqrt{n}} \cdot \text{fft}.$$

Putting the pieces together, this corresponds to the following operations:

$$\sqrt{p} \cdot \text{ifft}_{[p]} \sqrt{\frac{p}{n}} \frac{1}{\sqrt{n}} \cdot \text{fft}_{[n]} = \frac{p}{n} \cdot \text{ifft}_{[p]} \cdot \text{fft}_{[n]}. \quad (10.22)$$

Of course, F_p^{-1} can only be applied to a length p vector, so we need to place the degree n Fourier coefficients into a length p vector before inverting. The short program `dftinterp.m` carries out these steps.

```
%Program 10.1 Fourier interpolation
%Interpolate n data points on [c,d] with trig function P(t)
% and plot interpolant at p (>=n) evenly spaced points.
%Input: interval [c,d], data points x, even number of data
% points n, even number p>=n
%Output: data points of interpolant xp
function xp=dftinterp(inter,x,n,p)
c=inter(1);d=inter(2);t=c+(d-c)*(0:n-1)/n; tp=c+(d-c)*(0:p-1)/p;
y=fft(x); % apply DFT
yp=zeros(p,1); % yp will hold coefficients for ifft
yp(1:n/2+1)=y(1:n/2+1); % move n frequencies from n to p
yp(p-n/2+2:p)=y(n/2+2:n); % same for upper tier
xp=real(ifft(yp))*(p/n); % invert fft to recover data
plot(t,x,'o',tp,xp) % plot data points and interpolant
```

Running the function `dftinterp([0, 1], [-2.2 -2.8 -6.1 -3.9 0.0 1.1 -0.6 -1.1], 8, 100)`, for example, produces the $p = 100$ plotted points in Figure 10.6 without explicitly using sines or cosines. A few comments on the code are in order. The goal is to apply `fft[n]`, followed by `ifft[p]`, and then multiply by p/n . After applying `fft` to the n values in x , the coefficients in the vector y are moved from the n frequencies in $P_n(t)$ to a vector yp holding p frequencies, where $p \geq n$. There are many higher frequencies among the p frequencies that are not used by P_n , which leads to zero coefficients in those high frequencies, in positions $n/2 + 2$ to $p/2 + 1$. The upper half of the entries in yp gives a recapitulation of the lower half, with complex conjugates and in reverse order, following (10.13). After the DFT is inverted with the `ifft` command, although theoretically the result is real, computationally there may be a small imaginary part due to rounding. This is removed by applying the `real` command.

A particularly simple and useful case is $c = 0, d = n$. The data points x_j are collected at the integer interpolation nodes $s_j = j$ for $j = 0, \dots, n - 1$. The points (j, x_j) are interpolated by the trigonometric function

$$P_n(s) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} \left(a_k \cos \frac{2k\pi}{n} s - b_k \sin \frac{2k\pi}{n} s \right) + \frac{a_{n/2}}{\sqrt{n}} \cos \pi s. \quad (10.23)$$

In Chapter 11, we will use integer interpolation nodes exclusively, for compatibility with the usual conventions for audio and image data compression algorithms.

10.2 Exercises

1. Use the DFT and Corollary 10.8 to find the trigonometric interpolating function for the following data:

t		x	t		x	t		x	t		x
(a)	0	0	(b)	0	1	(c)	0	-1	(d)	0	1
	$\frac{1}{4}$	1		$\frac{1}{4}$	1		$\frac{1}{4}$	1		$\frac{1}{4}$	1
	$\frac{1}{2}$	0		$\frac{1}{2}$	-1		$\frac{1}{2}$	-1		$\frac{1}{2}$	1
	$\frac{3}{4}$	-1		$\frac{3}{4}$	-1		$\frac{3}{4}$	1		$\frac{3}{4}$	1

2. Use (10.23) to find the trigonometric interpolating function for the following data:

t		x	t		x	t		x	t		x
(a)	0	0	(b)	0	1	(c)	0	1	(d)	0	1
	1	1		1	1		1	2		1	0
	2	0		2	-1		2	4		2	1
	3	-1		3	-1		3	1		3	0

3. Find the trigonometric interpolating function for the following data:

t		x	t		x	t		x	t		x
(a)	0	0	(b)	0	1	(c)	0	1	(d)	0	1
	$\frac{1}{8}$	1		$\frac{1}{8}$	2		$\frac{1}{8}$	1		$\frac{1}{8}$	-1
	$\frac{1}{4}$	0		$\frac{1}{4}$	1		$\frac{1}{4}$	1		$\frac{1}{4}$	1
	$\frac{3}{8}$	-1		$\frac{3}{8}$	0		$\frac{3}{8}$	1		$\frac{3}{8}$	-1
	$\frac{1}{2}$	0		$\frac{1}{2}$	1		$\frac{1}{2}$	0		$\frac{1}{2}$	1
	$\frac{5}{8}$	1		$\frac{5}{8}$	2		$\frac{5}{8}$	0		$\frac{5}{8}$	-1
	$\frac{3}{4}$	0		$\frac{3}{4}$	1		$\frac{3}{4}$	0		$\frac{3}{4}$	1
	$\frac{7}{8}$	-1		$\frac{7}{8}$	0		$\frac{7}{8}$	0		$\frac{7}{8}$	-1

4. Find the trigonometric interpolating function for the following data:

t	x	t	x	t	x	t	x
0	0	0	1	0	1	0	-1
1	1	1	2	1	0	1	0
2	0	2	1	2	1	2	0
(a) 3	-1	(b) 3	0	(c) 3	0	(d) 3	0
4	0	4	1	4	1	4	1
5	1	5	2	5	0	5	0
6	0	6	1	6	1	6	0
7	-1	7	0	7	0	7	0

5. Find a version of (10.19) for the interpolating function in the case where n is odd.

10.2 Computer Problems

1. Find the order 8 trigonometric interpolating function $P_8(t)$ for the following data:

t	x	t	x	t	x	t	x
0	0	0	2	0	3	1	1
$\frac{1}{8}$	1	$\frac{1}{8}$	-1	1	1	2	-2
$\frac{1}{4}$	2	$\frac{1}{4}$	0	2	4	3	5
(a) $\frac{3}{8}$	3	(b) $\frac{3}{8}$	1	(c) 3	2	(d) 4	3
$\frac{1}{2}$	4	$\frac{1}{2}$	1	4	3	5	-2
$\frac{5}{8}$	5	$\frac{5}{8}$	3	5	1	6	-3
$\frac{3}{4}$	6	$\frac{3}{4}$	-1	6	4	7	1
$\frac{7}{8}$	7	$\frac{7}{8}$	-1	7	2	8	2

Plot the data points and $P_8(t)$.

2. Find the order 8 trigonometric interpolating function $P_8(t)$ for the following data:

t	x	t	x	t	x	t	x
0	6	0	3	0	1	-7	2
$\frac{1}{8}$	5	$\frac{1}{8}$	1	2	2	-5	1
$\frac{1}{4}$	4	$\frac{1}{4}$	2	4	4	-3	0
(a) $\frac{3}{8}$	3	(b) $\frac{3}{8}$	-1	(c) 6	-1	(d) -1	5
$\frac{1}{2}$	2	$\frac{1}{2}$	-1	8	0	1	7
$\frac{5}{8}$	1	$\frac{5}{8}$	-2	10	1	3	2
$\frac{3}{4}$	0	$\frac{3}{4}$	3	12	0	5	1
$\frac{7}{8}$	-1	$\frac{7}{8}$	0	14	2	7	-4

Plot the data points and $P_8(t)$.

3. Find the order $n = 8$ trigonometric interpolating function for $f(t) = e^t$ at the evenly spaced points $(j/8, f(j/8))$ for $j = 0, \dots, 7$. Plot $f(t)$, the data points, and the interpolating function.
4. Plot the interpolating function $P_n(t)$ on $[0, 1]$ in Computer Problem 3, along with the data points and $f(t) = e^t$ for (a) $n = 16$ (b) $n = 32$.

5. Find the order 8 trigonometric interpolating function for $f(t) = \ln t$ at the evenly spaced points $(1 + j/8, f(1 + j/8))$ for $j = 0, \dots, 7$. Plot $f(t)$, the data points, and the interpolating function.
6. Plot the interpolating function $P_n(t)$ on $[0, 1]$ in Computer Problem 5, along with the data points and $f(t) = \ln t$ for (a) $n = 16$ (b) $n = 32$.

10.3 THE FFT AND SIGNAL PROCESSING

The DFT Interpolation Theorem 10.6 is just one application of the Fourier transform. In this section, we look at interpolation from a more general point of view, which will show how to find least squares approximations by using trigonometric functions. These ideas form the basis of modern signal processing. They will make a second appearance in Chapter 11, applied to the Discrete Cosine Transform.

10.3.1 Orthogonality and interpolation

The deceptively simple interpolation result of Theorem 10.6 was made possible by the fact that $F_n^{-1} = \overline{F_n}^T = \overline{F_n}$, making F_n a unitary matrix. We encountered the real version of this definition in Chapter 4, where we called a matrix U orthogonal if $U^{-1} = U^T$. Now we study a particular form for an orthogonal matrix that will translate immediately into a good interpolant.

THEOREM 10.9 Orthogonal Function Interpolation Theorem. Let $f_0(t), \dots, f_{n-1}(t)$ be functions of t and t_0, \dots, t_{n-1} be real numbers. Assume that the $n \times n$ matrix

$$A = \begin{bmatrix} f_0(t_0) & f_0(t_1) & \cdots & f_0(t_{n-1}) \\ f_1(t_0) & f_1(t_1) & \cdots & f_1(t_{n-1}) \\ \vdots & \vdots & \cdots & \vdots \\ f_{n-1}(t_0) & f_{n-1}(t_1) & \cdots & f_{n-1}(t_{n-1}) \end{bmatrix} \quad (10.24)$$

is a real $n \times n$ orthogonal matrix. If $y = Ax$, the function

$$F(t) = \sum_{k=0}^{n-1} y_k f_k(t)$$

interpolates $(t_0, x_0), \dots, (t_{n-1}, x_{n-1})$, that is $F(t_j) = x_j$ for $j = 0, \dots, n-1$. ■

Proof. The fact $y = Ax$ implies that

$$x = A^{-1}y = A^T y,$$

and it follows that

$$x_j = \sum_{k=0}^{n-1} a_{kj} y_k = \sum_{k=0}^{n-1} y_k f_k(t_j)$$

for $j = 0, \dots, n-1$, which completes the proof. □

► **EXAMPLE 10.4** Let $[c, d]$ be an interval and let n be an even positive integer. Show that the assumptions of Theorem 10.9 are satisfied for $t_j = c + j(d - c)/n$, $j = 0, \dots, n - 1$, and

$$\begin{aligned} f_0(t) &= \sqrt{\frac{1}{n}} \\ f_1(t) &= \sqrt{\frac{2}{n}} \cos \frac{2\pi(t - c)}{d - c} \\ f_2(t) &= \sqrt{\frac{2}{n}} \sin \frac{2\pi(t - c)}{d - c} \\ f_3(t) &= \sqrt{\frac{2}{n}} \cos \frac{4\pi(t - c)}{d - c} \\ f_4(t) &= \sqrt{\frac{2}{n}} \sin \frac{4\pi(t - c)}{d - c} \\ &\vdots \\ f_{n-1}(t) &= \frac{1}{\sqrt{n}} \cos \frac{n\pi(t - c)}{d - c}. \end{aligned}$$

The matrix is

$$A = \sqrt{\frac{2}{n}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ 1 & \cos \frac{2\pi}{n} & \cdots & \cos \frac{2\pi(n-1)}{n} \\ 0 & \sin \frac{2\pi}{n} & \cdots & \sin \frac{2\pi(n-1)}{n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \cos \pi & \cdots & \frac{1}{\sqrt{2}} \cos(n-1)\pi \end{bmatrix}. \quad (10.25)$$

Lemma 10.10 shows that the rows of A are pairwise orthogonal. 

LEMMA 10.10 Let $n \geq 1$ and k, l be integers. Then

$$\begin{aligned} \sum_{j=0}^{n-1} \cos \frac{2\pi jk}{n} \cos \frac{2\pi jl}{n} &= \begin{cases} n & \text{if both } (k-l)/n \text{ and } (k+l)/n \text{ are integers} \\ \frac{n}{2} & \text{if exactly one of } (k-l)/n \text{ and } (k+l)/n \text{ is an integer} \\ 0 & \text{if neither is an integer} \end{cases} \\ \sum_{j=0}^{n-1} \cos \frac{2\pi jk}{n} \sin \frac{2\pi jl}{n} &= 0 \\ \sum_{j=0}^{n-1} \sin \frac{2\pi jk}{n} \sin \frac{2\pi jl}{n} &= \begin{cases} 0 & \text{if both } (k-l)/n \text{ and } (k+l)/n \text{ are integers} \\ \frac{n}{2} & \text{if } (k-l)/n \text{ is an integer and } (k+l)/n \text{ is not} \\ -\frac{n}{2} & \text{if } (k+l)/n \text{ is an integer and } (k-l)/n \text{ is not} \\ 0 & \text{if neither is an integer} \end{cases} \end{aligned}$$

The proof of this lemma follows from Lemma 10.1. See Exercise 5.

Returning to Example 10.4, let $y = Ax$. Theorem 10.9 immediately gives the interpolating function

$$\begin{aligned}
F(t) = & \frac{1}{\sqrt{n}} y_0 \\
& + \sqrt{\frac{2}{n}} y_1 \cos \frac{2\pi(t-c)}{d-c} + \sqrt{\frac{2}{n}} y_2 \sin \frac{2\pi(t-c)}{d-c} \\
& + \sqrt{\frac{2}{n}} y_3 \cos \frac{4\pi(t-c)}{d-c} + \sqrt{\frac{2}{n}} y_4 \sin \frac{4\pi(t-c)}{d-c} \\
& \vdots \\
& + \frac{1}{\sqrt{n}} y_{n-1} \cos \frac{n\pi(t-c)}{d-c}
\end{aligned} \tag{10.26}$$

for the points (t_j, x_j) , in agreement with (10.19).

► **EXAMPLE 10.5** Use the basis functions of Example 10.4 to interpolate the data points $x = [-2.2, -2.8, -6.1, -3.9, 0.0, 1.1, -0.6, -1.1]$ from Example 10.3.

Computing the product of the 8×8 matrix A with x yields

$$Ax = \sqrt{\frac{2}{8}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ 1 & \cos 2\pi \frac{1}{8} & \cos 2\pi \frac{2}{8} & \cdots & \cos 2\pi \frac{7}{8} \\ 0 & \sin 2\pi \frac{1}{8} & \sin 2\pi \frac{2}{8} & \cdots & \sin 2\pi \frac{7}{8} \\ 1 & \cos 4\pi \frac{1}{8} & \cos 4\pi \frac{2}{8} & \cdots & \cos 4\pi \frac{7}{8} \\ 0 & \sin 4\pi \frac{1}{8} & \sin 4\pi \frac{2}{8} & \cdots & \sin 4\pi \frac{7}{8} \\ 1 & \cos 6\pi \frac{1}{8} & \cos 6\pi \frac{2}{8} & \cdots & \cos 6\pi \frac{7}{8} \\ 0 & \sin 6\pi \frac{1}{8} & \sin 6\pi \frac{2}{8} & \cdots & \sin 6\pi \frac{7}{8} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \cos \pi & \frac{1}{\sqrt{2}} \cos 2\pi & \cdots & \frac{1}{\sqrt{2}} \cos 7\pi \end{bmatrix} \begin{bmatrix} -2.2 \\ -2.8 \\ -6.1 \\ -3.9 \\ 0.0 \\ 1.1 \\ -0.6 \\ -1.1 \end{bmatrix} = \begin{bmatrix} -5.5154 \\ -1.4889 \\ -5.1188 \\ 2.2500 \\ 1.6500 \\ -0.7111 \\ 0.3812 \\ -0.7778 \end{bmatrix}.$$

The formula (10.26) gives the interpolating function,

$$\begin{aligned}
P(t) = & -1.95 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t \\
& + 1.125 \cos 4\pi t + 0.825 \sin 4\pi t \\
& - 0.3555 \cos 6\pi t + 0.1906 \sin 6\pi t \\
& - 0.2750 \cos 8\pi t,
\end{aligned}$$

in agreement with Example 10.3. ◀

10.3.2 Least squares fitting with trigonometric functions

Corollary 10.8 showed how the DFT makes it easy to interpolate n evenly spaced data points on $[0, 1]$ by a trigonometric function of form

$$P_n(t) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{n/2-1} (a_k \cos 2k\pi t - b_k \sin 2k\pi t) + \frac{a_{n/2}}{\sqrt{n}} \cos n\pi t. \tag{10.27}$$

Note that the number of terms is n , equal to the number of data points. (As usual in this chapter, we assume that n is even.) The more data points there are, the more cosines and sines are added to help with the interpolation.

SPOTLIGHT ON

Orthogonality

In Chapter 4, we established the normal equations $A^T A \bar{x} = A^T b$ for solving least squares approximation to data by basis functions. The point of Theorem 10.9 is to find special cases that make the normal equations trivial, greatly simplifying the least squares procedure. This leads to an extremely useful theory of so-called orthogonal functions. Major examples include the Fourier transform in this chapter and the cosine transform in Chapter 11.

As we found in Chapter 3, when the number of data points n is large, it becomes less common to fit a model function exactly. In fact, a common application of a model is to forget a few details (lossy compression) in order to simplify matters. A second reason to move away from interpolation, discussed in Chapter 4, is the case where the data points themselves are assumed to be inexact, so that rigorous enforcement of an interpolating function is inappropriate.

In either of these situations, we are motivated to do a least squares fit with a function of type (10.27). Since the coefficients a_k and b_k occur linearly in the model, we can proceed with the same program described in Chapter 4, using the normal equations to solve for the best coefficients. When we try this, we find a surprising result, which will send us right back to the DFT.

Return to Theorem 10.9. Let n denote the number of data points x_j , which we think of as occurring at evenly spaced times $t_j = j/n$ in $[0, 1]$, for simplicity. We will introduce the even positive integer m to denote the number of basis functions to use in the least squares fit. That is, we will fit to the first m of the basis functions, $f_0(t), \dots, f_{m-1}(t)$. The function used to fit the n data points will be

$$P_m(t) = \sum_{k=0}^{m-1} c_k f_k(t), \quad (10.28)$$

where the c_k are to be determined. When $m = n$, the problem is still interpolation. When $m < n$, we have changed to the compression problem. In this case, we expect to match the data points using P_m with minimum squared error.

The least squares problem is to find coefficients c_0, \dots, c_{m-1} such that the equality

$$\sum_{k=0}^{m-1} c_k f_k(t_j) = x_j$$

is met with as little error as possible. In matrix terms,

$$A_m^T c = x, \quad (10.29)$$

where A_m is the matrix of the first m rows of A . Under the assumptions of Theorem 10.9, A_m^T has pairwise orthonormal columns. When we set up the normal equations

$$A_m A_m^T c = A_m x$$

for c , $A_m A_m^T$ is the identity matrix. Therefore, the least squares solution,

$$c = A_m x, \quad (10.30)$$

is easy to calculate. We have proved the following useful result, which extends Theorem 10.9:

THEOREM 10.11 Orthogonal Function Least Squares Approximation Theorem. Let $m \leq n$ be integers, and assume that data $(t_0, x_0), \dots, (t_{n-1}, x_{n-1})$ are given. Set $y = Ax$, where A is an orthogonal matrix of form (10.24). Then the interpolating polynomial for basis functions $f_0(t), \dots, f_{n-1}(t)$ is

$$F_n(t) = \sum_{k=0}^{n-1} y_k f_k(t), \quad (10.31)$$

and the best least squares approximation, using only the functions f_0, \dots, f_{m-1} , is

$$F_m(t) = \sum_{k=0}^{m-1} y_k f_k(t). \quad (10.32)$$

This is a beautiful and useful fact. It says that, given n data points, to find the best least squares trigonometric function with $m < n$ terms fitting the data, it suffices to compute the actual interpolant with n terms and keep only the desired first m terms. In other words, the interpolating coefficients Ax for x degrade as gracefully as possible when terms are dropped from the highest frequencies. Keeping the m lowest terms in the n -term expansion guarantees the best fit possible with m lowest frequency terms. This property reflects the “orthogonality” of the basis functions.

The reasoning preceding Theorem 10.11 is easily adapted to prove something more general. We showed how to find the least squares solution for the first m basis functions, but in truth, the order was not relevant; we could have specified any subset of the basis functions. The least squares solution is found simply by dropping all terms in (10.31) that are not included in the subset. The version (10.32) is a “low-pass” filter, assuming that the lower index functions go with lower “frequencies”; but by changing the subset of basis functions kept, we can pass any frequencies of interest simply by dropping the undesired coefficients.

Now we return to the trigonometric polynomial (10.27) and demonstrate how to fit an order m version to n data points, where $m < n$. The basis functions used are the functions of Example 10.4, which satisfy the assumptions of Theorem 10.9. Theorem 10.11 shows that, whatever the interpolating coefficients, the coefficients of the best least squares approximation of order m are found by dropping all terms above order m . We have arrived at the following application:

COROLLARY 10.12 Let $[c, d]$ be an interval, let $m < n$ be even positive integers, $x = (x_0, \dots, x_{n-1})$ a vector of n real numbers, and let $t_j = c + j(d - c)/n$ for $j = 0, \dots, n - 1$. Let $\{a_0, a_1, b_1, a_2, b_2, \dots, a_{n/2-1}, b_{n/2-1}, a_{n/2}\} = F_n x$ be the interpolating coefficients for x so that

$$\begin{aligned} x_j = P_n(t_j) &= \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{\frac{n}{2}-1} \left(a_k \cos \frac{2k\pi(t_j - c)}{d - c} - b_k \sin \frac{2k\pi(t_j - c)}{d - c} \right) \\ &\quad + \frac{a_{\frac{n}{2}}}{\sqrt{n}} \cos \frac{n\pi(t_j - c)}{d - c} \end{aligned}$$

for $j = 0, \dots, n - 1$. Then

$$P_m(t) = \frac{a_0}{\sqrt{n}} + \frac{2}{\sqrt{n}} \sum_{k=1}^{\frac{m}{2}-1} \left(a_k \cos \frac{2k\pi(t - c)}{d - c} - b_k \sin \frac{2k\pi(t - c)}{d - c} \right) + \frac{2a_{\frac{m}{2}}}{\sqrt{n}} \cos \frac{n\pi(t - c)}{d - c}$$

is the best least squares fit of order m to the data (t_j, x_j) for $j = 0, \dots, n - 1$.

Another way of appreciating the power of Theorem 10.11 is to compare it with the monomial basis functions we have used previously for least squares models. The best least squares parabola fit to the points $(0, 3)$, $(1, 3)$, $(2, 5)$ is $y = x^2 - x + 3$. In other words, the best coefficients for the model $y = a + bx + cx^2$ for this data are $a = 3$, $b = -1$, and $c = 1$ (in this case because the squared error is zero—this is the interpolating parabola). Now let's fit to a subset of the basis functions—say, change the model to $y = a + bx$. We calculate the best line fit to be $a = 8/3$, $b = 1$. Note that the coefficients for the degree 1 fit have no apparent relation to their corresponding coefficients for the degree 2 fit. This is exactly what *doesn't* happen for trigonometric basis functions. An interpolating fit, or any least squares fit to the form (10.28), explicitly contains all the information about lower order least squares fits.

Because of the extremely simple answer DFT has for least squares, it is especially simple to write a computer program to carry out the steps. Let $m < n < p$ be integers, where n is the number of data points, m is the order of the least squares trigonometric model, and p governs the resolution of the plot of the best model. We can think of least squares as “filtering out” the highest frequency contributions of the order n interpolant and leaving only the lowest m frequency contributions. That explains the name of the following MATLAB function:

```
% Program 10.2 Least squares trigonometric fit
% Least squares fit of n data points on [0,1] with trig function
%   where 2 <= m <= n. Plot best fit at p (>=n) points.
% Input: interval [c,d], data points x, even number m,
%   even number of data points n, even number p >= n
% Output: filtered points xp
function xp=dftfilter(inter,x,m,n,p)
c=inter(1); d=inter(2);
t=c+(d-c)*(0:n-1)/n;          % time points for data (n)
tp=c+(d-c)*(0:p-1)/p         % time points for interpolant (p)
y=fft(x);                     % compute interpolation coefficients
yp=zeros(p,1);                % will hold coefficients for ifft
yp(1:m/2)=y(1:m/2);           % keep only first m frequencies
yp(m/2+1)=real(y(m/2+1));     % since m is even, keep cos term only
if(m<n)                        % unless at the maximum frequency,
    yp(p-m/2+1)=yp(m/2+1);    %   add complex conjugate to
end                            %   corresponding place in upper tier
yp(p-m/2+2:p)=y(n-m/2+2:n);  % more conjugates for upper tier
xp=real(ifft(yp))*(p/n);      % invert fft to recover data
plot(t,x,'o',tp,xp)           % plot data and least square approx
```

► **EXAMPLE 10.6** Fit the temperature data from Example 10.3 by least squares trigonometric functions of orders 4 and 6.

The point of Corollary 10.12 is that we can just interpolate the data points by applying F_n and then chop off terms at will to get the least squares fit of lower orders. The result from Example 10.3 was that

$$\begin{aligned}
 P_8(t) = & -1.95 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t \\
 & + 1.125 \cos 4\pi t + 0.825 \sin 4\pi t \\
 & - 0.3555 \cos 6\pi t + 0.1906 \sin 6\pi t \\
 & - 0.2750 \cos 8\pi t.
 \end{aligned} \tag{10.33}$$

Therefore, the least squares models of orders 4 and 6 are

$$P_4(t) = -1.95 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t + 1.125 \cos 4\pi t$$

$$P_6(t) = -1.95 - 0.7445 \cos 2\pi t - 2.5594 \sin 2\pi t$$

$$+ 1.125 \cos 4\pi t + 0.825 \sin 4\pi t - 0.3555 \cos 6\pi t.$$

Figure 10.7 shows both least squares fits, generated by

```
dftfilter([0,1],[-2.2,-2.8,-6.1,-3.9,0.0,1.1,-0.6,-1.1],m,8,200)
```

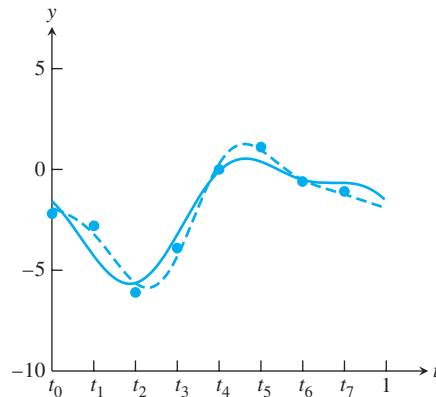


Figure 10.7 Least squares trigonometric fits for Example 10.6. Fits for $m=4$ (solid curve) and 6 (dashed curve) are shown. The input vector x is $[-2.2, -2.8, -6.1, -3.9, 0.0, 1.1, -0.6, -1.1]^T$. The fit for $m=8$ is trigonometric interpolation, shown in Figure 10.6.

for $m=4$ and 6, respectively. The $m=4$ fit matches the explicit least squares fit to the basis functions $1, \cos 2\pi t, \sin 2\pi t, \cos 4\pi t$ carried out in Example 4.6 and plotted in Figure 4.5(b). ◀

The program `dftfilter.m` can be made more efficient. It computes the order n interpolant and then ignores $n-m$ coefficients. Of course, one look at the Fourier matrix F_n shows that if we want to know only the first m Fourier coefficients of n data points, we can multiply x by only the top m rows of F_n and leave it at that. In other words, it would suffice to replace the $n \times n$ matrix F_n by an $m \times n$ submatrix. An improved version of `dftfilter.m` would make use of this fact.

10.3.3 Sound, noise, and filtering

The `dftfilter.m` code of the last section is an introduction to the vast area of digital signal processing. We are using the Fourier transform as a way to transfer the information of a signal $\{x_0, \dots, x_{n-1}\}$ from the “time domain” to the “frequency domain,” where it is easier to manipulate. When we finish changing what we want to change, we send the signal back to the time domain by an inverse FFT.

If x represents an audio signal, this is helpful because of the way our hearing system is constructed. The human ear contains structures that respond to frequencies, and so the building blocks in the frequency domain are directly meaningful. We will illustrate this by introducing some basic concepts of audio and signal processing and a few convenient MATLAB commands.

An audio signal consists of a set of real numbers indexed by time. Each real number represents a sound intensity. When an audio signal is played back, the speaker head is made to vibrate so that the amplitude matches the signal, causing the surrounding air to vibrate at the same frequencies. When the sound waves reach your ear, you perceive sound.

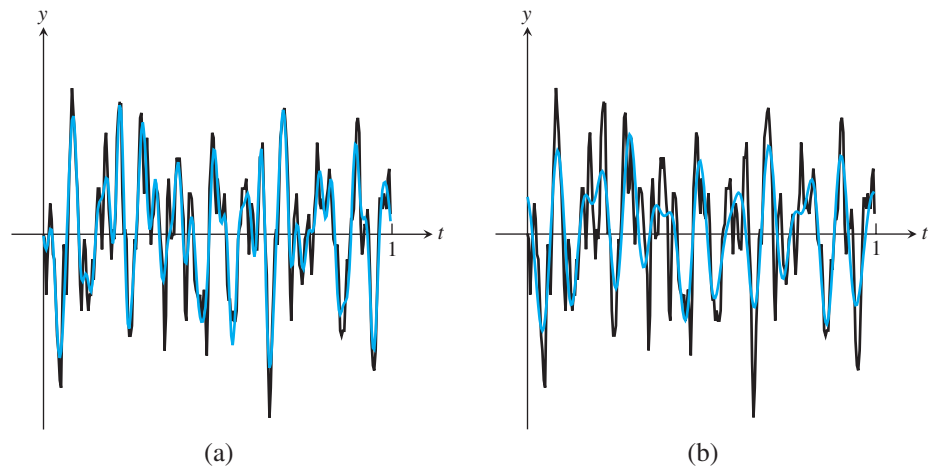


Figure 10.8 Sound curve along with filtered versions. First 1/32 second of *Hallelujah Chorus* (256 points on black curve) along with filtered version (blue curve) with (a) 64 basis functions, a 4:1 compression ratio and (b) 32 basis functions, an 8:1 compression ratio.

MATLAB provides an audio signal of the first 9 seconds of Handel's *Hallelujah Chorus* for us to sample. The curve in Figure 10.8 shows the first $2^8 = 256$ values of the file, which consists of sound intensities. The sampling rate of the music is $2^{13} = 8192$ Hz, meaning that intensities are represented at the rate of 2^{13} per second, evenly spaced. To access the signal, type

```
>> load handel
```

which puts the variables F_s and y in the workspace. The former variable is the sampling rate $F_s = 8192$. The variable y is a length 73113 vector containing the sound signal. The MATLAB command

```
>> sound(y, F_s)
```

plays the signal on your computer speakers, if available, at the correct sampling rate F_s .

The *Hallelujah Chorus* data can be used to implement the filtering of Corollary 10.12. Using `dftfilter.m` with the first $n = 256$ samples of the signal, and $m = 64$ and 32 basis functions, results in the blue curves of Figure 10.8. The reader may want to explore filtering with other audio files.

One common audio file format is the .wav format. A stereo .wav file carries two paired signals to be played from two different speakers. For example, using the MATLAB command

```
>> [y, F_s] = wavread('castanets')
```

will extract the stereo signal from the file `castanets.wav` and load it into MATLAB as an $n \times 2$ matrix y , each column a separate sound signal. (The file `castanets.wav` is a common audio test file and can be easily found by a web search.) The MATLAB command `wavwrite` reverses the process, creating a .wav file from simple sound signals.

Filtering is used in two ways. It can be used to match the original sound wave as closely as possible with a simpler function. This is a form of compression. Instead of using 256

SPOTLIGHT ON**Compression**

Filtering is a form of lossy compression. In the case of an audio signal, the goal is to reduce the amount of data required to store or transmit the sound without compromising the musical effect or spoken information the signal is designed to represent. This is best done in the frequency domain, which means applying the DFT, manipulating the frequency components, and then inverting the DFT.

numbers to store the wave, we could instead just store the lowest m frequency components and then reconstruct the wave when needed by using Corollary 10.12. In Figure 10.8(a), we used $m = 64$ real numbers in place of the original 256, a 4:1 compression ratio. Note that the compression is lossy, in that the original wave has not been reproduced exactly.

The second major application of filtering is to remove noise. Given a music file where the music or speech was corrupted by high-frequency noise (or hiss), eliminating the higher frequency contributions may be important to enhancing the sound. Of course, so-called low-pass filters are blunt hammers—a high-frequency part of the desired sound, possibly in overtones not even obvious to the listener, may be deleted as well. The topic of filtering is part of a vast literature on signal processing, and the reader is referred to Oppenheim and Schaffer [2009] for further study. In Reality Check 10, we investigate a filter of widespread application called the Wiener filter.

10.3 Exercises

- Find the best order 2 least squares approximation to the data in Exercise 10.2.1, using the basis functions 1 and $\cos 2\pi t$.
- Find the best order 3 least squares approximation to the data in Exercise 10.2.1, using the basis functions 1 , $\cos 2\pi t$, and $\sin 2\pi t$.
- Find the best order 4 least squares approximation to the data in Exercise 10.2.3, using the basis functions 1 , $\cos 2\pi t$, $\sin 2\pi t$, and $\cos 4\pi t$.
- Find the best order 4 least squares approximation to the data in Exercise 10.2.4, using the basis functions 1 , $\cos \frac{\pi}{4} t$, $\sin \frac{\pi}{4} t$, and $\cos \frac{\pi}{2} t$.
- Prove Lemma 10.10. (*Hint*: Express $\cos 2\pi jk/n$ as $(e^{i2\pi jk/n} + e^{-i2\pi jk/n})/2$, and write everything in terms of $\omega = e^{-i2\pi/n}$, so that Lemma 10.1 can be applied.)

10.3 Computer Problems

- Find the least squares trigonometric approximating functions of orders $m = 2$ and 4 for the following data points:

t	y	t	y	t	y	t	y
0	3	0	2	0	5	1	-1
$\frac{1}{4}$	0	$\frac{1}{4}$	0	1	2	2	1
$\frac{1}{2}$	-3	$\frac{1}{2}$	5	2	6	3	4
$\frac{3}{4}$	0	$\frac{3}{4}$	1	3	1	4	3
						5	3
						6	2

Using `dftfilter.m`, plot the data points and the approximating functions, as in Figure 10.7.

2. Find the least squares trigonometric approximating functions of orders 4, 6, and 8 for the following data points:

t		y	t		y	t		y	t		y
(a)	0	3	(b)	0	1	(c)	0	1	(d)	0	4.2
	$\frac{1}{8}$	0		$\frac{1}{8}$	0		$\frac{1}{8}$	2		$\frac{1}{8}$	5.0
	$\frac{1}{4}$	-3		$\frac{1}{4}$	-2		$\frac{1}{4}$	3		$\frac{1}{4}$	3.8
	$\frac{3}{8}$	0		$\frac{3}{8}$	1		$\frac{3}{8}$	1		$\frac{3}{8}$	1.6
	$\frac{1}{2}$	3		$\frac{1}{2}$	3		$\frac{1}{2}$	-1		$\frac{1}{2}$	-2.0
	$\frac{5}{8}$	0		$\frac{5}{8}$	0		$\frac{5}{8}$	-1		$\frac{5}{8}$	-1.4
	$\frac{3}{4}$	-6		$\frac{3}{4}$	-2		$\frac{3}{4}$	-3		$\frac{3}{4}$	0.0
	$\frac{7}{8}$	0		$\frac{7}{8}$	1		$\frac{7}{8}$	0		$\frac{7}{8}$	1.0

Plot the data points and the approximating functions, as in Figure 10.7.

- Plot the least squares trigonometric approximation function of orders $m = n/2$, $n/4$, and $n/8$, along with the vector x containing the first 2^{14} sound intensity values from MATLAB's `handel` sound file. (This covers about 2 seconds of audio. The MATLAB code `dftfilter` can be used with $p = n$. Make three separate plots.) Use the MATLAB `sound` command to compare the original with the approximation. What has been lost?
- Download `castanets.wav` from an appropriate website, and form a vector containing the signal at the first 2^{14} sample times. Carry out the steps of Computer Problem 3 for each stereo channel separately.
- Gather 24 consecutive hourly temperature readings from a newspaper or website. Plot the data points along with (a) the trigonometric interpolating function and least squares approximating functions of order (b) $m = 6$ and (c) $m = 12$.



10 The Wiener Filter

Let c be a clean audio signal, and add a vector r of the same length to c . Is the resulting signal $x = c + r$ noisy? If $r = c$, we would not consider r noise, since the result would be a louder, but still clean, version of c . By definition, noise is uncorrelated with the signal. In other words, if r is noise, the expected value of the inner product $c^T r$ is zero. We will exploit this lack of correlation next.

In a typical application, we are presented with a noisy signal x and asked to find c . The signal c might be the value of an important system variable, being monitored in a noisy environment. Or, as in our example below, c might be an audio sample that we want to bring out of noise. In the middle of the 20th century, Norbert Wiener suggested looking for the optimal filter for removing the noise from x , in the sense of least squares error. He suggested finding a real, diagonal matrix Φ such that the Euclidean norm of

$$F^{-1}\Phi Fx - c$$

is as small as possible, where F denotes the Discrete Fourier Transform. The idea is to clean up the signal x by applying the Fourier transform, operating on the frequency components by multiplying by Φ , and then inverting the Fourier transform. This is called filtering in the frequency domain, since we are changing the Fourier-transformed version of x rather than x itself.

To find the best diagonal matrix Φ , note that

$$\begin{aligned} \|F^{-1}\Phi Fx - c\|_2 &= \|\Phi Fx - Fc\|_2 \\ &= \|\Phi F(c + r) - Fc\|_2 \\ &= \|(\Phi - I)C + \Phi R\|_2, \end{aligned} \quad (10.34)$$

where we set $C = Fc$ and $R = Fr$ to be the Fourier transforms. Note also that the definition of noise implies

$$\overline{C}^T R = \overline{F}^T c^T F r = c^T \overline{F}^T F r = c^T r = 0.$$

We will use this as motivation to ignore the cross-terms in the norm, so that the squared magnitude reduces to

$$\begin{aligned} \left(\overline{(\Phi - I)C + \Phi R} \right)^T ((\Phi - I)C + \Phi R) &= \left(\overline{C}^T (\Phi - I) + \overline{R}^T \Phi \right) ((\Phi - I)C + \Phi R) \\ &\approx \overline{C}^T (\Phi - I)^2 C + \overline{R}^T \Phi^2 R \\ &= \sum_{i=1}^n (\phi_i - 1)^2 |C_i|^2 + \phi_i^2 |R_i|^2. \end{aligned} \quad (10.35)$$

To find the diagonal entries ϕ_i that minimize this expression, differentiate with respect to each ϕ_i separately to obtain

$$2(\phi_i - 1)|C_i|^2 + 2\phi_i|R_i|^2 = 0$$

for each i , or, solving for ϕ_i ,

$$\phi_i = \frac{|C_i|^2}{|C_i|^2 + |R_i|^2}. \quad (10.36)$$

This formula gives Wiener's values for the entries of the diagonal matrix Φ , to minimize the difference between the filtered version $F^{-1}\Phi Fx$ and the clean signal c . The only problem is that in typical cases, we don't know C or R and must make some approximations to apply the formula.

Your job is to investigate ways of putting together an approximation. Let $X = Fx$ be the Fourier transform. Again using the uncorrelatedness of signal and noise, approximate

$$|X_i|^2 \approx |C_i|^2 + |R_i|^2.$$

Then we can write the optimal choice as

$$\phi_i \approx \frac{|X_i|^2 - |R_i|^2}{|X_i|^2} \quad (10.37)$$

and use our best knowledge of the noise level. For example, if the noise is uncorrelated Gaussian noise (modeled by adding a normal random number independently to each sample of the clean signal), we could replace $|R_i|^2$ in (10.37) with the constant $(p\sigma)^2$, where σ is the standard deviation of the noise and p is a parameter near one to be chosen. Note that

$$\sum_{i=1}^n |R_i|^2 = \overline{R}^T R = r^T \overline{F}^T F r = r^T r = \sum_{i=1}^n r_i^2.$$

In the following code, we add 50 percent noise to the Handel signal, and use $p = 1.3$ standard deviations to approximate R_i :

```

load handel                                % y is clean signal
c=y(1:40000);                             % work with first 40K samples
p=1.3;                                    % parameter for cutoff
noise=std(c)*.50;                          % 50 percent noise
n=length(c);                             % n is length of signal
r=noise*randn(n,1);                       % pure noise
x=c+r;                                    % noisy signal
fx=fft(x); sfx=conj(fx).*fx;               % take fft of signal, and
sfcapprox=max(sfx-n*(p*noise)^2,0);        % apply cutoff
phi=sfcapprox./sfx;                       % define phi as derived
xout=real(ifft(phi.*fx));                 % invert the fft
% then compare sound(x) and sound(xout)

```

Suggested activities:

1. Run the code to form the filtered signal y_f , and use MATLAB's `sound` command to compare the input and output signals.
2. Compute the mean squared error (MSE) of the input (y_s) and output (y_f) by comparing with the clean signal (y_c).
3. Find the best value of the parameter p for 50 percent noise. Compare the value that minimizes MSE to the one that sounds best to the ear.
4. Change the noise level to 10 percent, 25 percent, 100 percent, 200 percent, and repeat Step 3. Summarize your conclusions.
5. Design a fair comparison of the Wiener filter with the low-pass filter described in Section 10.2, and carry out the comparison.
6. Download a .wav file of your choice, add noise, and carry out the aforementioned steps. ✓

Software and Further Reading

Good sources for further reading on the Discrete Fourier Transform include Briggs [1995], Brigham [1988], and Briggs and Henson [1995]. The original breakthrough of Cooley and Tukey appeared in Cooley and Tukey [1965], and computational improvements that have continued as the central place of the Fast Fourier Transform in modern signal processing have been acknowledged (Winograd [1978], Van Loan [1992], and Chu and George [1999]). The FFT is an important algorithm in its own right and, additionally, is used as a building block in other algorithms because of its efficient implementation. For example, it is used by MATLAB to compute the Discrete Cosine Transform, defined in Chapter 11. Interestingly, the divide-and-conquer strategy used by Cooley and Tukey was later successfully applied to many other computational problems.

MATLAB's `fft` command is based on the “Fastest Fourier Transform in the West” (FFTW), developed in the 1990s at MIT (Frigo and Johnson [1998]). In case the size n is not a power of two, the program breaks down the problem, using the prime factors of n , into smaller “codelets” optimized for particular fixed sizes. More information on the FFTW, including downloadable code, is available at <http://www.fftw.org>. IMSL provides the forward transform FFTCF and inverse transform FFTCB, based on Netlib's FFTPACK (Swarztrauber [1982]), a package of Fortran subprograms for the Fast Fourier Transform, optimized for use in parallel implementations.



Eigenvalues and Singular Values

The World Wide Web makes vast amounts of information easily accessible to the casual user—so vast, in fact, that navigation with a powerful search engine is essential. Technology has also provided miniaturization and low-cost sensors, making great quantities of data available to researchers. How can access to large amounts of information be exploited in an efficient way?

Many aspects of search technology, and knowledge discovery in general, benefit from treatment as an

eigenvalue or singular value problem. Numerical methods to solve these high-dimensional problems generate projections to distinguished lower dimensional subspaces. This is exactly the simplification that complex data environments most need.

Reality Check

Reality Check 12 on page 549 explores what has been called the largest ongoing eigenvalue computation in the world, used by one of the well-known web search providers.

Computational methods for locating eigenvalues are based on the fundamental idea of power iteration, a type of fixed-point iteration for eigenspaces. A sophisticated version of the idea, called the QR Algorithm, is the standard algorithm for determining all eigenvalues of typical matrices.

The singular value decomposition reveals the basic structure of a matrix and is heavily used in statistical applications to find relations between data. In this chapter, we survey methods for finding the eigenvalues and eigenvectors of a square matrix, and the singular values and singular vectors of a general matrix.

12.1 POWER ITERATION METHODS

There is no direct method for computing eigenvalues. The situation is analogous to root-finding, in that all feasible methods depend on some type of iteration. To begin the section, we consider whether the problem might be reducible to root-finding.

Appendix A shows a method for calculating eigenvalues and eigenvectors of an $m \times m$ matrix. This approach, based on finding the roots of the degree m characteristic polynomial, works well for 2×2 matrices. For larger matrices, the procedure requires a rootfinder of the type studied in Chapter 1.

The difficulty of this approach to finding eigenvalues becomes clear if we recall the example of the Wilkinson polynomial of Chapter 1. There we found that very small changes in the coefficients of a polynomial can change the roots of the polynomial by arbitrarily large amounts. In other words, the condition number of the input/output problem taking coefficients to roots can be extremely large. Because our calculation of the coefficients of the characteristic polynomial will be subject to errors on the order of machine roundoff or larger, calculation of eigenvalues by this approach is susceptible to large errors. This difficulty is serious enough to warrant eliminating the method of finding roots of the characteristic polynomial as a pathway to the accurate calculation of eigenvalues.

A simple example of poor accuracy for this method follows from the existence of the Wilkinson polynomial. If we are trying to find the eigenvalues of the matrix

$$A = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 2 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 20 \end{bmatrix}, \quad (12.1)$$

we will calculate the coefficients of the characteristic polynomial $P(x) = (x - 1)(x - 2) \cdots (x - 20)$ and use a rootfinder to find the roots. However, as shown in Chapter 1, some of the roots of the machine version of $P(x)$ are far from the roots of the true version of $P(x)$, which are the eigenvalues of A .

This section introduces methods based on multiplying high powers of the matrix times a vector, which usually will turn into an eigenvector as the power is raised. We will refine the idea later, but it is the main thrust of the most sophisticated methods.

12.1.1 Power Iteration

The motivation behind Power Iteration is that multiplication by a matrix tends to move vectors toward the dominant eigenvector direction.

SPOTLIGHT ON

Conditioning

The large errors that the “characteristic polynomial method” are subject to are not the fault of the rootfinder. A perfectly accurate rootfinder would fare no better. When the polynomial is multiplied out to determine its coefficients for entry into the rootfinder, the coefficients will, in general, be subject to errors on the order of machine epsilon. The rootfinder will then be asked to find the roots of the slightly wrong polynomial, which, as we have seen, can have disastrous consequences. There is no general fix to this problem. The only way to fight the problem would be to increase the size of the mantissa representing floating point numbers, which would have the effect of lowering machine epsilon. If machine epsilon could be made lower than $1/\text{cond}(P)$, then accuracy could be assured for the eigenvalues. Of course, this is not really a solution, but just another step in an unwinnable arms race. If higher precision computing is used, we can always extend the Wilkinson polynomial to a higher degree to find an even higher condition number.

DEFINITION 12.1 Let A be an $m \times m$ matrix. A **dominant eigenvalue** of A is an eigenvalue λ whose magnitude is greater than all other eigenvalues of A . If it exists, an eigenvector associated to λ is called a **dominant eigenvector**. \square

The matrix

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

has a dominant eigenvalue of 4 with eigenvector $[1, 1]^T$, and an eigenvector that is smaller in magnitude, -1 , with associated eigenvector $[-3, 2]^T$. Let us observe the result of multiplying the matrix A times a “random” vector, say $[-5, 5]^T$:

$$\begin{aligned} x_1 &= Ax_0 = \begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} -5 \\ 5 \end{bmatrix} = \begin{bmatrix} 10 \\ 0 \end{bmatrix} \\ x_2 &= A^2x_0 = \begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} 10 \\ 0 \end{bmatrix} = \begin{bmatrix} 10 \\ 20 \end{bmatrix} \\ x_3 &= A^3x_0 = \begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \end{bmatrix} = \begin{bmatrix} 70 \\ 60 \end{bmatrix} \\ x_4 &= A^4x_0 = \begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} 70 \\ 60 \end{bmatrix} = \begin{bmatrix} 250 \\ 260 \end{bmatrix} = 260 \begin{bmatrix} \frac{25}{26} \\ 1 \end{bmatrix} \end{aligned}$$

Multiplying a random starting vector repeatedly by the matrix A has resulted in moving the vector very close to the dominant eigenvector of A . This is no coincidence, as can be seen by expressing x_0 as a linear combination of the eigenvectors

$$x_0 = 1 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} -3 \\ 2 \end{bmatrix}$$

and reviewing the calculation in this light:

$$\begin{aligned} x_1 &= Ax_0 = 4 \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 2 \begin{bmatrix} -3 \\ 2 \end{bmatrix} \\ x_2 &= A^2x_0 = 4^2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} -3 \\ 2 \end{bmatrix} \\ x_3 &= A^3x_0 = 4^3 \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 2 \begin{bmatrix} -3 \\ 2 \end{bmatrix} \\ x_4 &= A^4x_0 = 4^4 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} -3 \\ 2 \end{bmatrix} \\ &= 256 \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 2 \begin{bmatrix} -3 \\ 2 \end{bmatrix}. \end{aligned}$$

The point is that the eigenvector corresponding to the eigenvalue that is largest in magnitude will dominate the calculation after several steps. In this case, the eigenvalue 4 is largest, and so the calculation moves closer and closer to an eigenvector in its direction $[1, 1]^T$.

To keep the numbers from getting out of hand, it is necessary to normalize the vector at each step. One way to do this is to divide the current vector by its length prior to each step. The two operations, normalization and multiplication by A constitute the method of Power Iteration.

As the steps deliver improved approximate eigenvectors, how do we find approximate eigenvalues? To pose the question more generally, assume that a matrix A and an approximate eigenvector are known. What is the best guess for the associated eigenvalue?

SPOTLIGHT ON

Convergence

Power Iteration is essentially a fixed-point iteration with normalization at each step. Like FPI, it converges linearly, meaning that during convergence, the error decreases by a constant factor on each iteration step. Later in this section, we will encounter a quadratically convergent variant of Power Iteration called Rayleigh Quotient Iteration.

We will appeal to least squares. Consider the eigenvalue equation $x\lambda = Ax$, where x is an approximate eigenvector and λ is unknown. Looked at this way, the coefficient matrix is the $n \times 1$ matrix x . The normal equations say that the least squares answer is the solution of $x^T x \lambda = x^T Ax$, or

$$\lambda = \frac{x^T Ax}{x^T x}, \quad (12.2)$$

known as the **Rayleigh quotient**. Given an approximate eigenvector, the Rayleigh quotient is the best approximate eigenvalue. Applying the Rayleigh quotient to the normalized eigenvector adds an eigenvalue approximation to Power Iteration.

Power Iteration

Given initial vector x_0 .

```
for    j = 1, 2, 3, ...
        uj-1 = xj-1 / ||xj-1||2
        xj = Auj-1
        λj = uTj-1 Auj-1
end
uj = xj / ||xj||2
```

To find the dominant eigenvector of the matrix A , begin with an initial vector. Each iteration consists of normalizing the current vector and multiplying by A . The Rayleigh quotient is used to approximate the eigenvalue. The MATLAB `norm` command makes this simple to implement, as shown in the following code:

```
% Program 12.1 Power Iteration
% Computes dominant eigenvector of square matrix
% Input: matrix A, initial (nonzero) vector x, number of steps k
% Output: dominant eigenvalue lam, eigenvector u
function [lam,u]=powerit(A,x,k)
for j=1:k
    u=x/norm(x);           % normalize vector
    x=A*u;                 % power step
    lam=u'*x;              % Rayleigh quotient
end
u=x/norm(x);
```

12.1.2 Convergence of Power Iteration

We will prove the convergence of Power Iteration under certain conditions on the eigenvalues. Although these conditions are not completely general, they serve to show why the method succeeds in the clearest possible case. Later, we will assemble successively more sophisticated eigenvalue methods, built on the basic concept of Power Iteration, that cover more general matrices.

THEOREM 12.2 Let A be an $m \times m$ matrix with real eigenvalues $\lambda_1, \dots, \lambda_m$ satisfying $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \dots \geq |\lambda_m|$. Assume that the eigenvectors of A span R^m . For almost every initial vector,

Power Iteration converges linearly to an eigenvector associated to λ_1 with convergence rate constant $S = |\lambda_2/\lambda_1|$. ■

Proof. Let v_1, \dots, v_n be the eigenvectors that form a basis of R^n , with corresponding eigenvalues $\lambda_1, \dots, \lambda_n$, respectively. Express the initial vector in this basis as $x_0 = c_1 v_1 + \dots + c_n v_n$ for some coefficients c_i . The phrase “for almost every initial vector” means we can assume that $c_1, c_2 \neq 0$. Applying Power Iteration yields

$$\begin{aligned} Ax_0 &= c_1 \lambda_1 v_1 + c_2 \lambda_2 v_2 + \dots + c_n \lambda_n v_n \\ A^2 x_0 &= c_1 \lambda_1^2 v_1 + c_2 \lambda_2^2 v_2 + \dots + c_n \lambda_n^2 v_n \\ A^3 x_0 &= c_1 \lambda_1^3 v_1 + c_2 \lambda_2^3 v_2 + \dots + c_n \lambda_n^3 v_n \\ &\vdots \end{aligned}$$

with normalization at each step. As the number of steps $k \rightarrow \infty$, the first term on the right-hand side will dominate, no matter how the normalization is done, because

$$\frac{A^k x_0}{\lambda_1^k} = c_1 v_1 + c_2 \left(\frac{\lambda_2}{\lambda_1} \right)^k v_2 + \dots + c_n \left(\frac{\lambda_n}{\lambda_1} \right)^k v_n.$$

The assumption that $|\lambda_1| > |\lambda_i|$ for $i > 1$ implies that all but the first term on the right will converge to zero with convergence rate $S \leq |\lambda_2/\lambda_1|$, and exactly that rate, as long as $c_2 \neq 0$. As a result, the method converges to a multiple of the dominant eigenvector v_1 , with eigenvalue λ_1 . ■

The term “almost every” in the theorem’s conclusion means that the set of initial vectors x_0 for which the iteration fails is a set of lower dimension in R^m . Specifically, the iteration will succeed at the specified rate if x_0 is not contained in the union of the dimension $m - 1$ planes spanned by $\{v_1, v_3, \dots, v_m\}$ and $\{v_2, v_3, \dots, v_m\}$.

12.1.3 Inverse Power Iteration

Power Iteration is limited to locating the eigenvalue of largest magnitude (absolute value). If Power Iteration is applied to the inverse of the matrix, the smallest eigenvalue can be found.

LEMMA 12.3 Let the eigenvalues of the $m \times m$ matrix A be denoted by $\lambda_1, \lambda_2, \dots, \lambda_m$. (a) The eigenvalues of the inverse matrix A^{-1} are $\lambda_1^{-1}, \lambda_2^{-1}, \dots, \lambda_m^{-1}$, assuming that the inverse exists. The eigenvectors are the same as those of A . (b) The eigenvalues of the shifted matrix $A - sI$ are $\lambda_1 - s, \lambda_2 - s, \dots, \lambda_m - s$ and the eigenvectors are the same as those of A . ■

Proof. (a) $Av = \lambda v$ implies that $v = \lambda A^{-1}v$, and therefore, $A^{-1}v = (1/\lambda)v$. Note that the eigenvector is unchanged. (b) Subtract sIv from both sides of $Av = \lambda v$. Then $(A - sI)v = (\lambda - s)v$ is the definition of eigenvalue for $(A - sI)$, and again the same eigenvector can be used. ■

According to Lemma 12.3, the largest magnitude eigenvalue of the matrix A^{-1} is the reciprocal of the smallest magnitude eigenvalue of A . Applying Power Iteration to the inverse matrix, followed by inverting the resulting eigenvalue of A^{-1} , gives the smallest magnitude eigenvalue of A .

To avoid explicit calculation of the inverse of A , we rewrite the application of Power Iteration to A^{-1} , namely,

$$x_{k+1} = A^{-1}x_k \quad (12.3)$$

as the equivalent

$$Ax_{k+1} = x_k, \quad (12.4)$$

which is then solved for x_{k+1} by Gaussian elimination.

Now we know how to find the largest and smallest eigenvalues of a matrix. In other words, for a 100×100 matrix, we are 2 percent finished. How do we find the other 98 percent?

One approach is suggested by Lemma 12.3(b). We can make any of the other eigenvalues small by shifting A by a value close to the eigenvalue. If we happen to know that there is an eigenvalue near 10 (say, 10.05), then $A - 10I$ has an eigenvalue $\lambda = 0.05$. If it is the smallest magnitude eigenvalue of $A - 10I$, then the Inverse Power Iteration $x_{k+1} = (A - 10I)^{-1}x_k$ will locate it. That is, the Inverse Power Iteration will converge to the reciprocal $1/(.05) = 20$, after which we invert to .05 and add the shift back to get 10.05. This trick will locate the eigenvalue that is smallest after the shift—which is another way of saying the eigenvalue nearest to the shift. To summarize, we write

Inverse Power Iteration

Given initial vector x_0 and shift s

```

for     $j = 1, 2, 3, \dots$ 
         $u_{j-1} = x_{j-1} / \|x_{j-1}\|_2$ 
        Solve  $(A - sI)x_j = u_{j-1}$ 
         $\lambda_j = u_{j-1}^T x_j$ 
end
 $u_j = x_j / \|x_j\|_2$ 

```

To find the eigenvalue of A nearest to the real number s , apply Power Iteration to $(A - sI)^{-1}$ to get the largest magnitude eigenvalue b of $(A - sI)^{-1}$. The power iterations should be done by Gaussian elimination on $(A - sI)y_{k+1} = x_k$. Then $\lambda = b^{-1} + s$ is the eigenvalue of A nearest to s . The eigenvector associated to λ is given directly from the calculation.

```

% Program 12.2 Inverse Power Iteration
% Computes eigenvalue of square matrix nearest to input s
% Input: matrix A, (nonzero) vector x, shift s, steps k
% Output: dominant eigenvalue lam, eigenvector of inv(A-sI)
function [lam,u]=invpowerit(A,x,s,k)
As=A-s*eye(size(A));
for j=1:k
    u=x/norm(x);           % normalize vector
    x=As\u;                % power step
    lam=u'*x;              % Rayleigh Quotient
end
lam=1/lam+s; u=x/norm(x);

```

► **EXAMPLE 12.1** Assume that A is a 5×5 matrix with eigenvalues $-5, -2, 1/2, 3/2, 4$. Find the eigenvalue and convergence rate expected when applying (a) Power Iteration (b) Inverse Power Iteration with shift $s = 0$ (c) Inverse Power Iteration with shift $s = 2$.

(a) Power Iteration with a random initial vector will converge to the largest magnitude eigenvalue -5 , with convergence rate $S = |\lambda_2|/|\lambda_1| = 4/5$. (b) Inverse Power Iteration

(with no shift) will converge to the smallest, $1/2$, because its reciprocal 2 is larger than the other reciprocals $-1/5$, $-1/2$, $2/3$, and $1/4$. The convergence rate will be the ratio of the two largest eigenvalues of the inverse matrix, $S = (2/3)/2 = 1/3$. (c) The Inverse Power Iteration with shift $s = 2$ will locate the eigenvalue nearest to 2, which is $3/2$. The reason is that, after shifting the eigenvalues to -7 , -4 , $-3/2$, $-1/2$, and 2, the largest of the reciprocals is -2 . After inverting to get $-1/2$ and adding back the shift $s = 2$, we get $3/2$. The convergence rate is again the ratio $(2/3)/2 = 1/3$. ◀

12.1.4 Rayleigh Quotient Iteration

The Rayleigh quotient can be used in conjunction with Inverse Power Iteration. We know that it converges to the eigenvector associated to the eigenvalue with the smallest distance to the shift s , and that convergence is fast if this distance is small. If at any step along the way an approximate eigenvalue were known, it could be used as the shift s , to speed convergence.

Using the Rayleigh quotient as the updated shift in Inverse Power Iteration leads to Rayleigh Quotient Iteration (RQI).

Rayleigh Quotient Iteration

Given initial vector x_0 .

```

for     $j = 1, 2, 3, \dots$ 
         $u_{j-1} = x_{j-1} / \|x_{j-1}\|$ 
         $\lambda_{j-1} = u_{j-1}^T A u_{j-1}$ 
        Solve  $(A - \lambda_{j-1} I)x_j = u_{j-1}$ 
end
 $u_j = x_j / \|x_j\|_2$ 

```

```

% Program 12.3 Rayleigh Quotient Iteration
% Input: matrix A, initial (nonzero) vector x, number of steps k
% Output: eigenvalue lam and eigenvector u
function [lam,u]=rqi(A,x,k)
for j=1:k
    u=x/norm(x);           % normalize
    lam=u'*A*u;             % Rayleigh quotient
    x=(A-lam*eye(size(A)))\u; % inverse power iteration
end
u=x/norm(x);
lam=u'*A*u;                % Rayleigh quotient

```

While Inverse Power Iteration converges linearly, Rayleigh Quotient Iteration is quadratically convergent for simple (nonrepeated) eigenvalues and will converge cubically if the matrix is symmetric. This means that very few steps are needed to converge to machine precision for this method. After convergence, the matrix $A - \lambda_{j-1} I$ is singular and no more steps can be performed. As a result, trial and error should be used with Program 12.3 to stop the iteration just before this occurs. Note that the complexity has grown for RQI. Inverse Power Iteration requires only one LU factorization; but for RQI, each step requires a new factorization, since the shift has changed. Even so, Rayleigh Quotient Iteration is the fastest converging method we have presented in this section on finding one eigenvalue at a time. In the next section, we discuss ways to find all eigenvalues of a matrix in the same calculation. The basic engine will remain Power Iteration—it is only the organizational details that will become more sophisticated.

12.1 Exercises

1. Find the characteristic polynomial and the eigenvalues and eigenvectors of the following symmetric matrices:

$$(a) \begin{bmatrix} 3.5 & -1.5 \\ -1.5 & 3.5 \end{bmatrix} \quad (b) \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix} \quad (c) \begin{bmatrix} -0.2 & -2.4 \\ -2.4 & 1.2 \end{bmatrix} \quad (d) \begin{bmatrix} 136 & -48 \\ -48 & 164 \end{bmatrix}$$

2. Find the characteristic polynomial and the eigenvalues and eigenvectors of the following matrices:

$$(a) \begin{bmatrix} 7 & 9 \\ -6 & -8 \end{bmatrix} \quad (b) \begin{bmatrix} 2 & 6 \\ 1 & 3 \end{bmatrix} \quad (c) \begin{bmatrix} 2.2 & 0.6 \\ -0.4 & 0.8 \end{bmatrix} \quad (d) \begin{bmatrix} 32 & 45 \\ -18 & -25 \end{bmatrix}$$

3. Find the characteristic polynomial and the eigenvalues and eigenvectors of the following matrices:

$$(a) \begin{bmatrix} 1 & 0 & 1 \\ 0 & 3 & -2 \\ 0 & 0 & 2 \end{bmatrix} \quad (b) \begin{bmatrix} 1 & 0 & -\frac{1}{3} \\ 0 & 1 & \frac{2}{3} \\ -1 & 1 & 1 \end{bmatrix} \quad (c) \begin{bmatrix} -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{6} \\ -1 & 0 & \frac{1}{3} \\ -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

4. Prove that a square matrix and its transpose have the same characteristic polynomial, and therefore the same set of eigenvalues.
5. Assume that A is a 3×3 matrix with the given eigenvalues. Decide to which eigenvalue Power Iteration will converge, and determine the convergence rate constant S . (a) $\{3, 1, 4\}$ (b) $\{3, 1, -4\}$ (c) $\{-1, 2, 4\}$ (d) $\{1, 9, 10\}$
6. Assume that A is a 3×3 matrix with the given eigenvalues. Decide to which eigenvalue Power Iteration will converge, and determine the convergence rate constant S . (a) $\{1, 2, 7\}$ (b) $\{1, 1, -4\}$ (c) $\{0, -2, 5\}$ (d) $\{8, -9, 10\}$
7. Assume that A is a 3×3 matrix with the given eigenvalues. Decide to which eigenvalue Inverse Power Iteration with the given shift s will converge, and determine the convergence rate constant S . (a) $\{3, 1, 4\}, s = 0$ (b) $\{3, 1, -4\}, s = 0$ (c) $\{-1, 2, 4\}, s = 0$ (d) $\{1, 9, 10\}, s = 6$
8. Assume that A is a 3×3 matrix with the given eigenvalues. Decide to which eigenvalue Inverse Power Iteration with the given shift s will converge, and determine the convergence rate constant S . (a) $\{3, 1, 4\}, s = 5$ (b) $\{3, 1, -4\}, s = 4$ (c) $\{-1, 2, 4\}, s = 1$ (d) $\{1, 9, 10\}, s = 8$
9. Let $A = \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}$. (a) Find all eigenvalues and eigenvectors of A . (b) Apply three steps of Power Iteration with initial vector $x_0 = (1, 0)$. At each step, approximate the eigenvalue by the current Rayleigh quotient. (c) Predict the result of applying Inverse Power Iteration with shift $s = 0$ (d) with shift $s = 3$.
10. Let $A = \begin{bmatrix} -2 & 1 \\ 3 & 0 \end{bmatrix}$. Carry out the steps of Exercise 9 for this matrix.
11. If A is a 6×6 matrix with eigenvalues $-6, -3, 1, 2, 5, 7$, which eigenvalue of A will the following algorithms find? (a) Power Iteration (b) Inverse Power Iteration with shift $s = 4$ (c) Find the linear convergence rates of the two computations. Which converges faster?

12.1 Computer Problems

- Using the supplied code (or code of your own) for the Power Iteration method, find the dominant eigenvector of A , and estimate the dominant eigenvalue by calculating a Rayleigh quotient. Compare your conclusions with the corresponding part of Exercise 5.

$$\begin{array}{ll}
 \text{(a)} \quad \begin{bmatrix} 10 & -12 & -6 \\ 5 & -5 & -4 \\ -1 & 0 & 3 \end{bmatrix} & \text{(b)} \quad \begin{bmatrix} -14 & 20 & 10 \\ -19 & 27 & 12 \\ 23 & -32 & -13 \end{bmatrix} \\
 \text{(c)} \quad \begin{bmatrix} 8 & -8 & -4 \\ 12 & -15 & -7 \\ -18 & 26 & 12 \end{bmatrix} & \text{(d)} \quad \begin{bmatrix} 12 & -4 & -2 \\ 19 & -19 & -10 \\ -35 & 52 & 27 \end{bmatrix}
 \end{array}$$

- Using the supplied code (or code of your own) for the Inverse Power Iteration method, verify your conclusions from Exercise 7, using the appropriate matrix from Computer Problem 1.
- For the Inverse Power Iteration method, verify your conclusions from Exercise 8, using the appropriate matrix from Computer Problem 1.
- Apply Rayleigh Quotient Iteration to the matrices in Computer Problem 1. Try different starting vectors until all three eigenvalues are found.

12.2 QR ALGORITHM

The goal of this section is to develop methods for finding all eigenvalues at once. We begin with a method that works for symmetric matrices, and later supplement it to work in general. Symmetric matrices are easiest to handle because their eigenvalues are real and their unit eigenvectors form an orthonormal basis of R^m (see Appendix A). This motivates applying Power Iteration with m vectors in parallel, where we actively work at keeping the vectors orthogonal to one another.

12.2.1 Simultaneous iteration

Assume that we begin with m pairwise orthogonal initial vectors v_1, \dots, v_m . After one step of Power Iteration applied to each vector, Av_1, \dots, Av_m are no longer guaranteed to be orthogonal to one another. In fact, under further multiplications by A , they all would prefer to converge to the dominant eigenvector, according to Theorem 12.2.

To avoid this, we re-orthogonalize the set of m vectors at each step. The simultaneous multiplication by A of the m vectors is efficiently written as the matrix product

$$A[v_1 | \dots | v_m].$$

As we found in Chapter 4, the orthogonalization step can be viewed as factoring the resulting product as QR . If the elementary basis vectors are used as initial vectors, then the first step of Power Iteration followed by re-orthogonalization is $AI = \bar{Q}_1 R_1$, or

$$\left[A \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \mid A \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \mid \dots \mid A \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \right] = [\bar{q}_1^1 \mid \dots \mid \bar{q}_m^1] \begin{bmatrix} r_{11}^1 & r_{12}^1 & \cdots & r_{1m}^1 \\ & r_{22}^1 & & \vdots \\ & & \ddots & \vdots \\ & & & r_{mm}^1 \end{bmatrix}. \quad (12.5)$$

Appendix A: Matrix Algebra

We begin with a short review of the basic definitions in matrix algebra.

A.1 MATRIX FUNDAMENTALS

A **vector** is an array of numbers

$$u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}.$$

If the list contains n numbers, it is called an n -dimensional vector. We will often make a distinction between the foregoing vertically arranged array, or **column vector**, and a horizontally arranged array

$$u = [u_1, \dots, u_n]$$

called a **row vector**. An $m \times n$ **matrix** is an $m \times n$ array of numbers having the form

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

Each (horizontal) row of A can be considered as a row vector of A , and each (vertical) column as a column vector.

Matrix–vector multiplication makes a vector out of a matrix and a vector. The matrix–vector product is defined as

$$Au = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} a_{11}u_1 + a_{12}u_2 + \cdots + a_{1n}u_n \\ \vdots \\ a_{m1}u_1 + a_{m2}u_2 + \cdots + a_{mn}u_n \end{bmatrix}. \quad (\text{A.1})$$

Note that in order to multiply an $m \times n$ matrix by a d -dimensional vector, it is required that $n = d$.

In matrix–matrix multiplication, an $m \times n$ matrix is multiplied by an $n \times p$ matrix to yield an $m \times p$ matrix as the product. Multiplying matrices can be expressed in terms of matrix–vector multiplication. Let C be an $n \times p$ matrix written in terms of its column vectors

$$C = [c_1 \mid \cdots \mid c_p].$$

Then the matrix–matrix product of A and C is

$$AC = A [c_1 \mid \cdots \mid c_p] = [Ac_1 \mid \cdots \mid Ac_p].$$

A system of m linear equations in n unknowns can be written in matrix form as

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix},$$

which we call a **matrix equation**.

The $n \times n$ **identity matrix** I_n is the matrix with $I_{ii} = 1$ for $1 \leq i \leq n$ and $I_{ij} = 0$ for $i \neq j$. The identity matrix serves as the identity for the operation of matrix multiplication, as $AI_n = I_nA = A$ for each $n \times n$ matrix A . For an $n \times n$ matrix A , the **inverse** A^{-1} of A is an $n \times n$ matrix satisfying $AA^{-1} = A^{-1}A = I_n$. If A has an inverse, it is called **invertible**; a noninvertible matrix is called **singular**.

The **transpose** of an $m \times n$ matrix A is the matrix A^T whose entries are $A_{ij}^T = A_{ji}$. The rule for the transpose of a product is $(AB)^T = B^T A^T$.

There are two important ways to multiply two vectors together. Let

$$u = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} \text{ and } v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}.$$

The **inner product** $u^T v$ transposes u to a row vector; then ordinary matrix multiplication gives

$$u^T v = u_1 v_1 + \cdots + u_n v_n.$$

Thus the product of $1 \times n$ by $n \times 1$ yields a 1×1 matrix, or real number, as the result. Two column vectors are **orthogonal** if $u^T v = 0$. The **outer product** uv^T multiplies an $n \times 1$ column by a $1 \times n$ row. Ordinary matrix multiplication gives an $n \times n$ matrix result

$$uv^T = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \cdots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \cdots & u_2 v_n \\ \vdots & & & \vdots \\ u_n v_1 & \cdots & \cdots & u_n v_n \end{bmatrix}.$$

An outer product is a rank-one matrix.

Each matrix product AB can be represented as the sum of outer products of the columns of A with the rows of B . More precisely,

Outer product sum rule

Let A and B be $m \times p$ and $p \times n$ matrices, respectively. Then

$$AB = \sum_{i=1}^p a_i b_i^T$$

where a_i is the i th column of A and b_i^T is the i th row of B .

The case $n = 1$ is sometimes called the “alternate form of matrix-vector multiplication.” For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} -3 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix} \begin{bmatrix} -3 \end{bmatrix} + \begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} + \begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix}$$

illustrates why, when viewed as a linear transformation, the range of a matrix is equivalent to its column space.

Because of the high computational complexity of computing the matrix inverse, it is avoided or minimized whenever possible. One trick that helps is the Sherman–Morrison formula. Assume that the inverse of an $n \times n$ matrix A is already known, and that the inverse of the modified matrix $A + uv^T$ is needed, where u and v are n -vectors.

THEOREM A.1 (Sherman–Morrison Formula) If $v^T A^{-1} u \neq -1$, then $A + uv^T$ is invertible and

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

The Sherman–Morrison formula is proved by multiplying $A + uv^T$ times the expression in the formula. The matrix $A + uv^T$ is called a **rank-one update** of A , since uv^T is a rank-one matrix. (See the discussion of Broyden’s Method in Chapter 2 for an important application of the Sherman–Morrison formula. Elementary facts about matrices can be found in linear algebra texts such as Strang [2005] and Lay [2005].)

A.2 BLOCK MULTIPLICATION

Matrix multiplication can be done blockwise, a fact that will be very helpful in Chapter 12. If two matrices are divided into blocks whose sizes are compatible with matrix multiplication, then the matrix product can be carried out by matrix multiplication of the blocks. For example, the product of two 3×3 matrices can be carried out in the following blocks:

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

Here A_{11} and B_{11} are 1×1 matrices, A_{12} and B_{12} are 1×2 matrices, and so forth. For example,

$$\begin{aligned} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 3 \\ 2 & 2 & 4 \end{bmatrix} \begin{bmatrix} 2 & 4 & 1 \\ 1 & 0 & 1 \\ 3 & 1 & 2 \end{bmatrix} &= \begin{bmatrix} 1 \cdot 2 + [2 \ 3] \begin{bmatrix} 1 \\ 3 \end{bmatrix} & 1[4 \ 1] + [2 \ 3] \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \\ [0]2 + \begin{bmatrix} 1 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} & [0] [4 \ 1] + \begin{bmatrix} 1 & 3 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \\ [2]2 + \begin{bmatrix} 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} & [2] [4 \ 1] + \begin{bmatrix} 2 & 4 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} 13 & 7 & 9 \\ 10 & 3 & 7 \\ 18 & 12 & 12 \end{bmatrix}. \end{aligned}$$

Doing the multiplication blockwise gives the same result as doing it without blocks. This alternative way of looking at matrix multiplication is not meant to reduce computations, but to assist with bookkeeping, especially with eigenvalue computations in Chapter 12.

The only necessary compatibility required of the blocks is that the column groupings of A must exactly match the row groupings of B . In the preceding example, the first column of A is in one group, and the last two columns are in another. For matrix B , the first row is

in one group and the last two *rows* are in another. As another example, we can multiply the 3×5 matrix A and the 5×2 matrix B in the following blocks:

$$\begin{aligned}
 & \left[\begin{array}{c|c|c|c|c} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{array} \right] \left[\begin{array}{c|c} x & x \\ x & x \\ x & x \\ x & x \\ x & x \end{array} \right] \\
 &= \left[\begin{array}{c|c|c} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{array} \right] \left[\begin{array}{c|c} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{array} \right] \\
 &= \left[\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} & A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} \\ A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31} & A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} \end{array} \right]
 \end{aligned}$$

In this case, the three groups of columns of A are matched by the three groups of rows of B . On the other hand, the groupings of rows of A and columns of B do not need to match; they may be done arbitrarily.

A.3 EIGENVALUES AND EIGENVECTORS

We begin with a short review of the basic concepts of eigenvalues and eigenvectors.

DEFINITION A.2 Let A be an $m \times m$ matrix and x a nonzero m -dimensional real or complex vector. If $Ax = \lambda x$ for some real or complex number λ , then λ is called an **eigenvalue** of A and x is the corresponding **eigenvector**. \square

For example, the matrix $A = \begin{bmatrix} 1 & 3 \\ 2 & 2 \end{bmatrix}$ has an eigenvector $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$, and corresponding eigenvalue 4.

Eigenvalues are the roots λ of the **characteristic polynomial** $\det(A - \lambda I)$. If λ is an eigenvalue of A , then any nonzero vector in the nullspace of $A - \lambda I$ is an eigenvector corresponding to λ . For this example,

$$\det(A - \lambda I) = \det \begin{bmatrix} 1 - \lambda & 3 \\ 2 & 2 - \lambda \end{bmatrix} = (\lambda - 1)(\lambda - 2) - 6 = (\lambda - 4)(\lambda + 1), \quad (\text{A.2})$$

so the eigenvalues are $\lambda = 4, -1$. The eigenvectors corresponding to $\lambda = 4$ are found in the nullspace of

$$A - 4I = \begin{bmatrix} -3 & 3 \\ 2 & -2 \end{bmatrix} \quad (\text{A.3})$$

and so consist of all nonzero multiples of $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Similarly, the eigenvectors corresponding to $\lambda = -1$ are all nonzero multiples of $\begin{bmatrix} 3 \\ -2 \end{bmatrix}$.

DEFINITION A.3 The $m \times m$ matrices A_1 and A_2 are **similar**, denoted $A_1 \sim A_2$, if there exists an invertible $m \times m$ matrix S such that $A_1 = SA_2S^{-1}$. \square

Similar matrices have identical eigenvalues, because their characteristic polynomials are identical:

$$A_1 - \lambda I = SA_2S^{-1} - \lambda I = S(A_2 - \lambda I)S^{-1} \quad (\text{A.4})$$

implies that

$$\det(A_1 - \lambda I) = (\det S) \det(A_2 - \lambda I) \det S^{-1} = \det(A_2 - \lambda I). \quad (\text{A.5})$$

If a matrix A has eigenvectors that form a basis for R^m , then A is similar to a diagonal matrix, and A is called **diagonalizable**. In fact, assume that $Ax_i = \lambda_i x_i$ for $i = 1, \dots, m$, and define the matrix

$$S = [x_1 \quad \cdots \quad x_m].$$

Then you can check that the matrix equation

$$AS = S \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_m \end{bmatrix} \quad (\text{A.6})$$

holds. The matrix S is invertible because its columns span R^m . Therefore, A is similar to the diagonal matrix containing its eigenvalues.

Not all matrices are diagonalizable, even in the 2×2 case. In fact, all 2×2 matrices are similar to one of the following three types:

$$\begin{aligned} A_1 &= \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \\ A_2 &= \begin{bmatrix} a & 1 \\ 0 & a \end{bmatrix} \\ A_3 &= \begin{bmatrix} a & -b \\ b & a \end{bmatrix}. \end{aligned}$$

Remember that eigenvalues are identical for similar matrices. A matrix is similar to a matrix of form A_1 if there are two eigenvectors that span R^2 ; a matrix is similar to a matrix of form A_2 if there is a repeated eigenvalue with only one dimensional space of eigenvectors; and to A_3 if it has a complex pair of eigenvalues.

A.4 SYMMETRIC MATRICES

For a symmetric matrix, all eigenvectors are orthogonal to one another, and together they span the underlying space. In other words, symmetric matrices always have an orthonormal basis of eigenvectors.

DEFINITION A.4 A set of vectors is **orthonormal** if the elements of the set are unit vectors that are pairwise orthogonal. \square

In terms of dot products, orthonormality of the set $\{w_1, \dots, w_m\}$ means $w_i^T w_j = 0$ if $i \neq j$, and $w_i^T w_i = 1$, for $1 \leq i, j \leq m$. For example, the sets $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$ and $\{(\sqrt{2}/2, \sqrt{2}/2), (\sqrt{2}/2, -\sqrt{2}/2)\}$ are orthonormal sets.

THEOREM A.5 Assume that A is a symmetric $m \times m$ matrix with real entries. Then the eigenvalues are real numbers, and the set of unit eigenvectors of A is an orthonormal set $\{w_1, \dots, w_m\}$ that forms a basis of R^m . ■

► **EXAMPLE A.1** Find the eigenvalues and eigenvectors of

$$A = \begin{bmatrix} 0 & 1 \\ 1 & \frac{3}{2} \end{bmatrix}. \quad (\text{A.7})$$

Calculating as before, the eigenvalue/eigenvector pairs are $2, (1, 2)^T$ and $-1/2, (-2, 1)^T$. Note that as the theorem promises, the eigenvectors are orthogonal. The corresponding orthonormal basis of unit eigenvectors is

$$\left\{ \begin{bmatrix} \frac{1}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} \end{bmatrix}, \begin{bmatrix} -\frac{2}{\sqrt{5}} \\ \frac{1}{\sqrt{5}} \end{bmatrix} \right\}. \quad \triangleleft$$

The following theorem will be helpful for studying iterative methods in Chapter 2:

DEFINITION A.6 The **spectral radius** $\rho(A)$ of a square matrix A is the maximum magnitude of its eigenvalues. □

THEOREM A.7 If the $n \times n$ matrix A has spectral radius $\rho(A) < 1$, and b is arbitrary, then, for any vector x_0 , the iteration $x_{k+1} = Ax_k + b$ converges. In fact, there exists a unique x_* such that $\lim_{k \rightarrow \infty} x_k = x_*$ and $x_* = Ax_* + b$. ■

Moreover, if $b = 0$, then x_* is either the zero vector or an eigenvector of A with eigenvalue 1. The latter is ruled out because of the spectral radius, leading to the following fact that is useful in Chapter 8:

COROLLARY A.8 If the $n \times n$ matrix A has spectral radius $\rho(A) < 1$, then, for any initial vector x_0 , the iteration $x_{k+1} = Ax_k$ converges to 0. ■

A.5 VECTOR CALCULUS

In this section, the derivatives of scalar-valued and vector-valued functions are defined, and the product rules involving them are collected for later use.

Let $f(x_1, \dots, x_n)$ be a scalar-valued function of n variables. The **gradient** of f is the vector-valued function

$$\nabla f(x_1, \dots, x_n) = [f_{x_1}, \dots, f_{x_n}],$$

where the subscripts denote partial derivatives of f with respect to that variable.

Let

$$F(x_1, \dots, x_n) = \begin{bmatrix} f_1(x_1, \dots, x_n) \\ \vdots \\ f_n(x_1, \dots, x_n) \end{bmatrix}$$

be a vector-valued function of n variables. The **Jacobian** of F is the matrix

$$DF(x_1, \dots, x_n) = \begin{bmatrix} \nabla f_1 \\ \vdots \\ \nabla f_n \end{bmatrix}.$$

Now we can state the product rules for two typical products in matrix algebra. Both have straightforward proofs when they are written in components and the single-variable product rule is applied. Let $u(x_1, \dots, x_n)$ and $v(x_1, \dots, x_n)$ be vector-valued functions, and let $A(x_1, \dots, x_n)$ be an $n \times n$ matrix function. The dot product $u^T v$ is a scalar function. The first formula shows how to take its gradient. The matrix vector product Av is a vector whose Jacobian is expressed in the second rule.

Vector dot product rule

$$\nabla(u^T v) = v^T Du + u^T Dv$$

Matrix/vector product rule

$$D(Av) = A \cdot Dv + \sum_{i=1}^n v_i Da_i,$$

where a_i denotes the i th column of A .

Appendix B: Introduction to MATLAB

MATLAB is a general-purpose computing environment that is ideally suited for implementing mathematical and numerical methods. It is used as a high-powered calculator for small problems and as a full-featured programming language for large problems. A helpful feature of MATLAB is its long list of high-quality library functions that can make complicated calculations short, precise, and easy to write in high-level code.

This section contains a brief introduction to MATLAB's commands and features. Much more detailed accounts can be found in MATLAB's help facilities, the *MATLAB User's Guide*, in books such as Sigmon [2002], Hahn [2002], and on websites devoted to the package.

B.1 STARTING MATLAB

On PC-based systems, MATLAB is started by clicking the appropriate icon and ended by clicking on File/Exit. On Unix-based systems, type MATLAB at the system prompt:

```
$ matlab
```

Then type

```
>> exit
```

to exit.

Type the command

```
>> a=5
```

followed by the return key. MATLAB will echo the information back to you. Type the additional commands

```
>> b=3
```

```
>> c=a+b
```

```
>> c=a*b
```

```
>> d=log(c)
```

```
>> who
```

to get an idea of how MATLAB works. You may include a semicolon after a statement to suppress echoing of the value. The `who` command gives a list of all variables you have defined.

MATLAB has an extensive online help facility. Type `help log` for information on the `log` command. The PC version of MATLAB has a Help menu that contains descriptions and usage suggestions on all commands.

To erase the value of the variable *a*, type `clear a`. Typing `clear` will erase all previously defined variables. To recover a previous command, use the up cursor key. If you run out of room on the current command line, end the line with three periods and a return; then resume typing on the next line.

To save values of variables for your next login, type `save`, then `load` on your next login to MATLAB. For a transcript of part or all of the MATLAB session, type `diary filename` to start logging, and `diary off` to end. Use a filename of your choice for *filename*. This is helpful for submitting your work for an assignment. The `diary` command produces a file that can be viewed or printed once your MATLAB session is over.

MATLAB normally performs all computations in IEEE double precision, about 16 decimal digits of accuracy. The numeric display format can be changed with the `format`

statement. Typing `format long` will change the way numbers are displayed until further notice. For example, the number $1/3$ will be displayed differently depending on the current format:

```
format short      0.3333
format short e    3.3333E-001
format long       0.33333333333333
format long e     3.33333333333333E-001
format bank       0.33
format hex        3fd5555555555555
```

More control over formatting output is given by the `fprintf` command. The commands

```
>> x=0:0.1:1;
>> y=x.^2;
>> fprintf('%8.5f %8.5f \n', [x;y])
```

print the table

```
0.00000 0.00000
0.10000 0.01000
0.20000 0.04000
0.30000 0.09000
0.40000 0.16000
0.50000 0.25000
0.60000 0.36000
0.70000 0.49000
0.80000 0.64000
0.90000 0.81000
1.00000 1.00000
```

B.2 GRAPHICS

To plot data, express the data as vectors in the X and Y directions. For example, the commands

```
>> a=[0.0 0.4 0.8 1.2 1.6 2.0];
>> b=sin(a);
>> plot(a,b)
```

will draw a piecewise-linear approximation to the graph of $y = \sin x$ on $0 \leq x \leq 2$, as shown in Figure B.1(a). In this case, `a` and `b` are 6-dimensional vectors, or 6-element arrays. The font of the axis numbers can be set to 16-point, for example, by the command `set(gca, 'FontSize', 16)`. A shorter way to define the vector `a` is the command

```
>> a=0:0.4:2;
```

This command defines `a` to be a vector whose entries begin at 0, increment by 0.4, and end at 2, identical to the previous longer definition. A more accurate version of one entire cycle of the sine curve results from

```
>> a=0:0.02:2*pi;
>> b=sin(a);
>> plot(a,b)
```

and is shown in Figure B.1(b).

To draw the graph of $y = x^2$ on $0 \leq x \leq 2$, one could use

```
>> a=0:0.02:2;
>> b=a.^2;
>> plot(a,b)
```

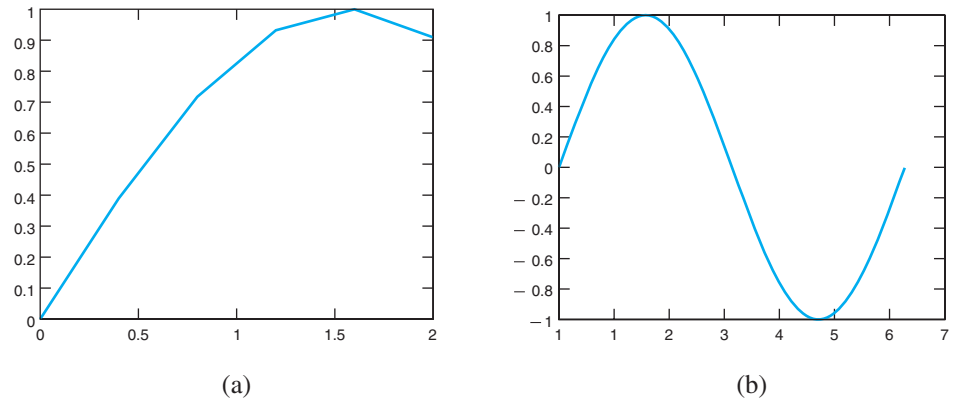


Figure B.1 MATLAB figures. (a) Piecewise-linear plot of $f(x) = \sin x$, with x increment of 0.4. (b) Another piecewise plot looks smooth because the x increment is 0.02.

The “.” character preceding the power operator may be unexpected. It causes the power operator to be vectorized, that is, to square each entry of the vector a . As we will see in the next section, MATLAB treats every variable as a matrix. Omitting the period in this instance would mean multiplying the 101×1 matrix a by itself, under the rules of matrix multiplication, which is impossible. If you ask MATLAB to do this, it will complain. In general, MATLAB interprets an operation preceded by a period to mean that the operation should be applied entry-wise, not as matrix multiplication.

There are more advanced techniques for plotting graphs. MATLAB will choose axis scaling automatically if it is not specified, as in Figure B.1. To choose the axis scaling manually, use the `axis` command. For example, following a plot with the command

```
>> v=[-1 1 0 10]; axis(v)
```

sets the graphing window to $[-1, 1] \times [0, 10]$. The `grid` command draws a grid behind the plot.

Use the command `plot(x1,y1,x2,y2,x3,y3)` to plot three curves in the same graph window, where x_i , y_i are pairs of vectors of the same lengths. Type `help plot` to see the choices of solid, dotted, and dashed line types and various symbol types (circles, dots, triangles, squares, etc.) for plots. Semilog plots are available through the `semilogy` and `semilogx` commands.

The subplot command splits the graph window into multiple parts. The statement `subplot(abc)` breaks the window into an $a \times b$ grid and uses the c box for the plot. For example,

```
>> subplot(121),plot(x,y)
>> subplot(122),plot(x,z)
```

plots the first graph on the left side of the screen and the second on the right. The `figure` command opens up new plot windows and moves among them, if you need to view several different plots at once.

Three-dimensional surface plots are drawn with the command `mesh`. For example, the function $z = \sin(x^2 + y^2)$ on the domain $[-1, 1] \times [-2, 2]$ can be graphed by

```
>> [x,y]=meshgrid(-1:0.1:1,-2:0.1:2);
>> z=sin(x.^2+y.^2);
>> mesh(x,y,z)
```

The vector x created by `meshgrid` is 41 rows of the 21-vector $-1:0.1:1$, and similarly, y is 21 columns of the column vector $-2:0.1:2$. The graph produced by this code is shown in Figure B.2. Replacing `mesh` with `surf` plots a colored surface over the mesh.

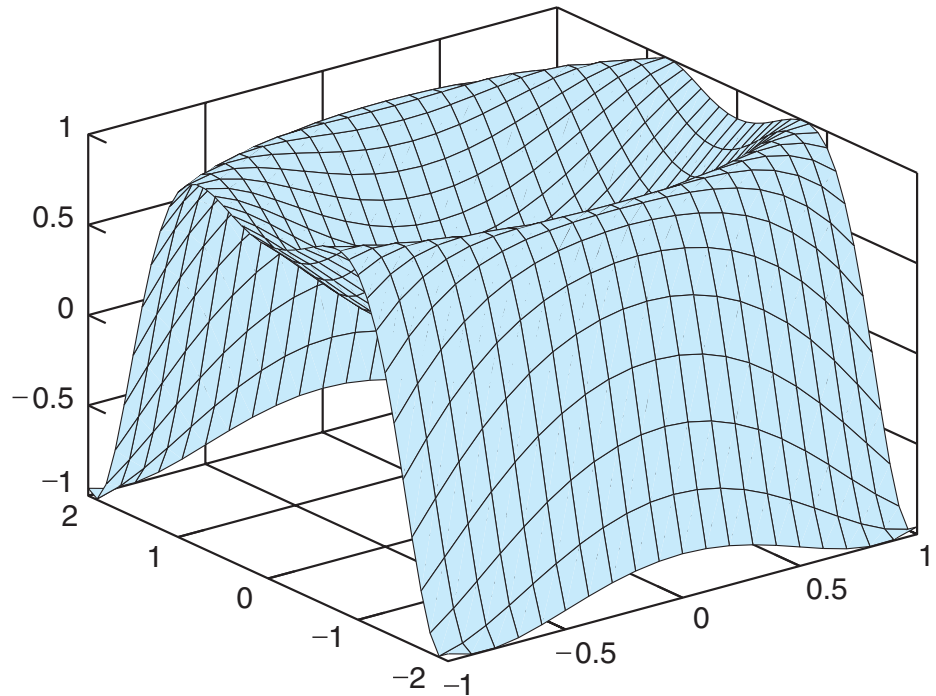


Figure B.2 Three-dimensional MATLAB plot. The `mesh` command is used to plot surfaces.

B.3 PROGRAMMING IN MATLAB

More sophisticated results can be achieved by writing programs in the MATLAB language. A **script file** is a file containing a list of MATLAB commands. The filename of a script file has a suffix of `.m`, so such files are sometimes called **m-files**. For example, you might use your favorite editor, or the MATLAB editor if available, to create the file `cubrt.m`, containing the following lines:

```
% The program cubrt.m finds a cube root by iteration
y=1;
n=15;
z=input('Enter z:');
for i = 1:n
    y = 2*y/3 + z/(3*y^2)
end
```

To run the program, type `cubrt` at the MATLAB prompt. The reason that this code converges to the cube root will become evident from our study of Newton's Method in Chapter 1. Notice that the semicolon was dropped from the line that defines the new `y` by iteration. This allows you to see the progression of approximants as they approach the cube root.

With the graphics ability of MATLAB, we can analyze the data from the cube root algorithm. Consider the program `cubrt1.m`:

```
% The program cubrt1.m finds cube roots and displays its progress
y(1)=1;
n=15;
z=input('Enter z:');
for i = 1:n-1
    y(i+1) = 2*y(i)/3 + z/(3*y(i)^2);
end
```

```

plot(1:n,y)
title('Iterative method for cube roots')
xlabel('Iteration number')
ylabel('Approximate cube root')

```

Run the foregoing program with $z = 64$. When finished, type the commands

```

>> e=y-4;
>> plot(1:n,e)
>> semilogy(1:n,e)

```

The first command subtracts the correct cube root 4 from each entry of the vector y . This remainder is the error e at each step of the iteration. The second command plots the error, and the third plots the error in a semilog plot, using logarithmic units in the y -direction.

Creating a script file to hold MATLAB code is preferred if the calculation will take more than a few lines. A script file can call other script files, including itself. (Typing `<ctrl>-C` will usually abort runaway MATLAB processes.)

B.4 FLOW CONTROL

The `for` loop was introduced in the previous cube root program. MATLAB has a number of commands to control the flow of a program. A number of these, including `while` loops and `if` and `break` statements, will be familiar to anyone with knowledge of a high-level programming language. For example,

```

n=5;
for i=1:n
    for j=1:n
        a(i,j)=1/(i+j-1);
    end
end
a

```

creates and displays the 5×5 Hilbert matrix. The semicolon avoids repeated printing of partial results, and the final `a` displays the final result. Note that each `for` must be matched with an `end`. It is a good idea, though not required by MATLAB, to indent loops for greater readability.

The `while` command works similarly:

```

n=5;i=1;
while i<=n
    j=1;
    while j<=n
        a(i,j)=1/(i+j-1);
        j=j+1;
    end
    i=i+1;
end
a

```

This produces the same result as the double `for` loop.

The `if` statement is used to make flow decisions, and the `break` command provides an exit jump out of the next inner loop. Both are illustrated as follows:

```

% To compute the nth derivative of sin(x) at x=0
n=input('Enter n, negative number to quit:')

```

```

if n<=0,break,end
r=rem(n,4) % rem is the remainder function
if r==0
    y=0
elseif r==1
    y=1
elseif r==2
    y=0
else
    y=-1
end
y

```

The logical operators & and | stand for AND, OR, respectively. The `error` command stops execution of the m-file and reports information to the user.

B.5 FUNCTIONS

In addition to built-in library functions like `sin` and `exp`, MATLAB allows the definition of user-defined functions. The command

```
>> f=@(x) exp(sin(2*x))
```

creates a function with input x and output $f(x) = e^{\sin 2x}$. After defining f as above, the command

```
>> f(0)
```

returns the correct result $e^{\sin 2(0)} = 1$. Moreover, the definition with @ assigns a **function handle** to f that can be passed to another function. If we create another function

```
>> firstderiv=@(f,x,h) (f(x+h)-f(x-h))/(2*h)
```

with three inputs f, x, h , the command

```
>> firstderiv(f,0,0.0001)
```

returns an approximation to the derivative at 0. Here, we have used the user-defined function handle f as an input to the user-defined MATLAB function `firstderiv`.

A MATLAB function may have several inputs and several outputs. An example of a vector-valued function of several variables having three inputs and three outputs is the following function that converts rectangular to spherical coordinates:

```
>> rec2sph=@(x,y,z) [sqrt(x^2+y^2+z^2) acos(z/sqrt(x^2+y^2+z^2)) ...
    atan2(y,x)]
```

This method of defining functions is useful when the function can be defined on one line. For more complicated examples, MATLAB allows a second way to define a function, through a special m-file. The syntax of the first line must be adhered to, as in the following example, where the filename is `cubrtf.m`:

```

function y=cubrtf(x)
% Approximates the cube root of x
% Input real number x, output its cube root
y=1;
n=15;
for i = 1:n
    y = 2*y/3 + x/(3*y^2)
end

```


Here, we have transferred the script-file version of the cube root approximator to a MATLAB function. The function can be evaluated by

```
>> c=cubrtf(8)
```

Note that a MATLAB function differs from a script m-file in the first line. The filename, with the .m omitted, should agree with the function name in the first line. Variables in a function file are local by default, but can be made global with the `global` command.

Combining the two above approaches, a previously defined MATLAB function, such as an m-file function, can be assigned a function handle by prefixing with the `@` sign. The function handle can then be passed into another function. For example,

```
>> firstderiv(@cubrtf,1,0.0001)
```

returns the approximation 0.3333 for the derivative of $x^{1/3}$ at $x = 1$.

A more complicated function can use several variables as inputs and several as outputs. For example, here is a function that calls the existing MATLAB functions `mean` and `std` and collects both in an array:

```
function [m,sigma]=stat(x)
% Returns sample mean and standard deviation of input vector x
m=mean(x);
sigma=std(x);
```

If this file `stat.m` resides in your MATLAB path, typing `stat(x)`, where x is a vector, will return the mean and standard deviation of the entries of the vector.

The `nargin` command provides the number of input arguments to a function. With this command, the work of a function can change, depending on how many arguments are presented to it. An example of `nargin` is given in Program 0.1 on nested multiplication.

An example of a piecewise-defined function is

$$h(x) = \begin{cases} x + 2 & \text{for } x \leq -1 \\ 1 & \text{for } -1 < x \leq 0 \\ \cos x & \text{for } x > 0. \end{cases}$$

The function $h(x)$ can be represented by the creating the MATLAB function file `h.m` containing

```
function y=h(x)
p1=(x<=-1);
p2=(x>-1).*(x<=0);
p3=(x>0);
y=p1.*(x+2)+p2.*1+p3.*cos(x);
```

Here we are making use of Boolean evaluation of the conditional expressions as 1 if true and 0 if false. We are also using the period preceding arithmetic operations to vectorize them, allowing the input x to be a vector of numbers. Now h can be passed to other MATLAB functions via its function handle `@h`. For example,

```
>> ezplot(@h,[-3 3])
```

plots the piecewise function h , and

```
>> fzero(@h,1)
```

finds a root of $h(x)$ near 1. Should the result of

```
>> firstderiv(@h,-1,0.0001)
```

be trusted?

B.6 MATRIX OPERATIONS

The key to MATLAB's power and versatility is the sophistication of its variables' data structure. Each variable in MATLAB is an $m \times n$ matrix of double precision floating point numbers. A scalar is simply the special case of a 1×1 matrix. The syntax

```
>> A=[1 2 3
4 5 6]
```

or

```
>> A=[1 2 3;4 5 6]
```

defines a 2×3 matrix A . The command $B=A'$ creates a 3×2 matrix B that is the transpose of A . Matrices of the same size can be added and subtracted with the $+$ and $-$ operators. The command $\text{size}(A)$ returns the dimensions of the matrix A , and $\text{length}(A)$ returns the maximum of the two dimensions.

MATLAB provides many commands that allow matrices to be easily built. For example, $\text{zeros}(m,n)$ produces a matrix full of zeros of size $m \times n$. If A is a matrix, then $\text{zeros}(\text{size}(A))$ produces a matrix of zeros of the same size as A . The commands $\text{ones}(m,n)$ and $\text{eye}(m,n)$ (for the identity matrix) work essentially the same way. For example,

```
>> A=[eye(2) zeros(2,2);zeros(2,2) eye(2)]
```

is a convoluted, but accurate way to construct the 4×4 identity matrix.

The colon operator can be used to extract a submatrix from a matrix. For example,

```
>> b=A(1:3,2)
```

assigns to b the first three entries of the second column of A . The command

```
>> b=A(:,2)
```

assigns to b the entire second column of A , and

```
>> B=A(:,1:3)
```

assigns to B the submatrix consisting of the first three columns of A .

The $m \times n$ matrix A and the $n \times p$ matrix B can be multiplied by the command $C=A*B$. If the matrices have inappropriate sizes, MATLAB will refuse to do the operation and return an error message.

B.7 ANIMATION AND MOVIES

The field of differential equations includes the study of dynamic systems, or “things that move.” MATLAB makes animation easy, and these aspects are exploited in Chapter 6 to follow solutions that are changing with time.

The sample MATLAB program `bounce.m` given next shows a tennis ball bouncing from wall to wall in a unit square. The first `set` command sets up parameters of the current figure (`gca`), including the axis limits $0 \leq x, y \leq 1$. The `cla` command clears the figure window, and `axis square` equalizes the units in the x and y directions.

Next, the `line` command is used to define a line object called `ball`, along with its properties. The `erase` parameter set to `xor` means that each time the ball is drawn, its previous position is erased. The four `if` statements in the `while` loop cause the ball to reverse velocity when it hits one of the four walls. The loop also contains a `set` command that updates the current x and y coordinates of the line object `ball`, by setting its `xdata`

and `ydata` attributes, respectively. The `drawnow` command draws all defined objects to the current figure window. The speed of the moving ball can be adjusted with the `pause` command and through the step sizes `hx0` and `hy0`. The while loop is infinite and can be interrupted by (cntl)-C. Here is the program in its entirety:

```
%bounce.m
% Illustrates Matlab animation using the drawnow command
% Usage: Save this file in bounce.m, then type "bounce"
set(gca,'XLim',[0 1],'YLim',[0 1],'Drawmode','fast', ...
    'Visible','on');
cla
axis square
ball = line('color','r','Marker','o','MarkerSize',10, ...
    'LineWidth',2,'erase','xor','xdata',[],'ydata',[]);
hx0=.005;hy0=.0039;hx=hx0;hy=hy0;
xl=.02;xr=.98;yb=xl;yt=xr;x=.1;y=.1;
while 1 == 1
    if x < xl
        hx= hx0;
    end
    if x > xr
        hx = -hx0;
    end
    if y < yb
        hy = hy0;
    end
    if y > yt
        hy = -hy0;
    end
    x=x+hx;y=y+hy;
    set(ball,'xdata',x,'ydata',y);drawnow;pause(0.01)
end
```

Using the file `MakeQTMovie.m`, it is straightforward to make QuickTime movies in Matlab. Each frame of the movie will be a single Matlab figure. To begin the process of making a movie, acquire the file `MakeQTMovie.m` from the Internet. This file was written by Malcolm Slaney of Interval Research and is free to download and distribute. Place the file so that it can be found by Matlab, either in your current working directory or your search path. Then the example code segment

```
MakeQTMovie('start','filename.mov')
for i=1:n
    (plot a figure)
    MakeQTMovie('addfigure')
end
MakeQTMovie('finish')
```

will capture the n still figures and place them into a QuickTime movie file named `filename.mov`.

Answers to Selected Exercises

CHAPTER 0

0.1 Exercises

- (a) $P(x) = 1 + x(1 + x(5 + x(1 + x(6))))$, $P(1/3) = 2$.
(b) $P(x) = 1 + x(-5 + x(5 + x(4 + x(-3))))$, $P(1/3) = 0$
(c) $P(x) = 1 + x(0 + x(-1 + x(1 + x(2))))$, $P(1/3) = 77/81$
- $P(x) = 1 + x^2(2 + x^2(-4 + x^2(1)))$, $P(1/2) = 81/64$
- (a) 5 (b) $41/4$
- n multiplications and $2n$ additions

0.1 Computer Problems

- Correct answer from Q is 51.01275208275, error $= 4.76 \times 10^{-12}$

0.2 Exercises

- (a) 1000000 (b) 10001 (c) 1001111 (d) 11100011
- (a) 1010.1 (b) $0.\overline{01}$ (c) $0.\overline{101}$ (d) $1100.\overline{1100}$ (e) $110111.\overline{0110}$ (f) $0.000\overline{11}$
- 11.0010010000111
- (a) 85 (b) $93/8$ (c) $70/3$ (d) $20/3$ (e) $20/7$ (f) $48/7$ (g) $283/120$ (h) 8

0.3 Exercises

- (a) $1.0000\dots0000 \times 2^{-2}$ (b) $1.0101\dots0101 \times 2^{-2}$
(c) $1.0101\dots0101 \times 2^{-1}$ (d) $1.11001100\dots11001101 \times 2^{-1}$
- $1 \leq k \leq 50$
- (a) $2\epsilon_{\text{mach}}$ (b) $4\epsilon_{\text{mach}}$
- (a) 4020000000000000 (b) 4035000000000000 (c) 3fc0000000000000 (d) 3fd5555555555555
(e) 3fe5555555555555 (f) 3fb999999999999a (g) bfb999999999999a (h) bfc999999999999a
- (a) Note that $(7/3 - 4/3) - 1 = \epsilon_{\text{mach}}$ in double precision. (b) No, $(4/3 - 1/3) - 1 = 0$.
- No, associative law fails.
- (a) 2, represented by 010...0 (b) 2^{-511} , represented by 0010...0 (c) 0, represented by 10...0
- (a) 2^{-50} (b) 0 (c) 2^{-50}

0.4 Exercises

- (a) Loss of significance near $x = 2\pi n$, n integer. Rewrite as $-1/(1 + \sec x)$ (b) Loss of significance near $x = 0$. Rewrite as $3 - 3x + x^2$ (c) Loss of significance near $x = 0$. Rewrite as $2x/(x^2 - 1)$
- $x_1 = -(b + \sqrt{b^2 + 4 \times 10^{-12}})/2$, $x_2 = (2 \times 10^{-12})/(b + \sqrt{b^2 + 4 \times 10^{-12}})$

0.4 Computer Problems

1. (a)

x	original	revised
0.100000000000000	-0.49874791371143	-0.49874791371143
0.010000000000000	-0.49998749979096	-0.49998749979166
0.001000000000000	-0.49999987501429	-0.49999987499998
0.000100000000000	-0.49999999362793	-0.49999999875000
0.000010000000000	-0.50000004133685	-0.49999999998750
0.000001000000000	-0.50004445029084	-0.49999999999987
0.000000100000000	-0.51070259132757	-0.50000000000000
0.000000010000000	0	-0.50000000000000
0.000000001000000	0	-0.50000000000000
0.000000000100000	0	-0.50000000000000
0.000000000010000	0	-0.50000000000000
0.000000000001000	0	-0.50000000000000
0.000000000000100	0	-0.50000000000000
0.000000000000010	0	-0.50000000000000
0.000000000000001	0	-0.50000000000000

(b)

x	original	revised
0.100000000000000	2.71000000000000	2.71000000000000
0.010000000000000	2.97010000000001	2.97010000000000
0.001000000000000	2.99700100000000	2.99700100000000
0.000100000000000	2.99970000999905	2.99970001000000
0.000010000000000	2.99997000008379	2.99997000010000
0.000001000000000	2.99999700015263	2.99999700000100
0.000000100000000	2.99999969866072	2.99999970000001
0.000000010000000	2.99999998176759	2.99999997000000
0.000000001000000	2.99999991515421	2.99999999700000
0.000000000100000	3.00000024822111	2.99999999970000

x	original	revised
0.000000000010000	3.00000024822111	2.99999999997000
0.000000000001000	2.99993363483964	2.99999999999700
0.000000000000100	3.00093283556180	2.99999999999970
0.000000000000001	2.99760216648792	2.99999999999997

3. 6.127×10^{-13}

5. 2.23322×10^{-10}

0.5 Exercises

- (a) $f(0)f(1) = -2 < 0$ implies $f(c) = 0$ for some c in $(0, 1)$ by the Intermediate Value Theorem.

(b) $f(0)f(1) = -9 < 0$ implies $f(c) = 0$ for some c in $(0, 1)$ (c) $f(0)f(1/2) = -1/2 < 0$ implies $f(c) = 0$ for some c in $(0, 1/2)$.
- (a) $c = 2/3$ (b) $c = 1/\sqrt{2}$ (c) $c = 1/(e - 1)$
- (a) $P(x) = 1 + x^2 + 1/2x^4$ (b) $P(x) = 1 - 2x^2 + 2/3x^4$ (c) $P(x) = x - x^2/2 + x^3/3 - x^4/4 + x^5/5$

(d) $P(x) = x^2 - x^4/3$
- (a) $P(x) = (x - 1) - (x - 1)^2/2 + (x - 1)^3/3 - (x - 1)^4/4$ (b) $P(0.9) = -0.105358\bar{3}$, $P(1.1) = 0.095308\bar{3}$

(c) error bound = 0.000003387 for $x = 0.9$, 0.000002 for $x = 1.1$ (d) Actual error ≈ 0.00000218 at $x = 0.9$, 0.00000185 at $x = 1.1$
- $\sqrt{1+x} = 1 + x/2 \pm x^2/8$. For $x = 1.02$, $\sqrt{1.02} \approx 1.01 \pm 0.00005$. Actual value is $\sqrt{1.02} = 1.0099505$, error = 0.0000495

CHAPTER 1

1.1 Exercises

1. (a) $[2, 3]$ (b) $[1, 2]$ (c) $[6, 7]$
3. (a) 2.125 (b) 1.125 (c) 6.875
5. (a) $[2, 3]$ (b) 33 steps

1.1 Computer Problems

1. (a) 2.080084 (b) 1.169726 (c) 6.776092
3. (a) Intervals $[-2, -1]$, $[-1, 0]$, $[1, 2]$, roots -1.641784 , -0.168254 , 1.810038 (b) Intervals $[-2, -1]$, $[-0.5, 0.5]$, $[0.5, 1.5]$, roots -1.023482 , 0.163822 , 0.788941 (c) Intervals $[-1.7, -0.7]$, $[-0.7, 0.3]$, $[0.3, 1.3]$, roots -0.818094 , 0 , 0.506308
5. (a) $[1, 2]$, 27 steps, 1.25992105 (b) $[1, 2]$, 27 steps, 1.44224957 (c) $[1, 2]$, 27 steps, 1.70997595
7. first root -17.188498 , determinant correct to 2 places; second root 9.708299 , determinant correct to 3 places.
9. $H = 635.5\text{mm}$

1.2 Exercises

1. (a) $-\sqrt{3}, \sqrt{3}$ (b) 1, 2 (c) $(5 \pm \sqrt{17})/2$
3. Check by substitution.
5. B, D
7. (a) loc. convergent (b) divergent (c) divergent
9. (a) 0 is locally convergent, 1 is divergent (b) $1/2$ is locally convergent, $3/4$ is divergent
11. (a) For example, $x = x^3 + e^x$, $x = (x - e^x)^{1/3}$, and $x = \ln(x - x^3)$; (b) For example, $x = 9x^2 + 3/x^3$, $x = 1/9 - 1/(3x^4)$, and $x = (x^5 - 9x^6)/3$
13. (a) 0.3, -1.3 (b) 0.3 (c) slower
15. All converge to $\sqrt{5}$. From faster to slowest: (B), (C), (A).
17. $g(x) = \sqrt{(1-x)/2}$ is locally convergent to $1/2$, and $g(x) = -\sqrt{(1-x)/2}$ is locally convergent to -1 .
19. $g(x) = (x + A/x^2)/2$ converges to $A^{1/3}$.
21. (a) Substitute and check (b) $|g'(r)| > 1$ for all three fixed points r
23. $g'(r_2) > 1$
27. (a) $x = x - x^3$ implies $x = 0$ (b) If $0 < x_i < 1$, then $x_{i+1} = x_i - x_i^3 = x_i(1 - x_i^2) < x_i$, and $0 < x_{i+1} < x_i < 1$.
(c) The bounded monotonic sequence x_i converges to a limit L , which must be a fixed point. Therefore $L = 0$.
29. (a) $c < -2$ (b) $c = -4$
31. The open interval $(-5/4, 5/4)$ of initial guesses converge to the fixed point $1/4$; the two initial guesses $-5/4, 5/4$ lead to $-5/4$.
33. (a) Choose $a = 0$ and $|b| < 1$, c arbitrary. (b) Choose $a = 0$ and $|b| > 1$, c arbitrary.

1.2 Computer Problems

1. (a) 1.76929235 (b) 1.67282170 (c) 1.12998050
3. (a) 1.73205081 (b) 2.23606798
5. fixed point is $r = 0.641714$ and $S = |g'(r)| \approx 0.959$
7. (a) $0 < x_0 < 1$ (b) $1 < x_0 < 2$ (c) $x_0 > 2.2$, for example

1.3 Exercises

- (a) $FE = 0.01, BE = 0.04$ (b) $FE = 0.01, BE = 0.0016$ (c) $FE = 0.01, BE = 0.000064$ (d) $FE = 0.01, BE = 0.342$
- (a) 2 (b) $FE = 0.0001, BE = 5 \times 10^{-9}$
- $BE = |a| FE$
- (b) $(-1)^j (j-1)!(20-j)!$

1.3 Computer Problems

- (a) $m = 3$ (b) $x_a = -2.0735 \times 10^{-8}, FE = 2.0735 \times 10^{-8}, BE = 0$
- (a) $x_a = FE = 0.000169, BE = 0$ (b) Terminates after 13 steps, $x_a = -0.00006103$
- Predicted root $= r + \Delta r = 4 + 4^6 10^{-6}/6 = 4.000682\bar{6}$, actual root $= 4.0006825$

1.4 Exercises

- (a) $x_1 = 2, x_2 = 18/13$ (b) $x_1 = 1, x_2 = 1$ (c) $x_1 = -1, x_2 = -2/3$
- (a) $r = -1, e_{i+1} = \frac{5}{2}e_i^2; r = 0, e_{i+1} = 2e_i^2; r = 1, e_{i+1} = \frac{2}{3}e_i$ (b) $r = -1/2, e_{i+1} = 2e_i^2; r = 1, e_{i+1} = 2/3e_i$
- $r = 0$, Newton's Method; $r = 1/2$, Bisection Method
- No, $2/3$
- $x_{i+1} = (x_i + A/x_i)/2$
- $x_{i+1} = (n-1)x_i/n + A/(nx_i^{n-1})$
- (a) 0.75×10^{-12} (b) 0.5×10^{-18}

1.4 Computer Problems

- (a) 1.76929235 (b) 1.67282170 (c) 1.12998050
- (a) $r = -2/3, m = 3$ (b) $r = 1/6, m = 2$
- $r = 3.2362$ m
- -1.197624 , quadratic conv.; 0, linear conv., $m = 4$; 1.530134 , quadratic conv.
- 0.857143 , quadratic conv., $M = 2.414$; 2, linear conv., $m = 3, S = 2/3$
- initial guess $= 1.75$, solution $V = 1.70$ L
- (a) $3/4$ (c) $f(x)$ fails to be differentiable at $x = 3/4$.

1.5 Exercises

- (a) $x_2 = 8/5, x_3 = 1.742268$ (b) $x_2 = 1.578707, x_3 = 1.66016$ (c) $x_2 = 1.092907, x_3 = 1.119357$
- (a) $x_3 = -1/5, x_4 = -0.11996018$ (b) $x_3 = 1.757713, x_4 = 1.662531$ (c) $x_3 = 1.139481, x_4 = 1.129272$
- From fastest to slowest, (B), (D), (A), and (C), which does not converge (b) Newton's Method will converge faster.

1.5 Computer Problems

- (a) 1.76929235 (b) 1.67282170 (c) 1.12998050
- (a) 1.76929235 (b) 1.67282170 (c) 1.12998050
- fzero converges to the non-root zero, same as Bisection Method

CHAPTER 2

2.1 Exercises

- (a) $[4, 2]$ (b) $[5, -3]$ (c) $[1, 3]$
- (a) $[1/3, 1, 1]$ (b) $[2, -1/2, -1]$
- Approximately 27 times longer.
- Approximately 61 seconds.

2.1 Computer Problems

- (a) $[1, 1, 2]$ (b) $[1, 1, 1]$ (c) $[-1, 3, 2]$

2.2 Exercises

- (a) $\begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & -2 \end{bmatrix}$ (b) $\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 0 & -4 \end{bmatrix}$ (c) $\begin{bmatrix} 1 & 0 \\ -5/3 & 1 \end{bmatrix} \begin{bmatrix} 3 & -4 \\ 0 & -14/3 \end{bmatrix}$
- (a) $[-2, 1]$ (b) $[-1, 1]$
- $[1, -1, 1, -1]$
- 5 min., 33 sec.
- 300

2.3 Exercises

- (a) 7 (b) 8
- (a) $FE = 2, BE = 0.0002, EMF = 20001$ (b) $FE = 1, BE = 0.0001, EMF = 20001$ (c) $FE = 1, BE = 2.0001, EMF = 1$ (d) $FE = 3, BE = 0.0003, EMF = 20001$ (e) $FE = 3.0001, BE = 0.0002, EMF = 30002.5$
- (a) $RFE = 3, RBE = 3/7, EMF = 7$ (b) $RFE = 3, RBE = 1/7, EMF = 21$ (c) $RFE = 1, RBE = 1/7, EMF = 7$ (d) $RFE = 2, RBE = 6/7, EMF = 7/3$ (e) 21
- 137/60
- (a) $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ (b) $\begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$
- $LU = \begin{bmatrix} 1 & 0 & 0 \\ 0.1 & 1 & 0 \\ 0 & -5000 & 1 \end{bmatrix} \begin{bmatrix} 10 & 20 & 1 \\ 0 & -0.01 & 5.9 \\ 0 & 0 & 29501 \end{bmatrix}$, largest multiplier = 5000

2.3 Computer Problems

Answers given to Computer Problems in this section are illustrative only; results will vary slightly with implementation details.

	n	FE	EMF	$\text{cond}(A)$
1. (a)	6	5.35×10^{-10}	3.69×10^6	7.03×10^7
(b)	10	1.10×10^{-3}	9.05×10^{12}	1.31×10^{14}

n	FE	EMF	$\text{cond}(A)$
100	4.62×10^{-12}	3590	9900
200	4.21×10^{-11}	23010	39800
300	7.37×10^{-11}	50447	89700
400	1.20×10^{-10}	55019	159600
500	2.56×10^{-10}	91495	249500

5. $n \geq 13$

2.4 Exercises

1. (a) $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 0 & \frac{3}{2} \end{bmatrix}$ (b) $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 0 & 1 \end{bmatrix}$

(c) $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 5 \\ 5 & 12 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \frac{1}{5} & 1 \end{bmatrix} \begin{bmatrix} 5 & 12 \\ 0 & \frac{13}{5} \end{bmatrix}$ (d) $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

3. (a) $[-2, 1]$ (b) $[-1, 1, 1]$

5. $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$

7. $\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$

9. (a) $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 8 \end{bmatrix}$

(b) $P = I$, L is lower triangular with all non-diagonal entries -1 , the nonzero entries of U are $u_{ii} = 1$ for $1 \leq i \leq n-1$, and $u_{in} = 2^{i-1}$ for $1 \leq i \leq n$.

2.5 Exercises

1. (a) Jacobi $[u_2, v_2] = [7/3, 17/6]$ Gauss-Seidel $[u_2, v_2] = [47/18, 119/36]$ (b) Jacobi $[u_2, v_2, w_2] = [1/2, 1, 1/2]$
Gauss-Seidel $[u_2, v_2, w_2] = [1/2, 3/2, 3/4]$ (c) Jacobi $[u_2, v_2, w_2] = [10/9, -2/9, 2/3]$ Gauss-Seidel
 $[u_2, v_2, w_2] = [43/27, 14/81, 262/243]$

3. (a) $[u_2, v_2] = [59/16, 213/64]$ (b) $[u_2, v_2, w_2] = [9/8, 39/16, 81/64]$ (c) $[u_2, v_2, w_2] = [1, 1/2, 5/4]$

2.5 Computer Problems

1. $n = 100$, 36 steps, $\text{BE} = 4.58 \times 10^{-7}$; $n = 100000$, 48 steps, $\text{BE} = 2.70 \times 10^{-6}$
5. (a) 21 steps, $\text{BE} = 4.78 \times 10^{-7}$ (b) 16 steps, $\text{BE} = 1.55 \times 10^{-6}$

2.6 Exercises

1. (a) $x^T A x = x_1^2 + 3x_2^2 > 0$ for $x \neq 0$
(b) $x^T A x = (x_1 + 3x_2)^2 + x_2^2 > 0$ for $x \neq 0$
(c) $x_1^2 + 2x_2^2 + 3x_3^2 > 0$ for $x \neq 0$

3. (a) $R = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{3} \end{bmatrix}$ (b) $R = \begin{bmatrix} 1 & 3 \\ 0 & 1 \end{bmatrix}$ (c) $R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sqrt{2} & 0 \\ 0 & 0 & \sqrt{3} \end{bmatrix}$
5. (a) $R = \begin{bmatrix} 1 & 2 \\ 0 & 2 \end{bmatrix}$ (b) $R = \begin{bmatrix} 2 & -1 \\ 0 & 1/2 \end{bmatrix}$ (c) $R = \begin{bmatrix} 5 & 1 \\ 0 & 5 \end{bmatrix}$ (d) $R = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$
7. (a) $[2, -1]$ (b) $[3, 1]$
9. $x^T Ax = (x_1 + 2x_2)^2 + (d - 4)x_2^2$. If $d > 4$, the expressions can be 0 only if $0 = x_2 = x_1 + 2x_2$, which implies $x_1 = x_2 = 0$.
11. $d > 1$
13. (a) $[3, -1]$ (b) $[-1, 1]$
15. $\alpha_1 = 1/A, x_1 = b/A, r_1 = b - Ab/A = 0$

2.6 Computer Problems

1. (a) $[2, 2]$ (b) $[3, -1]$
3. (a) $[-4, 60, -180, 140]$ (b) $[-8, 504, -7560, 46200, -138600, 216216, -168168, 51480]$

2.7 Exercises

1. (a) $\begin{bmatrix} 3u^2 & 0 \\ v^3 & 3uv^2 \end{bmatrix}$ (b) $\begin{bmatrix} v \cos uv & u \cos uv \\ ve^{uv} & ue^{uv} \end{bmatrix}$ (c) $\begin{bmatrix} 2u & 2v \\ 2(u-1) & 2v \end{bmatrix}$
- (d) $\begin{bmatrix} 2u & 1 & -2w \\ vw \cos uvw & uw \cos uvw & uv \cos uvw \\ vw^4 & uw^4 & 4uvw^3 \end{bmatrix}$
3. (a) $(1/2, \pm\sqrt{3}/2)$ (b) $(\pm 2/\sqrt{5}, \pm 2/\sqrt{5})$ (c) $(4(1 + \sqrt{6})/5, \pm\sqrt{3 + 8\sqrt{6}}/5)$
5. (a) $x_1 = [0, 1], x_2 = [0, 0]$ (b) $x_1 = [0, 0], x_2 = [0.8, 0.8]$ (c) $x_1 = [8, 4], x_2 = [9.0892, -12.6103]$

2.7 Computer Problems

1. (a) $(1/2, \pm\sqrt{3}/2)$ (b) $(\pm 2/\sqrt{5}, \pm 2/\sqrt{5})$ (c) $(4(1 + \sqrt{6})/5, \pm\sqrt{3 + 8\sqrt{6}}/5)$
3. $\pm[0.50799200040795, 0.86136178666199]$
5. (a) $[1, 1, 1], [1/3, 1/3, 1/3]$ (b) $[1, 2, 3], [17/9, 22/9, 19/9]$
7. (a) 11 steps give the root $(1/2, \sqrt{3}/2)$ to 15 places (b) 13 steps give the root $(2/\sqrt{5}, 2/\sqrt{5})$ to 15 places
(c) 14 steps give the root $(4(1 + \sqrt{6})/5, \sqrt{3 + 8\sqrt{6}}/5)$ to 15 places
9. Same answers as Computer Problem 5
11. Same answers as Computer Problem 5

CHAPTER 3

3.1 Exercises

1. (a) $P(x) = \frac{(x-2)(x-3)}{(0-2)(0-3)} + 3 \frac{x(x-3)}{(2-0)(2-3)}$
- (b) $P(x) = \frac{(x+1)(x-3)(x-5)}{(2+1)(2-3)(2-5)} + \frac{(x+1)(x-2)(x-5)}{(3+1)(3-2)(3-5)} + 2 \frac{(x+1)(x-2)(x-3)}{(5+1)(5-2)(5-3)}$
- (c) $P(x) = -2 \frac{(x-2)(x-4)}{(0-2)(0-4)} + \frac{x(x-4)}{(2-0)(2-4)} + 4 \frac{x(x-2)}{4(4-2)}$

3. (a) One, $P(x) = 3 + (x + 1)(x - 2)$ (b) None (c) Infinitely many, for example $P(x) = 3 + (x + 1)(x - 2) + C(x + 1)(x - 1)(x - 2)(x - 3)^3$, where C is a nonzero constant
5. (a) $P(x) = 4 - 2x$ (b) $P(x) = 4 - 2x + A(x + 2)x(x - 1)(x - 3)$ for $A \neq 0$
7. 4
9. (a) $P(x) = 10(x - 1) \cdots (x - 6)/6!$ (b) Same as (a)
11. None
13. $4/2$
15. $P(x) = -x - (x - 1)(x - 2) \cdots (x - 25)/24!$
17. (a) 316 (b) 465

3.1 Computer Problems

1. (a) 4494564854 (b) 4454831984 (c) 4472888288

3.2 Exercises

1. (a) $P_2(x) = \frac{2}{\pi}x - \frac{4}{\pi^2}x(x - \pi/2)$ (b) $P_2(\pi/4) = 3/4$ (c) $\pi^3/128 \approx 0.242$ (d) $|\sqrt{2}/2 - 3/4| \approx 0.043$
3. (a) 7.06×10^{-11} (b) at least 9 decimal places, since $7.06 \times 10^{-11} < 0.5 \times 10^{-9}$
5. Expect errors at $x = 0.35$ to be smaller; approximately $5/21$ the size of the error at $x = 0.55$.

3.2 Computer Problems

1. (a) $P_4(x) = 1.433329 + (x - 0.6)(1.98987 + (x - 0.7)(3.2589 + (x - 0.8)(3.680667 + (x - 0.9)(4.000417))))$ (b) $P_4(0.82) = 1.95891$, $P_4(0.98) = 2.612848$ (c) Upper bound for error at $x = 0.82$ is 0.0000537, actual error is 0.0000234. Upper bound for error at $x = 0.98$ is 0.000217, actual error is 0.000107.
3. -1.952×10^{12} bbl/day. The estimate is nonsensical, due to the Runge phenomenon.

3.3 Exercises

1. (a) $\cos \pi/12, \cos \pi/4, \cos 5\pi/12, \cos 7\pi/12, \cos 3\pi/4, \cos 11\pi/12$
 (b) $2\cos \pi/8, 2\cos 3\pi/8, 2\cos 5\pi/8, 2\cos 7\pi/8$
 (c) $8 + 4\cos \pi/12, 8 + 4\cos \pi/4, 8 + 4\cos 5\pi/12, 8 + 4\cos 7\pi/12, 8 + 4\cos 3\pi/4, 8 + 4\cos 11\pi/12$
 (d) $1/5 + 1/2\cos \pi/10, 1/5 + 1/2\cos 3\pi/10, 1/5, 1/5 + 1/2\cos 7\pi/10, 1/5 + 1/2\cos 9\pi/10$
3. 0.000118, 3 correct digits
5. 0.00521
7. $d = 14$
9. (a) -1 (b) 1 (c) 0 (d) 1 (e) 1 (f) $-1/2$

3.4 Exercises

1. (a) not a cubic spline (b) cubic spline
3. (a) $c = 9/4$, natural (b) $c = 4$, parabolically-terminated and not-a-knot (c) $c = 5/2$, not-a-knot
5. One, $S_1(x) = S_2(x) = x$
7. (a)
$$\begin{cases} \frac{1}{2}x + \frac{1}{2}x^3 & \text{on } [0, 1] \\ 1 + 2(x - 1) + \frac{3}{2}(x - 1)^2 - \frac{1}{2}(x - 1)^3 & \text{on } [1, 2] \end{cases}$$
- (b)
$$\begin{cases} 1 - (x + 1) + \frac{1}{4}(x + 1)^3 & \text{on } [-1, 1] \\ 1 + 2(x - 1) + \frac{3}{2}(x - 1)^2 - \frac{1}{2}(x - 1)^3 & \text{on } [1, 2] \end{cases}$$

9. $-3, -12$
11. (a) One, $S_1(x) = S_2(x) = 2 - 4x + 2x^2$ (b) Infinitely many,
 $S_1(x) = S_2(x) = 2 - 4x + 2x^2 + cx(x-1)(x-2)$ for arbitrary c .
13. (a) $b_1 = 1, c_3 = -8/9$ (b) No. (c) The clamps are $S'(0) = 1$ and $S'(3) = -1/3$.
15. Yes. The leftmost and rightmost sections of the spline must be linear.
17. $S_2(x) = 1 + dx^3$ for arbitrary d
19. There are infinitely many parabolas through two arbitrary points with $x_1 \neq x_2$; each is a parabolically-terminated cubic spline.
21. (a) infinitely many (b) $S_1(x) = S_2(x) = x^2 + dx(x-1)(x-2)$ where $d \neq 0$.

3.4 Computer Problems

1. (a)
$$S(x) = \begin{cases} 3 + \frac{8}{3}x - \frac{2}{3}x^3 & \text{on } [0, 1] \\ 5 + \frac{2}{3}(x-1) - 2(x-1)^2 + \frac{1}{3}(x-1)^3 & \text{on } [1, 2] \\ 4 - \frac{7}{3}(x-2) - (x-2)^2 + \frac{1}{3}(x-2)^3 & \text{on } [2, 3] \end{cases}$$
- (b)
$$S(x) = \begin{cases} 3 + 2.5629(x+1) - 0.5629(x+1)^3 & \text{on } [-1, 0] \\ 5 + 0.8742x - 1.6887x^2 + 0.3176x^3 & \text{on } [0, 3] \\ 1 - 0.6824(x-3) + 1.1698(x-3)^2 - 0.4874(x-3)^3 & \text{on } [3, 4] \\ 1 + 0.1950(x-4) - 0.2925(x-4)^2 + 0.0975(x-4)^3 & \text{on } [4, 5] \end{cases}$$
3.
$$S(x) = \begin{cases} 1 + \frac{149}{56}x - \frac{37}{56}x^3 & \text{on } [0, 1] \\ 3 + \frac{19}{28}(x-1) - \frac{111}{56}(x-1)^2 + \frac{73}{56}(x-1)^3 & \text{on } [1, 2] \\ 3 + \frac{5}{8}(x-2) + \frac{27}{14}(x-2)^2 - \frac{87}{56}(x-2)^3 & \text{on } [2, 3] \\ 4 - \frac{5}{28}(x-3) - \frac{153}{56}(x-3)^2 + \frac{51}{56}(x-3)^3 & \text{on } [4, 5] \end{cases}$$
1.
$$S(x) = \begin{cases} 1 + 1.8006x + \frac{3}{2}x^2 - 1.3006x^3 & \text{on } [0, 1] \\ 3 + 0.8988(x-1) - 2.4018(x-1)^2 + 1.5030(x-1)^3 & \text{on } [1, 2] \\ 3 + 0.6042(x-2) + 2.1071(x-2)^2 - 1.7113(x-2)^3 & \text{on } [2, 3] \\ 4 - 0.3155(x-3) - 3.0268(x-3)^2 + 1.3423(x-3)^3 & \text{on } [4, 5] \end{cases}$$
3.
$$S(x) = \begin{cases} 1 - 2x + \frac{57}{7}x^2 - \frac{29}{7}x^3 & \text{on } [0, 1] \\ 3 + \frac{13}{7}(x-1) - \frac{30}{7}(x-1)^2 + \frac{17}{7}(x-1)^3 & \text{on } [1, 2] \\ 3 + \frac{4}{7}(x-2) + 3(x-2)^2 - \frac{18}{7}(x-2)^3 & \text{on } [2, 3] \\ 4 - \frac{8}{7}(x-3) - \frac{33}{7}(x-3)^2 + \frac{27}{7}(x-3)^3 & \text{on } [4, 5] \end{cases}$$
5.
$$S(x) = \begin{cases} x - 0.0006x^2 - 0.1639x^3 & \text{on } [0, \frac{\pi}{8}] \\ \sin \frac{\pi}{8} + 0.9237(x - \frac{\pi}{8}) - 0.1937(x - \frac{\pi}{8})^2 - 0.1396(x - \frac{\pi}{8})^3 & \text{on } [\frac{\pi}{8}, \frac{\pi}{4}] \\ \frac{\sqrt{2}}{2} + 0.7070(x - \frac{\pi}{4}) - 0.3582(x - \frac{\pi}{4})^2 - 0.0931(x - \frac{\pi}{4})^3 & \text{on } [\frac{\pi}{4}, \frac{3\pi}{8}] \\ \sin \frac{3\pi}{8} + 0.3826(x - \frac{3\pi}{8}) - 0.4679(x - \frac{3\pi}{8})^2 - 0.0327(x - \frac{3\pi}{8})^3 & \text{on } [\frac{3\pi}{8}, \frac{\pi}{2}] \end{cases}$$

7. $n = 48$

9. (a) 322.6 (b) 318.8 (c) not-a-knot spline is identical to solution of Exercise 3.1.13

3.5 Exercises

1. (a) $\begin{cases} x(t) = 6t^2 - 5t^3 \\ y(t) = 6t - 12t^2 + 6t^3 \end{cases}$ (b) $\begin{cases} x(t) = 1 - 3t - 3t^2 + 3t^3 \\ y(t) = 1 - 3t + 3t^2 \end{cases}$ (c) $\begin{cases} x(t) = 1 + 3t^2 - 2t^3 \\ y(t) = 2 + 3t - 3t^2 \end{cases}$
3. $\begin{cases} x(t) = 1 + 6t^2 - 4t^3 \\ y(t) = 2 + 6t^2 - 4t^3 \end{cases}$ $\begin{cases} x(t) = 3 + 6t^2 - 4t^3 \\ y(t) = 4 - 9t^2 + 6t^3 \end{cases}$ $\begin{cases} x(t) = 5 - 12t^2 + 8t^3 \\ y(t) = 1 + 3t^2 - 2t^3 \end{cases}$
5. The number 3.
7. $\begin{cases} x(t) = -1 + 6t^2 - 4t^3 \\ y(t) = 4t - 4t^2 \end{cases}$
9. (a) $\begin{cases} x(t) = 1 + 3t - 9t^2 + 5t^3 \\ y(t) = 6t^2 - 5t^3 \\ z(t) = 3t^2 - 3t^3 \end{cases}$ (b) $\begin{cases} x(t) = 1 - 6t^2 + 6t^3 \\ y(t) = 1 + 3t - 9t^2 + 6t^3 \\ z(t) = 2 + 3t - 12t^2 + 8t^3 \end{cases}$
 (c) $\begin{cases} x(t) = 2 + 3t - 12t^2 + 10t^3 \\ y(t) = 1 \\ z(t) = 1 + 6t^2 - 4t^3 \end{cases}$

CHAPTER 4

4.1 Exercises

1. (a) $\bar{x} = [-1/7, 10/7], \|e\|_2 = \sqrt{14}/7$ (b) $\bar{x} = [-1/2, 2], \|e\|_2 = \sqrt{6}/2$
 (c) $\bar{x} = [16/19, 16/19], \|e\|_2 = 2.013$
3. $\bar{x} = [4, x_2]$ for arbitrary x_2
7. (a) $y = 1/5 - 6/5t$, RMSE = $\sqrt{2/5} \approx 0.6325$ (b) $y = 6/5 + 1/2t$, RMSE = $\sqrt{26}/10 \approx 0.5099$
9. (a) $y = 0.3481 + 1.9475t - 0.1657t^2$, RMSE = 0.5519 (b) $y = 2.9615 - 1.0128t + 0.1667t^2$, RMSE = 0.4160 (c) $y = 4.8 - 1.2t$, RMSE = 0.4472
11. $h(t) = 0.475 + 141.525t - 4.905t^2$, max height = 1021.3m, landing time = 28.86 sec.

4.1 Computer Problems

1. (a) $\bar{x} = [2.5246, 0.6616, 2.0934], \|e\|_2 = 2.4135$ (b) $\bar{x} = [1.2739, 0.6885, 1.2124, 1.7497], \|e\|_2 = 0.8256$
3. (a) 2,996,236,899 + 76,542,140($t - 1960$), RMSE = 36,751,088
 (b) 3,028,751,748 + 67,871,514($t - 1960$) + 216,766($t - 1960$)², RMSE = 17,129,714;
 1980 estimates: (a) 4,527,079,702 (b) 4,472,888,288; Parabola gives better estimate.
5. (a) $c_1 = 9510.1, c_2 = -8314.36$, RMSE = 518.3 (b) selling price = 68.7 cents maximizes profit.
7. (a) $y = 0.0769$, RMSE = 0.2665 (b) $y = 0.1748 - 0.02797t^2$, RMSE = 0.2519
9. (a) 4 correct decimal places, $P_5(t) = 1.000009 + 0.999983t + 1.000012t^2 + 0.999996t^3 + 1.000000t^4 + 1.000000t^5$;
 $\text{cond}(A^T A) = 2.72 \times 10^{13}$ (b) 1 correct decimal place, $P_6(t) = 0.99 + 1.02t + 0.98t^2 + 1.01t^3 + t^4 + t^5 + t^6$;
 $\text{cond}(A^T A) = 2.55 \times 10^{16}$ (c) $P_8(t)$ has no correct places, $\text{cond}(A^T A) = 1.41 \times 10^{19}$

4.2 Exercises

1. (a) $y = 3/2 - 1/2 \cos 2\pi t + 3/2 \sin 2\pi t, \|e\|_2 = 0$, RMSE = 0 (b) $y = 7/4 - 1/2 \cos 2\pi t + \sin 2\pi t, \|e\|_2 = 1/2$, RMSE = 1/4 (c) $y = 9/4 + 3/4 \cos 2\pi t, \|e\|_2 = 1/\sqrt{2}$, RMSE = $1/(2\sqrt{2})$
3. (a) $y = 1.932e^{0.3615t}, \|e\|_2 = 1.2825$, (b) $y = 2^{t-1/4}, \|e\|_2 = 0.9982$
5. (a) $y = 5.5618t^{-1.3778}$, RMSE = 0.2707 (b) $y = 2.8256t^{0.7614}$, RMSE = 0.7099

4.2 Computer Problems

1. $y = 5.5837 + 0.7541 \cos 2\pi t + 0.1220 \sin 2\pi t + 0.1935 \cos 4\pi t$ M bbls/day, RMSE = 0.1836
3. $P(t) = 3,079,440,361e^{0.0174(t-1960)}$, 1980 estimate is $P(20) = 4,361,485,000$, estimation error ≈ 91 million
5. (a) $t_{\max} = -1/c_2$ (b) half-life ≈ 7.81 hrs.

4.3 Exercises

1. (a) $\begin{bmatrix} 0.8 & -0.6 \\ 0.6 & 0.8 \end{bmatrix} \begin{bmatrix} 5 & 0.6 \\ 0 & 0.8 \end{bmatrix}$ (b) $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \sqrt{2} & \frac{3\sqrt{2}}{2} \\ 0 & \frac{\sqrt{2}}{2} \end{bmatrix}$
 (c) $\begin{bmatrix} \frac{2}{3} & \frac{\sqrt{2}}{6} & \frac{\sqrt{2}}{2} \\ \frac{1}{3} & -\frac{2\sqrt{2}}{3} & 0 \\ \frac{2}{3} & \frac{\sqrt{2}}{6} & -\frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 0 & \sqrt{2} \\ 0 & 0 \end{bmatrix}$ (d) $\begin{bmatrix} \frac{4}{5} & 0 & -\frac{3}{5} \\ 0 & 1 & 0 \\ \frac{3}{5} & 0 & \frac{4}{5} \end{bmatrix} \begin{bmatrix} 5 & 10 & 5 \\ 0 & 2 & -2 \\ 0 & 0 & 5 \end{bmatrix}$
3. (a) – (d) same as Exercise 1
5. (a) – (d) same as Exercise 1
7. (a) $\bar{x} = [4, -1]$ (b) $\bar{x} = [-11/18, 4/9]$

4.3 Computer Problems

5. (a) $\bar{x} = [1.6154, 1.6615]$, $\|e\|_2 = 0.3038$ (b) $\bar{x} = [2.0588, 2.3725, 1.5784]$, $\|e\|_2 = 0.2214$
7. (a) $\bar{x} = [1, \dots, 1]$ to 10 correct decimal places (b) $\bar{x} = [1, \dots, 1]$ to 6 correct decimal places

4.4 Exercises

1. (a) $x_1 = [0.5834, -0.0050, -0.5812]$, $x_2 = [1.0753, -0.1039, -0.9417]$, $x_3 = [1, 0, -1]$
 (b) $x_1 = [0.3896, 0.1674, 0.3045]$, $x_2 = [0.7650, 0.2107, 0.2502]$, $x_3 = [1/2, 1/2, 0]$
 (c) $x_1 = [0.0332, 0.8505, 0.9668]$, $x_2 = [0.0672, 0.8479, 0.9696]$, $x_3 = [0, 0, 1]$

4.5 Exercises

1. (a) $(x_1, y_1) = (2 - \sqrt{2}, 0)$ (b) $(x_1, y_1) = (1 - \sqrt{2}/2, 0)$
5. (a) $\begin{bmatrix} t_1^{c_2} & c_1 t_1^{c_2} \ln t_1 \\ t_2^{c_2} & c_1 t_2^{c_2} \ln t_2 \\ t_3^{c_2} & c_1 t_3^{c_2} \ln t_3 \end{bmatrix}$ (b) $\begin{bmatrix} t_1 e^{c_2 t_1} & c_1 t_1^2 e^{c_2 t_1} \\ t_2 e^{c_2 t_2} & c_1 t_2^2 e^{c_2 t_2} \\ t_3 e^{c_2 t_3} & c_1 t_3^2 e^{c_2 t_3} \end{bmatrix}$

4.5 Computer Problems

1. (a) $(\bar{x}, \bar{y}) = (0.410623, 0.055501)$ (b) $(\bar{x}, \bar{y}) = (0.275549, 0)$
3. (a) $(x, y) = (0, -0.586187)$, $K = 0.329572$ (b) $(x, y) = (0.556853, 0)$, $K = 1.288037$
5. $c_1 = 15.9$, $c_2 = 2.53$, RMSE = 0.755
7. Same as Computer Problem 5.
9. (a) $c_1 = 11.993468$, $c_2 = 0.279608$, $c_3 = 1.802342$, RMSE = 0.441305
 (b) $c_1 = 12.702778$, $c_2 = 0.159591$, $c_3 = 5.682764$, RMSE = 0.802834
11. (a) $c_1 = 8.670956$, $c_2 = 0.274184$, $c_3 = 0.981070$, $c_4 = 1.232813$, RMSE = 0.102660
 (b) $c_1 = 8.683823$, $c_2 = 0.131945$, $c_3 = 0.620292$, $c_4 = -1.921257$, RMSE = 0.199789

CHAPTER 5

5.1 Exercises

- (a) 0.9531, error = 0.0469 (b) 0.9950, error = 0.0050 (c) 0.9995, error = 0.0005
- (a) 0.455902, error = 0.044098; error must satisfy $0.0433 \leq \text{error} \leq 0.0456$ (b) 0.495662, error = 0.004338; error must satisfy $0.004330 \leq \text{error} \leq 0.004355$ (c) 0.499567, error = 0.000433; error must satisfy $0.0004330 \leq \text{error} \leq 0.0004333$
- (a) 2.02020202, error = 0.02020202 (b) 2.00020002, error = 0.00020002 (c) 2.00000200, error = 0.00000200
- $f'(x) = [(f(x) - f(x-h))/h + hf''(c)]/2$
- $f'(x) = [3f(x) - 4f(x-h) + f(x-2h)]/(2h) + O(h^2)$
- $f'(x) \approx [4f(x+h/2) - 3f(x) - f(x+h)]/h$
- $f'(x) = [f(x+3h) + 8f(x) - 9f(x-h)]/(12h) - h^2 f'''(c)/2$, where $x-h < c < x+3h$
- $f''(x) = [f(x+3h) - 4f(x) + 3f(x-h)]/(6h^2) - 2hf'''(c)/3$, where $x-h < c < x+3h$
- $f'(x) = [4f(x+3h) + 5f(x) - 9f(x-2h)]/(30h) - h^2 f'''(c)$, where $x-2h < c < x+3h$

5.1 Computer Problems

- minimum error at $h = 10^{-5} \approx \epsilon_{\text{mach}}^{1/3}$
- minimum error at $h = 10^{-8} \approx \epsilon_{\text{mach}}^{1/2}$
- (a) minimum error at $h = 10^{-4} \approx \epsilon_{\text{mach}}^{1/4}$ (b) same as (a)

5.2 Exercises

- (a) $m = 1 : 0.500000$, err = 0.166667; $m = 2 : 0.375000$, err = 0.041667; $m = 4 : 0.343750$, err = 0.010417
(b) $m = 1 : 0.785398$, err = 0.214602; $m = 2 : 0.948059$, err = 0.051941; $m = 4 : 0.987116$, err = 0.012884
(c) $m = 1 : 1.859141$, err = 0.140859; $m = 2 : 1.753931$, err = 0.035649; $m = 4 : 1.727222$, err = 0.008940
- (a) $m = 1 : 1/3$, err = 0; $m = 2 : 1/3$, err = 0; $m = 4 : 1/3$, err = 0 (b) $m = 1 : 1.002280$, err = 0.002280; $m = 2 : 1.000135$, err = 0.000135; $m = 4 : 1.000008$, err = 0.000008 (c) $m = 1 : 1.718861$, err = 0.000579; $m = 2 : 1.718319$, err = 0.000037; $m = 4 : 1.718284$, err = 0.000002
- (a) $m = 1 : 1.414214$, err = 0.585786; $m = 2 : 1.577350$, err = 0.422650; $m = 4 : 1.698844$, err = 0.301156
(b) $m = 1 : 1.259921$, err = 0.240079; $m = 2 : 1.344022$, err = 0.155978; $m = 4 : 1.400461$, err = 0.099539
(c) $m = 1 : 2.000000$, err = 0.828427; $m = 2 : 2.230710$, err = 0.597717; $m = 4 : 2.402528$, err = 0.425899
- (a) 1.631729, err = 0.368271
(b) 1.372055, err = 0.127945
(c) 2.307614, err = 0.520814
- (a) 1 (b) 1 (c) 3
- $\frac{4h}{3} \sum_{i=1}^m [2f(u_i) + 2f(v_i) - f(w_i)] + \frac{7(b-a)h^4}{90} f^{(iv)}(c)$
- 5

5.2 Computer Problems

- (a) exact = 2; $m = 16$ approx = 1.998638, err = 1.36×10^{-3} ; $m = 32$ approx = 1.999660, err = 3.40×10^{-4}
(b) exact = $1/2(1 - \ln 2)$; $m = 16$ approx = 0.153752, err = 3.26×10^{-4} ; $m = 32$ approx = 0.153508, err = 8.14×10^{-5} (c) exact = 1; $m = 16$ approx = 1.001444, err = 1.44×10^{-3} ; $m = 32$ approx = 1.000361, err = 3.61×10^{-4} (d) exact = $9 \ln 3 - 26/9$; $m = 16$ approx = 7.009809, err = 1.12×10^{-2} ; $m = 32$ approx = 7.001419, err = 2.80×10^{-3} (e) exact = $\pi^2 - 4$; $m = 16$ approx = 5.837900, err = 3.17×10^{-2} ; $m = 32$ approx = 5.861678, err = 7.93×10^{-3} (f) exact = $2\sqrt{5} - \sqrt{15}/2$; $m = 16$ approx = 2.535672, err = 2.80×10^{-5} ;

- $m = 32$ approx = 2.535651, err = 7.00×10^{-6} (g) exact = $\ln(\sqrt{3} + 2)$; $m = 16$ approx = 1.316746, err = 2.11×10^{-4} ; $m = 32$ approx = 1.316905, err = 5.29×10^{-5} (h) exact = $\ln(\sqrt{2} + 1)/2$; $m = 16$ approx = 0.440361, err = 3.26×10^{-4} ; $m = 32$ approx = 0.440605, err = 8.14×10^{-5}
3. (a) $m = 16$ approx = 1.464420; $m = 32$ approx = 1.463094 (b) $m = 16$ approx = 0.891197; $m = 32$ approx = 0.893925 (c) $m = 16$ approx = 3.977463; $m = 32$ approx = 3.977463 (d) $m = 16$ approx = 0.264269; $m = 32$ approx = 0.264025 (e) $m = 16$ approx = 0.160686; $m = 32$ approx = 0.160936 (f) $m = 16$ approx = -0.278013; $m = 32$ approx = -0.356790 (g) $m = 16$ approx = 0.785276; $m = 32$ approx = 0.783951 (h) $m = 16$ approx = 0.369964; $m = 32$ approx = 0.371168
5. (a) $m = 10$: 1.808922, err = 0.191078; $m = 100$: 1.939512, err = 0.060488; $m = 1000$: 1.980871, err = 0.019129 (b) $m = 10$: 1.445632, err = 0.054368; $m = 100$: 1.488258, err = 0.011742; $m = 1000$: 1.497470, err = 0.002530 (c) $m = 10$: 2.558203, err = 0.270225; $m = 100$: 2.742884, err = 0.085543; $m = 1000$: 2.801375, err = 0.027052
7. (a) $m = 16$ approx = 1.8315299; $m = 32$ approx = 1.83183081 (b) $m = 16$ approx = 2.99986658; $m = 32$ approx = 3.00116293 (c) $m = 16$ approx = 0.91601205; $m = 32$ approx = 0.91597721

5.3 Exercises

1. (a) $1/3$ (b) 0.99999157 (c) 1.71828269

5.3 Computer Problems

1. (a) correct = 2, approx = 2.00000010, err = 1.0×10^{-7} (b) correct $1/2(1 - \ln 2)$, approx = 0.15342640, err = 1.23×10^{-8} (c) correct 1, approx = 1.00000000, err = 3.5×10^{-13} (d) correct $9 \ln 3 - 26/9$, approx = 6.99862171, err = 3.00×10^{-9} (e) correct $\pi^2 - 4$, approx = 5.86960486, err = 4.56×10^{-7} (f) correct $2\sqrt{5} - \sqrt{15}/2$, approx = 2.53564428, err = 1.21×10^{-10} (g) correct $\ln(\sqrt{3} + 2)$, approx = 1.31695765, err = 2.46×10^{-7} (h) correct $\ln(\sqrt{2} + 1)/2$, approx = 0.44068686, err = 6.98×10^{-8}

5.4 Exercises

1. (a) 0.3750, error = 0.0417 (b) 0.9871, error = 0.0129 (c) 1.7539, error = 0.0356
 3. Use same tolerance test as Adaptive Quadrature with Trapezoid Rule, replace Trapezoid Rule with Midpoint Rule.

5.4 Computer Problems

1. (a) 2.00000000, 12606 subintervals (b) 0.15342641, 6204 subintervals (c) 1.00000000, 12424 subintervals (d) 6.99862171, 32768 subintervals (e) 5.86960440, 73322 subintervals (f) 2.53564428, 1568 subintervals (g) 1.31695790, 7146 subintervals (h) 0.44068679, 5308 subintervals
3. first eight decimal places identical to Computer Problem 1 (a) 56 subintervals (b) 46 subintervals (c) 40 subintervals (d) 56 subintervals (e) 206 subintervals (f) 22 subintervals (g) 54 subintervals (h) 52 subintervals
5. first eight decimal places identical to Computer Problem 1 (a) 50 subintervals (b) 44 subintervals (c) 36 subintervals (d) 54 subintervals (e) 198 subintervals (f) 22 subintervals (g) 50 subintervals (h) 52 subintervals
7. Same as Computer Problem 6
9. $\text{erf}(1) = 0.84270079$, $\text{erf}(3) = 0.99997791$

5.5 Exercises

1. (a) 0, error = 0 (b) 0.222222, error = 0.1777778 (c) 2.342696, error = 0.007706 (d) -0.481237, error = 0.481237
3. (a) 0, error = 0 (b) 0.4, error = 0 (c) 2.350402, error = 2.95×10^{-7} (d) -0.002136, error = 0.002136
5. (a) 1.999825 (b) 0.15340700 (c) 0.99999463 (d) 6.99867782

CHAPTER 6

6.1 Exercises

3. (a) $y(t) = 1 + t^2/2$ (b) $y(t) = e^{t^3/3}$ (c) $y(t) = e^{t^2+2t}$ (d) $y = e^{t^5}$ (e) $y(t) = (3t + 1)^{1/3}$ (f) $y(t) = (3t^4/4 + 1)^{1/3}$
5. (a) $w = [1.0000, 1.0000, 1.0625, 1.1875, 1.3750]$, error = 0.1250
 (b) $w = [1.0000, 1.0000, 1.0156, 1.0791, 1.2309]$, error = 0.1648
 (c) $w = [1.0000, 1.5000, 2.4375, 4.2656, 7.9980]$, error = 12.0875
 (d) $w = [1.0000, 1.0000, 1.0049, 1.0834, 1.5119]$, error = 1.2064
 (e) $w = [1.0000, 1.2500, 1.4100, 1.5357, 1.6417]$, error = 0.0543
 (f) $w = [1.0000, 1.0000, 1.0039, 1.0349, 1.1334]$, error = 0.0717
7. (b) $c = \arctan y_0$
9. (a) $L = 0$, has unique solution (b) $L = 1$, has unique solution (a) $L = 1$, has unique solution
 (d) No Lipschitz constant
11. (a) Solutions are $Y(t) = t^2/2$ and $Z(t) = t^2/2 + 1$. $|Y(t) - Z(t)| = 1 \leq e^0|1| = 1$ (b) Solutions are $Y(t) = 0$ and $Z(t) = e^t$. $|Y(t) - Z(t)| = e^t \leq e^{1(t-0)}|1|$ (c) Solutions are $Y(t) = 0$ and $Z(t) = e^{-t}$. $|Y(t) - Z(t)| = e^{-t} \leq e^{1(t-0)}|1| = 1$ (d) Lipschitz condition not satisfied
13. $y(t) = 1/(1 - t)$
15. (a) $[a, b]$

6.1 Computer Problems

1.	(a)	t_i	w_i	error	(b)	t_i	w_i	error	(c)	t_i	w_i	error
		0.0	1.0000	0.0000		0.0	1.0000	0.0000		0.0	1.0000	0.0000
		0.1	1.0000	0.0050		0.1	1.0000	0.0003		0.1	1.2000	0.0337
		0.2	1.0100	0.0100		0.2	1.0010	0.0017		0.2	1.4640	0.0887
		0.3	1.0300	0.0150		0.3	1.0050	0.0040		0.3	1.8154	0.1784
		0.4	1.0600	0.0200		0.4	1.0140	0.0075		0.4	2.2874	0.3243
		0.5	1.1000	0.0250		0.5	1.0303	0.0123		0.5	2.9278	0.5625
		0.6	1.1500	0.0300		0.6	1.0560	0.0186		0.6	3.8062	0.9527
		0.7	1.2100	0.0350		0.7	1.0940	0.0271		0.7	5.0241	1.5952
		0.8	1.2800	0.0400		0.8	1.1477	0.0384		0.8	6.7323	2.6610
		0.9	1.3600	0.0450		0.9	1.2211	0.0540		0.9	9.1560	4.4431
		1.0	1.4500	0.0500		1.0	1.3200	0.0756		1.0	12.6352	7.4503
	(d)	t_i	w_i	error	(e)	t_i	w_i	error	(f)	t_i	w_i	error
		0.0	1.0000	0.0000		0.0	1.0000	0.0000		0.0	1.0000	0.0000
		0.1	1.0000	0.0000		0.1	1.1000	0.0086		0.1	1.0000	0.0000
		0.2	1.0001	0.0003		0.2	1.1826	0.0130		0.2	1.0001	0.0003
		0.3	1.0009	0.0016		0.3	1.2541	0.0156		0.3	1.0009	0.0011
		0.4	1.0049	0.0054		0.4	1.3177	0.0171		0.4	1.0036	0.0028
		0.5	1.0178	0.0140		0.5	1.3753	0.0181		0.5	1.0099	0.0054
		0.6	1.0496	0.0313		0.6	1.4282	0.0187		0.6	1.0222	0.0092
		0.7	1.1176	0.0654		0.7	1.4772	0.0191		0.7	1.0429	0.0139
		0.8	1.2517	0.1360		0.8	1.5230	0.0193		0.8	1.0744	0.0190
		0.9	1.5081	0.2968		0.9	1.5661	0.0195		0.9	1.1188	0.0239
		1.0	2.0028	0.7154		1.0	1.6069	0.0195		1.0	1.1770	0.0281

6.2 Exercises

1. (a) $w = [1.0000, 1.0313, 1.1250, 1.2813, 1.5000]$, error = 0 (b) $w = [1.0000, 1.0078, 1.0477, 1.1587, 1.4054]$, error = 0.0097 (c) $w = [1.0000, 1.7188, 3.3032, 7.0710, 16.7935]$, error = 3.2920
 (d) $w = [1.0000, 1.0024, 1.0442, 1.3077, 2.7068]$, error = 0.0115
 (e) $w = [1.0000, 1.2050, 1.3570, 1.4810, 1.5871]$, error = 0.0003
 (f) $w = [1.0000, 1.0020, 1.0193, 1.0823, 1.2182]$, error = 0.0132
3. (a) $w_{i+1} = w_i + ht_i w_i + 1/2h^2(w_i + t_i^2 w_i)$
 (b) $w_{i+1} = w_i + h(t_i w_i^2 + w_i^3) + 1/2h^2(w_i^2 + (2t_i w_i + 3w_i^2)(t_i w_i^2 + w_i^3))$
 (c) $w_{i+1} = w_i + hw_i \sin w_i + 1/2h^2(\sin w_i + w_i \cos w_i)w_i \sin w_i$
 (d) $w_{i+1} = w_i + he^{w_i t_i^2} + 1/2h^2 e^{w_i t_i^2} (2t_i w_i + t_i^2 e^{w_i t_i^2})$

6.2 Computer Problems

1.	(a)	t_i	w_i	error	(b)	t_i	w_i	error	(c)	t_i	w_i	error
		0.0	1.0000	0		0.0	1.0000	0.0000		0.0	1.0000	0.0000
		0.1	1.0050	0		0.1	1.0005	0.0002		0.1	1.2320	0.0017
		0.2	1.0200	0		0.2	1.0030	0.0003		0.2	1.5479	0.0048
		0.3	1.0450	0		0.3	1.0095	0.0005		0.3	1.9832	0.0106
		0.4	1.0800	0		0.4	1.0222	0.0007		0.4	2.5908	0.0209
		0.5	1.1250	0		0.5	1.0434	0.0008		0.5	3.4509	0.0394
		0.6	1.1800	0		0.6	1.0757	0.0010		0.6	4.6864	0.0725
		0.7	1.2450	0		0.7	1.1224	0.0012		0.7	6.4878	0.1316
		0.8	1.3200	0		0.8	1.1875	0.0014		0.8	9.1556	0.2378
0.9	1.4050	0	0.9	1.2767	0.0016	0.9	13.1694	0.4297				
1.0	1.5000	0	1.0	1.3974	0.0018	1.0	19.3063	0.7792				
(d)	t_i	w_i	error	(e)	t_i	w_i	error	(f)	t_i	w_i	error	
	0.0	1.0000	0.0000		0.0	1.0000	0.0000		0.0	1.0000	0.0000	
	0.1	1.0000	0.0000		0.1	1.0913	0.0001		0.1	1.0001	0.0000	
	0.2	1.0005	0.0001		0.2	1.1695	0.0001		0.2	1.0005	0.0001	
	0.3	1.0029	0.0004		0.3	1.2384	0.0001		0.3	1.0022	0.0002	
	0.4	1.0114	0.0011		0.4	1.3005	0.0001		0.4	1.0068	0.0004	
	0.5	1.0338	0.0021		0.5	1.3571	0.0001		0.5	1.0160	0.0006	
	0.6	1.0845	0.0037		0.6	1.4093	0.0001		0.6	1.0323	0.0009	
	0.7	1.1890	0.0060		0.7	1.4580	0.0001		0.7	1.0579	0.0011	
	0.8	1.3967	0.0090		0.8	1.5036	0.0001		0.8	1.0948	0.0014	
0.9	1.8158	0.0109	0.9	1.5466	0.0001	0.9	1.1443	0.0017				
1.0	2.7164	0.0018	1.0	1.5873	0.0001	1.0	1.2069	0.0018				

6.3 Exercises

$$\begin{aligned}
1. \quad (a) \quad \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} &= \begin{bmatrix} 1 & 1.25 & 1.5 & 1.7188 & 1.8594 \\ 0 & -0.25 & -0.625 & -1.1563 & -1.875 \end{bmatrix} \quad \text{error} = \begin{bmatrix} 0.3907 \\ 0.4124 \end{bmatrix} \\
(b) \quad \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} &= \begin{bmatrix} 1 & 0.7500 & 0.5000 & 0.2813 & 0.1094 \\ 0 & 0.2500 & 0.3750 & 0.4063 & 0.3750 \end{bmatrix} \quad \text{error} = \begin{bmatrix} 0.0894 \\ 0.0654 \end{bmatrix} \\
(c) \quad \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} &= \begin{bmatrix} 1 & 1.0000 & 0.9375 & 0.8125 & 0.6289 \\ 0 & 0.2500 & 0.5000 & 0.7344 & 0.9375 \end{bmatrix} \quad \text{error} = \begin{bmatrix} 0.0886 \\ 0.0960 \end{bmatrix} \\
(d) \quad \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} &= \begin{bmatrix} 5 & 6.2500 & 9.6875 & 17.2656 & 32.9492 \\ 0 & 2.5000 & 6.8750 & 15.1563 & 31.3672 \end{bmatrix} \quad \text{error} = \begin{bmatrix} 77.3507 \\ 77.0934 \end{bmatrix}
\end{aligned}$$

1. (a) $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 1.2500 & 1.4648 & 1.5869 & 1.5354 \\ 0 & -0.3125 & -0.7813 & -1.4343 & -2.2888 \end{bmatrix}$ error = $\begin{bmatrix} 0.0667 \\ 0.0015 \end{bmatrix}$
- (b) $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0.7500 & 0.5273 & 0.3428 & 0.1990 \\ 0 & 0.1875 & 0.2813 & 0.3098 & 0.2966 \end{bmatrix}$ error = $\begin{bmatrix} 0.0002 \\ 0.0129 \end{bmatrix}$
- (c) $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 0.9688 & 0.8760 & 0.7275 & 0.5327 \\ 0 & 0.2500 & 0.4844 & 0.6882 & 0.8486 \end{bmatrix}$ error = $\begin{bmatrix} 0.0076 \\ 0.0071 \end{bmatrix}$
- (d) $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 5 & 7.3438 & 14.3311 & 32.6805 & 79.2426 \\ 0 & 3.4375 & 11.2793 & 30.2963 & 77.3799 \end{bmatrix}$ error = $\begin{bmatrix} 31.0574 \\ 31.0806 \end{bmatrix}$
3. (a) $y_1 = [1.0000, 1.2500, 1.5195, 1.8364, 2.2388]$ (b) $[1, 1.1875, 1.2378, 1.1229, 0.7832]$
 (c) $[1, 1.2813, 1.6617, 2.1999, 2.9933]$

6.3 Computer Problems

1. errors in $[y_1, y_2]$: (a) $[0.1973, 0.1592]$ for $h = 0.1$, $[0.0226, 0.0149]$ for $h = 0.01$ (b) $[0.0328, 0.0219]$ for $h = 0.1$, $[0.0031, 0.0020]$ for $h = 0.01$ (c) $[0.0305, 0.0410]$ for $h = 0.1$, $[0.0027, 0.0042]$ for $h = 0.01$
 (d) $[51.4030, 51.3070]$ for $h = 0.1$, $[8.1919, 8.1827]$ for $h = 0.01$. Note that the errors decline roughly by a factor of 10 for a first-order method.
5. (a) Roughly speaking, periodic trajectory consisting of $3\frac{1}{2}$ revolutions clockwise, $2\frac{1}{2}$ revolutions counterclockwise, $3\frac{1}{2}$ revolutions clockwise, $2\frac{1}{2}$ revolutions counterclockwise. The other periodic trajectory is the same with clockwise replaced by counterclockwise.

6.4 Exercises

1. (a) $w = [1.0000, 1.0313, 1.1250, 1.2813, 1.5000]$, error = 0 (b) $w = [1.0000, 1.0039, 1.0395, 1.1442, 1.3786]$, error = 0.0171 (c) $w = [1.0000, 1.7031, 3.2399, 6.8595, 16.1038]$, error = 3.9817
 (d) $w = [1.0000, 1.0003, 1.0251, 1.2283, 2.3062]$, error = 0.4121
 (e) $w = [1.0000, 1.1975, 1.3490, 1.4734, 1.5801]$, error = 0.0073
 (f) $w = [1.0000, 1.0005, 1.0136, 1.0713, 1.2055]$, error = 0.0004
3. (a) $w = [1, 1.0313, 1.1250, 1.2813, 1.5000]$, error = 0 (b) $w = [1, 1.0052, 1.0425, 1.1510, 1.3956]$, error = 1.2476×10^{-5} (c) $w = [1, 1.7545, 3.4865, 7.8448, 19.975]$, error = 0.11007
 (d) $w = [1, 1.001, 1.0318, 1.2678, 2.7103]$, error = 7.9505×10^{-3}
 (e) $w = [1, 1.2051, 1.3573, 1.4813, 1.5874]$, error = 4.1996×10^{-5}
 (f) $w = [1, 1.0010, 1.0154, 1.0736, 1.2051]$, error = 6.0464×10^{-5}

6.4 Computer Problems

1.		t_i	w_i	error		t_i	w_i	error		t_i	w_i	error
	(a)	0.0	1.0000	0	(b)	0.0	1.0000	0.0000	(c)	0.0	1.0000	0.0000
		0.1	1.0050	0		0.1	1.0003	0.0001		0.1	1.2310	0.0027
		0.2	1.0200	0		0.2	1.0025	0.0002		0.2	1.5453	0.0074
		0.3	1.0450	0		0.3	1.0088	0.0003		0.3	1.9780	0.0158
		0.4	1.0800	0		0.4	1.0212	0.0004		0.4	2.5814	0.0303
		0.5	1.1250	0		0.5	1.0420	0.0005		0.5	3.4348	0.0555
		0.6	1.1800	0		0.6	1.0740	0.0007		0.6	4.6594	0.0995
		0.7	1.2450	0		0.7	1.1201	0.0010		0.7	6.4430	0.1764
		0.8	1.3200	0		0.8	1.1847	0.0014		0.8	9.0814	0.3120
		0.9	1.4050	0		0.9	1.2730	0.0020		0.9	13.0463	0.5528
1.0	1.5000	0	1.0	1.3926	0.0030	1.0	19.1011	0.9845				

	t_i	w_i	error		t_i	w_i	error		t_i	w_i	error
	0.0	1.0000	0.0000		0.0	1.0000	0.0000		0.0	1.0000	0.0000
	0.1	1.0000	0.0000		0.1	1.0907	0.0007		0.1	1.0000	0.0000
	0.2	1.0003	0.0001		0.2	1.1686	0.0010		0.2	1.0003	0.0000
	0.3	1.0022	0.0002		0.3	1.2375	0.0011		0.3	1.0019	0.0001
(d)	0.4	1.0097	0.0005	(e)	0.4	1.2995	0.0011	(f)	0.4	1.0062	0.0002
	0.5	1.0306	0.0012		0.5	1.3561	0.0011		0.5	1.0151	0.0003
	0.6	1.0785	0.0024		0.6	1.4083	0.0011		0.6	1.0311	0.0003
	0.7	1.1778	0.0052		0.7	1.4570	0.0011		0.7	1.0564	0.0003
	0.8	1.3754	0.0124		0.8	1.5026	0.0011		0.8	1.0931	0.0003
	0.9	1.7711	0.0338		0.9	1.5456	0.0010		0.9	1.1426	0.0001
	1.0	2.6107	0.1076		1.0	1.5864	0.0010		1.0	1.2051	0.0001

6.6 Exercises

- (a) $w = [0, 0.0833, 0.2778, 0.6204, 1.1605]$, error = 0.4422
 (b) $w = [0, 0.0500, 0.1400, 0.2620, 0.4096]$, error = 0.0417
 (c) $w = [0, 0.1667, 0.4444, 0.7963, 1.1975]$, error = 0.0622

6.6 Computer Problems

- (a) $y = 1$, Euler step size ≤ 1.8 (b) $y = 1$, Euler step size $\leq 1/3$

6.7 Exercises

- (a) $w = [1.0000, 1.0313, 1.1250, 1.2813, 1.5000]$, error = 0
 (b) $w = [1.0000, 1.0078, 1.0314, 1.1203, 1.3243]$, error = 0.0713
 (c) $w = [1.0000, 1.7188, 3.0801, 6.0081, 12.7386]$, error = 7.3469
 (d) $w = [1.0000, 1.0024, 1.0098, 1.1257, 1.7540]$, error = 0.9642
 (e) $w = [1.0000, 1.2050, 1.3383, 1.4616, 1.5673]$, error = 0.0201
 (f) $w = [1.0000, 1.0020, 1.0078, 1.0520, 1.1796]$, error = 0.0255
- $w_{i+1} = -4w_i + 5w_{i-1} + h[4f_i + 2f_{i-1}]$; No.
- (a) $0 < a_1 < 2$ (b) $a_1 = 0$
- (a) second order unstable (b) second order strongly stable (c) third order strongly stable (d) third order unstable (e) third order unstable
- For example, $a_1 = 0, a_2 = 1, b_1 = 2 - 2b_0, b_2 = b_0$, where $b_0 \neq 0$ is arbitrary.
- (a) $a_1 + a_2 + a_3 = 1, -a_2 - 2a_3 + b_1 + b_2 + b_3 = 1, a_2 + 4a_3 - 2b_2 - 4b_3 = 1, -a_2 - 8a_3 + 3b_2 + 12b_3 = 1$ (c) $P(x) = x^3 - x^2$ has double root at 0, simple root at 1.
 (d) $w_{i+1} = w_{i-1} + h[\frac{7}{3}f_i - \frac{2}{3}f_{i-1} + \frac{1}{3}f_{i-2}]$
- (a) $a_1 + a_2 + a_3 = 1, -a_2 - 2a_3 + b_0 + b_1 + b_2 + b_3 = 1, a_2 + 4a_3 + 2b_0 - 2b_2 - 4b_3 = 1, -a_2 - 8a_3 + 3b_0 + 3b_2 + 12b_3 = 1, a_2 + 16a_3 + 4b_0 - 4b_2 - 32b_3 = 1$ (c) $P(x) = x^3 - x^2 = x^2(x - 1)$ has simple root at 1.

6.7 Computer Problems

1.

	t_i	w_i	error		t_i	w_i	error		t_i	w_i	error
(a)	0.0	1.0000	0	(b)	0.0	1.0000	0.0000	(c)	0.0	1.0000	0.0000
	0.1	1.0050	0		0.1	1.0005	0.0002		0.1	1.2320	0.0017
	0.2	1.0200	0		0.2	1.0020	0.0007		0.2	1.5386	0.0141
	0.3	1.0450	0		0.3	1.0075	0.0015		0.3	1.9569	0.0368
	0.4	1.0800	0		0.4	1.0191	0.0025		0.4	2.5355	0.0762
	0.5	1.1250	0		0.5	1.0390	0.0035		0.5	3.3460	0.1443
	0.6	1.1800	0		0.6	1.0698	0.0048		0.6	4.4967	0.2621
	0.7	1.2450	0		0.7	1.1146	0.0065		0.7	6.1533	0.4661
	0.8	1.3200	0		0.8	1.1773	0.0088		0.8	8.5720	0.8214
	0.9	1.4050	0		0.9	1.2630	0.0121		0.9	12.1548	1.4443
1.0	1.5000	0	1.0	1.3788	0.0168	1.0	17.5400	2.5455			

	t_i	w_i	error		t_i	w_i	error		t_i	w_i	error
(d)	0.0	1.0000	0.0000	(e)	0.0	1.0000	0.0000	(f)	0.0	1.0000	0.0000
	0.1	1.0000	0.0000		0.1	1.0913	0.0001		0.1	1.0001	0.0000
	0.2	1.0001	0.0002		0.2	1.1673	0.0023		0.2	1.0002	0.0002
	0.3	1.0013	0.0012		0.3	1.2354	0.0032		0.3	1.0013	0.0007
	0.4	1.0070	0.0033		0.4	1.2970	0.0036		0.4	1.0050	0.0014
	0.5	1.0243	0.0075		0.5	1.3534	0.0038		0.5	1.0131	0.0022
	0.6	1.0658	0.0150		0.6	1.4055	0.0039		0.6	1.0282	0.0032
	0.7	1.1534	0.0296		0.7	1.4542	0.0039		0.7	1.0528	0.0039
	0.8	1.3266	0.0611		0.8	1.4998	0.0039		0.8	1.0890	0.0044
	0.9	1.6649	0.1400		0.9	1.5428	0.0038		0.9	1.1383	0.0044
1.0	2.3483	0.3700	1.0	1.5836	0.0038	1.0	1.2011	0.0040			

3.

	t_i	w_i	error		t_i	w_i	error		t_i	w_i	error
(a)	0.0	0.0000	0.0000	(b)	0.0	0.0000	0.0000	(c)	0.0	0.0000	0.0000
	0.1	0.0050	0.0002		0.1	0.0050	0.0002		0.1	0.0200	0.0013
	0.2	0.0213	0.0002		0.2	0.0187	0.0000		0.2	0.0700	0.0003
	0.3	0.0493	0.0005		0.3	0.0413	0.0005		0.3	0.1530	0.0042
	0.4	0.0916	0.0002		0.4	0.0699	0.0004		0.4	0.2435	0.0058
	0.5	0.1474	0.0013		0.5	0.1082	0.0016		0.5	0.3855	0.0176
	0.6	0.2222	0.0001		0.6	0.1462	0.0027		0.6	0.4645	0.0367
	0.7	0.3105	0.0032		0.7	0.2032	0.0066		0.7	0.7356	0.0890
	0.8	0.4276	0.0020		0.8	0.2360	0.0134		0.8	0.5990	0.2029
	0.9	0.5510	0.0086		0.9	0.3363	0.0297		0.9	1.4392	0.4739
1.0	0.7283	0.0100	1.0	0.3048	0.0631	1.0	0.0394	1.0959			

CHAPTER 7

7.1 Exercises

3. (a) $\sin 2t, \cos 2t$ (b) $y_a - y_b = 0$ (c) $y_a + y_b = 0$ (d) no condition, solution always exists

$$5. y(t) = \frac{y_1 - e^{-\sqrt{k}} y_0}{e^{\sqrt{k}} - e^{-\sqrt{k}}} e^{\sqrt{k}t} + \frac{e^{\sqrt{k}} y_0 - y_1}{e^{\sqrt{k}} - e^{-\sqrt{k}}} e^{-\sqrt{k}t}$$

7.1 Computer Problems

1. (a) $y(t) = 1/3te^t$ (b) $y(t) = e^{t^2}$
3. (a) $y(t) = 1/(3t^2)$ (b) $y(t) = \ln(t^2 + 1)$
5. (a) $s = y_2(0) = 1$, exact solution is $y_1(t) = \arctan t$, $y_2 = t^2 + 1$ (b) $s = y_2(0) = 1/3$, exact solution is $y_1(t) = e^{t^3}$, $y_2(t) = 1/3 - t^2$

7.2 Computer Problems

5. (a) $y(t) = \frac{e^{1+t} - e^{1-t}}{e^2 - 1}$

(c)

n	h	error
3	1/4	0.00026473
7	1/8	0.00006657
15	1/16	0.00001667
31	1/32	0.00000417
63	1/64	0.00000104
127	1/128	0.00000026

7. Extrapolate by $N_2(h) = (4N(h/2) - N(h))/3$ and $N_3(h) = (16N_2(h/2) - N_2(h))/15$ to arrive at estimate $y(1/2) \approx 0.443409442296$, error $\approx 3.11 \times 10^{-10}$.

11. 11.786

CHAPTER 8

8.1 Computer Problems

1. Approximate solution at representative points:

	$x = 0.2$	$x = 0.5$	$x = 0.8$		$x = 0.2$	$x = 0.5$	$x = 0.8$
(a) $t = 0.2$	3.0432	3.3640	3.9901	(b) $t = 0.2$	1.8219	2.4593	3.3199
$t = 0.5$	5.5451	6.1296	7.2705	$t = 0.5$	3.3198	4.4811	6.0492
$t = 0.8$	10.1039	11.1688	13.2477	$t = 0.8$	6.0490	8.1651	11.0224

Forward Difference Method is unstable on both parts for $h = 0.1$, $K > 0.003$.

3.

	h	k	$u(0.5, 1)$	$w(0.5, 1)$	error		h	k	$u(0.5, 1)$	$w(0.5, 1)$	error
(a)	0.02	0.02	16.6642	16.7023	0.0381	(b)	0.02	0.02	12.1825	12.2104	0.0279
	0.02	0.01	16.6642	16.6834	0.0192		0.02	0.01	12.1825	12.1965	0.0140
	0.02	0.005	16.6642	16.6738	0.0097		0.02	0.005	12.1825	12.1896	0.0071

5.

	h	k	$u(0.5, 1)$	$w(0.5, 1)$	error
(a)	0.02	0.02	16.664183	16.664504	0.000321
	0.01	0.01	16.664183	16.664263	0.000080
	0.005	0.005	16.664183	16.664203	0.000020
	h	k	$u(0.5, 1)$	$w(0.5, 1)$	error
(b)	0.02	0.02	12.182494	12.182728	0.000235
	0.01	0.01	12.182494	12.182553	0.000059
	0.005	0.005	12.182494	12.182509	0.000015

7. $C = \pi^2/100$

8.2 Computer Problems

1. Approximate solution at representative points:

	$x = 0.2$	$x = 0.5$	$x = 0.8$
(a) $t = 0.2$	-0.4755	-0.8090	-0.4755
$t = 0.5$	0.5878	1.0000	0.5878
$t = 0.8$	-0.4755	-0.8090	-0.4755

	$x = 0.2$	$x = 0.5$	$x = 0.8$
(b) $t = 0.2$	0.5489	0.4067	0.3012
$t = 0.5$	0.3012	0.2231	0.1653
$t = 0.8$	0.1652	0.1224	0.0907

	$x = 0.2$	$x = 0.5$	$x = 0.8$
(c) $t = 0.2$	0.3364	0.5306	0.6931
$t = 0.5$	0.5306	0.6930	0.8329
$t = 0.8$	0.6931	0.8329	0.9554

- 3.

	h	k	$w(1/4, 3/4)$	error
(a) 2^{-4}	2^{-6}		-0.70710678	0.0
2^{-5}	2^{-7}		-0.70710678	0.0
2^{-6}	2^{-8}		-0.70710678	0.0
2^{-7}	2^{-9}		-0.70710678	0.0
2^{-8}	2^{-10}		-0.70710678	0.0

	h	k	$w(1/4, 3/4)$	error
(b) 2^{-4}	2^{-5}		0.17367424	0.00009971
2^{-5}	2^{-6}		0.17374901	0.00002493
2^{-6}	2^{-7}		0.17376771	0.00000623
2^{-7}	2^{-8}		0.17377238	0.00000156
2^{-8}	2^{-9}		0.17377355	0.00000039

	h	k	$w(1/4, 3/4)$	error
(c) 2^{-4}	2^{-4}		0.69308400	0.00006318
2^{-5}	2^{-5}		0.69313136	0.00001582
2^{-6}	2^{-6}		0.69314323	0.00000396
2^{-7}	2^{-7}		0.69314619	0.00000099
2^{-8}	2^{-8}		0.69314693	0.00000025

8.3 Computer Problems

1. Approximate solution at representative points:

	$x = 0.2$	$x = 0.5$	$x = 0.8$
(a) $y = 0.2$	0.3151	0.5362	0.3151
$y = 0.5$	0.1236	0.2103	0.1236
$y = 0.8$	0.0482	0.0821	0.0482

	$x = 0.2$	$x = 0.5$	$x = 0.8$
(b) $y = 0.2$	0.4006	1.3686	3.6222
$y = 0.5$	0.6816	2.3284	6.1624
$y = 0.8$	0.4006	1.3686	3.6222

3. Approximate solution at representative points:

	$x = 0.2$	$x = 0.5$	$x = 0.8$
(a) $y = 0.2$	0.0347	0.0590	0.0347
$y = 0.5$	0.1185	0.2016	0.1185
$y = 0.8$	0.3136	0.5336	0.3136

	$x = 0.2$	$x = 0.5$	$x = 0.8$
(b) $y = 0.2$	0.4579	0.6752	0.8417
$y = 0.5$	0.6752	0.6708	0.6752
$y = 0.8$	0.8417	0.6752	0.4579

5. 11.4 meters

- 7.

	h	k	$w(1/4, 3/4)$	error
(a) 2^{-2}	2^{-2}		0.072692	0.005672
2^{-3}	2^{-3}		0.068477	0.001457
2^{-4}	2^{-4}		0.067387	0.000367
2^{-5}	2^{-5}		0.067112	0.000092

	h	k	$w(1/4, 3/4)$	error
(b) 2^{-2}	2^{-2}		0.673903	0.059660
2^{-3}	2^{-3}		0.629543	0.015300
2^{-4}	2^{-4}		0.618094	0.003851
2^{-5}	2^{-5}		0.615207	0.000964

11. Approximate solution at representative points:

	$x = 0.2$	$x = 0.5$	$x = 0.8$		$x = 0.2$	$x = 0.5$	$x = 0.8$
(a) $y = 0.2$	0.0631	0.1571	0.2493	(b) $y = 0.2$	1.0405	1.1046	1.1731
$y = 0.5$	0.1571	0.3839	0.5887	$y = 0.5$	1.1046	1.2830	1.4910
$y = 0.8$	0.2493	0.5887	0.8448	$y = 0.8$	1.1731	1.4910	1.8956

13. Approximate solution at representative points:

	$x = 1.25$	$x = 1.50$	$x = 1.75$		$x = 1.25$	$x = 1.50$	$x = 1.75$
(a) $y = 1.25$	3.1250	3.8125	4.6250	(b) $y = 0.50$	0.1999	0.1666	0.1428
$y = 1.50$	3.8125	4.5000	5.3125	$y = 1.00$	0.7999	0.6666	0.5714
$y = 1.75$	4.6250	5.3125	6.1250	$y = 1.50$	1.7999	1.4999	1.2857

15.

	h	k	$w(1/4, 3/4)$	error		h	k	$w(1/4, 3/4)$	error
(a) 2^{-2}	2^{-2}		0.294813	0.004528	(b) 2^{-2}	2^{-2}		1.202628	0.003602
2^{-3}	2^{-3}		0.291504	0.001219	2^{-3}	2^{-3}		1.205310	0.000920
2^{-4}	2^{-4}		0.290596	0.000311	2^{-4}	2^{-4}		1.205999	0.000231
2^{-5}	2^{-5}		0.290363	0.000078	2^{-5}	2^{-5}		1.206172	0.000058

8.4 Computer Problems

- Solution approaches $u = 0$.
- (a) Solution approaches $u = 0$ (b) Solution approaches $u = 2$

CHAPTER 9

9.1 Exercises

- (a) 4 (b) 9
- (a) 0.3 (b) 0.28

9.1 Computer Problems

- 0.000273, compared with correct volume ≈ 0.000268 .
- (The minimal standard LCG with seed 1 is used in the following answers:)

	n	Type 1 estimate	error		n	Type 2 estimate	error
(a) $1/3$	10^2	0.327290	0.006043	(c) $1/3$	10^2	0.28	0.053333
	10^3	0.342494	0.009161		10^3	0.354	0.020667
	10^4	0.332705	0.000628		10^4	0.3406	0.007267
	10^5	0.333610	0.000277		10^5	0.33382	0.000487
	10^6	0.333505	0.000172		10^6	0.333989	0.000656

- (a) $n = 10^4$: 0.5128, error = 0.010799; $n = 10^6$: 0.524980, error = 0.001381 (b) $n = 10^4$: 0.1744, error = 0.000133; $n = 10^6$: 0.174851, error = 0.000318
- (a) $1/12$ (b) 0.083566, error = 0.000232

9.2 Computer Problems

- | n | Type 1 estimate | error | n | Type 2 estimate | error |
|--------|-----------------|----------|--------|-----------------|----------|
| 10^2 | 0.335414 | 0.002080 | 10^2 | 0.35 | 0.016667 |
| 10^3 | 0.333514 | 0.000181 | 10^3 | 0.333 | 0.000333 |
| 10^4 | 0.333339 | 0.000006 | 10^4 | 0.3339 | 0.000567 |
| 10^5 | 0.333334 | 0.000001 | 10^5 | 0.33338 | 0.000047 |
1. (a) $1/3$ (b) 10^2 : 0.335414, error = 0.002080; 10^3 : 0.333514, error = 0.000181; 10^4 : 0.333339, error = 0.000006; 10^5 : 0.333334, error = 0.000001 (c) 10^2 : 0.35, error = 0.016667; 10^3 : 0.333, error = 0.000333; 10^4 : 0.3339, error = 0.000567; 10^5 : 0.33338, error = 0.000047
3. (a) $n = 10^4$: 0.5232, error = 0.000399; $n = 10^5$: 0.52396, error = 0.000361 (b) $n = 10^4$: 0.1743, error = 0.000233; $n = 10^5$: 0.17455, error = 0.000017
5. Typical results: Monte Carlo estimate 4.9656, error = 0.030798; quasi-Monte Carlo estimate 4.92928, error = 0.005522.
7. (a) exact value = $1/2$; $n = 10^6$ Monte Carlo estimate 0.500313 (b) exact value $4/9$; $n = 10^6$ Monte Carlo estimate 0.444486
9. $1/24 \approx 4.167\%$

9.3 Computer Problems

Answers in this section use the minimal standard LCG.

1. (a) Monte Carlo = 0.2907, error = 0.0050 (b) 0.6323, error 0.0073. (c) 0.7322, error 0.0049.
3. (a) 0.8199, error = 0.0014 (b) 0.9871, error = 0.0004 (c) 0.9984, error = 0.0006
5. (a) 0.2969, error = 0.0112 (b) 0.3939, error = 0.0049 (c) 0.4600, error = 0.0106
7. (a) 0.5848, error = 0.0207 (b) 0.3106, error = 0.0154 (c) 0.7155, error = 0.0107

9.4 Computer Problems

5. Typical results:

Δt	avg. error
10^{-1}	0.2657
10^{-2}	0.0925
10^{-3}	0.0256

The results show approximate order $1/2$.

Δt	avg. error
10^{-1}	0.1394
10^{-2}	0.0202
10^{-3}	0.0026

The results show approximate order 1.

CHAPTER 10

10.1 Exercises

1. (a) $[0, -i, 0, i]$ (b) $[2, 0, 0, 0]$ (c) $[0, i, 0, -i]$ (d) $[0, 0, -\sqrt{2}i, 0, 0, 0, \sqrt{2}i, 0]$
3. (a) $[1/2, 1/2, 1/2, 1/2]$ (b) $[1, 1, -1, 1]$ (c) $[1, 1, 1, -1]$ (d) $[2, -1, 2, -1, 2, -1, 2, -1]/\sqrt{2}$
5. (a) 4th roots of unity: $-i, -1, i, 1$; primitive: $-i, i$ (b) $\omega, \omega^2, \omega^3, \omega^4, \omega^5, \omega^6$ where $\omega = e^{-2\pi i/7}$ (c) $p - 1$
7. (a) $a_0 = a_1 = a_2 = 0, b_1 = -1$ (b) $a_0 = 2, a_1 = a_2 = 0, b_1 = 0$ (c) $a_0 = a_1 = a_2 = 0, b_1 = 1$
- (d) $b_2 = -\sqrt{2}, a_0 = a_1 = a_2 = a_3 = a_4 = b_1 = b_3 = 0$

10.2 Exercises

- (a) $P_4(t) = \sin 2\pi t$ (b) $P_4(t) = \cos 2\pi t + \sin 2\pi t$ (c) $P_4(t) = -\cos 4\pi t$ (d) $P_4(t) = 1$
- (a) $P_8(t) = \sin 4\pi t$ (b) $P_8(t) = 1 + \sin 4\pi t$ (c) $P_8(t) = \frac{1}{2} + \frac{1}{4} \cos 2\pi t + \frac{\sqrt{2}+1}{4} \sin 2\pi t + \frac{1}{4} \cos 6\pi t + \frac{\sqrt{2}-1}{4} \sin 6\pi t$ (d) $P_8(t) = \cos 8\pi t$

10.2 Computer Problems

- (a) $P_8(t) = \frac{7}{2} - \cos 2\pi t - (1 + \sqrt{2}) \sin 2\pi t - \cos 4\pi t - \sin 4\pi t - \cos 6\pi t + (1 - \sqrt{2}) \sin 6\pi t - \frac{1}{2} \cos 8\pi t$ (b) $P_8(t) = \frac{1}{2} - 0.8107 \cos 2\pi t - 0.1036 \sin 2\pi t + \cos 4\pi t + \frac{1}{2} \sin 4\pi t + 1.3107 \cos 6\pi t - 0.6036 \sin 6\pi t$ (c) $P_8(t) = \frac{5}{2} - \frac{1}{2} \cos \frac{\pi}{2} t - \frac{1}{2} \sin \frac{\pi}{2} t + \cos \pi t$ (d) $P_8(t) = \frac{5}{8} + \frac{3}{4} \cos \frac{\pi}{4} (t-1) + 1.3536 \sin \frac{\pi}{4} (t-1) - \frac{7}{4} \cos \frac{\pi}{2} (t-1) - \frac{5}{2} \sin \frac{\pi}{2} (t-1) + \frac{3}{4} \cos \frac{3\pi}{4} (t-1) - 0.6464 \sin \frac{3\pi}{4} (t-1) + \frac{5}{8} \cos \pi (t-1)$
- $P_8(t) = 1.6131 - 0.1253 \cos 2\pi t - 0.5050 \sin 2\pi t - 0.1881 \cos 4\pi t - 0.2131 \sin 4\pi t - 0.1991 \cos 6\pi t - 0.0886 \sin 6\pi t - 0.1007 \cos 8\pi t$
- $P_8(t) = 0.3423 - 0.1115 \cos 2\pi (t-1) - 0.2040 \sin 2\pi (t-1) - 0.0943 \cos 4\pi (t-1) - 0.0859 \sin 4\pi (t-1) - 0.0912 \cos 6\pi (t-1) - 0.0357 \sin 6\pi (t-1) - 0.0453 \cos 8\pi (t-1)$

10.3 Exercises

- (a) $F_2(t) = 0$ (b) $F_2(t) = \cos 2\pi t$ (c) $F_2(t) = 0$ (d) $F_2(t) = 1$
- (a) $F_4(t) = 0$ (b) $F_4(t) = 1$ (c) $F_4(t) = \frac{1}{2} + \frac{1}{4} \cos 2\pi t + \frac{\sqrt{2}+1}{4} \sin 2\pi t$ (d) $F_4(t) = 0$

10.3 Computer Problems

- (a) $F_2(t) = F_4(t) = 3 \cos 2\pi t$
- (b) $F_2(t) = 2 - \frac{3}{2} \cos 2\pi t$, $F_4(t) = 2 - \frac{3}{2} \cos 2\pi t - \frac{1}{2} \sin 2\pi t + \frac{3}{2} \cos 4\pi t$
- (c) $F_2(t) = \frac{7}{2} - \frac{1}{2} \cos \frac{\pi}{2} t$, $F_4(t) = \frac{7}{2} - \frac{1}{2} \cos \frac{\pi}{2} t + \frac{1}{2} \sin \frac{\pi}{2} t + 2 \cos \pi t$
- (d) $F_2(t) = 2 - 2 \cos \frac{\pi}{3} (t-1)$, $F_4(t) = 2 - 2 \cos \frac{\pi}{3} (t-1) - \cos \frac{2\pi}{3} (t-1)$

CHAPTER 11

11.1 Exercises

- The DCT matrix is $C = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, and $P_2(t) = \frac{1}{\sqrt{2}} y_0 + y_1 \cos \frac{(2t+1)\pi}{4}$
 - $y = [3\sqrt{2}, 0]$, $P_2(t) = 3$
 - $y = [0, 2\sqrt{2}]$, $P_2(t) = 2\sqrt{2} \cos \frac{(2t+1)\pi}{4}$
 - $y = [2\sqrt{2}, \sqrt{2}]$, $P_2(t) = 2 + \sqrt{2} \cos \frac{(2t+1)\pi}{4}$
 - $y = [3\sqrt{2}/2, 5\sqrt{2}/2]$, $P_2(t) = 3/2 + (5\sqrt{2}/2) \cos \frac{(2t+1)\pi}{4}$
- (a) $y = [1, b-c, 0, b+c]$, $P_4(t) = \frac{1}{2} + \left((b-c)/\sqrt{2} \right) \cos \frac{(2t+1)\pi}{8} + \left((b+c)/\sqrt{2} \right) \cos \frac{3(2t+1)\pi}{8}$
 - $y = [2, 0, 0, 0]$, $P_4(t) = 1$
 - $y = [1/2, b, 1/2, c]$, $P_4(t) = 1/2 + \left(b/\sqrt{2} \right) \cos \frac{(2t+1)\pi}{8} + (1/2\sqrt{2}) \cos \frac{2(2t+1)\pi}{8} + \left(c/\sqrt{2} \right) \cos \frac{3(2t+1)\pi}{8}$
 - $y = [5, -(c+3b), 0, (b-3c)]$, $P_4(t) = \frac{5}{2} - \left((c+3b)/\sqrt{2} \right) \cos \frac{(2t+1)\pi}{8} + \left((b-3c)/\sqrt{2} \right) \cos \frac{3(2t+1)\pi}{8}$

11.2 Exercises

1. (a) $Y = \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix}$, $P_2(s, t) = \frac{1}{4} + \frac{1}{2\sqrt{2}} \cos \frac{(2s+1)\pi}{4} + \frac{1}{2\sqrt{2}} \cos \frac{(2t+1)\pi}{4} + \frac{1}{2} \cos \frac{(2s+1)\pi}{4} \cos \frac{(2t+1)\pi}{4}$ (b) $Y = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$, $P_2(s, t) = \frac{1}{2} + \frac{1}{\sqrt{2}} \cos \frac{(2t+1)\pi}{4}$
- (c) $Y = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}$, $P_2(s, t) = 1$. (d) $Y = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $P_2(s, t) = \frac{1}{2} + \cos \frac{(2s+1)\pi}{4} \cos \frac{(2t+1)\pi}{4}$
3. (a) $P(t) = ((b+c)/\sqrt{2}) \cos \frac{(2t+1)\pi}{8}$ (b) $P(t) = 1/4$ (c) $P(t) = 1/4$
- (d) $P(t) = 2 + \sqrt{2}(b-c) \cos \frac{(2s+1)\pi}{8}$

11.2 Computer Problems

1. (a) $\begin{bmatrix} 0 & -3.8268 & 0 & -9.2388 \\ 0 & 1.7071 & 0 & 4.1213 \\ 0 & 0 & 0 & 0 \\ 0 & 0.1213 & 0 & 0.2929 \end{bmatrix}$ (b) $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2.1213 & -0.7654 & -0.8787 \\ 0 & 0 & 0 & 0 \\ 0 & 5.1213 & -1.8478 & -2.1213 \end{bmatrix}$
- (c) $\begin{bmatrix} 4.7500 & 1.4419 & 0.2500 & 0.2146 \\ -0.7886 & 0.5732 & -1.4419 & -1.0910 \\ 0.2500 & 2.6363 & -2.2500 & -0.8214 \\ 0.0560 & -2.0910 & -0.2146 & 0.9268 \end{bmatrix}$ (d) $\begin{bmatrix} 0 & -4.4609 & 0 & -0.3170 \\ -4.4609 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -0.3170 & 0 & 0 & 0 \end{bmatrix}$

11.3 Exercises

1. (a) $P(A) = 1/4$, $P(B) = 5/8$, $P(C) = 1/8$, 1.30 (b) $P(A) = 3/8$, $P(B) = 1/4$, $P(C) = 3/8$, 1.56
- (c) $P(A) = 1/2$, $P(B) = 3/8$, $P(C) = 1/8$, 1.41
3. (a) 34 bits needed, $34/11 = 3.09$ bits/symbol $> 3.03 =$ Shannon inf. (b) 73 bits needed, $73/21 = 3.48$ bits/symbol $> 3.42 =$ Shannon inf. (c) 108 bits needed, $108/35 = 3.09$ bits/symbol $> 3.04 =$ Shannon inf.

11.4 Exercises

1. (a) $[-12b - 2c, 2b - 12c]$ (b) $[-3b - c, b - 3c]$ (c) $[-8b + 5c, -5b - 8c]$
3. (a) +101., error = 0 (b) +101., error = $1/15$ (c) +011., error = $1/35$
5. (a) +0110000., error = $1/170$ (b) -0101101., error = $1/85$ (c) +1011100., error = $7/510$
- (d) +1100100., error ≈ 0.0043
7. (a) $\frac{1}{2}(w_2 + w_3) = [-1.2246, 0.9184] \approx [-1, 1]$ (b) $\frac{1}{2}(w_2 + w_3) = [2.1539, -0.9293] \approx [2, -1]$
- (c) $\frac{1}{2}(w_2 + w_3) = [-1.7844, -3.0832] \approx [-2, -3]$
9. $c_{5n} = -c_{n-1}$, $c_{6n} = -c_0$

CHAPTER 12

12.1 Exercises

1. (a) $P(\lambda) = (\lambda - 5)(\lambda - 2)$, 2 and $[1, 1]$, 5 and $[1, -1]$ (b) $P(\lambda) = (\lambda + 2)(\lambda - 2)$, -2 and $[1, -1]$, 2 and $[1, 1]$
- (c) $P(\lambda) = (\lambda - 3)(\lambda + 2)$, 3 and $[-3, 4]$, -2 and $[4, 3]$ (d) $P(\lambda) = (\lambda - 100)(\lambda - 200)$, 200 and $[-3, 4]$, 100 and $[4, 3]$

3. (a) $P(\lambda) = -(\lambda - 1)(\lambda - 2)(\lambda - 3)$, 3 and $[0, 1, 0]$, 2 and $[1, 2, 1]$, 1 and $[1, 0, 0]$
 (b) $P(\lambda) = -\lambda(\lambda - 1)(\lambda - 2)$, 2 and $[-1, 2, 3]$, 1 and $[1, 1, 0]$, 0 and $[1, -2, 3]$
 (c) $P(\lambda) = -\lambda(\lambda - 1)(\lambda + 1)$, 1 and $[1, -2, -3]$, 0 and $[1, -2, 3]$, -1 and $[1, 1, 0]$
5. (a) $\lambda = 4, S = 3/4$ (b) $\lambda = -4, S = 3/4$ (c) $\lambda = 4, S = 1/2$ (d) $\lambda = 10, S = 9/10$
7. (a) $\lambda = 1, S = 1/3$ (b) $\lambda = 1, S = 1/3$ (c) $\lambda = -1, S = 1/2$ (d) $\lambda = 9, S = 3/4$
9. (a) 5 and $[1, 2]$, -1 and $[-1, 1]$ (b) $u_1 = [1/\sqrt{17}, 4/\sqrt{17}]$, RQ = 1; $u_2 = [0.4903, 0.8716]$, RQ = 4.29;
 $u_3 = [0.4386, 0.8987]$, RQ = 5.08 (c) IPI converges to $\lambda = -1$. (d) IPI converges to $\lambda = 5$.
11. (a) 7 (b) 5 (c) $S = 6/7, S = 1/2$; IPI with $s = 4$ is faster.

12.1 Computer Problems

1. (a) converges to 4 and $[1, 1, -1]$ (b) converges to -4 and $[1, 1, -1]$ (c) converges to 4 and $[1, 1, -1]$
 (d) converges to 10 and $[1, 1, -1]$
3. (a) $\lambda = 4$ (b) $\lambda = 3$ (c) $\lambda = 2$ (d) $\lambda = 9$

12.2 Exercises

1. (a)
$$\begin{bmatrix} 1 & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\sqrt{2} & \frac{1}{2} & \frac{1}{2} \\ 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$
 (b)
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$
 (c)
$$\begin{bmatrix} 2 & -\frac{4}{5} & -\frac{3}{5} \\ -5 & \frac{37}{25} & -\frac{16}{25} \\ 0 & \frac{9}{25} & \frac{13}{25} \end{bmatrix}$$

(d)
$$\begin{bmatrix} 1 & -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\sqrt{8} & \frac{5}{2} & \frac{3}{2} \\ 0 & \frac{3}{2} & \frac{1}{2} \end{bmatrix}$$

5. (a) NSI fails: \overline{Q}_k does not converge, alternates with period of 2. (b) NSI fails: \overline{Q}_k does not converge, alternates with period of 2.
7. (a) before: does not converge; after: same (already in Hessenberg form) (b) before: does not converge; after: does not converge

12.2 Computer Problems

1. (a) $\{-6, 4, -2\}$ (b) $\{6, 4, 2\}$ (c) $\{20, 18, 16\}$ (d) $\{10, 2, 1\}$
3. (a) $\{3, 3, 3\}$ (b) $\{1, 9, 10\}$ (c) $\{3, 3, 18\}$ (d) $\{-2, 2, 0\}$
5. (a) $\{2, i, -i\}$ (b) $\{1, i, -i\}$ (c) $\{2 + 3i, 2 - 3i, 1\}$ (d) $\{5, 4 + 3i, 4 - 3i\}$

12.3 Exercises

1. (a)
$$\begin{bmatrix} -3 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

Expands by factor of 3 and flips along x -axis, expands by factor of 2 along y -axis.

(b)
$$\begin{bmatrix} 0 & 0 \\ 0 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Projects onto y axis and expands by 3 in y -direction.

(c)
$$\begin{bmatrix} \frac{3}{2} & -\frac{1}{2} \\ -\frac{1}{2} & \frac{3}{2} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Expands into ellipse with major axis of length 4 along the line $y = -x$.

$$(d) \begin{bmatrix} -\frac{3}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{3}{2} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \text{ Same as (c), but rotated } 180^\circ.$$

$$(e) \begin{bmatrix} \frac{3}{4} & \frac{5}{4} \\ \frac{5}{4} & \frac{3}{4} \end{bmatrix} = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} -\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Expands by factor of 2 along line $y = x$ and contracts by factor of 2 along line $y = -x$, and flips the points on the circle.

$$\begin{aligned} 3. \text{ Four: } \begin{bmatrix} 3 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\ &= \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \end{aligned}$$

12.4 Computer Problems

$$1. (a) \begin{bmatrix} 1.1708 & 1.8944 \\ 1.8944 & 3.0652 \end{bmatrix} \quad (b) \begin{bmatrix} 1.5607 & 3.7678 \\ 1.3536 & 3.2678 \end{bmatrix} \quad (c) \begin{bmatrix} 1.0107 & 2.5125 & 3.6436 \\ 0.9552 & 2.3746 & 3.4436 \\ 0.1787 & 0.4442 & 0.6441 \end{bmatrix}$$

$$(d) \begin{bmatrix} -0.5141 & 5.2343 & 1.9952 \\ 0.2070 & -2.1076 & -0.8033 \\ -0.1425 & 1.4510 & 0.5531 \end{bmatrix}$$

$$3. (a) \text{ Best line } y = 3.3028x; \text{ projections are } \begin{bmatrix} 1.1934 \\ 3.9415 \end{bmatrix}, \begin{bmatrix} 1.4707 \\ 4.8575 \end{bmatrix}, \begin{bmatrix} 1.2774 \\ 4.2188 \end{bmatrix}.$$

$$(b) \text{ Best line } y = 0.3620x; \text{ projections are } \begin{bmatrix} 1.7682 \\ 0.6402 \end{bmatrix}, \begin{bmatrix} 3.8565 \\ 1.3963 \end{bmatrix}, \begin{bmatrix} 3.2925 \\ 1.1921 \end{bmatrix}.$$

$$(c) \text{ Best line } (x(t), y(t), z(t)) = [0.3015, 0.3416, 0.8902]t; \text{ projections are } \begin{bmatrix} 1.3702 \\ 1.5527 \\ 4.0463 \end{bmatrix}, \begin{bmatrix} 1.8325 \\ 2.0764 \\ 5.4111 \end{bmatrix},$$

$$\begin{bmatrix} 1.8949 \\ 2.1471 \\ 5.5954 \end{bmatrix}, \begin{bmatrix} 0.9989 \\ 1.1319 \\ 2.9498 \end{bmatrix}.$$

5. See Exercise 12.3.2 answers.

CHAPTER 13

13.1 Exercises

$$1. (a) (0, 1) \quad (b) (0, 0) \quad (c) (-1/2, -3/8) \quad (d) (1, 1)$$

13.1 Computer Problems

$$1. (a) 1/2 \quad (b) -2, 1 \quad (c) 0.47033 \quad (d) 1.43791$$

$$3. (a), (b): (0.358555, 2.788973)$$

5. $(1.20881759, 1.20881759)$, about 8 correct places
7. $(1, 1)$

13.2 Computer Problems

1. Minimum is $(1.2088176, 1.2088176)$. Different initial conditions will yield answers that differ by about $\epsilon^{1/2}$.
3. $(1, 1)$. Newton's Method will be accurate to machine precision, since it is finding a simple root. Steepest Descent will have error of size $\approx \epsilon^{1/2}$.
5. same as Computer Problem 2

Bibliography

- Y. Achdou and O. Pironneau [2005] *Computational Methods for Options Pricing*. SIAM, Philadelphia, PA.
- A. Ackleh, E. J. Allen, R. B. Kearfott, and P. Seshaiyer [2009] *Classical and Modern Numerical Analysis: Theory, Methods, and Practice*. Chapman and Hall, New York.
- M. Agoston [2005] *Computer Graphics and Geometric Modeling*. Springer, New York.
- K. Alligood, T. Sauer, and J. A. Yorke [1996] *Chaos: An Introduction to Dynamical Systems*. Springer, New York.
- W. F. Ames [1992] *Numerical Methods for Partial Differential Equations*, 3rd ed. Academic Press, Boston.
- E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen [1990] “LAPACK: A Portable Linear Algebra Library for High-performance Computers,” Computer Science Dept. Technical Report CS-90-105, University of Tennessee, Knoxville.
- U. M. Ascher, R. M. Mattheij, and R. B. Russell [1995] *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. SIAM, Philadelphia, PA.
- U. M. Ascher and L. Petzold [1998] *Computer Methods for Ordinary Differential Equations and Differential-algebraic Equations*. SIAM, Philadelphia, PA.
- R. Ashino, M. Nagase, and R. Vaillancourt [2000] “Behind and Beyond the MATLAB ODE Suite.” *Computers and Mathematics with Application* **40**, 491–572.
- R. Aster, B. Borchers, and C. Thurber [2005] *Parameter Estimation and Inverse Problems*. Academic Press, New York.
- O. Axelsson [1994] *Iterative Solution Methods*. Cambridge University Press, New York.
- O. Axelsson and V. A. Barker [1984] *Finite Element Solution of Boundary Value Problems for Ordinary Differential Equations*. Academic Press, Orlando, FL.
- Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. Van der Vorst [2000] *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, PA.
- P. B. Bailey, L. F. Shampine, and P. E. Waltman [1968] *Nonlinear Two-Point Boundary-Value Problems*. Academic Press, New York.
- R. Bank [1998] “PLTMG, A Software Package for Solving Elliptic Partial Differential Equations”, *Users’ Guide 8.0*. SIAM, Philadelphia, PA.
- R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst [1987] *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA.
- V. Bhaskaran and K. Konstantinides [1995] *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, Boston, MA.
- G. Birkhoff and R. Lynch [1984] *Numerical Solution of Elliptic Problems*. SIAM, Philadelphia, PA.

- G. Birkhoff and G. Rota [1989] *Ordinary Differential Equations*, 4th ed. John Wiley & Sons, New York.
- F. Black and M. Scholes [1973] "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy* **81**, 637–654.
- P. Blanchard, R. Devaney, and G. R. Hall [2002] *Differential Equations*, 2nd ed. Brooks-Cole, Pacific Grove, CA.
- F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel [2004] *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*. SIAM, Philadelphia.
- W. E. Boyce and R. C. DiPrima [2008] *Elementary Differential Equations and Boundary Value Problems*, 9th ed. John Wiley & Sons, New York.
- G. E. P. Box and M. Muller [1958] "A Note on the Generation of Random Normal Deviates." *The Annals Mathematical Statistics* **29**, 610–611.
- R. Bracewell [2000] *The Fourier Transform and Its Application*, 3rd ed. McGraw-Hill, New York.
- J. H. Bramble [1993] *Multigrid Methods*. John Wiley & Sons, New York.
- K. Brandenburg and M. Bosi [1997] "Overview of MPEG Audio: Current and Future Standards for Low Bit Rate Audio Coding." *Journal of the Audio Engineering Society* **45**, 4–21.
- M. Braun [1993] *Differential Equations and Their Applications*, 4th ed. Springer-Verlag, New York.
- S. Brenner and L. R. Scott [2002] *The Mathematical Theory of Finite Element Methods*, 2nd ed. Springer Verlag, New York.
- R. P. Brent [1973] *Algorithms for Minimization without Derivatives*. Prentice Hall, Englewood Cliffs, NJ.
- W. Briggs [1987] *A Multigrid Tutorial*. SIAM, Philadelphia, PA.
- W. Briggs and V. E. Henson [1995] *The DFT: An Owner's Manual for the Discrete Fourier Transform*. SIAM, Philadelphia, PA.
- E. O. Brigham [1988] *The Fast Fourier Transform and Its Applications*. Prentice-Hall, Englewood Cliffs, NJ.
- S. Brin and L. Page [1998] "The Anatomy of a Large-scale Hypertextual Web Search Engine." *Computer Networks and ISDN systems* **30**, 107–117.
- C. G. Broyden [1965] "A Class of Methods for Solving Nonlinear Simultaneous Equations." *Mathematics of Computation* **19**, 577–593.
- C. G. Broyden, J. E. Dennis, Jr., and J. J. Moré [1973] "On the Local and Superlinear Convergence of Quasi-Newton Methods." *IMA Journal of Applied Mathematics* **12**, 223–245.
- K. Burrage [1995] *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford University Press, New York.
- J. C. Butcher [1987] *Numerical Analysis of Ordinary Differential Equations*. Wiley, London.
- E. Cheney [1966] *Introduction to Approximation Theory*. McGraw-Hill, New York.
- E. Chu and A. George [1999] *Inside the FFT Black Box*. CRC Press, Boca Raton, FL.

- P. G. Ciarlet [1978] *The Finite Element Method for Elliptic Problems*. North-Holland, Amsterdam.
- CODEE [1999] *ODE Architect Companion*. John Wiley & Sons, New York.
- T. F. Coleman and C. van Loan [1988] *Handbook for Matrix Computations*. SIAM, Philadelphia, PA.
- R. D. Cook [1995] *Finite Element Modeling for Stress Analysis*. Wiley, New York.
- J. W. Cooley and J. W. Tukey [1965] “An Algorithm for the Machine Calculation of Complex Fourier Series.” *Mathematics of Computation* **19**, 297–301.
- T. Cormen, C. Leiserson, R. Rivest and C. Stein [2009] *Introduction to Algorithms*, 3rd ed. MIT Press, Cambridge, MA.
- R. Courant, K. O. Friedrichs and H. Lewy [1928] “Über die Partiellen Differenzengleichungen der Mathematischen Physik.” *Mathematischen Annalen* **100**, 32–74.
- J. Crank and P. Nicolson [1947] “A Practical Method for Numerical Evaluation of Solutions of Partial Differential Equations of the Heat Conduction Type.” *Proceedings of the Cambridge Philosophical Society* **43**, 1–67.
- J. Cuppen [1981] “A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem.” *Numerische Mathematik* **36**, 177–195.
- B. Datta [2010] *Numerical Linear Algebra and Applications*, 2nd ed. SIAM, Philadelphia.
- A. Davies and P. Samuels [1996] *An Introduction to Computational Geometry for Curves and Surfaces*. Oxford University Press, Oxford.
- P. J. Davis [1975] *Interpolation and Approximation*. Dover, New York.
- P. Davis and P. Rabinowitz [1984] *Methods of Numerical Integration*, 2nd ed. Academic Press, New York.
- T. Davis [2006] *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA.
- C. de Boor [2001] *A Practical Guide to Splines*, 2nd ed. Springer-Verlag, New York.
- J. W. Demmel [1997] *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA.
- J. E. Dennis and Jr., R. B. Schnabel [1987] *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM Publications, Philadelphia, PA.
- C. S. Desai and T. Kundu [2001] *Introductory Finite Element Method*. CRC Press, Boca Raton, FL.
- P. Dierckx [1995] *Curve and Surface Fitting with Splines*. Oxford University Press, New York.
- J. R. Dormand [1996] *Numerical Methods for Differential Equations*. CRC Press: Boca Raton, FL.
- N. Draper and H. Smith [2001] *Applied Regression Analysis*, 3rd ed. John Wiley and Sons, New York.
- T. Driscoll [2009] *Learning MATLAB*. SIAM, Philadelphia, PA.
- P. Duhamel and M. Vetterli [1990] “Fast Fourier Transforms: A Tutorial Review and a State of the Art.” *Signal Processing* **19**, 259–299.

- C. Edwards and D. Penny [2004] *Differential Equations and Boundary Value Problems*, 5th ed. Prentice Hall, Upper Saddle River, NJ.
- H. Elman, D. J. Silvester and A. Wathen [2004] *Finite Elements and Fast Iterative Solvers*. Oxford University Press, Oxford, UK.
- H. Engels [1980] *Numerical Quadrature and Cubature*. Academic Press, New York.
- G. Evans [1993] *Practical Numerical Integration*. John Wiley and Sons, New York.
- L. C. Evans [2010] *Partial Differential Equations*, 2nd ed. AMS Publications, Providence, RI.
- G. Farin [1990] *Curves and Surfaces for Computer-aided Geometric Design*, 2nd ed. Academic Press, New York.
- G. S. Fishman [1996] *Monte Carlo: Concepts, Algorithms, and Applications*. Springer-Verlag, New York.
- C. A. Floudas, P. M. Pardalos, C. Adjiman, W. R. Esposito, Z. H. Gms, S. T. Harding, J. L. Klepeis, C. A. Meyer, and C. A. Schweiger [1999] *Handbook of Test Problems in Local and Global Optimization*, Vol. 33, Series titled Nonconvex Optimization and its Applications, Springer, Berlin, Germany.
- B. Fornberg [1998] *A Practical Guide to Pseudospectral Methods*. Cambridge University Press, Cambridge, UK.
- J. Fox [1997] *Applied Regression Analysis, Linear Models, and Related Methods*. Sage Publishing, New York.
- M. Frigo and S. G. Johnson [1998] “FFTW: An Adaptive Software Architecture for the FFT.” *Proceedings ICASSP* **3**, 1381–1384.
- C. W. Gear [1971] *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice-Hall, Englewood Cliffs, NJ.
- J. E. Gentle [2003] *Random Number Generation and Monte Carlo Methods*, 2nd ed. Springer-Verlag, New York.
- A. George and J. W. Liu [1981] *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, Englewood Cliff, NJ.
- M. Gockenbach [2006] *Understanding and Implementing the Finite Element Method*. SIAM, Philadelphia, PA.
- M. Gockenbach [2010] *Partial Differential Equations: Analytical and Numerical Methods*, 2nd ed. SIAM, Philadelphia, PA.
- D. Goldberg [1991] “What Every Computer Scientist Should Know about Floating Point Arithmetic.” *ACM Computing Surveys* **23**, 5–48.
- G. H. Golub and C. F. Van Loan [1996] *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore.
- D. Gottlieb and S. Orszag [1977] *Numerical Analysis of Spectral Methods: Theory and Applications*. SIAM, Philadelphia, PA.
- T. Gowers, J. Barrow-Green, and I. Leader [2008] *The Princeton Companion to Mathematics*. Princeton University Press, Princeton, NJ.
- I. Griva, S. Nash, and A. Sofer [2008] *Linear and Nonlinear Programming*, 2nd ed. SIAM, Philadelphia.

- C. Grossmann, H. Roos, and M. Stynes [2007] *Numerical Treatment of Partial Differential Equations*. Springer, Berlin, Germany.
- B. Guenter and R. Parent [1990] “Motion Control: Computing the Arc Length of Parametric Curves.” *IEEE Computer Graphics and Applications* **10**, 72–78.
- S. Haber [1970] “Numerical Evaluation of Multiple Integrals.” *SIAM Review* **12**, 481–526.
- R. Haberman [2004] *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*. Prentice Hall, Upper Saddle River, NJ.
- W. Hackbush [1994] *Iterative Solution of Large Sparse Systems of Equations*. Springer-Verlag, New York.
- S. Hacker [2000] *MP3: The Definitive Guide*. O’Reilly Publishing, Sebastopol, CA.
- B. Hahn [2002] *Essential MATLAB for Scientists and Engineers*, 3rd ed. Elsevier, Amsterdam.
- E. Hairer, S. P. Norsett, and G. Wanner [1993] *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd ed., Springer Verlag, Berlin.
- E. Hairer and G. Wanner [1996] *Solving Ordinary Differential Equations II: Stiff and Differential-algebraic Problems*, 2nd ed., Springer Verlag, Berlin.
- C. Hall and T. Porsching [1990] *Numerical Analysis of Partial Differential Equations*. Prentice Hall, Englewood Cliffs, NJ.
- J. H. Halton [1960] “On the Efficiency of Certain Quasi-Random Sequences of Points in Evaluating Multi-Dimensional Integrals.” *Numerische Mathematik* **2**, 84–90.
- M. Heath [2002] *Scientific Computing*, 2nd ed. McGraw-Hill, New York.
- P. Hellekalek [1998] “Good Random Number Generators Are (Not So) Easy to Find.” *Mathematics and Computers in Simulation* **46**, 485–505.
- P. Henrici [1962] *Discrete Variable Methods in Ordinary Differential Equations*. New York, John Wiley & Sons, New York.
- M. R. Hestenes and E. Steifel [1952] “Methods of Conjugate Gradients for Solving Linear Systems.” *Journal of Research National Bureau of Standards* **49**, 409–436.
- R. C. Hibbeler [2008] *Structural Analysis*, 7th ed. Prentice Hall, Englewood Cliffs, NJ.
- D. J. Higham [2001] “An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations.” *SIAM Review* **43**, 525–546.
- D. J. Higham and N. J. Higham [2005] *MATLAB Guide*, 2nd ed. SIAM, Philadelphia, PA.
- N. J. Higham [2002] *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM Publishing, Philadelphia, PA.
- B. Hoffmann-Wellenhof, H. Lichtenegger, and J. Collins [2001] *Global Positioning System: Theory and Practice*, 5th ed. Springer-Verlag, New York.
- J. Hoffman [2001] *Numerical Methods for Engineers and Scientists*, 2nd ed. CRC Press, New York.
- K. Höllig [2003] *Finite Element Methods with B-Splines*. SIAM, Philadelphia, PA.
- M. Holmes [2006] *Introduction to Numerical Methods in Differential Equations*. Springer, New York.

- M. Holmes [2009] *Introduction to the Foundations of Applied Mathematics*. Springer, New York.
- A. S. Householder [1970] *The Numerical Treatment of a Single Nonlinear Equation*. McGraw-Hill, New York.
- J. V. Huddleston [2000] *Extensibility and Compressibility in One-dimensional Structures*, 2nd ed. ECS Publishing, Buffalo, NY.
- D. A. Huffman [1952] "A Method for the Construction of Minimum-Redundancy Codes." *Proceedings of the IRE* **40**, 1098–1101.
- J. C. Hull [2008] *Options, Futures, and Other Derivatives*, 7th ed. Prentice Hall, Upper Saddle River, NJ.
- IEEE [1985] Standard for Binary Floating Point Arithmetic, IEEE Std. 754-1985, IEEE, New York.
- I. Ipsen [2009] *Numerical Matrix Analysis: Linear Systems and Least Squares*. SIAM, Philadelphia, PA.
- A. Iserles [1996] *A First Course in the Numerical Analysis of Differential Equations*, Cambridge University Press, Cambridge, UK.
- C. Johnson [2009] *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Dover Publications, New York.
- P. Kattan [2007] *MATLAB Guide to Finite Elements*, 2nd ed. Springer, New York.
- H. B. Keller [1968] *Numerical Methods of Two-Point Boundary-Value Problems*. Blaisdell, Waltham, MA.
- C. T. Kelley [1995] *Iterative Methods for Linear and Nonlinear Problems*. SIAM Publications, Philadelphia, PA.
- J. Kepner [2009] *Parallel MATLAB for Multicore and Multinode Computers*. SIAM, Philadelphia, PA.
- F. Klebaner [1998] *Introduction to Stochastic Calculus with Applications*. Imperial College Press, London.
- P. Kloeden and E. Platen [1992] *Numerical Solution of Stochastic Differential Equations*. Springer-Verlag, Berlin, Germany.
- P. Kloeden, E. Platen, and H. Schurz [1994] *Numerical Solution of SDE through Computer Experiments*. Springer-Verlag, Berlin, Germany.
- P. Knaber, and L. Angerman [2003] *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*. Springer, Berlin, Germany.
- D. Knuth [1981] *The Art of Computer Programming*. Addison-Wesley, Reading, MA.
- D. Knuth [1997] *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, Reading, MA.
- E. Kostelich and D. Armbruster [1997] *Introductory Differential Equations: From Linearity to Chaos*. Addison Wesley, Boston, MA.
- A. Krommer and C. Ueberhuber [1998] *Computational Integration*. SIAM, Philadelphia, PA.
- M. Kutner, C. Nachtsheim, J. Neter, and W. Li [2004] *Applied Linear Statistical Models*, 5th ed. McGraw-Hill, New York.

- J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright [1998] “Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions.” *SIAM Journal of Optimization* **9**, 112–147.
- J. D. Lambert [1991] *Numerical Methods for Ordinary Differential Systems*, John Wiley & Sons, New York.
- L. Lapidus and G. F. Pinder [1982] *Numerical Solution of Partial Differential Equations in Science and Engineering*. Wiley-Interscience, New York.
- S. Larsson and V. Thomee [2008] *Partial Differential Equations with Numerical Methods*. Springer, Berlin, Germany.
- C. L. Lawson and R. J. Hanson [1995] *Solving Least Squares Problems*. SIAM Publications, Philadelphia, PA.
- D. Lay [2011] *Linear Algebra and Its Applications*, 4th ed. Pearson Education, Boston, MA.
- K. Levenberg [1944] “A Method for the Solution of Certain Nonlinear Problems in Least Squares.” *The Quarterly of Applied Mathematics* **2**, 164–168.
- R. Leveque [2007] *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, Philadelphia, PA.
- J. D. Logan [2004] *Applied Partial Differential Equations*, 2nd ed. Springer, New York.
- D. L. Logan [2011] *A First Course in the Finite Element Method*, 5th ed. CL-Engineering, New York.
- H. S. Malvar [1992] *Signal Processing with Lapped Transforms*. Artech House, Norwood, MA.
- D. Marquardt [1963] “An Algorithm for Least-Squares Estimation of Nonlinear Parameters.” *SIAM J. on Applied Mathematics* **11**, 431–441.
- G. Marsaglia [1968] “Random Numbers Fall Mainly in the Planes.” *Proceedings of the National Academy of Sciences* **61**, 25.
- G. Marsaglia and A. Zaman [1991] “A New Class of Random Number Generators.” *Annals of Applied Probability* **1**, 462–480.
- G. Marsaglia and W. W. Tsang [2000] “The Ziggurat Method for Generating Random Variables,” *Journal of Statistical Software* **5**, 1–7.
- R. McDonald [2006] *Derivatives Markets*, 2nd ed. Pearson Education, Boston, MA.
- P. J. McKenna and C. Tuama [2001] “Large Torsional Oscillations in Suspension Bridges Visited Again: Vertical Forcing Creates Torsional Response.” *American Mathematical Monthly* **108**, 738–745.
- J.-P. Merlet [2000] *Parallel Robots*. Kluwer Academic Publishers, London.
- A. R. Mitchell and D. F. Griffiths [1980] *The Finite Difference Method in Partial Differential Equations*. Wiley, New York.
- C. Moler [2004] *Numerical Computing with MATLAB*. SIAM, Philadelphia, PA.
- J. Moré and S. Wright [1987] *Optimization Software Guide*. SIAM, Philadelphia, PA.
- K. W. Morton and D. F. Mayers [1996] *Numerical Solution of Partial Differential Equations*, Cambridge University Press, Cambridge, UK.

- J. A. Nelder and R. Mead [1965] "A Simplex Method for Function Minimization." *Computer Journal* **7**, 308–313.
- M. Nelson and J. Gailly [1995] *The Data Compression Book*, 2nd ed. M&T Books, Redwood City, CA.
- H. Niederreiter [1992] *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM Publications, Philadelphia, PA.
- J. Nocedal and S. Wright [1999] *Numerical Optimization*, Springer Series in Operations Research. Springer, New York.
- B. Oksendal [1998] *Stochastic Differential Equations: An Introduction with Applications*, 5th ed. Springer-Verlag, Berlin, Germany.
- A. Oppenheim and R. Schaffer [2009] *Discrete-time Signal Processing*, 3rd ed. Prentice Hall, Upper Saddle River, NJ.
- J. M. Ortega [1972] *Numerical Analysis: A Second Course*. Academic Press, New York.
- A. M. Ostrowski [1966] *Solution of Equations and Systems of Equations*, 2nd ed. Academic Press, New York.
- M. Overton [2001] *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM Publishing, Philadelphia, PA.
- S. Park and K. Miller [1988] "Random Number Generators: Good Ones Are Hard to Find." *Communications of the ACM* **31**, 1192–1201.
- B. Parlett [1998] *The Symmetric Eigenvalue Problem*. SIAM, Philadelphia, PA.
- B. Parlett [2000] "The QR Algorithm." *Computing in Science and Engineering* **2**, 38–42.
- W. Pennebaker and J. Mitchell [1993] *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York.
- R. Piessens, E. de Doncker-Kapenga, C. Ueberhuber, and D. Kahaner [1983] *QUADPACK: A Subroutine Package for Automatic Integration*, Springer, New York.
- G. Pinski and F. Narin [1976] "Citation Influence for Journal Aggregates of Scientific Publications: Theory, with Application to the Literature of Physics." *Information Processing and Management* **12**, 297–312.
- J. Polking [1999] *Ordinary Differential Equations Using MATLAB*. Prentice Hall, Upper Saddle River NJ.
- H. Prautzsch, W. Boehm, and M. Paluszny [2002] *Bézier and B-Spline Techniques*. Springer, Berlin, Germany.
- A. Quarteroni, R. Sacco, and F. Saleri [2000] *Numerical Mathematics*. Springer, Berlin, Germany.
- K. R. Rao and J. J. Hwang [1996] *Techniques and Standards for Image, Video, and Audio Coding*. Prentice Hall, Upper Saddle River, NJ.
- K. R. Rao and P. Yip [1990] *Discrete Cosine Transform: Algorithms, Advantages, Applications*. Academic Press, Boston, MA.
- J. R. Rice and R. F. Boisvert [1984] *Solving Elliptic Problems Using ELLPACK*. Springer Verlag, New York.
- T. J. Rivlin [1981] *An Introduction to the Approximation of Functions*, 2nd ed. Dover, New York.

- T. J. Rivlin [1990] *Chebyshev Polynomials*, 2nd ed. John Wiley and Sons, New York.
- S. Roberts and J. Shipman [1972] *Two-Point Boundary Value Problems: Shooting Methods*. Elsevier, New York.
- R. Y. Rubinstein [1981] *Simulation and the Monte Carlo Method*. John Wiley, New York.
- T. Ryan [1997] *Modern Regression Methods*. John Wiley and Sons.
- Y. Saad [2003] *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM Publishing, Philadelphia, PA.
- D. Salomon [2005] *Curves and Surfaces for Computer Graphics*. Springer, New York.
- K. Sayood [1996] *Introduction to Data Compression*. Morgan Kaufmann Publishers, San Francisco.
- M. H. Schultz [1973] *Spline Analysis*. Prentice Hall, Englewood Cliffs, NJ.
- L. L. Schumaker [1981] *Spline Functions: Basic Theory*. John Wiley, New York.
- L. F. Shampine [1994] *Numerical Solution of Ordinary Differential Equations*. Chapman & Hall, New York.
- L. F. Shampine, I. Gladwell, and S. Thompson [2003] *Solving ODEs with MATLAB*. Cambridge University Press, Cambridge, UK.
- L. F. Shampine and M. W. Reichelt [1997] "The MATLAB ODE Suite." *SIAM Journal on Scientific Computing* **18**, 1–22.
- K. Sigmon and T. Davis [2002] *MATLAB Primer*, 6th ed. CRC Press, Boca Raton, FL.
- S. Skiena [2008] *The Algorithm Design Manual*, 2nd ed. Springer, New York.
- I. Smith and D. Griffiths [2004] *Programming the Finite Element Method*. John Wiley, New York.
- B. T. Smith, J. M. Boyle, Y. Ikebe, V. Klema, and C. B. Moler [1970] *Matrix Eigensystem Routines: EISPACK Guide*, 2nd ed. Springer-Verlag, New York.
- W. Stallings [2003] *Computer Organization and Architecture*, 6th ed. Prentice Hall, Upper Saddle River, NJ.
- J. M. Steele [2001] *Stochastic Calculus and Financial Applications*. Springer-Verlag, New York.
- G. W. Stewart [1973] *Introduction to Matrix Computations*. Academic Press, New York.
- G. W. Stewart [1998] *Afternotes on Numerical Analysis: Afternotes Goes to Graduate School*. SIAM, Philadelphia, PA.
- J. Stoer and R. Bulirsch [2002] *Introduction to Numerical Analysis*, 3rd ed. Springer-Verlag, New York.
- J. A. Storer [1988] *Data Compression: Methods and Theory*. Computer Science Press, Rockville, MD.
- G. Strang [1988] *Linear Algebra and Its Applications*, 3rd ed. Saunders, Philadelphia.
- G. Strang [2007] *Computational Science and Engineering*. Wellesley-Cambridge Press, Cambridge, MA.
- G. Strang and K. Borre [1997] *Linear Algebra, Geodesy, and GPS*. Wellesley Cambridge Press, Cambridge, MA.
- G. Strang and G. J. Fix [1973] *An Analysis of the Finite Element Method*. Prentice-Hall, Englewood Cliffs, NJ.

- J. C. Strikwerda [1989] *Finite Difference Schemes and Partial Differential Equations*. Wadsworth and Brooks-Cole, Pacific Grove, CA.
- W. A. Strauss [1992] *Partial Differential Equations: An Introduction*. John Wiley and Sons, New York.
- A. Stroud and D. Secrest [1966] *Gaussian Quadrature Formulas*, Prentice Hall, Englewood Cliffs, NJ.
- P. N. Swarztrauber [1982] “Vectorizing the FFTs.” In: *Parallel Computations*, ed. G. Rodrigue, pp. 51–83. Academic Press, New York.
- D. S. Taubman and M. W. Marcellin [2002] *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Kluwer, Boston, MA.
- J. Traub [1964] *Iterative Methods for the Solution of Equations*. Prentice-Hall, Englewood Cliffs, NJ.
- N. Trefethen [2000] *Spectral Methods in MATLAB*. SIAM, Philadelphia.
- N. Trefethen and D. Bau [1997] *Numerical Linear Algebra*. SIAM, Philadelphia, PA.
- A. Turing [1952] “The Chemical Basis of Morphogenesis.” *Philosophical Transactions Royal of the Society Lond. B* **237**, 3772.
- C. Van Loan [1992] *Computational Frameworks for the Fast Fourier Transform*. SIAM Publications, Philadelphia, PA.
- C. Van Loan and K. Fan [2010] *Insight Through Computing: A MATLAB Introduction to Computational Science and Engineering*. SIAM, Philadelphia, PA.
- R. S. Varga [2000] *Matrix Iterative Analysis*, 2nd ed. Springer-Verlag, New York.
- J. Volder [1959] “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computing* **8**, 330–334.
- G. K. Wallace [1991] “The JPEG Still Picture Compression Standard.” *Communications of the ACM* **34**, 30–44.
- H. Wang, J. Kearney, and K. Atkinson [2003] “Arc-length Parameterized Spline Curves for Real-time Simulation.” In: *Curve and Surface Design: Saint Malo 2002*, Eds. T. Lyche, M. Mazure, and L. Schumaker. Nashboro Press, Brentwood, TN.
- Y. Wang and M. Vilermo [2003] “The Modified Discrete Cosine Transform: Its Implications for Audio Coding and Error Concealment.” *Journal of the Audio Engineering Society* **51**, 52–62.
- D. S. Watkins [1982] “Understanding the QR Algorithm.” *SIAM Review* **24**, 427–440.
- D. S. Watkins [2007] *The Matrix Eigenvalue Problem: GR and Krylow Subspace Methods*. SIAM, Philadelphia.
- J. Wilkinson [1965] *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford.
- J. Wilkinson [1984] “The Perfidious Polynomial.” In: *Studies in Numerical Analysis*, Ed: G. Golub. MAA, Washington, DC.
- J. Wilkinson [1994] *Rounding Errors in Algebraic Processes*. Dover, New York.
- J. Wilkinson and C. Reinsch [1971] *Handbook for Automatic Computation, Vol. 2: Linear Algebra*. Springer-Verlag, New York.
- P. Wilmott, S. Howison, and J. Dewynne [1995] *The Mathematics of Financial Derivatives*. Cambridge University Press, Oxford and New York.

- S. Winograd [1978] “On Computing the Discrete Fourier Transform.” *Mathematics of Computation* **32**, 175–199.
- F. Yamaguchi [1988] *Curves and Surfaces in Computer-aided Geometric Design*. Springer-Verlag, New York.
- D. M. Young [1971] *Iterative Solution of Large Linear Systems*. Academic Press, New York.

Index

- 2-norm, 192, 198
- AC component, 517
- Adams-Bashforth Method, 336, 339, 341
- Adams-Moulton Method, 342, 345
- Adaptive Quadrature, **269, 270**
- Adobe Corp., 138
- algorithm
 - stable, 50
- Apple Corp., 138
- arbitrage theory, 464
- arc length integral, 243
- arcsine law, 452
- atomic clock, 239
- audio file
 - aac, 495
 - mp3, 496
 - wav, 490, 529
- B-spline, 408
 - piecewise-linear, **369**
- Bézier curve, **179, 279**
 - in PDF file, 183
- Bézier, P., 138, 179
- Babylonian mathematics, 39
- back-substitution, 73, 76, 77, 83
- backsolving, *see* back-substitution
- Backward Difference Method, 380
- Backward Euler Method, **333**
- barrier option, 465
- barycenter, 409
- base 60, 39
- base points, 143
- basis
 - orthonormal, 539, 554
- beam
 - Timoshenko, 105
- bell curve, 438
- bifurcation
 - buckling, 356
- binary number, 5
 - infinitely repeating, 7
- Bisection Method, **25**, 44, 46, 51, 65, 69, 352, 354, 364
 - efficiency, 28
 - stopping criterion, 29
- bit, 6
- Black, F., 431, 464
- Black-Scholes formula, 431, 464
- Bogacki-Shampine Method, 327
- Boole's Rule, 264
- boundary conditions
 - convective, 405
 - Dirichlet, 383, **398**
 - homogeneous, 383
 - Neumann, 383, **398**
 - Robin, 405
- boundary value problem, 348
 - existence and uniqueness of solutions, 350
 - for systems, 353
 - nonlinear, 360
- Box-Muller method, 438
- bracket, 38, 62
- bracketing, 25
- Brent's Method, **64**, 69
- Brownian bridge, 461
- Brownian motion, 456
 - continuous, 450
 - discrete, 446
 - geometric, 464
- Broyden's Method, **134**, 357, 585
- Brusselator model, 426
- buckling
 - of circular ring, 348, 355
- Buffon needle, 445
- bulk temperature, 404
- Burgers' equation, 417, 419
- BVP, *see* boundary value problem
- byte, 11
- call option, 464
- cantilever, 71
- carbon dioxide, 150, 178, 211
- castanets.wav, 490, 492
- Casteljau, P., 138, 179
- Cauchy-Schwarz inequality, 198
- centered-difference formula, 376
- Central Limit Theorem, 450
- CFL condition, **396**
- chaotic attractor, 320
- chaotic dynamics, 43, 60
- characteristic function, 435
- characteristic polynomial, 532
- Chebyshev interpolation, **162**
- Cholesky factorization, 121
- chopping, 9
- cobweb diagram, 34, **34**, 42

- codec, 526
- Collocation Method
 - for BVP, 365
- color image
 - RGB, 505
 - YUV, **512**
- column vector, 583
- completing the square, 117
- complex number, **468**
 - polar representation, **468**
- compressibility, 355
- compression, 194
 - image, 561
 - lossy, 508, 514, 559
- computational neuroscience, 317
- computer animation, 243
- computer arithmetic, 45
- computer word, 8
- computer-aided manufacturing, 243
- computer-aided modeling, 278
- condition number, 50, **50**, 88, 197, 289, 532
- conditioning
 - normal equations, 197
- conduction, 403
- conic section, 311
- conjugate
 - of a complex number, **468**
- Conjugate Gradient Method, 122, 127
 - preconditioned, 127
- convection, 403
- convective heat transfer, 404
- convergence, 33
 - linear, **35**, 37, 40, 55
 - local, **36**, 53, 56, 57
 - quadratic, **53**, 57
 - superlinear, 61, 135
- conversion
 - binary to decimal, 7
 - decimal to binary, 6
- convex set, **288**
- Cooley, J., 473
- cooling fin, 403
- CORDIC, 165
- Crank-Nicolson Method, 254, 385
 - stability, 387
- cube root, 30
- cubic spline, **167**
 - clamped, 174
 - curvature-adjusted, 173
 - end conditions, 169
 - MATLAB default, 175
 - natural, 169
 - not-a-knot, 175
 - parabolically-terminated, 174
- cumulative distribution function, 437
- cuneiform, 39
- Dahlquist criterion, 341
- data
 - automobile supply, 204
 - height vs. weight, 207
 - Intel CPU, 205
 - Japan oil consumption, 210
 - temperature, 201
- data compression, 138
- data-fitting, 188
- DC component, 504, 517
- decimal number, 5
- decimal places
 - correct within, **28**
- deflation, 543
- degree of precision, **258**, **273**
- demand curve, 199
- derivative, 244
 - symbolic, 250
- determinant, 30, 557
- differential equation, 281
 - autonomous, **282**
 - first-order linear, 291
 - ordinary, 282
 - partial, 374
 - stiff, **333**
 - stochastic, 452
- differentiation
 - numerical, 244
- differentiation formula
 - centered difference, 246, 358
 - forward difference, 245
- diffusion, 453
- diffusion coefficient, 375
- dimension reduction, 559
- direct kinematics problem, *see* forward kinematics problem
- direct method, 106
- direction field, **282**
- direction vector, 309
- Discrete Cosine Transform, 495
 - one-dimensional, **496**
 - inverse, **497**
 - two-dimensional, **502**
 - inverse, **502**
 - version 4, 520
- Discrete Fourier Transform, **471**
 - inverse, **471**

- discretization, 71, 102, 357, 375
- divided differences, 141
- Dormand-Prince Method, 328
- dot product, 190
- dot product rule, 230
- double helix, 565
- double precision, 8, 43, 44, 92, 197
- downhill simplex method, 571
- DPCM tree, 517
- drift, 453
- DSP chip, 473

- eigenvalue, 30, 531, **586**
 - complex, 542
 - dominant, 539, 551
- eigenvector, 532
 - principal, 551
- electric field, 398
- electrostatic potential, 415
- ellipsoid, 554
- elliptic equation
 - weak form, 407
- engineering
 - structural, 71, 83
- equation
 - diffusion, 375
 - reaction-diffusion, 390, 421
- equations
 - inconsistent, **189**
- equilibrium solution, 334
- equipartition, 278
- error
 - absolute, **10**, 40
 - backward, **45**, 50, 86, 93
 - forward, **45**, 50, 86, 93, 197
 - global truncation, **293**
 - input, 88
 - interpolation, **151**, 155, 159
 - local truncation, **293**, 327, 376
 - quantization, **508**
 - relative, **10**, 40
 - relative backward, 87
 - relative forward, 87
 - root mean squared, **192**
 - rounding, 10, 248
 - squared, **192**
 - standard, 448
 - tolerance, 326
 - truncation, 248
- error magnification factor, **49**, 88, 241
- escape time, 448
- Euler formula, **468**, 477
- Euler's Method, 284, 333
 - convergence, 296
 - global truncation error, 296
 - local truncation error, 294
 - order, 296
- Euler-Bernoulli beam, 71, 102
- Euler-Maruyama Method, **456**
- exponent, 8
- exponent bias, 11
- extended precision, **8**
- extrapolation, 249, 254, 265, 360, 364

- factorization
 - Cholesky, 119
 - eigenvalue-revealing, 542
 - $PA = LU$, **98**
 - QR, 215, 539
- Fast Fourier Transform, 473
 - operation count, 475
- Fick's law, 375
- fill-in, 113, 115
- filtering
 - low pass, 507
- financial derivative, 464
- Finite Difference Method, 358, 375
 - explicit, 395
 - unstable, 378
- Finite Element Method, 367
- first passage time, 448
- Fisher's equation, 421
- fixed point, **31**
- Fixed-Point Iteration, **31**, 334
 - divergence, 34
 - geometry, 33
- $fl(x)$, 10
- flight simulator, 24
- floating point number, **8**
 - normalized, 8
 - subnormal, 12
 - zero, 13
- forward difference, 244
- forward difference formula, 376
- Forward Difference Method
 - conditionally stable, **380**
 - explicit, **376**
 - stability analysis, 379
- forward kinematics problem, **24**, 67
- Fourier
 - first law, 404
- Fourier, J., 468
- FPI, *see* Fixed-Point Iteration
- freezing temperature, 24

- FSAL, **327**, 329
- function
 - orthogonal, 483
 - Riemann integrable, 409
 - unimodal, **566**
- fundamental domain, 151
- Fundamental Theorem of Algebra, 141
- Galerkin Method, 367, 407
- Gauss, C.F., 188
- Gauss-Newton Method, **231**, **236**, 241
- Gauss-Seidel
 - Method, **109**
- Gaussian elimination, **72**, 92, 358
 - matrix form, 79
 - naive, 72, 95
 - operation count, 75–77
 - tableau form, 73
- Gaussian Quadrature, **276**
- Generalized Minimum Residual Method, 226, 228
- GIS, 240
- GMRES, 226
 - preconditioned, 228
 - restarted, 228
- Golden Section Search, 566
- google-bombing, 551
- Google.com, 549
- Gough, E., 24
- GPS, 188, 233, 238
 - conditioning of, 241
- gradient, 230, 576
- gradient search, 577
- Gram-Schmidt Orthogonalization, 214, 218
- Gram-Schmidt orthogonalization
 - operation count, 215
- Green's Theorem, 407
- Gronwall inequality, 289
- groundwater flow, 416
- half-life, 207
- Halton sequence, 443
- harmonic function, **398**
- heat equation, **375**, 385
- heat sink, 403
- heated plate, 416
- Heron of Alexandria, 39
- Hessian, 231
- Heun Method, 298
- hexadecimal number, **7**
- Hodgkin, A., 317
- Hodgkin-Huxley neuron, 317
- Hooke's Law, 322
- Horner's method, **3**
- Householder reflector, 220, **220**, 545, 546
- Huffman coding, 501, 515
 - in JPEG, 517
- Huffman tree, 517
- Huxley, A., 317
- hypotenuse, 19
- ice cream, 60
- ideal gas law, 60
- IEEE, 8, 23, 92
- ill-conditioned, **50**, 90, 367
- image compression, 505, 508, 561
- image file
 - baseline JPEG, 512
 - grayscale, 505
 - JPEG, 495, 512
- importance sampling, 529
- Improved Euler Method, 298
- IMSL, 23
- incompressible flow, 399
- inflection point, 169
- information
 - Shannon, 515
- initial condition, 282
- initial value problem, **282**
 - existence and uniqueness, 288
- initial-boundary conditions, 375
- inner product, 584
- integral
 - arc length, 265
 - improper, 263, 265
- integrating factor, 290
- integration
 - Romberg, 266
- Intel Corp., 374
- Intermediate Value Theorem, **20**, 25, 29
 - Generalized, **245**
- interpolating polynomial
 - Chebyshev, **159**
- interpolation, **139**
 - by orthogonal functions, 497
 - Chebyshev, 159
 - Lagrange, 64, **140**, 255
 - Newton's divided difference, 142, 153
 - polynomial, 254
 - trigonometric, 467, 476
- interpolation error formula, 152
- inverse kinematics problem, 67

- Inverse Quadratic Interpolation, 64, 65, 69
- IQI, *see* Inverse Quadratic Interpolation
- iterative method, 106
- Ito integral, **453**
- Jacobi Method, **106**
- Jacobian, *see* matrix Jacobian, 361
- JPEG standard, 495
 - Annex K, 512
- Keeling, C., 211
- knot
 - cubic spline, 167
- Krylov methods, 226
- Langevin equation, 457
- Laplace equation, **398**, 414
- Laplacian, 398
- least squares, 558
 - by QR factorization, 217
 - from DCT, 499
 - nonlinear, 203
 - parabola, 488
 - trigonometric, 485
- left-justified, 8
- Legendre polynomial, **275**
- Legendre, A., 188
- Lennard-Jones potential, 565, **580**
- Levenberg-Marquardt Method, 236
- line
 - least squares, 193
- linear congruential generator, **433**
- Lipschitz constant, **288**
- Lipschitz continuous, **288**
- local extrapolation, 327
- logistic equation, 282
- long-double precision, *see* extended precision
- Lorenz equations, 319
- Lorenz, E., 319
- loss of significance, 16, 248
- loss parameter, 508
- low-discrepancy sequence, 442
- LU factorization, **79**
- luminance, **512**
- machine epsilon, 9, 12, 13, 46, 248, 532
- magnitude
 - of a complex number, **468**
 - of a complex vector, 471
- mantissa, **8**
- Maple, 23
- Markov process, 551
- Mathematica, 23
- matrix
 - adjacency, 550
 - banded, 104
 - coefficient, 79
 - condition number, 88, **88**
 - diagonalizable, 587
 - Fourier, 471
 - full, 113
 - google, 551
 - Hessian, 576
 - Hilbert, **30**, 79, 94, 130, 200, 225, 594
 - identity, 584
 - inverse, 557
 - invertible, **584**
 - Jacobian, 131, 576
 - lower triangular, 79
 - nonsymmetric, 541
 - orthogonal, 215, 483, 495, 520, 542, 554
 - permutation, **97**, 98
 - positive-definite, **117**, 578
 - projection, **220**
 - quantization, 508
 - rank-one, 558, 584
 - similar, 542, 587
 - singular, **584**
 - sparse, 71, 113
 - stochastic, **547**
 - structure, 83
 - symmetric, **117**, 539
 - transpose, 190
 - tridiagonal, 171, 359, 379
 - unitary, 471
 - upper Hessenberg, 544
 - upper triangular, 79, 215, 542
 - Van der Monde, **197**
- matrix multiplication
 - blockwise, 585
- Mauna Loa, 150
- Maxwell's equation, 399
- Mean Value Theorem, **20**, 35
 - for Integrals, **22**, 256, 262
- Mersenne prime, **434**
- Method of False Position, **63**
 - slow convergence, 63
- midpoint, 26, 27, 62
- Midpoint Method, 314, 336
- Midpoint Rule, **262**
 - Composite, **263**
 - two-dimensional, 410
- Milne-Simpson Method, **344**
- Milstein Method, **458**
- MKS units, 102

model

- drug concentration, 208
- exponential, 203
- linearization, 204
- population, 282
- power law, 206

Modified Discrete Cosine

- Transform, 496, **521**

Modified Gram-Schmidt, 218

moment of inertia, 102

Monte Carlo

- convergence, 445
- pseudo-random, 440
- quasi-random, 444
- Type 1, 434
- Type 2, 435

Moore's Law, 206, 374

Moore, G.C., 206

motion of projectile, 349, 354

Muller's Method, **63**multiplicity, **46**, 50

multistep methods, 336

- consistent, **341**
- convergent, **341**
- local truncation error, 339
- stable, **340**, 341
- strongly stable, **340**
- weakly stable, **340**

MATLAB

- animation in, 279
- Symbolic Toolbox, 241

MATLAB code

- ab2step.m, 337, 343
- adapquad.m, 271
- amlstep.m, 343
- bezierdraw.m, 181
- bisect.m, 28, 353
- broyden2.m, 135
- brusselator.m, 427
- burgers.m, 419
- bvpfem.m, 372
- clickinterp.m, 147
- crank.m, 387
- cubrt.m, 593
- dftfilter.m, 488, 492
- dftinterp.m, 480
- euler.m, 286
- euler2.m, 303
- eulerstep.m, 286
- exmultistep.m, 337
- fisher2d.m, 425
- fpi.m, 32

gss.m, 568

halton.m, 443

heatbdn.m, 384

heatfd.m, 378, 381

hessen.m, 546

hh.m, 318

invpowerit.m, 536

jacobi.m, 115

nest.m, 3, 146, 148, 165

newtdd.m, 146, 148

nlbvpfd.m, 362

nsi.m, 540

orbit.m, 310

pend.m, 307

poisson.m, 402, 406

poissonfem.m, 412

powerit.m, 534

predcorr.m, 343

rk4step.m, 319

romberg.m, 267

rqi.m, 537

shiftedqr.m, 543

shiftedqr0.m, 543

sin2.m, 165

sparsesetup.m, 115

spi.m, 570

splinecoeff.m, 172

splineplot.m, 173

tacoma.m, 324

trapstep.m, 308, 324, 337

unshiftedqr.m, 541

unstable2step.m, 337

weaklystab2step.m, 337

wilkpoly.m, 47

MATLAB command

axis, 592, 597

backslash, 89, 94, 412

break, 594

button, 147

cla, 597

clear, 590

cond, 89

conj, 494

dct, 504

det, 30

diag, 115, 378

diary, 590

diff, 251

double, 505

drawnow, 307, 598

eig, 30, 547

erf, 273

- error, 75, 595
- fft, 472, 480, 494
- figure, 592
- fminunc, 582
- for, 594
- format, 591
- format hex, 7, 11
- fprintf, 591
- fzero, 44, 47, 51, 65, 69
- ginput, 147, 181
- global, 319, 596
- grid, 592
- handel, 490
- hilb, 30, 90
- ifft, 472, 480, 494
- imagesc, 505
- imread, 505, 513
- int, 251
- interp1, 187
- length, 115, 597
- line, 280, 324
- load, 590
- log, 590
- loglog, 265
- lu, 101, 115, 446
- max, 30, 534
- mean, 596
- mesh, 392, 402, 406, 592
- nargin, 596
- ode23s, 331, 335
- ode45, 329, 331, 353
- odeset, 329
- ones, 90, 115, 597
- pause, 598
- pi, 30
- plot, 30, 591
- plot3, 581
- polyfit, 187, 196
- polyval, 187, 196
- pretty, 251
- qr, 540, 541, 543
- rand, 437
- randn, 439, 456, 494
- rem, 594
- round, 286, 529
- semilogy, 592
- set, 280, 307
- simple, 251
- size, 597
- solve, 241
- sound, 490, 492, 529
- spdiags, 115, 371
- spline, 175, 187
- std, 494, 596
- subplot, 319, 592
- subs, 241
- surf, 413, 592
- svd, 555, 562
- syms, 241, 251
- wavread, 490, 529
- wavwrite, 490
- while, 594
- xdata, 598
- ydata, 598
- zeros, 115, 597
- NAG, 23
- Napoleon, 468
- Navier-Stokes equations, 428
- Nelder-Mead search, 571, 581
- nested multiplication, **2**, 139
- Newton
 - law of cooling, 404
 - second law of motion, 282, 305, 309, 322, 349
- Newton's Method, **52**, 69, 334, 576
 - convergence, 53
 - Modified, 57
 - Multivariate, **131**, 231, 233, 360
 - periodicity, 58
- Newton-Cotes formula, 255
 - closed, 259
 - open, 262
- Newton-Raphson Method, *see* Newton's Method
- noise, 492
 - Gaussian, 493
- norm
 - Euclidean, 212
 - infinity, 86
 - matrix, 88, **90**
 - maximum, 86
 - vector, **90**
- normal equations, 191, 498
- Normalized Simultaneous Iteration, 540
- numerical integration, 254
 - composite, 259
- objective function, **565**
- ODE solver
 - multistep, 336
 - convergence, 296
 - explicit, **332**
 - implicit, **333**
 - variable step size, 325
- one-body problem, 309

- option
 - barrier, 465
 - call, 464
 - put, 465
- order
 - of a differential equation, **303**
 - of approximation, 244
 - of ODE solver, **296**
- ordinary differential equation, 349
- Ornstein-Uhlenbeck process, 457
- orthogonal
 - functions, **368**
 - matrix, 215
- orthogonalization, 539
 - Gram-Schmidt, 212
 - Modified Gram-Schmidt, 218
- orthonormal, 552, 587
- outer product, 584
- page rank, 549
- panel, 259
- parabola, 64
 - interpolating, 139
 - least squares, 194
- partial derivative, 334
- partial differential equation, 374
 - elliptic, 398, 404
 - hyperbolic, 393
 - parabolic, 375
- PDF file, 183
- pencil, 44
- pendulum, 305
 - damped, 308
 - double, 309
- pivot, 75, 101
- pivoting
 - partial, **95**, 100
- Poincaré, H., 311
- Poincaré-Bendixson Theorem, 308
- Poisson equation, **398**
- polishing, 113
- polynomial
 - Chebyshev, **159**, 367
 - evaluation, 1
 - Legendre, 275
 - monic, **161**
 - orthogonal, **274**
 - Taylor, 48
 - Wilkinson, 47, 50, 51
- PostScript, 138
- potential, 398
- Power Iteration, **532**, 549
 - convergence, 534
 - inverse, 535
 - shifted, 536
- power law, 206, 445
- Prandtl number, 320
- preconditioner, 126
 - Gauss-Seidel, 127
 - Jacobi, 126
 - SSOR, 127
- preconditioning, 125
- predictor-corrector method, 342
- Prigogine, I., 426
- prismatic joint, 67
- probability distribution function, 437
- product rule
 - matrix/vector, 589
- progress curve, **280**
- projection
 - orthogonal, 559
- psychoacoustics, 528
- QR Algorithm, 544
 - shifted, 543
 - unshifted, 541
 - convergence, 541
- QR-factorization, **215**
 - operation count, 223
 - reduced, 213
- quadratic formula, 17
- quadrature, 254
 - Gaussian, 276
- quantization, 508, 561
 - JPEG standard, 512
 - linear, 508
- radix, 6
- random number
 - exponential, 437
 - normal, 438
 - pseudo-, 432
 - quasi-, 442
 - uniform, 432
- random number generator
 - minimal standard, **434**, 437
 - period, 433
 - RANDNUM, **439**
 - randu, 435
 - uniform, 432
- random seed, 432
- random variable
 - standard deviation, 440
 - standard normal, **438**, 456
 - variance, 440

- random walk, **447**
 - biased, 451
- rank, 557
- Rayleigh quotient, **534**
- Rayleigh Quotient Iteration, 537
- Rayleigh-Bénard convection, 319
- reaction-diffusion equation, 390, 421
- recursion relation
 - Chebyshev polynomials, 160
- Regula Falsi, *see* Method of False Position
- rejection method, 439
- relaxation parameter, 110
- residual, 86, 125, 234, 368
- Reynolds number, 320
- Richardson extrapolation, 249
- Riemann integral, 453
- right-hand side vector, 79
- RKF45, *see* Runge-Kutta-Fehlberg Method
- RMSE, **192**
- robot, 24
- Rolle's Theorem, **20**
- Romberg Integration, **267**
- root, **25**
 - double, 46
 - multiple, **46**, 56, 59
 - simple, **46**
 - triple, 46
- root of unity, **469**
 - primitive, **469**
- rounding, 9
 - to nearest, 9, 14, 15
- row exchange, 95
- row vector, 583
- run length encoding, 518
- Runge example, 155
- Runge Kutta Method, First-Order
 - Stochastic, 460
- Runge phenomenon, 155, 157, 158, 367
- Runge-Kutta Method, 314
 - global truncation error, 317
 - embedded pair, 326
 - order 2/3, 327
 - order four, 316, 339
- Runge-Kutta-Fehlberg Method, 328
- sample mean, **448**
- sample variance, **448**
- sampling rate, 490
- Scholes, M., 431, 464
- Schur form
 - real, **542**
- Scripps Institute, 211
- Secant Method, **61**, 64, 65
 - convergence, 61
 - slow convergence, 63
- sensitive dependence
 - on initial conditions, **311**, 320
- sensitivity, **48**
- Sensitivity Formula for Roots, **48**
- separation of variables, 287
- Shannon, C., 515
- Sherman-Morrison formula, 585
- shifted QR algorithm, 562
- Shooting Method, 352, 357
- sign, 8
- significant digits, 43
 - loss of, 248
- Simpson's Rule, **257**, 327, 344
 - adaptive, 272
 - Composite, **261**
- single precision, 8
- singular value, 552
- singular value decomposition, 554
 - calculation of, 562
 - nonuniqueness, 554
- singular vector, 552
- sinusoid
 - least squares, 201
- size
 - in JPEG code, 517
- slope field, **282**
- solution
 - least squares, **189**
- SOR, *see* Successive Over-Relaxation
- spectral method, 367
- spectral radius, 111, 382, 588
- spline
 - Bézier, 138, 179
 - cubic, **167**
 - linear, 166
- square root, 30, 38, 54
- squid axon, 318
- stability
 - conditional, 380, 395
 - unconditional, 382
- stage
 - of ODE solver, 315
- steepest descent, 577
- stencil, **376**
- step size, 284, 376, 417
- Stewart platform, **24**, 67
 - planar, 67
- stiffness, 71
- stochastic differential equation, **452**

- stochastic process, 447
 - continuous-time, 452
- stopping criterion, **40**, 47, 65, 575
- stress, 71
- strictly diagonally dominant, 107, 171
- strike price, 464
- strut, 67
- submatrix
 - principal, 118
- Successive Over-Relaxation, 109
- Successive Parabolic Interpolation, 569
- swamping, 91
- synthetic division, 3
- tableau form, 92
- Tacoma Narrows Bridge, 281, 322
- Taylor formula, 53
- Taylor Method, **300**
- Taylor polynomial, **21**
- Taylor remainder, **21**
- Taylor's Theorem, **21**, 244, 338
- thermal conductivity, 404
- thermal diffusivity, 375
- three-body problem, **311**
- time series, 476
- transpose
 - of a matrix, 584
- Trapezoid Method
 - explicit, **297**, 336
 - implicit, 342
- Trapezoid Rule, **257**, 298
 - adaptive, 269
 - Composite, **260**
- tridiagonal, 562
- trigonometric function
 - order n , **477**
 - plotting, 480
- Tukey, J., 473
- Turing patterns, 426
- Turing, A., 426
- unconstrained optimization, 566
- updating
 - interpolating polynomial, 144
- upper Hessenberg form, 544, 562
- Van der Corput sequence, 443
- Van der Waal's equation, 60
- Van der Waals force, 565, 580
- vector
 - orthogonal, 190
 - residual, 86
- vector calculus, 588
- volatility, 465
- Von Neumann stability, **379**
- Von Neumann, J., 432
- wave equation, **393**
- wave speed, 393
- Weather Underground, 210
- web search, 549
- well-conditioned, **50**
- Wiener, N., 492
- Wilkinson polynomial, **47**, 50, 51, 88, 532
- Wilkinson, J., 47
- wind turbine, 211
- window function, 529
- world oil production, 157
- world population, 151, 178
- Young's modulus, 71, 102
- zero-padding, 524
- zigurat algorithm, 439