Introduction to Scientific Computing

using MATLAB

Ian Gladwell
Department of Mathematics
Southern Methodist University
Dallas, TX 75275

James G. Nagy
Department of Mathematics and Computer Science
Emory University
Atlanta, GA 30322

Warren E. Ferguson, Jr.
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712

Contents

Preface

This text book is aimed at science and engineering majors who wish to learn the ideas behind elementary scientific computation including understanding how some of the best numerical software works. We devote a chapter to each of seven topics:

- A brief introduction to MATLAB.
- Computer arithmetic based on the IEEE 754 standard, and its effects on scientific computing
- The solution of systems of linear equations, and a discussion of residuals and conditioning
- Curve fitting including interpolation, splines and least squares fitting
- Numerical integration including a global adaptive algorithm
- Solving a single nonlinear equation including computing square roots and roots of polynomials
- Minimization of a function of one variable

These topics were chosen because they give a broad enough coverage to present the basic ideas of scientific computing and some of the theory behind it to students who have taken only the courses in univariate calculus in addition to high school mathematics. It is not intended a text for a graduate course nor for advanced undergraduates.

We do use partial derivatives at one point, but only at the level which can be taught within the context of the application. In addition, because we use MATLAB throughout the book, students must have some familiarity with basic matrix algebra. However, a linear algebra course is not prerequisite; instead, we introduce sufficient material about vectors, matrices and matrix algebra in Chapter 1, *Getting Started with Matlab*. If the book is used in a course for which linear algebra is not prerequisite, then we suggest going more slowly through this introduction.

For each topic we provide sufficient theory so that the student can follow the reasoning behind the choices made in quality numerical software and can interpret the results of using such software. We have provided a large number of problems, some theoretical and some computational. It is intended that students use Chapter 1 and the information on the MATLAB HELP pages to start these computational problems. It is partly our aim to provide an opportunity for students to practice their programming skills in a "real world" situation. Some problems require students to "code up" and use the pseudocodes in the book. Many problems require graphical output; it is assumed that the simple plotting package available through the `plot` function in MATLAB will be used.

A conscious choice has been made to largely divorce the MATLAB discussion from the algorithmic and theoretical development. So, almost all MATLAB related material is placed in a stand alone section at the end of each of chapters 2 – 7. We made this choice because we wanted the students to view MATLAB as tool with a strong library of mathematical software functions and an approachable user language but not to confuse MATLAB's features with the mathematical development. In fact, though there is no further mention of it in the book, almost all this same material has been taught by the authors at different times using as the computational base Fortran 77 or C with the mathematical software used originating from the respective NAG libraries.

The base versions of the numerical software library assumed in this text is MATLAB Version 7.2.0 (Release 2006A).

This textbook has been developed over a number of years. Versions of it have been used in the classroom a number of times by all three authors and also by other teachers. The students have included engineering, computer science, management science, physics, chemistry, economics, business and mathematics majors. A number of the computer projects and the problems have been used in assessment.

The chapters in this textbook can be taught in almost any order as many of the topics are almost independent. And, we have experimented with teaching the chapters in a variety of orders. It is best if the Curve Fitting chapter is taught before the Numerical Integration chapter, and if the Root Finding chapter is taught before the Univariate Minimization chapter. Sometimes, we have started the course by teaching from the Computer Arithmetic chapter, especially since its examples and computer projects make a good starting point for student assessment, but equally often we have started the course by teaching the Solution of Linear Systems chapter and occasionally we have started by first teaching the Root Finding chapter.

Each chapter is divided into a number of sections, and many of the sections are further subdivided into subsections where this is necessary to provide a structure that is easy for the instructor and the student. Figures and tables are numbered through the chapter but all other numbered entities (definitions, examples, problems, remarks and theorems) are numbered through the sections in which they appear.

In what ways is this book different to the many other scientific computing or numerical methods books currently available? First, it has relatively little in the way of mathematical prerequisites and the mathematical treatment involves no more theory than is necessary to follow the algorithmic argument. Second, the emphasis is on what is needed to be understood to make intelligent use of mathematical software. Specifically, the computer arithmetic chapter has multiple examples which illustrate the effects of catastrophic cancelation and the accumulation of rounding errors. These algorithmic difficulties are revisited in the linear equations chapter and to a lesser extent in later chapters. Ill-conditioning of the underlying problem is introduced in the linear equations chapter and reappears in the curve fitting chapter, for both interpolation and least squares fitting, and the roots of nonlinear equations chapter. Third, we emphasize good numerical practice. For example, to avoid problems with cancelation we emphasize incremental correction in iterative methods in the differentiation and integration, root finding, and minimization chapters. In the curve fitting chapter, we place some emphasis on the use of Chebyshev polynomials as an alternative, and usually superior, polynomial basis for data fitting and on the excellent properties of the Chebyshev points as sampling points. Generally, we identify where "standard" numerical algorithms may break down and we suggest alternatives. Fourth, we emphasize the use the MATLAB programming language and its excellent library of mathematical software but we acknowledge that for all its excellence there are deficiencies in this library, so we discuss alternative approaches. Where we have had to make choices about exactly what material to cover the content of the MATLAB library has been used as a guide. Finally, we discuss using symbolic algebra approaches to differentiation and integration by exploiting features available in the MATLAB Symbolic Math Toolbox and included in the MATLAB Student Version.

Acknowledgements

The authors wish to thank their friends and colleagues who have read prepublication versions of this text and, in some cases, used it in the classroom. In particular we wish to thank Michele Benzi and Eldad Haber of Emory University, Wayne Enright of the University of Toronto, Graeme Fairweather of the Colorado School of Mines, Leonard Freeman of the University of Manchester, Peter Moore, Johannes Tausch and Sheng Xu of Southern Methodist University, David Sayers of the Numerical Algorithms Group Ltd., and Wen Zhang of Oakland University.

The Authors

Ian Gladwell was awarded a BA in Mathematics from the University of Oxford in 1966, an M.Sc. in Numerical Analysis and Computing in 1967 and a Ph.D. in Mathematics in 1970 from the University of Manchester. From 1967–1987 he was a member of the faculty in the Department of Mathematics

at the University of Manchester. In 1986, he was a Royal Society Industrial Fellow at The Numerical Algorithms Group, Oxford. Since 1987 he has been a member of the faculty in the Department of Mathematics at Southern Methodist University, Dallas. He is the author of many published research papers on scientific computing and on mathematical software, and of publicly available numerical software. His particular research interests are in the numerical solution of both initial and boundary value problems in ordinary differential equations. Since 2005 he has been editor-in-chief of the Association of Computing Machinery (ACM) Transactions on Mathematical Software.

James Nagy was awarded a BS in Mathematics in 1986 and an MS in Computational Mathematics in 1998 both from Northern Illinois University. In 1991 he was awarded a Ph.D. in Applied Mathematics from North Carolina State University. He was a Postdoctoral Research Associate at the Institute of Mathematics and Its Applications, University of Minnesota, from 1991–1992, and a member of the faculty in the Department of Mathematics at Southern Methodist University, Dallas, from 1992–1999. Since 1999 he has been a member of the faculty in the Mathematics and Computer Science Department at Emory University, Atlanta. In 2001 he was selected to hold the Emory Professorship for Distinguished Teaching in the Social and Natural Sciences. He has published many research papers on scientific computing, numerical linear algebra, inverse problems, and image processing. His particular research interests are in the numerical solution of large scale structured linear systems.

Warren Ferguson was awarded a BS in Physics from Clarkson University in 1971 and a Ph.D. in Applied Mathematics from the California Institute of Technology in 1975. He was a member of the faculty in the Department of Mathematics at the University of Arizona from 1975–1980 and at Southern Methodist University, Dallas, from 1980–2000. In 2000, he joined the Intel Corporation. He is the author of many published research papers in applied mathematics and scientific computing. His particular interests are in the numerical solution of partial differential equations and in computer arithmetic.

Chapter 1

Getting Started with Matlab

The computational examples and exercises in this book have been computed using MATLAB, which is an interactive system designed specifically for scientific computation that is used widely in academia and industry. At its core, MATLAB contains an efficient, high level programming language and powerful graphical visualization tools which can be easily accessed through a development environment (that is, a graphical user interface containing various workspace and menu items). MATLAB has many advantages over computer languages such as C, C++, Fortran and Java. For example, when using the MATLAB programming language, essentially no declaration statements are needed for variables. In addition, MATLAB has a built-in extensive mathematical function library that contains such items as simple trigonometric functions, as well as much more sophisticated tools that can be used, for example, to compute difficult integrals. Some of these sophisticated functions are described as we progress through the book. MATLAB also provides additional *toolboxes* that are designed to solve specific classes of problems, such as for image processing. It should be noted that there is no free lunch; because MATLAB is an "interpreted" language, codes written in Fortran and C are usually more efficient for very large problems. (MATLAB may also be compiled.) Therefore, large production codes are usually written in one of these languages, in which case supplementary packages, such as the NAG or IMSL library, or free software from *netlib*, are recommended for scientific computing. However, because it is an excellent package for developing algorithms and problem solving environments, and it can be quite efficient when used properly, all computing in this book uses MATLAB.

We provide a very brief introduction to MATLAB. Though our discussion assumes the use of MATLAB 7.0 or higher, in most cases version 6.0 or 6.5 is sufficient. There are many good sources for more complete treatments on using MATLAB, both on-line, and as books. One excellent source is the *MATLAB Guide, 2nd ed.* by D.J. Higham and N.J. Higham published by SIAM Press, 2005. Another source that we highly recommend is MATLAB's built-in help system, which can be accessed once MATLAB is started. We explain how to access it in section 1.1. The remainder of the chapter then provides several examples that introduce various basic capabilities of MATLAB, such as graph plotting and writing functions.

1.1 Starting, Quitting, and Getting Help

.

The process by which you start MATLAB depends on your computer system; you may need to request specific commands from your instructor or system administrator. Generally, the process is as follows:

- On a PC with a Windows operating system, double-click the "MATLAB" shortcut icon on your Windows desktop. If there is no "MATLAB" icon on the desktop, you may bring up DOS (on Windows XP by going to the Command Prompt in Accessories) and entering `matlab` at the operating system prompt. Alternatively, you may search for the "MATLAB" icon in a

subdirectory and click on it. Where it is to be found depends on how MATLAB was installed; in the simplest case with a default installation it is found in the `C:\$MATLAB` directory, where `$MATLAB` is the name of the folder containing the MATLAB installation.

- On a Macintosh running OS X 10.1 or higher, there may be a "MATLAB" icon on the dock. If so, then clicking this icon should start MATLAB. If the "MATLAB" icon is not on the dock, then you need to find where it is located. Usually it is found in `/Applications/$MATLAB/`, or `/Applications/$MATLAB/bin/`, where `$MATLAB` is the name of the folder containing the MATLAB installation. Once you find the "MATLAB" icon, double clicking on it should start MATLAB.
- On Unix or Linux platforms, typically you enter `matlab` at the shell prompt.

When you have been successful in getting MATLAB to start, then the development tool Graphical User Interface (GUI) should appear on the screen. Although there are slight differences (such as key stroke short cuts) between platforms, in general the GUI should have the same look and feel independently of the platform.

The *command window*, where you will do much of your work, contains a prompt:

```
>>
```

We can enter data and execute commands at this prompt. One very useful command is `doc`, which displays the "help browser". For example, entering the command

```
>> doc matlab
```

opens the help browser to a good location for first time MATLAB users to begin reading. Alternatively, you can pull down the *Help* menu, and let go on *MATLAB Help*. We recommend that you read some of the information on these help pages now, but we also recommend returning periodically to read more as you gain experience using MATLAB.

Throughout this book we provide many examples using MATLAB. In all cases, we encourage readers to "play along" with the examples provided. While doing so, it may be helpful at times to use the `doc` command to find detailed information about various MATLAB commands and functions. For example,

```
>> doc plot
```

opens the help browser, and turns to a page containing detailed information on using the built-in `plot` function.

To exit MATLAB, you can pull down the *File* menu, and let go on or *Exit MATLAB*. Alternatively, in the command window, you can use the `exit` command:

```
>> exit
```

1.2 Basics of Matlab

MATLAB derives its name from MATrix LABoratory because the primary object involved in any MATLAB computation is a *matrix*. A matrix A is an array of values, with a certain number of rows and columns that define the "dimension" of the matrix. For example, the array A given by

$$A = \begin{bmatrix} 0 & -6 & 8 & 1 \\ -2 & 5 & 5 & -3 \\ 7 & 8 & 0 & 3 \end{bmatrix}$$

is a matrix with 3 rows and 4 columns, and so is typically referred to as a 3×4 matrix. It is two-dimensional. The values in the matrix are by default all Double Precision numbers which will be discussed in more detail in the next chapter. This fact permits MATLAB to avoid using declarations but involves a possible overhead in memory usage and speed of computation. A matrix with only

one row (that is, a $1 \times n$ matrix) is often called a row vector, while a matrix with only one column (that is, an $n \times 1$ matrix) is called a column vector. For example if

$$x = \begin{bmatrix} -2 \\ 8 \\ 4 \\ 0 \\ 5 \end{bmatrix} \quad \text{and} \quad y = \begin{bmatrix} -3 & 6 & 3 & -4 \end{bmatrix},$$

then we can say that x is a 5×1 matrix, or that it is a column vector of length 5. Similarly, we can say that y is a 1×4 matrix, or that it is a row vector of length 4. If the shape is obvious from the context, then we may omit the words *row* or *column*, and just refer to the object as a vector.

MATLAB is very useful when solving problems whose computations involve matrices and vectors. We explore some of these basic *linear algebra* manipulations, as well as some basic features of MATLAB, through a series of examples.

Initializing Vectors. We can easily create row and/or column vectors in MATLAB. For example, the following two statements create the same row vector:

```
>> x = [1 2 3 4]
>> x = [1, 2, 3, 4]
```

Similarly, the following two statements create the same column vector:

```
>> x = [1
2
3
4]
>> x = [1; 2; 3; 4]
```

Rather than thinking of these structures as row and column vectors, we should think of them as 1×4 and 4×1 matrices, respectively. Observe that elements in a row may be separated by using either a blank space or by using a comma. Similarly, to indicate that a row has ended, we can use either a carriage return, or a semicolon.

Initializing Matrices. In general, we can create matrices with more than one row and column. The following two statements generate the same matrix:

```
>> A = [1 2 3 4
5 6 7 8
9 10 11 12]
>> A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

Matrix Arithmetic. Matrices can be combined (provided certain requirements on their dimensions are satisfied) using the operations `+, -, *` to form new matrices. Addition and subtraction of matrices is intuitive; to add or subtract two matrices, we simply add or subtract corresponding entries. The only requirement is that the two matrices have the same dimensions. For example, if

$$A = \begin{bmatrix} -3 & -1 \\ 3 & 2 \\ 3 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 3 \\ -2 & 2 \\ 3 & -1 \end{bmatrix}, \quad \text{and} \quad C = \begin{bmatrix} -3 & -2 & -2 \\ 5 & 8 & -2 \end{bmatrix}$$

then the MATLAB commands

```
>> D = A + B
>> E = B - A
```

produce the matrices

$$D = \begin{bmatrix} 4 & 2 \\ 1 & 4 \\ 6 & -1 \end{bmatrix} \quad \text{and} \quad E = \begin{bmatrix} 10 & 4 \\ -5 & 0 \\ 0 & -1 \end{bmatrix},$$

but the command

```
>> F = C + A
```

produces an error message because the matrices C and A do not have the same dimensions.

Matrix multiplication, which is less intuitive than addition and subtraction, can be defined using linear combinations of vectors. We begin with something that is intuitive, namely the product of a scalar (that is, a number) and a vector. In general, if c is a scalar and a is a vector with entries a_i, then ca is a vector with entries ca_i. For example,

$$\text{if} \quad \mathrm{c} = 9 \quad \text{and} \quad a = \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} \quad \text{then} \quad ca = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 27 \\ 36 \\ -18 \end{bmatrix}.$$

Once multiplication by a scalar is defined, it is straight forward to form linear combinations of vectors. For example, if

$$\mathrm{c}_1 = 9,\ \mathrm{c}_2 = 3,\ \mathrm{c}_3 = 1, \quad \text{and} \quad a_1 = \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix},\ a_2 = \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix},\ a_3 = \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix},$$

then

$$\mathrm{c}_1 a_1 + \mathrm{c}_2 a_2 + \mathrm{c}_3 a_3 = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} + 3 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + 1 \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 40 \\ 60 \\ -14 \end{bmatrix}.$$

is a linear combination of the vectors a_1, a_2 and a_3 with the scalars c_1, c_2 and c_3, respectively.

Now, suppose we define a matrix A and column vector x as

$$A = \begin{bmatrix} 3 & 3 & 4 \\ 4 & 8 & 0 \\ -2 & 0 & 4 \end{bmatrix}, \quad x = \begin{bmatrix} 9 \\ 3 \\ 1 \end{bmatrix},$$

then the matrix–vector product Ax is simply a compact way to represent a linear combination of the columns of the matrix A, where the scalars in the linear combination are the entries in the vector x. That is,

$$Ax = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} + 3 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + 1 \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 40 \\ 60 \\ -14 \end{bmatrix}.$$

Thus, to form Ax, the number of columns in the matrix A must be the same as the number of entries in the vector x.

Finally we consider matrix-matrix multiplication. If A is an $m \times n$ matrix, and B is an $n \times p$ matrix (that is, the number of columns in A is the same as the number of rows in B), then the product $C = AB$ is an $m \times p$ matrix whose columns are formed by multiplying A by corresponding columns in B viewed as column vectors. For example, if

$$A = \begin{bmatrix} 3 & 3 & 4 \\ 4 & 8 & 0 \\ -2 & 0 & 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 9 & -1 & -2 & 0 \\ 3 & 0 & 1 & -3 \\ 1 & 2 & -1 & 5 \end{bmatrix}$$

then

$$C = AB = \begin{bmatrix} 40 & 5 & -7 & 11 \\ 60 & -4 & 0 & -24 \\ -14 & 10 & 0 & 20 \end{bmatrix}.$$

So, the jth column of AB is the linear combination of the columns of A with scalars drawn from the jth column of B. For example, the 3rd column of AB is formed by taking the linear combination of the columns of A with scalars drawn from the 3rd column of B. Thus, the 3rd column of AB is $(-2) * a_1 + (1) * a_2 + (-1) * a_3$ where a_1, a_2 and a_3 denote, respectively, the 1st, 2nd, and 3rd columns of A.

The $*$ operator can be used in MATLAB to multiply scalars, vectors, and matrices, provided the dimension requirements are satisfied. For example, if we define

```
>> A = [1 2; 3 4]
>> B = [5 6; 7 8]
>> c = [1; -1]
>> r = [2 0]
```

and we enter the commands

```
>> C = A*B
>> y = A*c
>> z = A*r
>> w = r*A
```

we find that

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}, \quad y = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \quad w = \begin{bmatrix} 2 & 4 \end{bmatrix},$$

but an error message is displayed when trying to calculate `z = A*r` because it is not a legal linear algebra operation; the number of columns in A is not the same as the number of rows in r (that is, the inner matrix dimensions do not agree).

Suppressing Output. Note that each time we execute a statement in the MATLAB command window, the result is redisplayed in the same window. This action can be suppressed by placing a semicolon at the end of the statement. For example, if we enter

```
>> x = 10;
>> y = 3;
>> z = x*y
```

then only the result of the last statement, $z = 30$, is displayed in the command window.

Special Characters In the above examples we observe that the semicolon can be used for two different purposes. When used inside brackets `[ ]` it indicates the end of a row, but when used at the end of a statement, it suppresses display of the result in the command window.

The comma can also be used for two different purposes. When used inside brackets `[ ]` it separates entries in a row, but it can also be used to separate statements in a single line of code. For example, the previous example could have been written in one line as follows:

```
>> x = 10; y = 3; z = x*y
```

The semicolon, comma and brackets are examples of certain special characters in MATLAB that are defined for specific purposes. For a full list of special characters, and their usage, see

```
>> doc 'special characters'
```

Transposing Matrices. The *transpose* operation is used in linear algebra to transform an $m \times n$ matrix into an $n \times m$ matrix by transforming rows to columns, and columns to rows. For example, if we have the matrices and vectors:

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \end{bmatrix}, \quad c = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad r = \begin{bmatrix} 2 & 0 \end{bmatrix},$$

then

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 5 & 6 \end{bmatrix}, \quad c^T = \begin{bmatrix} 1 & -1 \end{bmatrix}, \quad r^T = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

where superscript T denotes transposition. In MATLAB, the single quote ' is used to perform the transposition operation. For example, consider the following matrices:

```
>> A = [1 2 5; 3 4 6];
>> c = [1; -1];
```

When we enter the commands

```
>> D = A'
>> s = c'*c
>> H = c*c'
```

we obtain

$$D = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 5 & 6 \end{bmatrix}, \quad s = 2, \quad H = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

Other Array Operations. MATLAB supports certain array operations (not normally found in standard linear algebra books) that can be very useful in scientific computing applications. Some of these operations are:

```
.*    ./    .^
```

The *dot* indicates that the operation is to act on the matrices in an element by element (componentwise) way. For example,

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} .* \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 5 \\ 12 \\ 21 \\ 32 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} .^ 3 = \begin{bmatrix} 1 \\ 8 \\ 27 \\ 64 \end{bmatrix}$$

The rules for the validity of these operations are different than for linear algebra. A full list of arithmetic operations, and the rules for their use in MATLAB, can be found by referring to

```
doc 'arithmetic operators'
```

Remark on the Problems. The problems in this chapter are designed to help you become more familiar with MATLAB and some of its capabilities. Some problems may include issues not explicitly discussed in the text, but a little exploration in MATLAB (that is, executing the statements and reading appropriate `doc` pages) should provide the necessary information to solve all of the problems.

Problem 1.2.1. *If* `z = [0 -1 2 4 -2 1 5 3]`, *and* `J = [5 2 1 6 3 8 4 7]`, *determine what is produced by the following sequences of* MATLAB *statements:*

```
>> x = z', A = x*x', s = x'*x, w = x*J,
>> length(x), length(z)
>> size(A), size(x), size(z), size(s)
```

Note that to save space, we have placed several statements on each line. Use `doc length` *and* `doc size`, *or search the* MATLAB *Help index, for further information on these commands.*

1.3 Matlab as a Scientific Computing Environment

So far we have used MATLAB as a sophisticated calculator. It is far more powerful than that, containing many programming constructs, such as flow control (if, for, while, etc.), and capabilities for writing functions, and creating structures, classes, objects, etc. Since this is not a MATLAB programming book, we do not discuss these capabilities in great detail. But many MATLAB programming issues are discussed, when needed, as we progress through the book. Here, we introduce a few concepts before beginning our study of scientific computing.

1.3.1 Initializing Vectors with Many Entries

Suppose we want to create a vector, `x`, containing the values $1, 2, \ldots, 100$. We could do this using a `for` loop:

```
n = 100;
for i = 1:n
  x(i) = i;
end
```

In this example, the notation `1:n` is used to create the list of integers $1, 2, 3, \ldots, n$. When MATLAB begins to execute this code, it does not know that `x` will be a row vector of length 100, but it has a very smart *memory manager* that creates space as needed. Forcing the memory manager to work hard can make codes very inefficient. Fortunately, if we know how many entries `x` will have, then we can help out the memory manager by first allocating space using the `zeros` function. For example:

```
n = 100;
x = zeros(1,n);
for i = 1:n
  x(i) = i;
end
```

In general, the function `zeros(m,n)` creates an $m \times n$ array containing all zeros. Thus, in our case, `zeros(1,n)` creates a $1 \times n$ array, that is a row vector with n entries.

A much simpler, and better way, to initialize this simple vector using one of MATLAB's *vector operations*, is as follows:

```
n = 100;
x = 1:n;
```

The colon operator is very useful! Let's see another example where it can be used to create a vector. Suppose we want to create a vector, x, containing n entries equally spaced between $a = 0$ and $b = 1$. The distance between each of the equally spaced points is given by $h = \dfrac{b-a}{n-1} = \dfrac{1}{n-1}$, and the vector, x, should therefore contain the entries:

$$0, \quad 0+h, \quad 0+2*h, \quad \cdots, \quad (i-1)*h, \quad \cdots, \quad 1$$

We can create a vector with these entries, using the colon operator, as follows:

```
n = 101;
h = 1 / (n-1);
x = 0:h:1;
```

or even

```
n = 101;
x = 0:1/(n-1):1;
```

if we don't need the variable `h` later in our program. We often want to create vectors like this in mathematical computations. Therefore, MATLAB provides a function `linspace` for it. In general, `linspace(a, b, n)` generates a vector of n equally spaced points between a and b. So, in our example with $a = 0$ and $b = 1$, we could instead use:

```
n = 101;
x = linspace(0, 1, n);
```

Note that, for the interval $[0, 1]$, choosing $n = 101$ produces a nice rational spacing between points, namely $h = 0.01$. That is,

$$x = \begin{bmatrix} 0 & 0.01 & 0.02 & \cdots & 0.98 & 0.99 & 1 \end{bmatrix}.$$

A lesson is to be learned from the examples in this subsection. Specifically, if we need to perform a fairly standard mathematical calculation, then it is often worth using the *search* facility in the *help browser* to determine if MATLAB already provides an optimized function for the calculation.

Problem 1.3.1. *Determine what is produced by the* MATLAB *statements:*

```
>> i = 1:10
>> j = 1:2:11
>> x = 5:-2:-3
```

For more information on the use of `":"`*, see* `doc colon`*.*

Problem 1.3.2. *If* `z = [0 -1 2 4 -2 1 5 3]`*, and* `J = [5 2 1 6 3 8 4 7]`*, determine what is produced by the following* MATLAB *statements:*

```
>> z(2:5)
>> z(J)
```

Problem 1.3.3. *Determine what is produced by the following* MATLAB *statements:*

```
>> A = zeros(2,5)
>> B = ones(3)
>> R = rand(3,2)
>> N = randn(3,2)
```

What is the difference between the functions `rand` *and* `randn`*? What happens if you repeat the statement* `R = rand(3,2)` *several times? Now repeat the following pair of commands several times:*

```
>> rand('state', 0)
>> R = rand(3,2)
```

What do you observe? Note: The "up arrow" key can be used to recall statements previously entered in the command window.

Problem 1.3.4. *Determine what is produced by the* MATLAB *statements:*

```
>> x = linspace(1, 1000, 4)
>> y = logspace(0, 3, 4)
```

In each case, what is the spacing between the points?

Problem 1.3.5. *Given integers* `a` *and* `b`*, and a rational number* `h`*, determine a formula for* `n` *such that*

```
linspace(a, b, n) = [a, a+h, a+2h, ..., b]
```

1.3.2 Creating Plots

Suppose we want to plot the function $y = x^2 - \sqrt{x+3} + \cos 5x$ on the interval $-3 \leq x \leq 5$. The basic idea is first to plot several points, (x_i, y_i), and then to connect them together using lines. We can do this in MATLAB by creating a vector of x-coordinates, a vector of y-coordinates, and then using the MATLAB command `plot(x,y)` to draw the graph in a *figure* window. For example, we might consider the MATLAB code:

```
n = 81;
x = linspace(-3, 5, n);
y = zeros(1, n);
for i = 1:n
  y(i) = x(i)^2 - sqrt(x(i) + 3) + cos(5*x(i));
end
plot(x, y)
```

In this code we have used the `linspace` command to create the vector `x` efficiently, and we helped the memory manager by using the `zeros` command to pre-allocate space for the vector `y`. The `for` loop generates the entries of `y` one at time, and the `plot` command draws the graph.

MATLAB allows certain operations on arrays that can be used to shorten this code. For example, if `x` is a vector containing entries x_i, then:

- `x + 3` is a vector containing entries $x_i + 3$, and `sqrt(x + 3)` is a vector containing entries $\sqrt{x_i + 3}$.
- Similarly, `5*x` is a vector containing entries $5x_i$, and `cos(5*x)` is a vector containing entries $\cos(5x_i)$.
- Finally, recalling the previous section, we can use the *dot* operation `x.^2` to compute a vector containing entries x_i^2.

Using these properties, the `for` loop above may be replaced with a single *vector operation*:

```
n = 81;
y = zeros(1, n);
x = linspace(-3, 5, n);
y = x.^2 - sqrt(x + 3) + cos(5*x);
plot(x,y)
```

If you can use array operations instead of loops, then you should, as they are more efficient.

MATLAB has many more, very sophisticated, plotting capabilities. Three very useful commands are `axis`, `subplot`, and `hold`:

- `axis` is mainly used to scale the x and y-axes on the current plot as follows:

  ```
  axis([xmin xmax ymin ymax])
  ```

- `subplot` is mainly used to put several plots in a single figure window. Specifically,

  ```
  subplot(m, n, p)
  ```

  breaks the figure window into an "$m \times n$ matrix" of small axes, and selects the p^{th} set of axes for the current plot. The axes are counted along the top row of the figure window, then the second row, etc.

- `hold` allows you to overlay several plots on the same set of axes.

The following example illustrates how to use these commands.

Example 1.3.1. Chebyshev polynomials are used in a variety of engineering applications. The j^{th} Chebyshev polynomial $T_j(x)$ is defined by

$$T_j(x) = \cos(j \arccos(x)) , \quad -1 \leq x \leq 1 .$$

In section 4.1.4 we see that these strange objects are indeed polynomials.

(a) First we plot, in the same figure, the Chebyshev polynomials for $j = 1, 3, 5, 7$. This can be done by executing the following statements in the command window:

```
x = linspace(-1, 1, 201);
T1 = cos(acos(x));
T3 = cos(3*acos(x));
T5 = cos(5*acos(x));
T7 = cos(7*acos(x));
subplot(2,2,1), plot(x, T1)
subplot(2,2,2), plot(x, T3)
subplot(2,2,3), plot(x, T5)
subplot(2,2,4), plot(x, T7)
```

The resulting plot is shown in Fig. 1.1.

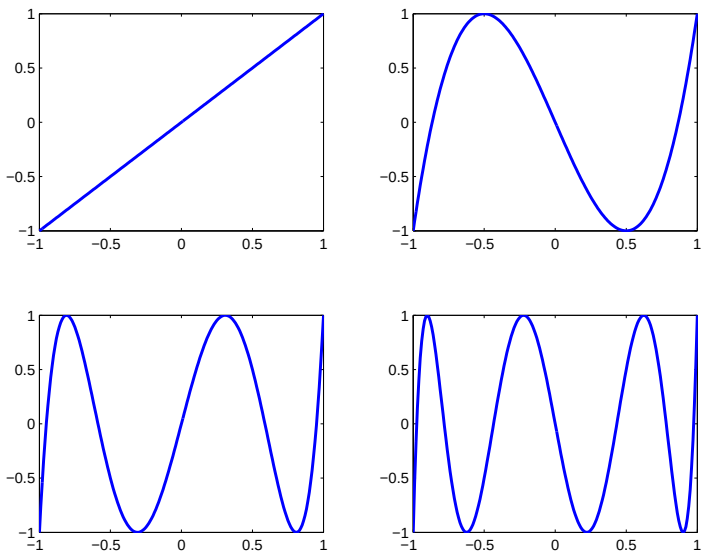

Figure 1.1: Using `subplot` in Example 1.3.1(a).

(b) When you use the `plot` command, MATLAB chooses (usually appropriately) default values for the axes. Here, it chooses precisely the domain and range of the Chebyshev polynomials. These can be changed using the `axis` command, for example:

```
subplot(2,2,1), axis([-1, 1, -2, 2])
subplot(2,2,2), axis([-2, 2, -1, 1])
subplot(2,2,3), axis([-2, 1, -1, 2])
subplot(2,2,4), axis([-2, 2, -2, 2])
```

The resulting plot is shown in Fig. 1.2.

(c) Finally, we can plot all of the polynomials on the same set of axes. This can be achieved as follows (here we use different colors, blue, red, green and cyan, for each):

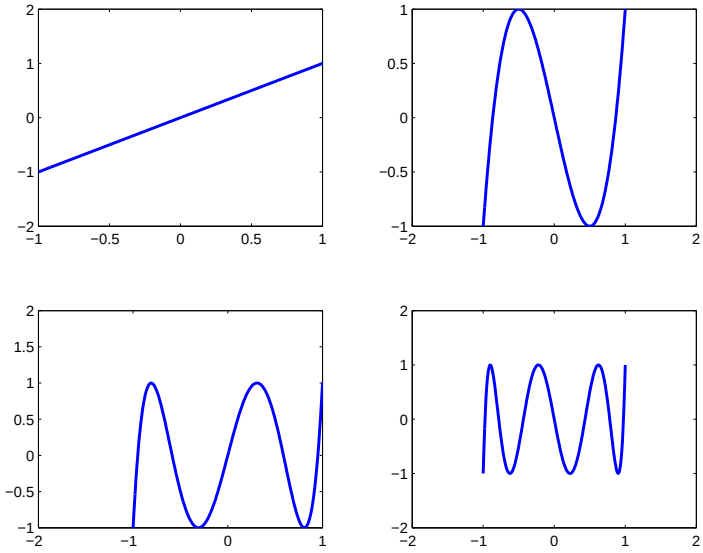

Figure 1.2: Using `axis` in Example 1.3.1(b).

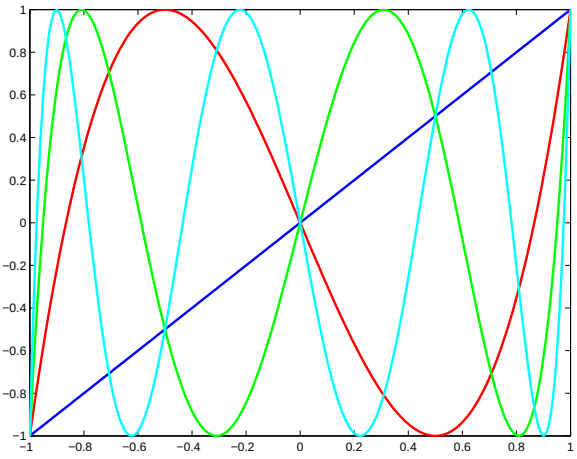

Figure 1.3: Using `hold` in Example 1.3.1(c).

```
subplot(1,1,1)
plot(x, T1, 'b')
hold on
plot(x, T3, 'r')
plot(x, T5, 'g')
plot(x, T7, 'c')
```

The resulting plot is shown in Fig. 1.3. (You should see the plot in color on your screen.)

Remarks on Matlab Figures. We have explained that `hold` can be used to overlay plots on the same axes, and `subplot` can be used to generate several different plots in the *same* figure. In some cases, it is preferable to generate several different plots in *different* figures, using the `figure` command. To clear a figure, so that a new set of plots can be drawn in it, use the `clf` command.

Problem 1.3.6. *Write* MATLAB *code that evaluates and plots the functions:*

(a) $y = 5\cos(3\pi x)$ *for 101 equally spaced points on the interval* $0 \le x \le 1$.

(b) $y = \dfrac{1}{1+x^2}$ *for 101 equally spaced points on the interval* $-5 \le x \le 5$.

(c) $y = \dfrac{\sin 7x - \sin 5x}{\cos 7x + \cos 5x}$ *using 200 equally spaced points on the interval* $-\pi/2 \le x \le \pi/2$. *Use the* `axis` *command to scale the plot so that* $-2 \le x \le 2$ *and* $-10 \le y \le 10$.

In each case, your code should not contain loops but should use arrays directly.

Problem 1.3.7. *A "fixed–point" of a function* $g(x)$ *is a point* x *that satisfies* $x = g(x)$. *An "educated guess' of the location of a fixed–point can be obtained by plotting* $y = g(x)$ *and* $y = x$ *on the same axes, and estimating the location of the intersection point. Use this technique to estimate the location of a fixed–point for* $g(x) = \cos x$.

Problem 1.3.8. *Use* MATLAB *to recreate the plot in Fig. 1.4. Hints: You will need the commands* `hold`, `xlabel`, `ylabel`, *and* `legend`. *The* $\pm$ *symbol in the legend can be created using* `\pm`.

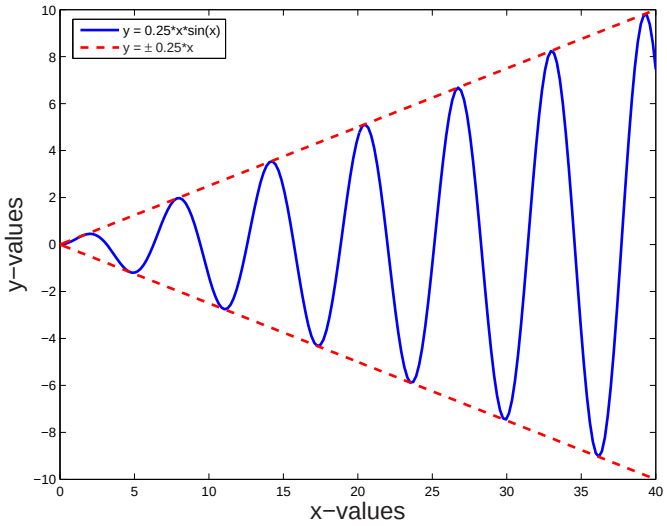

Figure 1.4: Example of a plot that uses MATLAB commands `hold`, `xlabel`, `ylabel`, and `legend`.

Problem 1.3.9. *Explain what happens when the following* MATLAB *code is executed.*

```
for k = 1:6
   x = linspace(0, 4*pi, 2^(k+1)+1);
   subplot(2,3,k), plot(x, sin(x))
   axis([0, 4*pi, -1.5, 1.5])
end
```

What happens if `subplot(2, 3, k)` *is changed to* `subplot(3, 2, k)`*? What happens if the* `axis` *statement is omitted?*

1.3.3 Script and Function M-Files

We introduce *script* and *function* M–files that should be used to write MATLAB programs.

- A *script* is a file containing MATLAB commands that are executed when the name of the file is entered at the prompt in the MATLAB command window. Scripts are convenient for conducting computational experiments that require entering many commands. However, care must be taken because variables in a script are global to the MATLAB session, and it is therefore easy to unintentionally change their values.

- A *function* is a file containing MATLAB commands that are executed when the function is called. The first line of a function must have the form:

  `function [`*out1, out2,...* `]` `=` *FunctionName(in1, in2, ...)*

  By default, any data or variables created within the function are private. You can specify any number of inputs to the function, and if you want it to return results, then you can specify any number of outputs. Functions can call other functions, so you can write sophisticated programs as in any conventional programming language.

In each case, the program must be saved in a M–file; that is, a file with a `.m` extension. Any editor may be used to create an M–file, but we recommend using MATLAB's built-in editor, which can be opened in one of several ways. In addition to clicking on certain menu items, the editor can be opened using the `edit` command. For example, if we enter

```
edit ChebyPlots.m
```

in the command window, then the editor will open the file ChebyPlots.m, and we can begin entering and modifying MATLAB commands.

It is important to consider carefully how the script and function M–files are to be named. As mentioned above, they should all have names of the form:

FunctionName`.m` or *ScriptName*`.m`

The names should be descriptive, but it is also important to avoid using a name already taken be one of MATLAB's many built-in functions. If we happen to use the same name as one of these built-in functions, then MATLAB has a way of choosing which function to use, but such a situation can be very confusing. The command `exist` can be used to determine if a function (or variable) is already defined by a specific name and the command `which` can be used to determine its path. Note, for a function file the name of the function and the name of the M–file must be the same.

To illustrate how to write functions and scripts, we provide two examples.

Example 1.3.2. In this example we write a simple function, `PlotCircle.m`, that generates a plot of a circle with radius r centered at the origin:

```
function PlotCircle(r)
%
%         PlotCircle(r)
%
%  This function plots a circle of radius r centered at the origin.
%  If no input value for r is specified, the default value is chosen
%  as r = 1.  We check for too many inputs and for negative input
%  and report errors in both cases.
%
if nargin < 2
   if nargin == 0
     r = 1;
   elseif r <= 0
     error('The input value should be > 0.')
   end
   theta = linspace(0, 2*pi, 200);
   x = r*cos(theta);
   y = r*sin(theta);
   plot(x, y)
   axis([-2*r,2*r,-2*r,2*r])
   axis square
else
  error('Too many input values.')
end
```

We can use it to generate plots of circles. For example, if we enter

```
>> PlotCircle(2)
```

then a circle of radius 2 centered at the origin is plotted, as shown in Figure 1.5. Note, we have chosen to plot 200 points equally spaced in the variable `theta`. If the resulting plotted circle seems a little "ragged" increase the number of points.

This code introduces some new MATLAB commands that require a little explanation:

- The percent symbol, %, is used to denote a comment in the program. On any line anything after a % symbol is treated as comment. All programs (functions and scripts) should contain comments to explain their purpose, and if there are any input and/or output variables.

- `nargin` is a built-in MATLAB command that can be used to count the number of input arguments to a function. If we run this code using `PlotCircle(2)`, `PlotCircle(7)`, etc., then on entry to this function, the value of `nargin` is 1. However, if we run this code using the command `PlotCircle` (with no specified input argument), then on entry to this function, the value of `nargin` is 0. We also use the command `nargin` to check for too many input values.

- The conditional command `if` is used to execute a statement if the expression is true. First we check that there are not too many inputs. Then, we check if `nargin` is 0 (that is, the input value has not been specified). In the latter case, the code sets the radius r to the default value of 1. Note, the difference between the statements `x = 0` and `x == 0`; the former sets the value of x to 0, while the latter checks to see if x and 0 are equal. Observe that the `if` statements are "nested"; the `end` statements are correspondingly nested. Each `end` encountered corresponds to the most recent `if`. The indenting used (and produced automatically by the MATLAB editor) should help you follow the command structure of the program. We also use an `elseif` statement to make sure the input value for the radius is not negative. The `doc` command can be used to find more detailed information on `if` and related statements such as `else` and `elseif`.

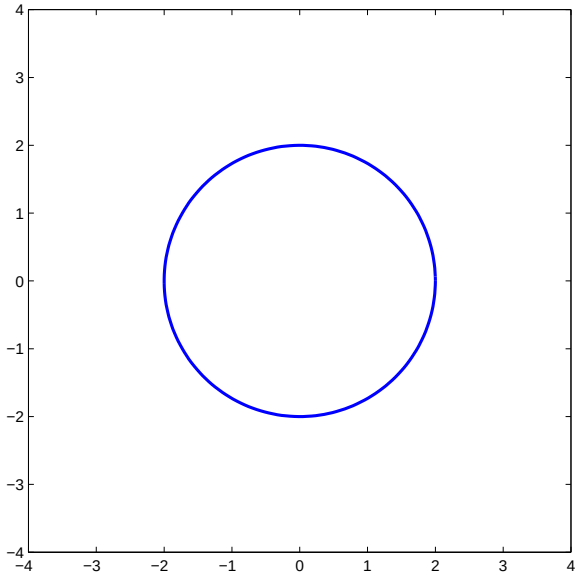

Figure 1.5: Figure created when entering `PlotCirle(2)`.

- `error` is a built-in MATLAB command. When this command is executed, the computation terminates, and the message between the single quotes is printed in the command window.

Example 1.3.3. Here, we illustrate how scripts can be used in combination with functions. Suppose that the concentration of spores of pollen per square centimeter are measured over a 15 day period, resulting in the following data:

day	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
pollen count	12	35	80	120	280	290	360	290	315	280	270	190	90	85	66

(a) First, we write a function that has as input a vector of data, and returns as output the mean and standard deviation of this data.

MATLAB has several built-in functions for statistical analysis. In particular, we can use `mean` and `std` to compute the mean and standard deviation, respectively. Thus, the function we write, named `GetStats.m`, is very simple:

```
function [m, s] = GetStats(x);
%
%   Compute the mean and standard deviation of entries
%   in a vector x.
%
%   Input:  x - vector (either row or column)
%
%   Output:  m - mean value of the elements in x.
%            s - standard deviation of the elements in x.
%
m = mean(x);
s = std(x);
```

```
%
%   Script:  PollenStats
%
%   This script shows a statistical analysis of measured pollen
%   counts on 15 consecutive days.
%

%
%  p is a vector containing the pollen count for each day.
%  d is a vector indicating the day (1 -- 15).
%
p = [12 35 80 120 280 290 360 290 315 280 270 190 90 85 66];
d = 1:length(p);
%
%  Get statistical data, and produce a plot of the results.
%
[m, s] = GetStats(p);
bar(d, p, 'b')
xlabel('Day of Measurement')
ylabel('Pollen Count')
hold on
plot([0, 16], [m, m], 'r')
plot([0, 16], [m-s, m-s], 'r--')
plot([0, 16], [m+s, m+s], 'r--')
text(16, m, '<-- m')
text(16, m-s, '<-- m - s')
text(16, m+s, '<-- m + s')
hold off
```

(b) We now write a script, `PollenStats.m`, to generate and display a statistical analysis of the pollen data.

Once the programs are written, we can run the script by simply entering its name in the command window:

```
>> PollenStats
```

The resulting plot is shown in Figure 1.6.

The script `PollenStats.m` uses several new commands. For detailed information on the specific uses of these commands, use `doc` (for example, `doc bar`). See if you can work out what is going on in the `plot` and `text` lines of `PollenStats.m`.

Problem 1.3.10. *Write a function M–file,* `PlotCircles.m`*, that accepts as input a vector,* `r`*, having positive entries, and plots circles of radius* `r(i)` *centered at the origin on the same axes. If no input value for* `r` *is specified, the default is to plot a single circle with radius* `r = 1`*. Test your function with* `r = 1:5`*. Hint:* MATLAB *commands that might be useful include* `max`*,* `min` *and* `any`*.*

1.3.4 Defining Mathematical Functions

In many computational science problems it is necessary to evaluate a function. For example, in order to numerically find the maximum of, say, $f(x) = 1/(1 + x^2)$, one way is to evaluate $f(x)$ a many different points. One approach is to write a function M–file, such as:

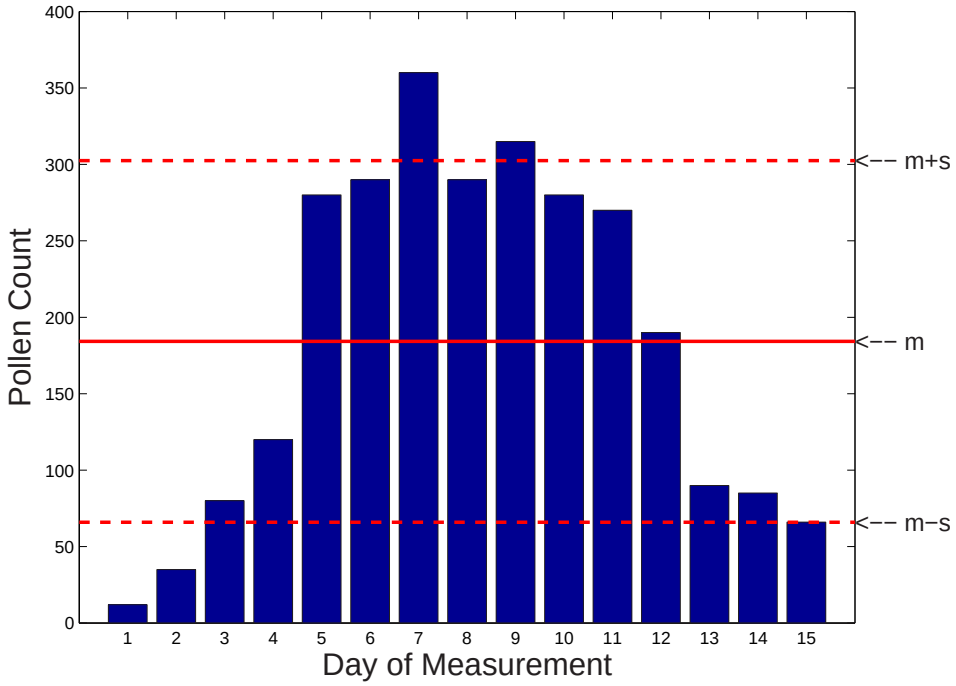

Figure 1.6: Bar graph summarizing the statistical analysis of pollen counts from Example 1.3.3

```
function f = Runge(x)
%
%     f = Runge(x);
%
%  This function evaluates Runge's function, f(x) = 1/(1 + x^2).
%
%  Input:  x - vector of x values
%
%  Output: f - vector of f(x) values.
%
f = 1 ./ (1 + x.*x);
```

To evaluate this function at, say, $x = 1.7$, we can use the MATLAB statement:

```
>> y = Runge(1.7)
```

Or, if we want to plot this function on the interval $-5 \leq x \leq 5$, we might use the MATLAB statements

```
>> x = linspace(-5, 5, 101);
>> y = Runge(x);
>> plot(x, y)
```

Although this approach works well, it is rather tedious to write a function M–file whose definition is just one line of code. Two alternative approaches are:

- We can use the `inline` construct to "define" the function:

```
>> Runge = inline('1 ./ (1 + x.*x)');
```

where the single quotes are needed. Here since there is only one variable, MATLAB assumes `Runge` is a function of x.

- Or, we can define `Runge` as an *anonymous* function:

```
>> Runge = @(x) 1 ./ (1 + x.*x);
```

Here the notation `@(x)` explicitly states that `Runge` is a function of x.

Once `Runge` has been defined, we can evaluate it at, say, $x = 1.7$, using the MATLAB statement:

```
>> y = Runge(1.7);
```

Or, to plot the function on the interval $-5 \leq x \leq 5$, we might use the MATLAB statements above.

Note that it is always best to use array operators (such as `./` and `.*`) when possible, so that functions can be evaluated at arrays of x-values. For example, if we did not use array operations in the above code, that is, if we had used the statement `Runge = @(x) 1 / (1 + x*x)`, (or the same definition for `f` in the function `Runge`) then we could compute the single value `Runge(1.7)`, but `y = Runge(x)` would produce an error if `x` was a vector, such as that defined by `x = linspace(-5, 5, 101)`.

Anonymous functions are new at MATLAB 7 and will eventually replace inline functions. They provide a powerful and easy to use construct for defining mathematical functions. Generally, if it is necessary to evaluate $f(x)$ many times, then MATLAB codes are more efficient if $f(x)$ is defined as a either a function M–file or an anonymous function, while the same code is (often substantially) less efficient if $f(x)$ is defined as an inline function. In general, if $f(x)$ can be defined with a single line of code, we use an anonymous function to define it, otherwise we use a function M–file.

Problem 1.3.11. *The* MATLAB *functions* `tic` *and* `toc` *can be used to find the (wall clock) time it takes to run a piece of code. For example, consider the following* MATLAB *statements:*

```
f = @(x) 1 ./ (1 + x.*x);
tic
for i = 1:n
  x = rand(n,1);
  y = f(x);
end
toc
```

When `tic` *is executed, the timer is started, and when* `toc` *is executed, the timer is stopped, and the time required to run the piece of code between the two statements is displayed in the* MATLAB *command window. Write a script M–file containing these statements. Replace the definition of $f(x)$ by an inline function and by a function M–file, in turn. Run the three codes for each of $n = 100, 200, 300, 400, 500$. What do you observe? Repeat this experiment. Do you observe any differences in the timings?*

Problem 1.3.12. *Repeat the experiment in Problem 1.3.11, but this time use* $f(x) = \dfrac{e^x + \sin \pi x}{x^2 + 7x + 4}$.

1.3.5 Creating Reports

Many problems in scientific computing (and in this book) require a combination of mathematical analysis, computational experiments, and a written summary of the results; that is, a report. Plots created in MATLAB can be easily printed for inclusion in the report. Probably the easiest approach is to pull down the *File* menu on the desired MATLAB figure window, and let go on *Print*.

Another option is to save the plot to a file on your computer using the *Save as* option under the *File* menu on the figure window. MATLAB supports many types of file formats, including encapsulated postscript (EPS) and portable document format (PDF), as well as many image compression formats such as TIFF, JPEG and PNG. These files can also be created in the command window, or

in any M–file, using the `print` command. For detailed information on creating such files, open the help browser to the associated reference page using the command `doc print`.

In addition to plots, it may be desirable to create tables of data for a report. Three useful commands that can be use for this purpose are `disp`, `sprintf` and `diary`. When used in combination, `disp` and `sprintf` can be used to write formatted output in the command window. The `diary` command can then be used to save the results in a file. For example, the following MATLAB code computes a compound interest table, with annual compounding:

```
a = 35000; n = 5;
r = 1:1:10;
rd = r/100;
t = a*(1 + rd).^n;
diary InterestInfo.dat
disp(' If a = 35,000 dollars is borrowed, to be paid in n = 5 years, then:')
disp(' ')
disp('      Interest rate, r%        Total paid after 5 years')
disp('     ================================================')
for i = 1:length(r)
   disp(sprintf('            %4.2f                    %8.2f     ', r(i), t(i)))
end
diary off
```

If the above code is put into a script or function M–file, then when the code is executed, it prints the following table of data in the command window:

```
 If a = 35,000 dollars is borrowed, to be paid in n = 5 years, then:

      Interest rate, r%        Total paid after 5 years
     ================================================
            1                     36785.35
            2                     38642.83
            3                     40574.59
            4                     42582.85
            5                     44669.85
            6                     46837.90
            7                     49089.31
            8                     51426.48
            9                     53851.84
           10                     56367.85
```

Here, the `diary` command is used to open a file named `InterestInfo.dat`, and any subsequent results displayed in the command window are automatically written in this file until it is closed with the `diary off` command. Thus, after execution, the above table is stored in the file `InterestInfo.dat`.

A word or warning about the `diary` command: If we create a script M–file containing the above code, and we run that script several times, the file `InterestInfo.dat` will collect the results from each run. In many cases the first set of runs may be used to debug the code, or to get the formatted output to look good, and it is only the final set that we want to save. In this case, it is recommended that you comment out the lines of code containing the `diary` commands (that is, put the symbol % in front of the commands) until you are ready to make the final run. Note that the % symbol is not treated as a comment when used in various print commands, such as `disp`, `sprintf` and `fprintf`.

Readers who wish to prepare more sophisticated reports and wish to have the freedom to choose their report's file format should consider using the more sophisticated MATLAB command `publish`.

Chapter 2

Numerical Computing

The first part of this chapter gives an elementary introduction to the representation of computer numbers and computer arithmetic. First, we introduce the computer numbers typically available on modern computers (and available through MATLAB). Next, we discuss the quality of the approximations produced by numeric computing.

In the second part of the chapter we present a number of simple examples that illustrate some of the pitfalls of numerical computing. There are seven such examples, each of them suitable for short computer projects. They illustrate a number of pitfalls, with particular emphasis on the effects of catastrophic cancelation, floating–point overflow and underflow, and the accumulation of roundoff errors.

2.1 Numbers

Most scientific programming languages (including MATLAB) provide users with at least two types of numbers: integers and floating–point numbers. In fact, MATLAB supports many other types too but we concentrate on integers and floating–point numbers as those most likely to be used in scientific computing. The floating–point numbers are a subset of the real numbers. All computer numbers are stored as a sequence of bits. A bit assumes one of two values, 0 or 1. So, if the number v is stored using 8 bits, we may write it

$$\text{stored}(v) = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

Here, b_i represents the i^{th} bit of v; the indices assigned to these bits start at 0 and increase to the left. Because each bit can assume the value 0 or 1, there are $2^8 = 256$ distinct patterns of bits. However, the value associated with each pattern depends on what *kind* of number v that represents.

2.1.1 Integers

The most commonly used type of integers are the **signed integers**. Most computers store signed integers using *two's complement notation*; in this representation the 8-bit signed integer i is stored as

$$\text{stored}(i) = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

which is assigned the value

$$\text{value}(i) = b_7(-2^7) + b_6(2^6) + \cdots + b_0(2^0)$$

When we compute with integers we normally use signed integers. So, for example, among the 256 different 8-bit signed integers, the smallest value is -128 and the largest is 127. This asymmetry arises because the total number of possible bit patterns, $2^8 = 256$, is even and one bit pattern, `00000000`, is assigned to zero.

An 8-bit **unsigned integer** j stored as

$$\text{stored}(j) = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

is assigned the value

$$\text{value}(j) = b_7(2^7) + b_6(2^6) + b_5(2^5) + \cdots + b_0(2^0)$$

Unsigned integers are used mainly in indexing and in the representation of floating–point numbers. Among the 256 different 8-bit unsigned integers, the smallest value, `00000000`, is 0 and the largest, `11111111`, is 255.

Example 2.1.1. Consider the decimal integer 13. It's representation as a signed integer is `00001101` and this is also its unsigned integer representation. Now consider the decimal integer -13. It has no representation as an unsigned integer. As a signed integer $-13 = -128+115$ so its two's complement binary representation is `11110011`.

Problem 2.1.1. *Determine a formula for the smallest and largest n-bit unsigned integers. What are the numerical values of the smallest and largest n-bit unsigned integers for each of $n = 8, 11, 16, 32, 64$?*

Problem 2.1.2. *Determine a formula for the smallest and largest n-bit signed integers. What are numerical values of the smallest and largest n-bit signed integers for each of $n = 8, 11, 16, 32, 64$?*

Problem 2.1.3. *List the value of each 8-bit signed integer for which the negative of this value is* not *also an 8-bit signed integer.*

2.1.2 Floating-Point Numbers

In early computers the types of floating–point numbers available depended not only on the programming language but also on the computer. The situation changed dramatically after 1985 with the widespread adoption of the *ANSI/IEEE 754-1985 Standard for Binary Floating–Point Arithmetic*. For brevity we'll call this the *Standard*. It defines several types of floating–point numbers. The most widely implemented is its 64**-bit double precision (DP)** type. In most computer languages the programmer can also use the 32**-bit single precision (SP)** type of floating–point numbers. If a program uses just one type of floating–point number, we refer to that type as the **working precision (WP)** floating–point numbers. (WP is usually either DPor SP. In MATLAB, WP=DP.) The important quantity **machine epsilon**, ϵ_{WP}, is the distance from 1 to the next larger number in the working precision.

Double Precision Numbers

An important property of double precision (DP) numbers is that *the computed result of every sum, difference, product, or quotient of a pair of DP numbers is a valid DP number*. For this property to be possible, the DP numbers in the *Standard* include representations of finite real numbers as well as the special numbers ± 0, $\pm\infty$ and NaN (Not-a-Number). For example, the DP quotient $1/0$ is the special number ∞ and the DP product $\infty \times 0$, which is mathematically undefined, has the value NaN. We focus on the normalized DP numbers.

In the Standard, a DP number y is stored as

$$\text{stored}(y) = \overbrace{\boxed{b_{63}}}^{s(y)}\overbrace{\boxed{b_{62}b_{61}\cdots b_{52}}}^{e(y)}\overbrace{\boxed{b_{51}b_{50}\cdots b_0}}^{f(y)}$$

The boxes illustrate how the 64 bits are partitioned into 3 fields: a 1-bit unsigned integer $s(y)$, the **sign bit of** y, an 11-bit unsigned integer $e(y)$, the **biased exponent of** y, and a 52-bit unsigned

integer $f(y)$, the **fraction of** y. Because the sign of y is stored explicitly, with $s(y) = 0$ for positive and $s(y) = 1$ for negative y, each DP number can be negated. The 11-bit unsigned integer $e(y)$ represents values from 0 through 2047:

- If $e(y) = 2047$, then y is a special number.
- If $0 < e(y) < 2047$, then y is a **normalized** floating–point number with value

$$\text{value}(y) = (-1)^{s(y)} \cdot \left\{1 + \frac{f(y)}{2^{52}}\right\} \cdot 2^{E(y)}$$

  where $E(y) = e(y) - 1023$ is the (unbiased) **exponent of** y.
- If $e(y) = 0$ and $f(y) \neq 0$, then y is a **denormalized** floating–point number with value

$$\text{value}(y) = (-1)^{s(y)} \cdot \left\{0 + \frac{f(y)}{2^{52}}\right\} \cdot 2^{-1022}$$

- If $e(y) = 0$ and $f(y) = 0$, then y has value **zero**.

The **significand of** y is $1 + \frac{f(y)}{2^{52}}$ for normalized numbers, $0 + \frac{f(y)}{2^{52}}$ for denormalized numbers, and 0 for zero. The normalized DP numbers y with exponent $E(y) = k$ belong to **binade** k. For example, binade -1 consists of the numbers y for which $|y| \in \left[\frac{1}{2}, 1\right)$ and binade 1 consists of the numbers y for which $|y| \in [2, 4)$. In general, binade k consists of the numbers y for which $2^k \leq |y| < 2^{k+1}$. Each DP binade contains the same number of positive DP numbers. We define the **DP machine epsilon** ϵ_{DP} as the distance from 1 to the next larger DP number. For a finite nonzero DP number y, we define $\text{ulp}_{\text{DP}}(y) \equiv \epsilon_{\text{DP}} 2^{E(y)}$ as the **DP unit-in-the-last-place of** y.

Example 2.1.2. Consider the real number $y = -1/6$. This number is not directly representable in the Standard but we will compute the nearest Standard floating–point number to it. Note that $-1/6 = -(8/6)2^{-3}$ so $E(y) = -3$, $s(y) = 1$, $e(y) = 1020$ (hence the number y is a normalized DP floating–point number), and $f(y)$ is the closest integer to $2^{52}/3$ (because $8/6 - 1 = 1/3$). This gives $f(y) = 1501199875790165$ as a decimal and 1010101010101010101010101010101010101010101010101010101 in binary.

Problem 2.1.4. *Write $y = 6.1$ as a normalized DP floating–point number*

Problem 2.1.5. *What are the smallest and largest possible significands for normalized DP numbers?*

Problem 2.1.6. *What are the smallest and largest positive normalized DP numbers?*

Problem 2.1.7. *How many DP numbers are in each binade? Sketch enough of the positive real number line so that you can place marks at the endpoints of each of the DP binades -3, -2, -1, 0, 1, 2 and 3. What does this sketch suggest about the spacing between successive positive DP numbers?*

Problem 2.1.8. *Show that $\epsilon_{DP} = 2^{-52}$.*

Problem 2.1.9. *Show that $y = q \cdot ulp_{DP}(y)$ where q is an integer with $1 \leq |q| \leq 2^{53} - 1$.*

Problem 2.1.10. *Show that neither $y = \frac{1}{3}$ nor $y = \frac{1}{10}$ is a DP number. Hint: If y is a DP number, then $2^m y$ is a 53-bit unsigned integer for some (positive or negative) integer m.*

Problem 2.1.11. *What is the largest positive integer n such that $2^n - 1$ is a DP number?*

Single Precision Numbers

In the Standard, a single precision (SP) number x is stored as

$$\text{stored}(x) = \overbrace{\boxed{b_{31}}}^{s(x)}\overbrace{\boxed{b_{30}b_{29}\cdots b_{23}}}^{e(x)}\overbrace{\boxed{b_{22}b_{21}\cdots b_0}}^{f(x)}$$

The boxes illustrate how the 32 bits are partitioned into 3 fields: a 1-bit unsigned integer $s(x)$, the **sign bit of** x, an 8-bit unsigned integer $e(x)$, the **biased exponent of** x, and a 23-bit unsigned integer $f(x)$, the **fraction of** x. The sign bit $s(x) = 0$ for positive and $s(x) = 1$ for negative SP numbers. The 8-bit unsigned integer $e(x)$ represents values in the range from 0 through 255:

- If $e(x) = 255$, then x is a special number.
- If $0 < e(x) < 255$, then x is a **normalized** floating–point number with value

$$\text{value}(x) = (-1)^{s(x)} \cdot \left\{1 + \frac{f(x)}{2^{23}}\right\} \cdot 2^{E(x)}$$

  where $E(x) \equiv e(x) - 127$ is the (unbiased) **exponent of** x.
- If $e(x) = 0$ and $f(x) \neq 0$, then x is a **denormalized** floating–point number with value

$$\text{value}(x) = (-1)^{s(x)} \cdot \left\{0 + \frac{f(x)}{2^{23}}\right\} \cdot 2^{-126}$$

- If both $e(x) = 0$ and $f(x) = 0$, then x is **zero**.

The **significand of** x is $1+\frac{f(x)}{2^{23}}$ for normalized numbers, $0+\frac{f(x)}{2^{23}}$ for denormalized numbers, and 0 for zero. The normalized SP numbers y with exponent $E(y) = k$ belong to **binade** k. For example, binade -1 consists of the numbers y for which $|y| \in \left[\frac{1}{2}, 1\right)$ and binade 1 consists of the numbers y for which $|y| \in [2, 4)$. In general, binade k consists of the numbers y for which $2^k \leq |y| < 2^{k+1}$. Each SP binade contains the same number of positive SP numbers. We define the **SP machine epsilon** ϵ_{SP} as the distance from 1 to the next larger SP number. For a finite nonzero SP number x, we define $\text{ulp}_{\text{SP}}(x) \equiv \epsilon_{\text{SP}} 2^{E(x)}$ as the **SP unit-in-the-last-place of** x.

Problem 2.1.12. *Write $y = 6.1$ as a normalized SP floating–point number*

Problem 2.1.13. *What are the smallest and largest possible significands for normalized SP numbers?*

Problem 2.1.14. *What are the smallest and largest positive normalized SP numbers?*

Problem 2.1.15. *How many SP numbers are in each binade? Sketch enough of the positive real number line so that you can place marks at the endpoints of each of the SP binades -3, -2, -1, 0, 1, 2 and 3. What does this sketch suggest about the spacing between consecutive positive SP numbers?*

Problem 2.1.16. *Compare the sketches produced in Problems 2.1.7 and 2.1.15. How many DP numbers lie between consecutive positive SP numbers? We say that distinct real numbers x and y are resolved by SP numbers if the SP number nearest x is not the same as the SP number nearest y. Write down the analogous statement for DP numbers. Given two distinct real numbers x and y, are x and y more likely to be resolved by SP numbers or by DP numbers? Justify your answer.*

Problem 2.1.17. *Show that $\epsilon_{SP} = 2^{-23}$.*

Problem 2.1.18. *Show that $x = p \cdot ulp_{SP}(x)$ where p is an integer with $1 \leq |p| \leq 2^{24} - 1$.*

Problem 2.1.19. *If x is a normalized SP number in binade k, what is the distance to the next larger SP number? Express your answer in terms of ϵ_{SP}.*

Problem 2.1.20. *Show that neither* $x = \frac{1}{3}$ *nor* $x = \frac{1}{10}$ *is a SP number. Hint: If* x *is a SP number, then* $2^m x$ *is an integer for some (positive or negative) integer* m*.*

Problem 2.1.21. *What is the largest positive integer* n *such that* $2^n - 1$ *is a SP number?*

2.2 Computer Arithmetic

What kind of numbers should a program use? Signed integers may be appropriate if the input data are integers and the only operations used are addition, subtraction, and multiplication. Floating–point numbers are appropriate if the data involves fractions, has large or small magnitudes, or if the operations include division, square root, or computing transcendental functions like the sine.

2.2.1 Integer Arithmetic

In integer computer arithmetic the computed sum, difference, or product of integers is the exact result, except when **integer overflow** occurs. This happens when to represent the result of an integer arithmetic operation more bits are needed than are available for the result. Thus, for the 8-bit signed integers $i = 126$ and $j = 124$ the computed value $i + j = 250$ is a number larger than the value of any 8-bit signed integer, so it overflows. So, *in the absence of integer overflow, integer arithmetic performed by the computer is exact*. That you have encountered integer overflow may not always be immediately apparent. The computer may simply return an (incorrect) integer result!

2.2.2 Floating-Point Arithmetic

The set of all integers is **closed** under the arithmetic operations of addition, subtraction, and multiplication. That is the sum, difference, or product of two integers is another integer. Similarly, the set of all real numbers is closed under the arithmetic operations of addition, subtraction, and multiplication, and division (except by zero). Unfortunately, *the set of all DP numbers is not closed under any of the operations of add, subtract, multiply, or divide.* For example, both $x = 2^{52} + 1$ and $y = 2^{52} - 1$ are DP numbers and yet their product $xy = 2^{104} - 1$ is not an DP number. To make the DP numbers closed under the arithmetic operations of addition, subtraction, multiplication, and division, the *Standard* modifies slightly the result produced by each operation. Specifically, it defines

$$\begin{aligned} x \oplus y &\equiv \mathrm{fl}_{\mathrm{DP}}(x + y) \\ x \ominus y &\equiv \mathrm{fl}_{\mathrm{DP}}(x - y) \\ x \otimes y &\equiv \mathrm{fl}_{\mathrm{DP}}(x \times y) \\ x \oslash y &\equiv \mathrm{fl}_{\mathrm{DP}}(x / y) \end{aligned}$$

Here, $\mathrm{fl}_{\mathrm{DP}}(z)$ is the DP number *closest* to the real number z; that is, $\mathrm{fl}_{\mathrm{DP}}()$ is the **DP rounding function**[1]. So, for example, the first equality states that $x \oplus y$, the value assigned to the sum of the DP numbers x and y, is the DP number $\mathrm{fl}_{\mathrm{DP}}(x + y)$. In a general sense, no approximate arithmetic on DP numbers can be more accurate than that specified by the *Standard*.

In summary, *floating–point arithmetic is inherently approximate; the computed value of any sum, difference, product, or quotient of DP numbers is equal to the exact value rounded to the nearest floating–point DP number.* In the next section we'll discuss how to measure the quality of this approximate arithmetic.

Problem 2.2.1. *Show that* $2^{104} - 1$ *is not a DP number. Hint: Recall Problem 2.1.11.*

Problem 2.2.2. *Show that each of* $2^{53} - 1$ *and* 2 *is a DP number, but that their sum is not a DP number. So, the set of all DP numbers is not closed under addition. Hint: Recall Problem 2.1.11.*

[1] When z is midway between two adjacent DP numbers, $\mathrm{fl}_{\mathrm{DP}}(z)$ is the one whose fraction is even.

Problem 2.2.3. *Show that the set of DP numbers is not closed under subtraction; that is, find two DP numbers whose difference is not a DP number.*

Problem 2.2.4. *Show that the set of DP numbers is not closed under division, that is find two nonzero DP numbers whose quotient is not a DP number. Hint: Consider Problem 2.1.10.*

Problem 2.2.5. *Assuming floating–point underflow does not occur, why can any DP number x be divided by 2 exactly? Hint: Consider the representation of x as a DP number.*

Problem 2.2.6. *Let x be a DP number. Show that $fl_{DP}(x) = x$.*

Problem 2.2.7. *Let each of x, y and $x + y$ be a DP number. What is the value of the DP number $x \oplus y$? State the extension of your result to the difference, product and quotient of DP numbers.*

Problem 2.2.8. *The real numbers x, y and z satisfy the* ***associative law of addition****:*

$$(x + y) + z = x + (y + z)$$

Consider the DP numbers $a = -2^{60}$, $b = 2^{60}$ and $c = 2^{-60}$. Show that

$$(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$$

So, in general DP addition is not associative. Hint: Show that $b \oplus c = b$.

Problem 2.2.9. *The real numbers x, y and z satisfy the* ***associative law of multiplication****:*

$$(x \times y) \times z = x \times (y \times z),$$

Consider the DP numbers $a = 1 + 2^{-52}$, $b = 1 - 2^{-52}$ and $c = 1.5 + 2^{-52}$. Show that

$$(a \otimes b) \otimes c \neq a \otimes (b \otimes c)$$

So, in general DP multiplication is not associative. Hint: Show that $a \otimes b = 1$ and $b \otimes c = 1.5 - 2^{-52}$.

Problem 2.2.10. *The real numbers x, y and z satisfy the* ***distributive law****:*

$$x \times (y + z) = (x \times y) + (x \times z)$$

Choose values of the DP numbers a, b and c such that

$$a \otimes (b \oplus c) \neq (a \otimes b) \oplus (a \otimes c)$$

So, in general DP arithmetic is not distributive.

Problem 2.2.11. *Define the SP rounding function $fl_{SP}()$ that maps real numbers into SP numbers. Define the values of $x \oplus y$, $x \ominus y$, $x \otimes y$ and $x \oslash y$ for SP arithmetic.*

Problem 2.2.12. *Show that $2^{24} - 1$ and 2 are SP numbers, but their sum is not a SP number. So, the set of all SP numbers is not closed under addition. Hint: Recall Problem 2.1.21.*

2.2.3 Quality of Approximations

To illustrate the use of approximate arithmetic we employ **normalized decimal scientific notation**. The *Standard* represents numbers in binary notation. We use decimal representation to simplify the development since the reader will be more familiar with decimal than binary arithmetic.

A nonzero real number T is represented as

$$T = \pm\, m(T) \cdot 10^{d(T)}$$

where $1 \le m(T) < 10$ is a real number and $d(T)$ is a positive, negative or zero integer. Here, $m(T)$ is the decimal **significand** and $d(T)$ is the decimal **exponent** of T. For example,

$$\begin{aligned} 120. &= (1.20) \cdot 10^2 \\ \pi &= (3.14159\ldots) \cdot 10^0 \\ -0.01026 &= (-1.026) \cdot 10^{-2} \end{aligned}$$

For the real number $T = 0$, we define $m(T) = 1$ and $d(T) = -\infty$.

For any integer k, **decade** k is the set of real numbers whose exponents $d(T) = k$. So, the decade k is the set of real numbers with values whose magnitudes are in the half open interval $[10^k, 10^{k+1})$.

For a nonzero number T, its i^{th} **significant digit** is the i^{th} digit of $m(T)$, counting to the right starting with the units digit. So, the units digit is the 1^{st} significant digit, the tenths digit is the 2^{nd} significant digit, the hundredths digit is the 3^{rd} significant digit, etc. For the value π listed above, the 1^{st} significant digit is 3, the 2^{nd} significant digit is 1, the 3^{rd} significant digit is 4, etc.

Frequently used measures of the error $A - T$ in A as an approximation to the true value T are

$$\begin{aligned} \text{absolute error} &= |A - T| \\ \text{absolute relative error} &= \left|\frac{A-T}{T}\right| \end{aligned}$$

with the relative error being defined only when $T \neq 0$. The approximation A to T is said to be q**–digits accurate** if the absolute error is less than $\frac{1}{2}$ of one unit in the q^{th} significant digit of T. Since $1 \le m(T) < 10$, A is a q–digits approximation to T if

$$|A - T| \le \frac{1}{2}|m(T)|10^{d(T)}10^{-q} \le \frac{10^{d(T)-q+1}}{2}$$

If the absolute relative error in A is less than r, then A is a q–digits accurate approximation to T provided that $q \le -\log_{10}(2r)$.

Returning now to binary representation, whenever $\text{fl}_{\text{DP}}(z)$ is a normalized DP number,

$$\text{fl}_{\text{DP}}(z) = z(1+\mu) \quad \text{where} \quad |\mu| \le \frac{\epsilon_{\text{DP}}}{2}$$

The value $|\mu|$, the absolute relative error in $\text{fl}_{\text{DP}}(z)$, depends on the value of z. Then

$$\begin{aligned} x \oplus y &= (x+y)(1+\mu_a) \\ x \ominus y &= (x-y)(1+\mu_s) \\ x \otimes y &= (x \times y)(1+\mu_m) \\ x \oslash y &= (x/y)(1+\mu_d) \end{aligned}$$

where the absolute relative errors $|\mu_a|$, $|\mu_s|$, $|\mu_m|$ and $|\mu_d|$ are each no larger than $\frac{\epsilon_{\text{DP}}}{2}$.

Problem 2.2.13. *Show that if A is an approximation to T with an absolute relative error less than $0.5 \cdot 10^{-16}$, then A is a 16–digit accurate approximation to T.*

Problem 2.2.14. *Let r be an upper bound on the absolute relative error in the approximation A to T, and let q be an integer that satisfies $q \le -\log_{10}(2r)$. Show that A is a q–digit approximation to T. Hint: Show that $|T| \le 10^{d(T)+1}$, $r \le \frac{10^{-q}}{2}$, and $|A - T| \le \frac{10^{-q}|T|}{2} \le \frac{10^{d(T)-q+1}}{2}$. This last inequality demonstrates that A is q–digits accurate.*

Problem 2.2.15. *Let z be a real number for which $fl_{DP}(z)$ is a normalized DP number. Show that $|fl_{DP}(z) - z|$ is at most half a DP ulp times the value of $fl_{DP}(z)$. Then show that $\mu \equiv \frac{fl_{DP}(z) - z}{z}$ satisfies the bound $|\mu| \le \frac{\epsilon_{DP}}{2}$.*

Problem 2.2.16. *Let x and y be DP numbers. The absolute relative error in $x \oplus y$ as an approximation to $x+y$ is no larger than $\frac{\epsilon_{DP}}{2}$. Show that $x \oplus y$ is about 15 digits accurate as an approximation to $x+y$. How does the accuracy change if you replace the addition operation by any one of subtraction, multiplication, or division (assuming $y \neq 0$)?*

2.2.4 Propagation of Errors

There are two types of errors in any computed sum, difference, product, or quotient. The first is the error that is inherent in the numbers, and the second is the error introduced by the arithmetic.

Let x' and y' be DP numbers and consider computing $x' \times y'$. Commonly, x' and y' are the result of a previous computation. Let x and y be the values that x' and y' would have been if they had been computed exactly; that is,

$$x' = x(1 + \mu_x), \quad y' = y(1 + \mu_y)$$

where μ_x and μ_y are the relative errors in the approximations of x' to x and of y' to y, respectively. Now, $x' \times y'$ is computed as $x' \otimes y'$, so

$$x' \otimes y' = \mathrm{fl}_{\mathrm{DP}}(x' \times y') = (x' \times y')(1 + \mu_m)$$

where μ_m is the relative error in the approximation $x' \otimes y'$ of $x' \times y'$. How well does $x' \otimes y'$ approximate the exact result $x \times y$? We find that

$$x' \otimes y' = (x \times y)(1 + \mu)$$

where $\mu = (1 + \mu_x)(1 + \mu_y)(1 + \mu_m) - 1$. Expanding the products $\mu = \mu_x + \mu_y + \mu_m + \mu_x\mu_y + \mu_x\mu_m + \mu_y\mu_m + \mu_x\mu_y\mu_m$. Generally, we can drop the terms involving products of the μ's because the magnitude of each value μ is small relative to 1, so the magnitudes of products of the μ's are smaller still. Consequently

$$\mu \approx \mu_x + \mu_y + \mu_m$$

So the relative error in $x' \otimes y'$ is (approximately) equal to the sum of

a. the relative errors *inherent* in each of the values x' and y'

b. the relative error *introduced* when the values x' and y' are multiplied

In an extended computation, we expect the error in the final result to come from the (hopefully slow) accumulation of the errors in the initial data and of the errors from the arithmetic operations on that data. This is a major reason why DP arithmetic is mainly used in general scientific computation. While the final result may be represented adequately by a SP number, we use DP arithmetic in an attempt to reduce the *effect* of the accumulation of errors introduced by the computer arithmetic because the relative errors μ in DP arithmetic are so much smaller than in SP arithmetic.

Relatively few DP computations produce exactly a DP number. However, subtraction of nearly equal DP numbers of the same sign is always exact and so is always a DP number. This **exact cancelation** result is stated mathematically as follows:

$$x' \ominus y' = x' - y'$$

whenever $\frac{1}{2} \leq \frac{x'}{y'} \leq 2$. So, $\mu_s = 0$ and we expect that $\mu \approx \mu_x + \mu_y$. It is easy to see that

$$x' \ominus y' = x' - y' = (x - y)(1 + \mu)$$

where $\mu = \frac{x\mu_x - y\mu_y}{x - y}$. We obtain an upper bound on μ by applying the triangle inequality:

$$\frac{|\mu|}{|\mu_x| + |\mu_y|} \leq g \equiv \frac{|x| + |y|}{|x - y|}$$

The left hand side measures how the relative errors μ_x and μ_y in the values x' and y', respectively, are *magnified* to produce the relative error μ in the value $x' - y'$. The right hand side, g, is an upper bound on this magnification. Observe that $g \geq 1$ and that g grows as $|x - y|$ gets smaller, that is as more cancelation occurs in computing $x - y$. When g is large, the relative error in $x' - y'$ *may be large* because **catastrophic cancelation** has occurred. In the next section we present examples where computed results suffer from the effect of catastrophic cancelation.

Example 2.2.1. We'll give a simple example to show how the floating–point addition works. To simplify, we'll work in three decimal digit floating–point arithmetic. Say, we are to compute 1/3+8/7. Since neither $x = 1/3$ nor $y = 8/7$ are representable we must first round so that $fl(1/3) = 3.33 * 10^{-1} = (1/3)(1 - 0.001)$ and $fl(8/7) = 1.14 * 10^0 = (8/7)(1 - 0.0025)$; note, $\epsilon \approx 0.005$. So, each of these numbers is a normalized three digit decimal number. Now, $x \oplus y = 1.47 * 10^0$ which is again a normalized three digit decimal number. And, $x \oplus y = (x + y)(1 - 0.0042)$, and we observe the effects of propagation of rounding error.

Problem 2.2.17. *Work through the analysis of $x \oplus y$ and $x \ominus y$ for $x = 1/11$ and $y = 4/3$.*

Problem 2.2.18. *Derive the expression*

$$x' \ominus y' = x' - y' = (x - y)(1 + \mu)$$

where $\mu \equiv \dfrac{x\mu_x - y\mu_y}{x - y}$. *Show that*

$$|x\mu_x - y\mu_y| \leq |x||\mu_x| + |y||\mu_y| \leq (|x| + |y|)(|\mu_x| + |\mu_y|)$$

and that

$$\left| \frac{x\mu_x - y\mu_y}{x - y} \right| \leq (|\mu_x| + |\mu_y|) \frac{|x| + |y|}{|x - y|}$$

2.3 Examples

The following examples illustrate some less obvious pitfalls in simple scientific computations.

2.3.1 Plotting a Polynomial

Consider the polynomial $p(x) = (1 - x)^{10}$, which can be written in power series form as

$$p(x) = x^{10} - 10\,x^9 + 45\,x^8 - 120\,x^7 + 210\,x^6 - 252\,x^5 + 210\,x^4 - 120\,x^3 + 45\,x^2 - 10\,x + 1$$

Suppose we use this power series to evaluate and plot $p(x)$. For example, if we use DP arithmetic to evaluate $p(x)$ at 101 equally spaced points in the interval $[0, 2]$ (so that the spacing between points is 0.02), and plot the resulting values, we obtain the curve shown on the left in Fig. 2.1. The curve has a shape we might expect. However, if we attempt to zoom in on the region near $x = 1$ by using DP arithmetic to evaluating $p(x)$ at 101 equally spaced points in the interval $[0.99, 1.01]$ (so that the spacing is 0.0002), and plot the resulting values, then we obtain the curve shown on the right in Fig. 2.1. In this case, the plot suggests that $p(x)$ has many zeros in the interval $[0.99, 1.01]$. (Remember, if $p(x)$ changes sign at two points then, by continuity, $p(x)$ has a zero somewhere between those points.) However, the factored form $p(x) = (1 - x)^{10}$ implies there is only a single zero, of multiplicity 10, at the point $x = 1$. Roundoff error incurred while evaluating the power series and the effects of cancelation of the rounded values induce this inconsistency.

In Fig. 2.1, the maximum amplitudes of the oscillations are larger to the right of $x = 1$ than to the left. Recall that $x = 1$ is the boundary between binade -1 and binade 0. As a result, the magnitude of the maximum error incurred by rounding can be a factor of 2 larger to the right of $x = 1$ than to the left, which is essentially what we observe.

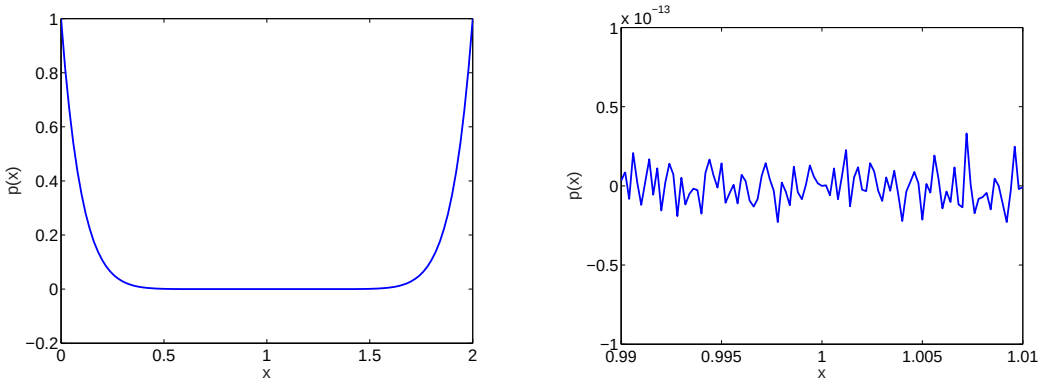

Figure 2.1: Plot of the power form of $p(x) \equiv (1-x)^{10}$ evaluated in DP arithmetic. The left shows a plot of $p(x)$ using 101 equally spaced points on the interval $0 \leq x \leq 2$. The right zooms in on the region near $x = 1$ by plotting $p(x)$ using 101 equally spaced points on the interval $0.99 \leq x \leq 1.01$.

2.3.2 Repeated Square Roots

Let n be a positive integer and let x be a DP number such that $1 \leq x < 4$. Consider the following procedure. First, initialize the DP variable $t := x$. Next, take the square root of t a total of n times via the assignment $t := \sqrt{t}$. Then, square the resulting value t a total of n times via the assignment $t := t * t$. Finally, print the value of error, $x - t$. The computed results of an experiment for various values of x, using $n = 100$, are shown in Table 2.1. But for rounding, the value of the error in the third column would be zero, but the results show errors much larger than zero! To understand what is happening, observe that taking the square root of a number discards information. The square root function maps the interval $[1, 4)$ onto the binade $[1, 2)$. Now $[1, 4)$ is the union of the binades $[1, 2)$ and $[2, 4)$. Furthermore, each binade contains 2^{53} DP numbers. So, the square root function maps each of $2 \cdot 2^{53} = 2^{54}$ arguments into one of 2^{53} possible square roots. On average, then, the square root function maps two DP numbers in $[1, 4)$ into one DP number in $[1, 2)$. So, generally, the DP square root of a DP argument does not contain sufficient information to recover that DP argument, that is taking the DP square root of a DP number usually loses information.

x	t	$x - t$
1.0000	1.0000	0.0000
1.2500	1.0000	0.2500
1.5000	1.0000	0.5000
1.7500	1.0000	0.7500
2.0000	1.0000	1.0000
2.2500	1.0000	1.2500
2.5000	1.0000	1.5000
2.7500	1.0000	1.7500
3.0000	1.0000	2.0000
3.2500	1.0000	2.2500
3.5000	1.0000	2.5000
3.7500	1.0000	2.7500

Table 2.1: Results of the repeated square root experiment. Here x is the exact value, t is the computed result (which in exact arithmetic should be the same as x), and $x - t$ is the error.

2.3.3 Estimating the Derivative

Consider the forward difference estimate

$$\Delta_{\text{E}} f(x) \equiv \frac{f(x+h) - f(x)}{h}$$

of the derivative $f'(x)$ of a continuously differentiable function $f(x)$. Assuming the function $f(x)$ is sufficiently smooth in an interval containing both x and $x+h$, for small values of the increment h, a Taylor series expansion (see Problem 2.3.1) gives the approximation

$$\Delta_{\text{E}} f(x) \approx f'(x) + \frac{h f''(x)}{2}$$

As the increment $h \to 0$ the value of $\Delta_{\text{E}} f(x) \to f'(x)$, a fact familiar from calculus.

Suppose, when computing $\Delta_{\text{E}} f(x)$, that the only errors that occur involve when rounding the exact values of $f(x)$ and $f(x+h)$ to working precision (WP) numbers, that is

$$\text{fl}_{\text{WP}}(f(x)) = f(x)(1+\mu_1), \quad \text{fl}_{\text{WP}}(f(x+h)) = f(x+h)(1+\mu_2)$$

where μ_1 and μ_2 account for the errors in the rounding. If $\Delta_{\text{WP}} f(x)$ denotes the computed value of $\Delta_{\text{E}} f(x)$, then

$$\begin{aligned}
\Delta_{\text{WP}} f(x) &\equiv \frac{\text{fl}_{\text{WP}}(f(x+h)) - \text{fl}_{\text{WP}}(f(x))}{h} \\
&= \frac{f(x+h)(1+\mu_2) - f(x)(1+\mu_1)}{h} \\
&= \Delta_{\text{E}} f(x) + \frac{f(x+h)\mu_2 - f(x)\mu_1}{h} \\
&\approx f'(x) + \frac{1}{2} h f''(x) + \frac{f(x+h)\mu_2 - f(x)\mu_1}{h}
\end{aligned}$$

where each absolute relative error $|\mu_i| \leq \dfrac{\epsilon_{\text{WP}}}{2}$. (For SP arithmetic $\epsilon_{\text{WP}} = 2^{-23}$ and for DP $\epsilon_{\text{WP}} = 2^{-52}$.) Hence, we obtain

$$\begin{aligned}
r = \left| \frac{\Delta_{\text{WP}} f(x) - f'(x)}{f'(x)} \right| &\approx h \left| \frac{f''(x)}{2f'(x)} \right| + \frac{1}{h} \left| \frac{f(x+h)\mu_2 - f(x)\mu_1}{f'(x)} \right| \\
&\approx h \left| \frac{f''(x)}{2f'(x)} \right| + \frac{1}{h} \left| \frac{f(x)(|\mu_2| + |\mu_1|)}{f'(x)} \right| \leq c_1 h + \frac{c_2}{h} \equiv R
\end{aligned}$$

an approximate upper bound in accepting $\Delta_{\text{WP}} f(x)$ as an approximation of the value $f'(x)$, where $c_1 \equiv \left| \dfrac{f''(x)}{2f'(x)} \right|$ and $c_2 \equiv \left| \dfrac{f(x)\epsilon_{\text{WP}}}{f'(x)} \right|$. In deriving this upper bound we have assumed that $f(x+h) \approx f(x)$ which is valid when h is sufficiently small and $f(x)$ is continuous on the interval $[x, x+h]$.

Consider the expression for R. The term $c_1 h \to 0$ as $h \to 0$, accounting for the error in accepting $\Delta_{\text{E}} f(x)$ as an approximation of $f'(x)$. The term $\dfrac{c_2}{h} \to \infty$ as $h \to 0$, accounting for the error arising from using the computed, rather than the exact, values of $f(x)$ and $f(x+h)$. So, as $h \to 0$, we might expect the absolute relative error in the forward difference estimate $\Delta_{\text{WP}} f(x)$ of the derivative first to decrease and then to increase. Consider using SP arithmetic, the function $f(x) \equiv \sin(x)$ with $x = 1$ radian, and the sequence of increments $h \equiv 2^{-n}$ for $n = 1, 2, 3, \cdots, 22$. Fig. 2.2 uses dots to display $-\log_{10}(2r)$, the digits of accuracy in the computed forward difference estimate of the derivative, and a solid curve to display $-\log_{10}(2R)$, an estimate of the minimum digits of accuracy obtained from the model of rounding. Note, the forward difference estimate becomes more accurate as the curve increases, reaching a maximum accuracy of about 4 digits at $n = 12$, and then becomes less accurate as the curve decreases. The maximum accuracy (that is the minimum value of R) predicted by the model is approximately proportional to the square root of the working-precision.

For h sufficiently small, what we have observed is *catastrophic cancelation* of the values of $f(x)$ and $f(x+h)$ followed by *magnification* of the effect of the resulting loss of significant digits by division by a small number, h.

We will revisit the problem of numerical differentiation in Chapter 5.

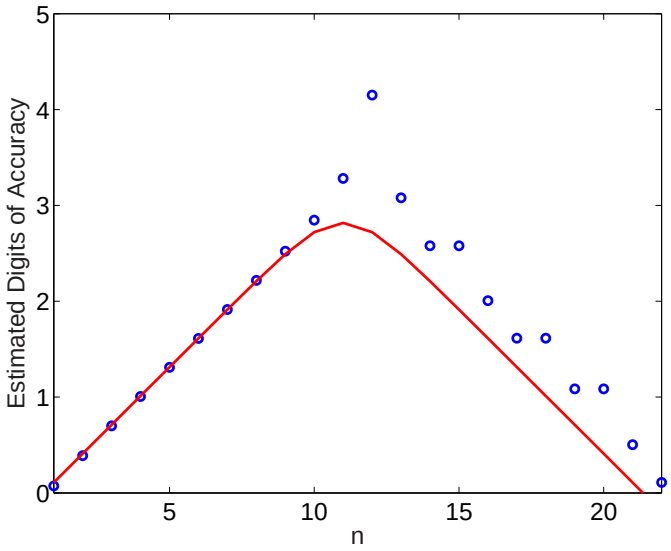

Figure 2.2: Forward difference estimates: $\Delta_{\text{SP}} f(x)$ (dots) and R (solid)

Problem 2.3.1. *A* **Taylor series** *expansion of a function $f(z)$ about the point c is defined to be*

$$f(z) = f(c) + (z-c)f'(c) + (z-c)^2 \frac{f''(c)}{2!} + (z-c)^3 \frac{f'''(c)}{3!} + \cdots$$

Conditions that guarantee when such a series expansion exists are addressed in a differential calculus course. The function must be sufficiently smooth in the sense that all of the derivatives $f^{(n)}(z)$, $n = 0, 1, 2, \ldots$, must exist in an open interval containing c. In addition, the series must converge, which means the terms of the series must approach zero sufficiently quickly, for all z values in an open interval containing c.

Suppose f is sufficiently smooth in an interval containing $z = x$ and $z = x + h$, where $|h|$ is small. Use the above Taylor series with $z = x + h$ and $c = x$ to show that

$$f(x+h) = f(x) + hf'(x) + h^2 \frac{f''(x)}{2!} + h^3 \frac{f'''(x)}{3!} + \cdots$$

and argue that

$$\Delta_{\text{E}} f(x) \equiv \frac{f(x+h) - f(x)}{h} \approx f'(x) + \frac{hf''(x)}{2}\,.$$

Problem 2.3.2. *Let the function $R(h) \equiv c_1 h + \dfrac{c_2}{h}$ where c_1, c_2 are positive constants. For what value of h does $R(h)$ attain its smallest value. Give an expression for this smallest value? Now, substitute $c_1 \equiv \left|\dfrac{f''(x)}{2f'(x)}\right|$ and $c_2 \equiv \left|\dfrac{f(x)\epsilon_{WP}}{f'(x)}\right|$ to show how the smallest value of $R(h)$ depends on $\sqrt{\epsilon_{WP}}$. For $h = 2^{-n}$, what integer n corresponds most closely to this smallest value?*

2.3.4 A Recurrence Relation

Consider the sequence of values $\{V_j\}_{j=0}^{\infty}$ defined by the definite integrals

$$V_j = \int_0^1 e^{x-1} x^j dx, \quad j = 0, 1, 2, \cdots$$

It can easily be seen that $0 < V_{j+1} < V_j < \frac{1}{j}$ for all $j \geq 1$; see Problem 2.3.4.

Integration by parts demonstrates that the values $\{V_j\}_{j=0}^{\infty}$ satisfy the *recurrence relation*

$$V_j = 1 - jV_{j-1}, \quad j = 1, 2, \cdots$$

Because we can calculate the value of

$$V_0 = \int_0^1 e^{x-1} dx = 1 - \frac{1}{e}$$

we can determine the values of $V_1, V_2, \cdots$ using the recurrence starting from the computed estimate for V_0. Table 2.2 displays the values $\hat{V}_j$ for $j = 0, 1, \cdots, 16$ computed by evaluating the recurrence in SP arithmetic.

j	$\hat{V}_j$	$\frac{\hat{V}_j}{j!}$
0	6.3212E-01	6.3212E-01
1	3.6788E-01	3.6788E-01
2	2.6424E-01	1.3212E-01
3	2.0728E-01	3.4546E-02
4	1.7089E-01	7.1205E-03
5	1.4553E-01	1.2128E-03
6	1.2680E-01	1.7611E-04
7	1.1243E-01	2.2307E-05
8	1.0056E-01	2.4941E-06
9	9.4933E-02	2.6161E-07
10	5.0674E-02	1.3965E-08
11	4.4258E-01	1.1088E-08
12	-4.3110E+00	-8.9999E-09
13	5.7043E+01	9.1605E-09
14	-7.9760E+02	-9.1490E-09
15	1.1965E+04	9.1498E-09
16	-1.9144E+05	-9.1498E-09

Table 2.2: Values of V_j determined using SP arithmetic.

The first few values $\hat{V}_j$ are positive and form a decreasing sequence. However, the values $\hat{V}_j$ for $j \geq 11$ alternate in sign and increase in magnitude, contradicting the mathematical properties of the sequence. To understand why, suppose that the only rounding error that occurs is in computing V_0. So, instead of the correct initial value V_0 we have used instead the rounded initial value $\hat{V}_0 = V_0 + \epsilon$. Let $\{\hat{V}_j\}_{j=0}^{\infty}$ be the modified sequence determined exactly from the value $\hat{V}_0$. Using the recurrence, the first few terms of this sequence are

$$\begin{array}{lllll} \hat{V}_1 = 1 - \hat{V}_0 & = & 1 - (V_0 + \epsilon) & = & V_1 - \epsilon \\ \hat{V}_2 = 1 - 2\hat{V}_1 & = & 1 - 2(V_1 - \epsilon) & = & V_2 + 2\epsilon \\ \hat{V}_3 = 1 - 3\hat{V}_2 & = & 1 - 3(V_2 + 2\epsilon) & = & V_3 - 3 \cdot 2\epsilon \\ \hat{V}_4 = 1 - 4\hat{V}_3 & = & 1 - 4(V_3 - 3 \cdot 2\epsilon) & = & V_4 + 4 \cdot 3 \cdot 2\epsilon \\ \vdots & & & & \end{array} \tag{2.1}$$

In general,

$$\hat{V}_j = V_j + (-1)^j j!\epsilon$$

Now, the computed value $\hat{V}_0$ should have an absolute error ϵ no larger than half a ulp in the SP value of V_0. Because $V_0 \approx 0.632$ is in binade -1, $\epsilon \approx \dfrac{\text{ulp}_{\text{SP}}(V_0)}{2} = 2^{-25} \approx 3 \cdot 10^{-8}$. Substituting this value for ϵ we expect the first few terms of the computed sequence to be positive and decreasing as theory predicts. However, because $j! \cdot (3 \cdot 10^{-8}) > 1$ for all $j \geq 11$ and because $V_j < 1$ for all values of j, the formula for $\hat{V}_j$ leads us to expect that the values $\hat{V}_j$ will ultimately be dominated by $(-1)^j j!\epsilon$ and so will alternate in sign and increase in magnitude. That this analysis gives a reasonable prediction is verified by Table 2.2, where the error grows like $j!$, and $\left|\dfrac{\hat{V}_j}{j!}\right|$ approaches a constant.

What we have observed is the potential for significant *accumulation* and *magnification* of rounding error in a simple process with relatively few steps.

We emphasize that we do not recommend using the recurrence to evaluate the integrals $\hat{V}_j$. They may be evaluated simply either symbolically or numerically using MATLAB integration software; see Chapter 5.

Problem 2.3.3. *Use integration by parts to show the recurrence*

$$V_j = 1 - jV_{j-1}, \quad j = 1, 2, \cdots$$

for evaluating the integral

$$V_j = \int_0^1 e^{x-1} x^j dx, \quad j = 0, 1, 2, \cdots$$

Problem 2.3.4. *In this problem we show that the sequence $\{V_j\}_{j=0}^{\infty}$ has positive terms and is strictly decreasing to 0.*

(a) Show that for $0 < x < 1$ we have $0 < x^{j+1} < x^j$ for $j \geq 0$. Hence, show that $0 < V_{j+1} < V_j$.

(b) Show that for $0 < x < 1$ we have $0 < e^{x-1} < 1$. Hence, show that $0 < V_j < \dfrac{1}{j}$.

(c) Hence, show that the sequence $\{V_j\}_{j=0}^{\infty}$ has positive terms and is strictly decreasing to 0.

Problem 2.3.5. *The error in the terms of the sequence $\{\hat{V}_j\}_{j=0}^{\infty}$ grows because $\hat{V}_{j-1}$ is multiplied by j. To obtain an accurate approximation of V_j, we can instead run the recurrence backwards*

$$\hat{V}_j = \frac{1 - \hat{V}_{j+1}}{j+1}, \quad j = M-1, M-2, \cdots, 1, 0$$

Now, the error in $\hat{V}_j$ is divided by j. More specifically, if you want accurate SP approximations of the values V_j for $0 \leq j \leq N$, start with $M = N + 12$ and $\hat{V}_M = 0$ and compute the values of $\hat{V}_j$ for all $j = M-1, M-2, \cdots, 1, 0$. We start 12 terms beyond the first value of $\hat{V}_N$ so that the error associated with using $\hat{V}_{N+12} = 0$ will be divided by at least $12! \approx 4.8 \cdot 10^8$, a factor large enough to make the contribution of the initial error in $\hat{V}_M$ to the error in $\hat{V}_N$ less than a SP ulp in V_N. (We know that $\hat{V}_M$ is in error – from Problem 2.3.4, V_M is a small positive number such that $0 < V_M < \frac{1}{M}$, so $|\hat{V}_M - V_M| = |V_M| < \frac{1}{M}$.)

2.3.5 Summing the Exponential Series

From calculus, the Taylor series for the exponential function

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

converges mathematically for all values of x both positive and negative. Let the term

$$T_n \equiv \frac{x^n}{n!}$$

and define the partial sum of the first n terms

$$S_n \equiv 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^{n-1}}{(n-1)!}$$

so that $S_{n+1} = S_n + T_n$ with $S_0 = 0$.

Let's use this series to compute $e^{-20.5} \approx 1.25 \cdot 10^{-9}$. For $x = -20.5$, the terms alternate in sign and their magnitudes increase until we reach the term $T_{20} \approx 7.06 \cdot 10^7$ and then they alternate in sign and their magnitudes steadily decrease to zero. So, for any value $n > 20$, from the theory of alternating series

$$|S_n - e^{-20.5}| < |T_n| .$$

That is, the absolute error in the partial sum S_n as an approximation to the value of $e^{-20.5}$ is less than the absolute value of the first neglected term, $|T_n|$. Note that DP arithmetic is about 16–digits accurate. So, if a value $n \geq 20$ is chosen so that $\frac{|T_n|}{e^{-20.5}} < 10^{-16}$, then S_n should be a 16–digit approximation to $e^{-20.5}$. The smallest value of n for which this inequality is satisfied is $n = 98$, so S_{98} should be a 16–digit accurate approximation to $e^{-20.5}$. Table 2.3 displays a selection of values of the partial sums S_n and the corresponding terms T_n computed using DP arithmetic.

n	T_n	S_{n+1}
0	1.0000e+00	1.0000E+00
5	-3.0171e+04	-2.4057e+04
10	3.6122e+06	2.4009e+06
15	-3.6292e+07	-2.0703e+07
20	7.0624e+07	3.5307e+07
25	-4.0105e+07	-1.7850e+07
30	8.4909e+06	3.4061e+06
35	-7.8914e+05	-2.8817e+05
40	3.6183e+04	1.2126e+04
45	-8.9354e+02	-2.7673e+02
50	1.2724e+01	3.6627e+00
55	-1.1035e-01	-2.9675e-02
60	6.0962e-04	1.5382e-04
65	-2.2267e-06	-5.2412e-07
70	5.5509e-09	6.2893e-09
75	-9.7035e-12	5.0406e-09
80	1.2178e-14	5.0427e-09
85	-1.1201e-17	5.0427e-09
90	7.6897e-21	5.0427e-09
95	-4.0042e-24	5.0427e-09
98	3.7802e-26	5.0427e-09

Table 2.3: Values of S_n and T_n determined using DP arithmetic.

From the table, it appears that the sequence of values S_n is converging. However, $S_{98} \approx 5.04 \cdot 10^{-9}$, and the true value should be $e^{-20.5} \approx 1.25 \cdot 10^{-9}$. Thus, S_{98} is an approximation of $e^{-20.5}$ that is 0–digits accurate! At first glance, it appears that the computation is wrong! However, the computed value of the sum, though very inaccurate, is reasonable. Recall that DP numbers are about 16 digits accurate, so the largest term in magnitude, T_{20}, could be in error by as much as about $|T_{20}| \cdot 10^{-16} \approx 7.06 \cdot 10^7 \cdot 10^{-16} = 7.06 \cdot 10^{-9}$. Of course, changing T_{20} by this amount will change the value of the partial sum S_{98} similarly. So, using DP arithmetic, it is unlikely that the computed value S_{98} will provide an estimate of $e^{-20.5}$ with an absolute error much less than $7.06 \cdot 10^{-9}$.

So, how can we calculate $e^{-20.5}$ accurately using Taylor series? Clearly we must avoid the *catastrophic cancelation* involved in summing an alternating series with large terms when the value of the sum is a much smaller number. There follow two alternative approaches which exploit simple mathematical properties of the exponential function.

1. Consider the relation $e^{-x} = \dfrac{1}{e^x}$. If we use a Taylor series for e^x for $x = 20.5$ it still involves large and small terms T_n but they are all positive and there is no cancellation in evaluating the sum S_n. In fact the terms T_n are the absolute values of those appearing in Table 2.3. In adding this sum we continue until adding further terms can have no impact on the sum. When the terms are smaller then $e^{20.5} \cdot 10^{-16} \approx 8.00 \cdot 10^8 \cdot 10^{-16} = 8.00 \cdot 10^{-8}$ they have no further impact so we stop at the first term smaller than this. Forming this sum and stopping when $|T_n| < 8.00 \cdot 10^{-8}$, it turns out that we need the first 68 terms of the series to give a DP accurate approximation to $e^{20.5}$. Since we are summing a series of positive terms we expect (and get) an approximation for $e^{20.5}$ with a small relative error. Next, we compute $e^{-20.5} = \dfrac{1}{e^{20.5}}$ using this approximation for $e^{20.5}$. The result has at least 15 correct digits in a DP calculation.

2. Another idea is to use a form of *range reduction*. We can use the alternating series as long as we avoid the catastrophic cancellation which leads to a large relative error. We will certainly avoid this problem if we evaluate e^{-x} only for values of $x \in (0, 1)$ since in this case the magnitudes of the terms of the series are monotonically decreasing. Observe that

$$e^{-20.5} = e^{-20}e^{-0.5} = (e^{-1})^{20}e^{-0.5}$$

So, if we can calculate e^{-1} accurately then raise it to the 20^{th} power, then evaluate $e^{-0.5}$ by Taylor series, and finally we can compute $e^{-20.5}$ by multiplying the results. (In the spirit of range reduction, we use, for example, the MATLAB exponential function to evaluate e^{-1} and then its power function to compute its 20^{th} power; this way these quantities are calculated accurately. If we don't have an exponential function available, we could use the series with $x = -1$ to calculate e^{-1}.) Since $e^{-0.5} \approx 0.6$, to get full accuracy we terminate the Taylor series when $|T_n| < 0.6 \cdot 10^{-16}$, that is after 15 terms. This approach delivers at least 15 digits of accuracy for $e^{-20.5}$.

Problem 2.3.6. *Quote and prove the alternating series theorem that shows that $|S_n - e^{-20.5}| < |T_n|$ in exact arithmetic.*

Problem 2.3.7. *Suppose that you use Taylor series to compute e^x for a value of x such that $|x| > 1$. Which is the largest term in the Taylor series? In what circumstances are there two equally sized largest terms in the Taylor series? [Hint: $\dfrac{T_n}{T_{n-1}} = \dfrac{x}{n}$.]*

Problem 2.3.8. *Suppose we are using SP arithmetic (with an accuracy of about 10^{-7}) and say we attempt to compute e^{-15} using the alternating Taylor series. What is the largest value T_n in magnitude. Using the fact that $e^{-15} \approx 3.06 \cdot 10^{-7}$ how many terms of the alternating series are needed to compute e^{-15} to full SP accuracy? What is the approximate absolute error in the sum as an approximation to e^{-15}? [Hint: There is no need to calculate the value of the terms T_n or the partial sums S_n to answer this question.]*

Problem 2.3.9. *For what value of n does the partial sum S_n form a 16–digit accurate approximation of $e^{-21.5}$? Using DP arithmetic, compare the computed value of S_n with the exact value $e^{-21.5} \approx 4.60 \cdot 10^{-10}$. Explain why the computed value of S_n is reasonable. [Hint: Because $\frac{T_n}{T_{n-1}} = \frac{x}{n}$, if the current value of* `term` *is T_{n-1}, then sequence of assignment statements*

```
term := x*term
term := term/n
```

converts `term` *into the value of T_n.]*

2.3.6 Euclidean Length of a Vector

Many computations require the **Euclidean length**

$$p = \sqrt{a^2 + b^2}$$

of the 2-vector $\begin{bmatrix} a \\ b \end{bmatrix}$. The value p can also be viewed as the **Pythagorean sum** of a and b. Now,

$$\min(|a|, |b|) \leq p \leq \sqrt{2} \cdot \max(|a|, |b|)$$

so we should be able to compute p in such a way that we never encounter floating–point underflow and we rarely encounter floating–point overflow. However, when computing p via the relation $p = \sqrt{a^2 + b^2}$ we can encounter one or both of *floating–point underflow* and *floating–point overflow* when computing the squares of a and b. To avoid floating–point overflow, we can choose a value c in such a way that we can scale a and b by this value of c and the resulting scaled quantities $\left[\frac{a}{c}\right]$ and $\left[\frac{b}{c}\right]$ may be safely squared. Using this scaling,

$$p = c \cdot \sqrt{\left[\frac{a}{c}\right]^2 + \left[\frac{b}{c}\right]^2}$$

An obvious choice for the scaling factor is $c = \max(|a|, |b|)$ so that one of $\left[\frac{a}{c}\right]$ and $\left[\frac{b}{c}\right]$ equals 1 and the other has magnitude less than 1. Another sensible choice of scaling factor is c a power of 2 just greater than $\max(|a|, |b|)$; the advantage of this choice is that with *Standard* arithmetic, division by 2 is performed exactly (ignoring underflow). If we choose the scaling factor c in one of these ways it is possible that the smaller squared term in that occurs when computing p will underflow but this is harmless; that is, the computed value of p will be essentially correct.

Another technique that avoids both unnecessary floating–point overflow and floating–point underflow is the Moler-Morrison Pythagorean sum algorithm displayed in the pseudocode in Fig. 2.3. In this algorithm, the value p converges to $\sqrt{a^2 + b^2}$ from below, and the value q converges rapidly to 0 from above. So, floating–point overflow can occur only if the exact value of p overflows. Indeed, only harmless floating–point underflow can occur; that is, the computed value of p will be essentially correct.

Problem 2.3.10. *For any values a and b show that*

$$\min(|a|, |b|) \leq p \equiv \sqrt{a^2 + b^2} \leq \sqrt{2} \cdot \max(|a|, |b|)$$

Problem 2.3.11. *Suppose $a = 1$ and $b = 10^{-60}$. If $c \approx \max(|a|, |b|)$, floating–point underflow occurs in computing $\left[\frac{b}{c}\right]^2$ in DP arithmetic. In this case, the* Standard *returns the value 0 for $\left[\frac{b}{c}\right]^2$. Why is the computed value of p still accurate?*

Moler-Morrison Pythagorean Sum Algorithm

Input: scalars a and b
Output: $p = \sqrt{a^2 + b^2}$

```
p := max(|a|, |b|)
q := min(|a|, |b|)
for i = 1 to N
    r := (q/p)^2
    s := r/(4 + r)
    p := p + 2sp
    q := sq
next i
```

Figure 2.3: The Moler-Morrison Pythagorean sum algorithm for computing $p \equiv \sqrt{a^2 + b^2}$. Typically $N = 3$ suffices for any SP or DP numbers p and q.

Problem 2.3.12. *In the Moler-Morrison Pythagorean sum algorithm, show that though each trip through the for-loop may change the values of p and q, in exact arithmetic it never changes the value of the loop invariant $p^2 + q^2$.*

Problem 2.3.13. *Design a pseudocode to compute $d = \sqrt{x^2 + y^2 + z^2}$. If floating–point underflow occurs it should be harmless, and floating–point overflow should occur only when the exact value of d overflows.*

2.3.7 Roots of a Quadratic Equation

We aim to design a reliable algorithm to determine the roots of the quadratic equation

$$ax^2 - 2bx + c = 0$$

(Note that the factor multiplying x is $-2b$ and not b as in the standard notation.) We assume that the coefficients a, b and c are real numbers, and that the coefficients are chosen so that the roots are real. The familiar quadratic formula for these roots is

$$x_\pm = \frac{b \pm \sqrt{d}}{a}$$

where $d \equiv b^2 - ac$ is the **discriminant**, assumed nonnegative here. The discriminant d is zero if there is a double root. Here are some problems that can arise.

First, the algorithm should check for the special cases $a = 0$, $b = 0$ or $c = 0$. These cases are trivial to eliminate. When $a = 0$ the quadratic equation degenerates to a linear equation whose solution $\frac{c}{2b}$ requires at most a division. When $b = 0$ the roots are $\pm\sqrt{-\frac{c}{a}}$. When $c = 0$ the roots are 0 and $\frac{2b}{a}$. In the latter two cases the roots will normally be calculated more accurately using these special formulas than by using the quadratic formula.

Second, computing d can lead to either *floating–point underflow* or *floating–point overflow*, typically when either b^2 or ac or both are computed. This problem can be eliminated using the technique described in the previous section; scale the coefficients a, b and c by a power of two chosen so that b^2 and ac can be safely computed.

Third, the computation can suffer from *catastrophic cancelation* when either

- the roots are nearly equal, i.e., $ac \approx b^2$
- the roots have significantly different magnitudes, i.e., $|ac| \ll b^2$

When $ac \approx b^2$ there is catastrophic cancelation when computing d. This may be eliminated by computing the discriminant $d = b^2 - ac$ using higher precision arithmetic, when possible. For example, if a, b and c are SP numbers, then we can use DP arithmetic to compute d. If a, b and c are DP numbers, maybe we can use QP (quadruple, extended, precision) arithmetic to compute d; MATLAB does not provide QP. The aim when using higher precision arithmetic is to discard as few digits as possible of b^2 and ac before forming $b^2 - ac$. For many arithmetic processors this is achieved without user intervention. That is, the discriminant is calculated to higher precision automatically, by computing the value of the whole expression $b^2 - ac$ in higher precision before rounding. When cancelation is inevitable, we might first use the quadratic formula to compute approximations to the roots and then using a Newton iteration (see Chapter 6) to improve these approximations.

When $|ac| \ll b^2$, $\sqrt{d} \approx |b|$ and one of the computations $b \pm \sqrt{d}$ suffers from catastrophic cancelation. To eliminate this problem note that

$$\frac{b+\sqrt{d}}{a} = \frac{c}{b-\sqrt{d}}, \quad \frac{b-\sqrt{d}}{a} = \frac{c}{b+\sqrt{d}}$$

So, when $b > 0$ use

$$x_+ = \frac{b+\sqrt{d}}{a}, \quad x_- = \frac{c}{b+\sqrt{d}}$$

and when $b < 0$ use

$$x_+ = \frac{c}{b-\sqrt{d}}, \quad x_- = \frac{b-\sqrt{d}}{a}$$

The following problems are intended as a "pencil and paper" exercises.

Problem 2.3.14. *Show that $a = 2049$, $b = 4097$ and $c = 8192$ are SP numbers. Use* MATLAB *SP arithmetic to compute the roots of $ax^2 - 2bx + c = 0$ using the quadratic formula.*

Problem 2.3.15. *Show that $a = 1$, $b = 4096$ and $c = 1$ are SP numbers. Use* MATLAB *SP arithmetic to compute the roots of $ax^2 - 2bx + c = 0$ using the quadratic formula.*

2.4 Matlab Notes

MATLAB provides a variety of data types, including integers and floating–point numbers. By default, all floating–point variables and constants, and the associated computations, are held in double precision. Indeed, they are all complex numbers but this is normally not visible to the user when using real data. However, there are cases where real data can lead to complex number results, for example the roots of a polynomial with real coefficients can be complex and this is handled seamlessly by MATLAB. The most recent versions of MATLAB (version 7.0 and higher) allow for the creation of single precision variables and constants. MATLAB also defines certain parameters (such as machine epsilon) associated with floating–point computations.

In addition to discussing these issues, we provide implementation details and several exercises associated with the examples discussed in Section 2.3.

2.4.1 Integers in Matlab

MATLAB supports both signed and unsigned integers. For each type, integers of 8, 16, 32 and 64 bits are supported. MATLAB provides functions to convert a numeric object (e.g. an integer of another type or length, or a double precision number) into an integer of the specified type and

length; e.g,. `int8` converts numeric objects into 8–bit signed integers and `uint16` converts numeric objects into 16–bit unsigned integers. MATLAB performs arithmetic between integers of the same type and between integers of any type and double precision numbers

2.4.2 Single Precision Computations in Matlab

By default, MATLAB assumes all floating–point variables and constants, and the associated computations, are double precision. To compute using single precision arithmetic, variables and constants must first be converted using the `single` function. Computations involving a mix of SP and DP variables generally produce SP results. For example,

```
theta1 = 5*single(pi)/6
s1 = sin(theta1)
```

produces the SP values `theta1`= 2.6179941 and `s1`= 0.4999998. Because we specify `single(pi)`, the constants 5 and 6 in `theta1` are assumed SP, and the computations use SP arithmetic.

As a comparison, if we do not specify `single` for any of the variables or constants,

```
theta2 = 5*pi/6
s2 = sin(theta2)
```

then MATLAB produces the DP values `theta2`= 2.61799387799149 and `s2`= 0.50000000000000.

However, if the computations are written

```
theta3 = single(5*pi/6)
s3 = sin(theta3)
```

then MATLAB produces the values `theta3`= 2.6179938, and `s3`= 0.5000001. The computation `5*pi/6` uses default DP arithmetic, then the result is converted to SP.

2.4.3 Special Constants

A nice feature of MATLAB is that many parameters discussed in this chapter can be generated easily. For example, `eps` is a function that can be used to compute ϵ_{DP} and $\text{ulp}_{\text{DP}}(y)$. In particular, `eps(1)`, `eps('double')` and `eps` all produce the same result, namely 2^{-52}. However, if `y` is a DP number, then `eps(y)` computes $\text{ulp}_{\text{DP}}(y)$, which is simply the distance from y to the next largest (in magnitude) DP number. The largest and smallest positive DP numbers can be computed using the functions `realmax` and `realmin`, respectively.

As with DP numbers, the functions `eps`, `realmax` and `realmin` can be used with SP numbers. For example, `eps('single')` and `eps(single(1))` produce the result 2^{-23}. Similarly, `realmax('single')` and `realmin('single')` return, respectively, the largest and smallest SP floating–point numbers.

MATLAB also defines parameters for ∞ (called `inf` or `Inf`) and NaN (called `nan` or `NaN`). It is possible to get, and to use, these quantities in computations. For example:

- The computation `1/0` produces `Inf`.
- The computation `1/Inf` produces 0.
- Computations of indeterminate quantities, such as `0*Inf`, `0/0` and `Inf/Inf` produce `NaN`.

Problem 2.4.1. *In* MATLAB*, consider the anonymous functions:*

```
f = @(x) x ./ (x.*(x-1));
g = @(x) 1 ./ (x-1);
```

What is computed by `f(0)`*,* `f(eps)`*,* `g(0)`*,* `g(eps)`*,* `f(Inf)`*, and* `g(Inf)`*? Explain these results.*

2.4.4 Floating-Point Numbers in Output

When displaying output, MATLAB rounds floating–point numbers to fit the number of digits to be displayed. For example, consider a MATLAB code that sets a DP variable x to the value 1.99 and prints it twice, first using a floating–point format with 1 place after the decimal point and second using a floating–point format with 2 places after the decimal point. MATLAB code for this is:

```
x = 1.99;
fprintf('Printing one decimal point produces %3.1f \n', x)
fprintf('Printing two decimal points produces %4.2f \n', x)
```

The first value printed is 2.0 while the second value printed is 1.99. Of course, 2.0 is not the true value of x. The number 2.0 appears because 2.0 represents the true value rounded to the number of digits displayed. If a printout lists the value of a variable x as precisely 2.0, that is it prints just these digits, then its actual value may be any number in the range $1.95 \leq x < 2.05$.

A similar situation occurs in the MATLAB command window. When MATLAB is started, numbers are displayed on the screen using the default "format" (called `short`) of 5 digits. For example, if we set a DP variable $x = 1.99999$, MATLAB displays the number as

```
2.0000
```

More correct digits can be displayed by changing the format. So, if we execute the MATLAB statement

```
format long
```

then 15 digits are displayed; that is, the number x is displayed as

```
1.99999000000000
```

Other formats can be set; see `doc format` for more information.

Problem 2.4.2. *Using the default* `format short`*, what is displayed in the command window when the following variables are displayed?*

```
x = exp(1)
y = single(exp(1))
z = x - y
```

Do the results make sense? What is displayed when using `format long`*?*

Problem 2.4.3. *Read* MATLAB*'s help documentation on* `fprintf`*. In the example* MATLAB *code given above, why did the first* `fprintf` *command contain* `\n`*? What happens if this is omitted?*

Problem 2.4.4. *Determine the difference between the following* `fprintf` *statements:*

```
fprintf('%6.4f \n',pi)
fprintf('%8.4f \n',pi)
fprintf('%10.4f \n',pi)
fprintf('%10.4e \n',pi)
```

In particular, what is the significance of the numbers `6.4`*,* `8.4`*, and* `10.4`*, and the letters* `f` *and* `e`*?*

2.4.5 Examples

Section 2.3 provided several examples that illustrate difficulties that can arise in scientific computations. Here we provide MATLAB implementation details and several associated exercises.

Plotting a Polynomial

Consider the example from Section 2.3.1, where a plot of

$$\begin{aligned} p(x) &= (1-x)^{10} \\ &= x^{10} - 10\,x^9 + 45\,x^8 - 120\,x^7 + 210\,x^6 - 252\,x^5 + 210\,x^4 - 120\,x^3 + 45\,x^2 - 10\,x + 1 \end{aligned}$$

on the interval $[0.99, 1.01]$ is produced using the power series form of $p(x)$. In MATLAB we use `linspace`, `plot`, and a very useful function called `polyval`, which is used to evaluate polynomials. Specifically, the following MATLAB code is used to produce the plot shown in Fig. 2.1:

```
x = linspace(0.99, 1.01, 101);
c = [1, -10, 45, -120, 210, -252, 210, -120, 45, -10, 1];
p = polyval(c, x);
plot(x, p)
xlabel('x'), ylabel('p(x)')
```

Of course, since we know the factored form of $p(x)$, we can use it to produce an accurate plot:

```
x = linspace(0.99, 1.01, 101);
p = (1 - x).^(10);
plot(x, p)
```

Not all polynomials can be factored easily, and it may be necessary to use the power series form.

Problem 2.4.5. *Use the code given above to sketch $y = p(x)$ for values $x \in [0.99, 1.01]$ using the power form of $p(x)$. Pay particular attention to the scaling of the y-axis. What is the largest value of y that you observe? Now modify the code to plot on the same graph the factored form of $y = p(x)$ and to put axes on the graph. Can you distinguish the plot of the factored form of $y = p(x)$? Explain what you observe.*

Problem 2.4.6. *Construct a figure analogous to Fig. 2.1, but using SP arithmetic rather than DP arithmetic to evaluate $p(x)$. What is the largest value of y that you observe in this case?*

Repeated Square Roots

The following MATLAB code performs the repeated square roots experiment outlined in Section 2.3.2 on a vector of 10 equally spaced values of x using 100 iterations (use a script M–file):

```
x = 1:0.25:3.75;
n = 100
t = x;
for i = 1:n
  t = sqrt(t);
end
for i = 1:n
  t = t .* t;
end
disp('   x         t        x-t ')
disp('==============================')
for k = 1:10
  disp(sprintf('%7.4f   %7.4f   %7.4f', x(k), t(k), x(k)-t(k)))
end
```

When you run this code, the `disp` and `sprintf` commands are used to display, in the command window, the results shown in Table 2.1. The command `sprintf` works just like `fprintf` but prints the result to a MATLAB string (displayed using `disp`) rather than to a file or the screen.

Problem 2.4.7. *Using the* MATLAB *script M–file above, what is the smallest number of iterations n for which you can reproduce the whole of Table 2.1 exactly?*

Problem 2.4.8. *Modify the* MATLAB *script M–file above to use SP arithmetic. What is the smallest number of iterations n for which you can reproduce the whole of Table 2.1 exactly?*

Estimating the Derivative

The following problems use MATLAB for experiments related to the methods of Section 2.3.3

Problem 2.4.9. *Use* MATLAB *(with its default DP arithmetic),* $f(x) \equiv \sin(x)$ *and* $x = 1$ *radian. Create a three column table with column headers "n", "*$-\log_{10}(2r)$*", and "*$-\log_{10}(2R)$*". Fill the column headed by "n" with the values* $1, 2, \cdots, 51$*. The remaining entries in each row should be filled with values computed using* $h = 2^{-n}$*. For what value of n does the forward difference estimate* $\Delta_{DP} f(x)$ *of the derivative* $f'(x)$ *achieve its maximum accuracy, and what is this maximum accuracy?*

Problem 2.4.10. *In* MATLAB*, open the help browser, and search for* `single precision mathematics`*. This search can be used to find an example of writing M–files for different data types. Use this example to modify the code from Problem 2.4.9 so that it can be used for either SP or DP arithmetic.*

A Recurrence Relation

The following problems use MATLAB for experiments related to the methods of Section 2.3.4 As mentioned at the end of Section 2.3.4, we emphasize that we do not recommend the use of the recurrence for evaluating the integrals $\hat{V}_j$. These integrals may be almost trivially evaluated using the MATLAB functions `quad` and `quadl`; see Chapter 5 for more details.

Problem 2.4.11. *Reproduce the Table 2.2 using* MATLAB *single precision arithmetic.*

Problem 2.4.12. *Repeat the above analysis, but use* MATLAB *with its default DP arithmetic, to compute the sequence* $\{\hat{V}_j\}_{j=0}^{23}$*. Generate a table analogous to Table 2.2 that displays the values of* j*,* $\hat{V}_j$*, and* $\dfrac{\hat{V}_j}{j!}$*. Hint: Assume that* ϵ*, the error in the initial value* $\hat{V}_0$*, has a magnitude equal to half a ulp in the DP value of* V_0*. Using this estimate, show that* $j!\epsilon > 1$ *when* $j \geq 19$*.*

Problem 2.4.13. *Use the* MATLAB *DP integration functions* `quad` *and* `quadl` *to compute the correct values for* V_j *to the number of digits shown in Table 2.2*

Summing the Exponential Series

The following problems use MATLAB for experiments related to the methods of Section 2.3.5

Problem 2.4.14. *Implement the scheme described in Section 2.3.5, Note 1, in* MATLAB *DP arithmetic to compute* $e^{-20.5}$*.*

Problem 2.4.15. *Implement the scheme described in Section 2.3.5, Note 2, in* MATLAB *DP arithmetic to compute* $e^{-20.5}$*.*

Euclidean Length of a Vector

The following problem uses MATLAB for experiments related to the methods of Section 2.3.6.

Problem 2.4.16. *Write a* MATLAB *DP program that uses the Moler-Morrison algorithm to compute the length of the vector* $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. *Your code should display the values of* p *and* q *initially as well as at the end of each trip through the* ***for*** *loop. It should also display the ratio of the new value of* q *to the cube of its previous value. Compare the value of these ratios to the value of* $\dfrac{1}{4(a^2+b^2)}$.

Problem 2.4.17. *Use DP arithmetic to compute the lengths of the vectors* $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 10^{60} \\ 10^{61} \end{bmatrix}$ *and* $\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 10^{-60} \\ 10^{-61} \end{bmatrix}$ *using both the unscaled and scaled formulas for* p. *Use the factor* $c = \max(|a|, |b|)$ *in the scaled computation.*

Roots of a Quadratic Equation

The following problem uses MATLAB for experiments related to the methods of Section 2.3.7.

Note that MATLAB does not provide software specifically for computing the roots of a quadratic equation. However, it provides a function `roots` for calculating the roots of general polynomials including quadratics; see Section 6.5 for details.

Problem 2.4.18. *Write a* MATLAB *function to compute the roots of the quadratic equation* $ax^2 - 2bx + c = 0$ *where the coefficients* a, b *and* c *are SP numbers. As output produce SP values of the roots. Use DP arithmetic to compute* $d = b^2 - ac$. *Test your program on the cases posed in Problems 2.3.14 and 2.3.15*

Chapter 3

Solution of Linear Systems

Linear systems of equations are ubiquitous in scientific computing – they arise when solving problems in many applications, including biology, chemistry, physics, engineering and economics, and they appear in nearly every chapter of this book. A fundamental numerical problem involving linear systems is that of finding a solution (if one exists) to a set of n linear equations in n unknowns. The first digital computers (developed in the 1940's primarily for scientific computing problems) required about an hour to solve linear systems involving only 10 equations in 10 unknowns. Modern computers are substantially more powerful, and we can now solve linear systems involving hundreds of equations in hundreds of unknowns in a fraction of a second. Indeed, many problems in science and industry involve millions of equations in millions of unknowns. In this chapter we study the most commonly used algorithm, *Gaussian elimination with partial pivoting*, to solve these important problems.

After a brief introduction to linear systems, we discuss computational techniques for problems (diagonal, lower and upper triangular) that are simple to solve. We then describe Gaussian elimination with partial pivoting, which is an algorithm that reduces a general linear system to one that is simple to solve. Important matrix factorizations associated with Gaussian elimination are described, and issues regarding accuracy of computed solutions are discussed. The chapter ends with a section describing MATLAB implementations, as well as the main tools provided by MATLAB for solving linear systems.

3.1 Linear Systems

A linear system of order n consists of the n linear algebraic equations

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

in the n unknowns $x_1, x_2, \ldots, x_n$. A solution of the linear system is a set of values $x_1, x_2, \ldots, x_n$ that satisfy all n equations simultaneously. Problems involving linear systems are typically formulated using the linear algebra language of matrices and vectors. To do this, first group together the quantities on each side of the equals sign as vectors,

$$\begin{bmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n \\ \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

and recall from Section 1.2 that the vector on the left side of the equal sign can be written as a linear combination of column vectors, or equivalently as a matrix–vector product:

$$\begin{bmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{n,1} \end{bmatrix} x_1 + \begin{bmatrix} a_{1,2} \\ a_{2,2} \\ \vdots \\ a_{n,2} \end{bmatrix} x_2 + \cdots + \begin{bmatrix} a_{1,n} \\ a_{2,n} \\ \vdots \\ a_{n,n} \end{bmatrix} x_n = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Thus, in matrix–vector notation, the linear system is represented as

$$Ax = b$$

where

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \quad \text{is the coefficient matrix of order } n,$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{is the unknown, or solution vector of length } n, \text{ and}$$

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad \text{is the right hand side vector of length } n.$$

We consider problems where the coefficients $a_{i,j}$ and the right hand side values b_i are real numbers. A **solution** of the linear system $Ax = b$ of order n is a vector x that satisfies the equation $Ax = b$. The **solution set** of a linear system is the set of all its solutions.

Example 3.1.1. The linear system of equations

$$\begin{aligned} 1x_1 + \ 1x_2 \ + \ 1x_3 \ &= 3 \\ 1x_1 + (-1)x_2 + \ 4x_3 \ &= 4 \\ 2x_1 + \ 3x_2 \ + (-5)x_3 &= 0 \end{aligned}$$

is of order $n = 3$ with unknowns x_1, x_2 and x_3. The matrix–vector form is $Ax = b$ where

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 4 \\ 2 & 3 & -5 \end{bmatrix}, \ x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \ b = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix}$$

This linear system has precisely one solution, given by $x_1 = 1$, $x_2 = 1$ and $x_3 = 1$, so

$$x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Linear systems arising in most realistic applications are usually much larger than the order $n = 3$ system of the previous example. However, a lot can be learned about linear systems by looking at small problems. Consider, for example, the 2×2 linear system:

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \Leftrightarrow \quad \begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 &= b_2. \end{aligned}$$

If $a_{1,2} \neq 0$ and $a_{2,2} \neq 0$, then we can write these equations as

$$x_2 = -\frac{a_{1,1}}{a_{1,2}}x_1 + b_1 \quad \text{and} \quad x_2 = -\frac{a_{2,1}}{a_{2,2}}x_1 + b_2,$$

which are essentially the slope-intercept form equations of two lines in a plane. Solutions of this linear system consist of all values x_1 and x_2 that satisfy both equations; that is, all points (x_1, x_2) where the two lines intersect. There are three possibilities for this simple example (see Fig. 3.1):

- Unique solution – the lines intersect at only one point.
- No solution – the lines are parallel, with different intercepts.
- Infinitely many solutions – the lines are parallel with the same intercept.

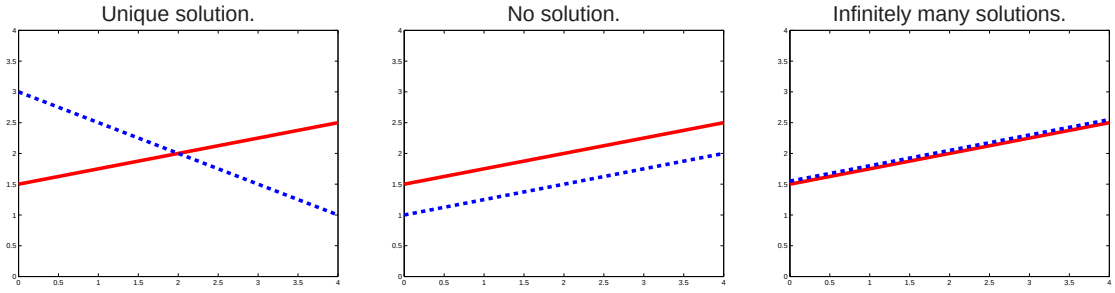

Figure 3.1: Possible solution sets for a general 2×2 linear system. The left plot shows two lines that intersect at only one point (unique solution), the middle plot shows two parallel lines that do not intersect at any points (no solution), and the right plot shows two identical lines (infinitely many solutions).

This conclusion holds for linear systems of any order; that is, **a linear system of order n has either no solution, 1 solution, or an infinite number of distinct solutions** . A linear system $Ax = b$ of order n is **nonsingular** if it has one and only one solution. A linear system is **singular** if it has either no solution or an infinite number of distinct solutions; which of these two possibilities applies depends on the relationship between the matrix A and the right hand side vector b, a matter that is considered in a first linear algebra course.

Whether a linear system $Ax = b$ is singular or nonsingular depends solely on properties of its coefficient matrix A. In particular, the linear system $Ax = b$ is nonsingular if and only if the matrix A is *invertible*; that is, if and only if there is a matrix, A^{-1}, such that $AA^{-1} = A^{-1}A = I$, where I is the identity matrix,

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix}.$$

So, we say A is nonsingular (invertible) if the linear system $Ax = b$ is nonsingular, and A is singular (non-invertible) if the linear system $Ax = b$ is singular. It is not always easy to determine, a-priori, whether or not a matrix is singular especially in the presence of roundoff errors. This point will be addressed in Section 3.5.

Example 3.1.2. Consider the linear systems of order 2:

(a) $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \quad \Rightarrow \quad \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{4}x_1 + \frac{6}{4} \end{matrix} \quad \Rightarrow \quad \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{4}x_1 + \frac{3}{2} \end{matrix}$

This linear system consists of two lines with unequal slopes. Thus the lines intersect at only one point, and the linear system has a unique solution.

(b) $\begin{bmatrix} 1 & 2 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix} \Rightarrow \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{6}x_1 + \frac{6}{6} \end{matrix} \Rightarrow \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{1}{2}x_1 + 1 \end{matrix}$

This linear system consists of two lines with equal slopes. Thus the lines are parallel. Since the intercepts are not identical, the lines do not intersect at any points, and the linear system has no solution.

(c) $\begin{bmatrix} 1 & 2 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \end{bmatrix} \Rightarrow \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{3}{6}x_1 + \frac{15}{6} \end{matrix} \Rightarrow \begin{matrix} x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \\ x_2 = -\frac{1}{2}x_1 + \frac{5}{2} \end{matrix}$

This linear system consists of two lines with equal slopes. Thus the lines are parallel. Since the intercepts are also equal, the lines are identical, and the linear system has infinitely many solutions.

Problem 3.1.1. *Consider the linear system of Example 3.1.1. If the coefficient matrix A remains unchanged, then what choice of right hand side vector b would lead to the solution vector $x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$?*

Problem 3.1.2. *Consider any linear system of equations of order 3. The solution of each equation can be portrayed as a plane in a 3-dimensional space. Describe geometrically how 3 planes can intersect in a 3-dimensional space. Why must two planes that intersect at two distinct points intersect at an infinite number of points? Explain how you would conclude that if a linear system of order 3 has 2 distinct solutions it must have an infinite number of distinct solutions.*

Problem 3.1.3. *Give an example of one equation in one unknown that has no solution. Give another example of one equation in one unknown that has precisely one solution, and another example of one equation in one unknown that has an infinite number of solutions.*

Problem 3.1.4. *The determinant of an $n \times n$ matrix, $\det(A)$, is a number that theoretically can be used computationally to indicate if a matrix is singular. Specifically, if $\det(A) \neq 0$ then A is nonsingular. The formula to compute $\det(A)$ for a general $n \times n$ matrix is complicated, but there are some special cases where it can be computed fairly easily. In the case of a 2×2 matrix,*

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc.$$

Compute the determinants of the 2×2 matrices in Example 3.1.2. Why do these results make sense? Important note: The determinant is a good theoretical test for singularity, but it is not a practical test in computational problems. This is discussed in more detail in Section 3.5.

3.2 Simply Solved Linear Systems

Some linear systems are easy to solve. Consider the linear systems of order 3 displayed in Fig. 3.2. By design, the solution of each of these linear systems is $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$. The structure of these linear systems makes them easy to solve; to explain this, we first name the structures exhibited.

The entries of a matrix $A = [a_{i,j}]_{i,j=1}^{n}$ are partitioned into three classes:

(a) the diagonal entries, i.e., the entries $a_{i,j}$ for which $i = j$,

(b) the strictly lower triangular entries, i.e., the entries $a_{i,j}$ for which $i > j$, and

(c) the strictly upper triangular entries, i.e., the entries $a_{i,j}$ for which $i < j$.

The locations of the diagonal, strictly lower triangular, and strictly upper triangular entries of A are illustrated in Fig. 3.3. The lower triangular entries are composed of the strictly lower triangular

$$
(a)\quad \begin{array}{rcl} (-1)x_1 + 0x_2 + 0x_3 &=& -1 \\ 0x_1 + 3x_2 + 0x_3 &=& 6 \\ 0x_1 + 0x_2 + (-5)x_3 &=& -15 \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 6 \\ -15 \end{bmatrix}
$$

$$
(b)\quad \begin{array}{rcl} (-1)x_1 + 0x_2 + 0x_3 &=& -1 \\ 2x_1 + 3x_2 + 0x_3 &=& 8 \\ (-1)x_1 + 4x_2 + (-5)x_3 &=& -8 \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} -1 & 0 & 0 \\ 2 & 3 & 0 \\ -1 & 4 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 8 \\ -8 \end{bmatrix}
$$

$$
(c)\quad \begin{array}{rcl} (-1)x_1 + 2x_2 + (-1)x_3 &=& 0 \\ 0x_1 + 3x_2 + 6x_3 &=& 24 \\ 0x_1 + 0x_2 + (-5)x_3 &=& -15 \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} -1 & 2 & -1 \\ 0 & 3 & 6 \\ 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 24 \\ -15 \end{bmatrix}
$$

Figure 3.2: Simply Solved Linear Systems

and diagonal entries, as illustrated in Fig. 3.4. Similarly, the upper triangular entries are composed of the strictly upper triangular and diagonal entries. Using this terminology, we say that the linear system in Fig. 3.2(a) is diagonal, the linear system in Fig. 3.2(b) is lower triangular, and the linear system in Fig. 3.2(c) is upper triangular.

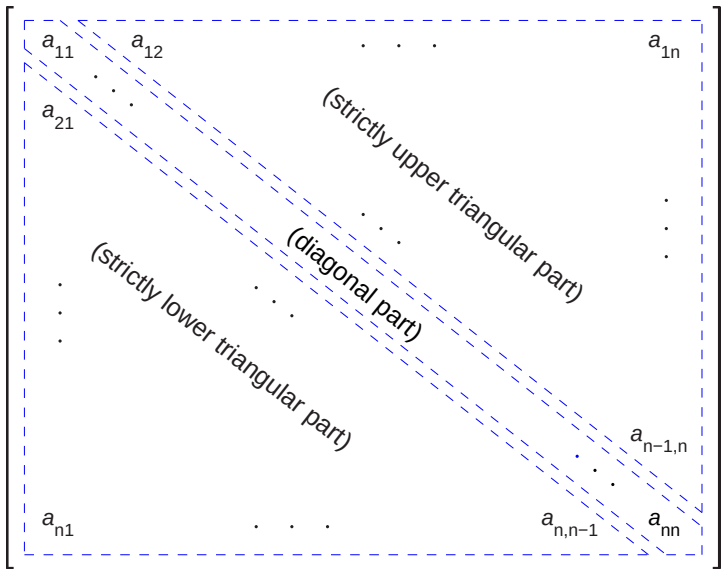

Figure 3.3: Illustration of strict triangular and diagonal matrix entries.

Problem 3.2.1. *Let A be a matrix of order n. Show that A has n^2 entries, that $n^2 - n = n(n-1)$ entries lie off the diagonal, and that each strictly triangular portion of A has $\dfrac{n(n-1)}{2}$ entries.*

3.2.1 Diagonal Linear Systems

A matrix A of order n is **diagonal** if all its nonzero entries are on its diagonal. (This description of a diagonal matrix *does not state* that the entries on the diagonal are nonzero.) A **diagonal linear**

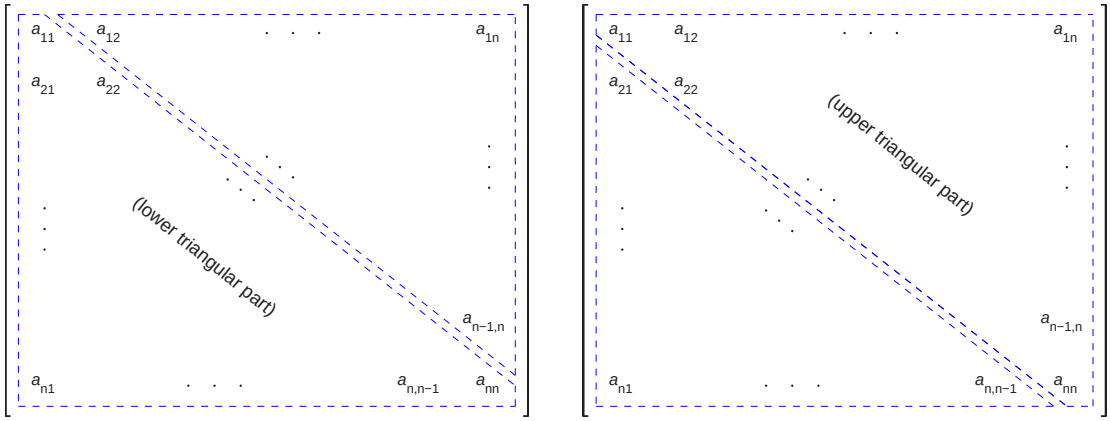

Figure 3.4: Illustration of the lower triangular and upper triangular parts of a matrix.

system of equations of order n is one whose coefficient matrix is diagonal. Solving a diagonal linear system of order n, like that in Fig. 3.2(a), is easy because each equation determines the value of one unknown, provided that the diagonal entry is nonzero. So, the first equation determines the value of x_1, the second x_2, etc. A linear system with a diagonal coefficient matrix is singular if it contains a diagonal entry that is zero. In this case the linear system may have no solutions or it may have infinitely many solutions.

Example 3.2.1. In Fig. 3.2(a) the solution is $x_1 = \frac{-1}{(-1)} = 1$, $x_2 = \frac{6}{3} = 2$ and $x_3 = \frac{-15}{(-5)} = 3$.

Example 3.2.2. Consider the following singular diagonal matrix, A, and the vectors b and d:

$$A = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}, \quad d = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}.$$

(a) The linear system $Ax = b$ has no solution. Although we can solve the first and last equations to get $x_1 = \frac{1}{3}$ and $x_3 = 0$, it is not possible to solve the second equation:

$$0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 = -1 \quad \Rightarrow \quad 0 \cdot \frac{1}{3} + 0 \cdot x_2 + 0 \cdot 0 = -1 \quad \Rightarrow \quad 0 \cdot x_2 = -1.$$

Clearly there is no value of x_2 that satisfies the equation $0 \cdot x_2 = -1$.

(b) The linear system $Ax = d$ has infinitely many solutions. From the first and last equations we obtain $x_1 = \frac{1}{3}$ and $x_3 = \frac{1}{2}$. The second equation is

$$0 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 = 0 \quad \Rightarrow \quad 0 \cdot \frac{1}{3} + 0 \cdot x_2 + 0 \cdot \frac{1}{2} = 0 \quad \Rightarrow \quad 0 \cdot x_2 = 0,$$

and thus x_2 can be any real number.

Problem 3.2.2. *Consider the matrix*

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

Why is this matrix singular? Find a vector b so that the linear system $Ax = b$ has no solution. Find a vector b so that the linear system $Ax = b$ has infinitely many solutions.

3.2.2 Column and Row Oriented Algorithms

Normally, an algorithm that operates on a matrix must choose between a row–oriented or a column–oriented version depending on how the programming language stores matrices in memory. Typically, a computer's memory unit is designed so that the CPU can quickly access consecutive memory locations. So, consecutive entries of a matrix usually can be accessed quickly if they are stored in consecutive memory locations.

MATLAB stores matrices in column–major order; numbers in the same column of the matrix are stored in consecutive memory locations, so column–oriented algorithms generally run faster.

Other scientific programming languages behave similarly. For example, FORTRAN 77 stores matrices in column–major order, and furthermore, consecutive columns of the matrix are stored contiguously; that is, the entries in the first column are succeeded immediately by the entries in the second column, etc. Generally, Fortran 90 follows the FORTRAN 77 storage strategy where feasible, and column–oriented algorithms generally run faster. In contrast, C and C++ store matrices in row–major order; numbers in the same row are stored in consecutive memory locations, so row–oriented algorithms generally run faster. However, there is no guarantee that consecutive rows of the matrix are stored contiguously, nor even that the memory locations containing the entries of one row are placed before the memory locations containing the entries in later rows.

In the sections that follow, we present both row– and column–oriented algorithms for solving linear systems.

3.2.3 Forward Substitution for Lower Triangular Linear Systems

A matrix A of order n is **lower triangular** if all its nonzero entries are either strictly lower triangular entries or diagonal entries. A **lower triangular linear system** of order n is one whose coefficient matrix is lower triangular. Solving a lower triangular linear system, like that in Fig. 3.2(b), is usually carried out by **forward substitution**. Forward substitution determines first x_1, then x_2, and so on, until all x_i are found. For example, in Fig. 3.2(b), the first equation determines x_1. Given x_1, the second equation then determines x_2. Finally, given both x_1 and x_2, the third equation determines x_3. This process is illustrated in the following example.

Example 3.2.3. In Fig. 3.2(b) the solution is $x_1 = \frac{-1}{(-1)} = 1$, $x_2 = \frac{8-2x_1}{3} = \frac{8-2\cdot 1}{3} = 2$ and $x_3 = \frac{-8-(-1)x_1-4x_2}{(-5)} = \frac{-8-(-1)\cdot 1-4\cdot 2}{(-5)} = 3$.

To write a computer code to implement forward substitution, we must formulate the process in a systematic way. To motivate the two most "popular" approaches to implementing forward substitution, consider the following lower triangular linear system.

$$\begin{array}{lllllll} a_{1,1}x_1 & & & & & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & & & = & b_2 \\ a_{3,1}x_1 & + & a_{3,2}x_2 & + & a_{3,3}x_3 & = & b_3 \end{array}$$

By transferring the terms involving the off-diagonal entries of A to the right hand side we obtain

$$\begin{array}{lllllll} a_{1,1}x_1 & & & = & b_1 & & \\ & a_{2,2}x_2 & & = & b_2 & - & a_{2,1}x_1 \\ & & a_{3,3}x_3 & = & b_3 & - & a_{3,1}x_1 \quad - \quad a_{3,2}x_2 \end{array}$$

The right hand sides can be divided naturally into rows and columns.

- Row–oriented forward substitution updates (modifies) the right hand side one row at a time. That is, after computing $x_1, x_2, \ldots, x_{i-1}$, we update b_i as:

$$b_i := b_i - (a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,i-1}x_{i-1}) = b_i - \sum_{j=1}^{i-1} a_{i,j}x_j$$

The symbol ":=" means assign the value computed on the right hand side to the variable on the left hand side. Here, and later, when the lower limit of the sum exceeds the upper limit the sum is considered to be "empty". In this process the variable b_i is "overwritten" with the value $b_i - \sum_{j=1}^{i-1} a_{i,j}x_j$. With this procedure to update the right hand side, an algorithm for row-oriented forward substitution could have the form:

$$\begin{aligned}&\text{for each } i = 1, 2, \ldots, n\\ &\qquad \text{update } b_i := b_i - \textstyle\sum_{j=1}^{i-1} a_{i,j}x_j\\ &\qquad \text{compute } x_i := b_i/a_{i,i}\end{aligned}$$

Notice that each update step uses elements in the ith *row* of A, $a_{i,1}, a_{i,2}, \ldots, a_{i,i-1}$.

- Column–oriented forward substitution updates (modifies) the right hand side one column at a time. That is, after computing x_j, we update $b_{j+1}, b_{j+2}, \ldots, b_n$ as:

$$\begin{aligned}b_{j+1} &:= b_{j+1} - a_{j+1,j}x_j\\ b_{j+2} &:= b_{j+2} - a_{j+2,j}x_j\\ &\vdots\\ b_n &:= b_n - a_{n,j}x_j\end{aligned}$$

With this procedure, an algorithm for column-oriented forward substitution could have the form:

$$\begin{aligned}&\text{for each } j = 1, 2, \ldots, n\\ &\qquad \text{compute } x_j := b_j/a_{j,j}\\ &\qquad \text{update } b_i := b_i - a_{i,j}x_j, \quad i = j+1, j+2, \ldots, n\end{aligned}$$

Notice in this case the update steps use elements in the jth *column* of A, $a_{j+1,j}, a_{j+2,j}, \ldots, a_{n,j}$.

Pseudocodes implementing row– and column–oriented forward substitution are presented in Fig. 3.5. Note that:

- We again use the symbol ":=" to assign values to variables.
- The "for" loops step in ones. So, "for $i = 1$ to n means execute the loop for each value $i = 1$, $i = 2$, until $i = n$, in turn. (Later, in Fig. 3.6 we use "downto" when we want a loop to count backwards in ones.)
- When a loop counting forward has the form, for example, "for $j = 1$ to $i - 1$" and for a given value of i we have $i - 1 < 1$ then the loop is considered *empty* and does not execute. A similar convention applies for empty loops counting backwards.
- The algorithms destroy the original entries of b. If these entries are needed for later calculations, they must be saved elsewhere. In some implementations, the entries of x are written over the corresponding entries of b.

The forward substitution process can break down if a diagonal entry of the lower triangular matrix is zero. In this case, the lower triangular matrix is singular, and a linear system involving such a matrix may have no solution or infinitely many solutions.

Problem 3.2.3. *Why is a diagonal linear system also lower triangular?*

Problem 3.2.4. *Illustrate the operation of column–oriented forward substitution when used to solve the lower triangular linear system in Fig. 3.2(b). [Hint: Show the value of b each time it has been modified by the for-loop and the value of each entry of x as it is computed.]*

Row–Oriented	*Column–Oriented*
Input: matrix $A = [a_{i,j}]$ vector $b = [b_i]$ Output: solution vector $x = [x_i]$	Input: matrix $A = [a_{i,j}]$ vector $b = [b_j]$ Output: solution vector $x = [x_j]$

for $i = 1$ to n
  for $j = 1$ to $i - 1$
  $b_i := b_i - a_{i,j}x_j$
  next j
  $x_i := b_i/a_{i,i}$
next i
	for $j = 1$ to n
	  $x_j := b_j/a_{j,j}$
	  for $i = j + 1$ to n
	  $b_i := b_i - a_{i,j}x_j$
	  next i
	next j

Figure 3.5: Pseudocode for *Row– and Column–Oriented Forward Substitution. Note that the algorithm assumes that the matrix A is lower triangular.*

Problem 3.2.5. *Use row–oriented forward substitution to solve the linear system:*

$$\begin{array}{rcl} 3x_1 + 0x_2 + 0x_3 + 0x_4 = & 6 \\ 2x_1 + (-3)x_2 + 0x_3 + 0x_4 = & 7 \\ 1x_1 + 0x_2 + 5x_3 + 0x_4 = & -8 \\ 0x_1 + 2x_2 + 4x_3 + (-3)x_4 = & -3 \end{array}$$

Problem 3.2.6. *Repeat Problem 3.2.5 but using the column–oriented version.*

Problem 3.2.7. *Modify the row– and the column–oriented pseudocodes in Fig. 3.5 so that the solution x is written over the right hand side b.*

Problem 3.2.8. *Consider a general lower triangular linear system of order n. Show that row–oriented forward substitution costs* $\frac{n(n-1)}{2}$ *multiplications,* $\frac{n(n-1)}{2}$ *subtractions, and n divisions.* *[Hint:* $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$*.]*

Problem 3.2.9. *Repeat Problem 3.2.8 for the column–oriented version of forward substitution.*

Problem 3.2.10. *Develop pseudocodes, analogous to those in Fig. 3.5, for row–oriented and column–oriented methods of solving the following linear system of order* 3

$$\begin{array}{rcrcrcl} a_{1,1}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & & & = & b_2 \\ a_{3,1}x_1 & & & & & = & b_3 \end{array}$$

Problem 3.2.11. *Consider the matrix and vector*

$$A = \begin{bmatrix} 3 & 0 & 0 \\ 1 & -2 & 0 \\ -1 & 1 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -1 \\ c \end{bmatrix}.$$

Why is A singular? For what values c does the linear system Ax = b have no solution? For what values c does the linear system Ax = b have infinitely many solutions?

3.2.4 Backward Substitution for Upper Triangular Linear Systems

A matrix A of order n is **upper triangular** if its nonzero entries are either strictly upper triangular entries or diagonal entries. An **upper triangular linear system** or order n is one whose coefficient matrix is upper triangular. Solving an upper triangular linear system, like that in Fig. 3.2(c), is usually carried out by **backward substitution**. Backward substitution determines first x_n, then x_{n-1}, and so on, until all x_i are found. For example, in Fig. 3.2(c), the third equation determines the value of x_3. Given the value of x_3, the second equation then determines x_2. Finally, given x_2 and x_3, the first equation determines x_1. This process is illustrated in the following example.

Example 3.2.4. In the case in Fig. 3.2(c) the solution is $x_3 = \frac{-15}{(-5)} = 3$, $x_2 = \frac{24-6x_3}{3} = \frac{24-6\cdot 3}{3} = 2$ and $x_1 = \frac{0-(-1)x_3-2x_2}{(-1)} = \frac{0-(-1)\cdot 3-2\cdot 2}{(-1)} = 1$.

As with forward substitution, there are two popular implementations of backward substitution. To motivate these implementations, consider the following upper triangular linear system.

$$\begin{array}{rcrcrcl} a_{1,1}x_1 & + & a_{1,2}x_2 & + & a_{1,3}x_3 & = & b_1 \\ & & a_{2,2}x_2 & + & a_{2,3}x_3 & = & b_2 \\ & & & & a_{3,3}x_3 & = & b_3 \end{array}$$

By transferring the terms involving the off-diagonal entries of A to the right hand side we obtain

$$\begin{array}{rrcrcrcl} a_{1,1}x_1 & & & = & b_1 & - & a_{1,3}x_3 & - & a_{1,2}x_2 \\ & a_{2,2}x_2 & & = & b_2 & - & a_{2,3}x_3 & & \\ & & a_{3,3}x_3 & = & b_3 & & & & \end{array}$$

The right hand sides of these equations can be divided naturally into rows and columns.

- Row–oriented backward substitution updates (modifies) the right hand side one row at a time. That is, after computing $x_n, x_{n-1}, \ldots, x_{i+1}$, we update b_i as:

$$b_i := b_i - (a_{i,i+1}x_{i+1} + a_{i,i+2}x_{i+2} + \cdots + a_{i,n}x_n) = b_i - \sum_{j=i+1}^{n} a_{i,j}x_j$$

  In this case the variable b_i is "overwritten" with the value $b_i - \sum_{j=i+1}^{n} a_{i,j}x_j$. With this procedure to update the right hand side, an algorithm for row-oriented backward substitution could have the form:

$$\begin{array}{l} \text{for each } i = n, n-1, \ldots, 1 \\ \quad \text{update } b_i := b_i - \sum_{j=i+1}^{n} a_{i,j}x_j \\ \quad \text{compute } x_i := b_i/a_{i,i} \end{array}$$

  Notice that each update step uses elements in the ith *row* of A, $a_{i,i+1}, a_{i,i+2}, \ldots, a_{i,n}$.

- Column–oriented backward substitution updates (modifies) the right hand side one column at a time. That is, after computing x_j, we update $b_1, b_2, \ldots, b_{j-1}$ as:

$$\begin{array}{rcl} b_1 & := & b_1 - a_{1,j}x_j \\ b_2 & := & b_2 - a_{2,j}x_j \\ & \vdots & \\ b_{j-1} & := & b_{j-1} - a_{j-1,j}x_j \end{array}$$

  With this procedure, an algorithm for column-oriented backward substitution could have the form:

for each $j = n, n-1, \ldots, 1$
compute $x_j := b_j / a_{j,j}$
update $b_i := b_i - a_{i,j} x_j, \quad i = 1, 2, \ldots, j-1$

Notice in this case the update steps use elements in the jth *column* of A, $a_{1,j}, a_{2,j}, \ldots, a_{j-1,j}$.

Pseudocodes implementing row– and column–oriented backward substitution are presented in Fig. 3.6.

Row–Oriented	*Column–Oriented*
Input: matrix $A = [a_{i,j}]$ vector $b = [b_i]$ Output: solution vector $x = [x_i]$	Input: matrix $A = [a_{i,j}]$ vector $b = [b_j]$ Output: solution vector $x = [x_j]$
for $i = n$ downto 1 do for $j = i+1$ to n $b_i := b_i - a_{i,j} x_j$ next j $x_i := b_i / a_{i,i}$ next i	for $j = n$ downto 1 do $x_j := b_j / a_{j,j}$ for $i = 1$ to $j-1$ $b_i := b_i - a_{i,j} x_j$ next i next j

Figure 3.6: Pseudocode *Row– and Column–Oriented Backward Substitution.* Note that the algorithm assumes that the matrix A is upper triangular.

The backward substitution process can break down if a diagonal entry of the upper triangular matrix is zero. In this case, the upper triangular matrix is singular, and a linear system involving such a matrix may have no solution or infinitely many solutions.

Problem 3.2.12. *Why is a diagonal linear system also upper triangular?*

Problem 3.2.13. *Illustrate the operation of column–oriented backward substitution when used to solve the upper triangular linear system in Fig. 3.2(c). Hint: Show the value of b each time it has been modified by the for-loop and the value of each entry of x as it is computed.*

Problem 3.2.14. *Use row–oriented backward substitution to solve the linear system:*

$$\begin{aligned} 2x_1 + 2x_2 + 3x_3 + 4x_4 &= 20 \\ 0x_1 + 5x_2 + 6x_3 + 7x_4 &= 34 \\ 0x_1 + 0x_2 + 8x_3 + 9x_4 &= 25 \\ 0x_1 + 0x_2 + 0x_3 + 10x_4 &= 10 \end{aligned}$$

Problem 3.2.15. *Repeat Problem 3.2.14 using the column–oriented backward substitution.*

Problem 3.2.16. *Modify the row– and column–oriented pseudocodes for backward substitution so that the solution x is written over b.*

Problem 3.2.17. *Consider a general upper triangular linear system of order n. Show that row–oriented backward substitution costs* $\frac{n(n-1)}{2}$ *multiplications,* $\frac{n(n-1)}{2}$ *subtractions, and n divisions.*

Problem 3.2.18. *Repeat Problem 3.2.17 using column–oriented backward substitution.*

Problem 3.2.19. *Develop pseudocodes, analogous to those in Fig. 3.6, for row– and column–oriented methods of solving the following linear system:*

$$\begin{array}{rcrcrcl} & & & & a_{1,3}x_3 & = & b_1 \\ & & a_{2,2}x_2 & + & a_{2,3}x_3 & = & b_2 \\ a_{3,1}x_1 & + & a_{3,2}x_2 & + & a_{3,3}x_3 & = & b_3 \end{array}$$

Problem 3.2.20. *Consider the matrix and vector*

$$A = \begin{bmatrix} -1 & 2 & 3 \\ 0 & 0 & 4 \\ 0 & 0 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ c \\ 4 \end{bmatrix}.$$

Why is A singular? For what values c does the linear system $Ax = b$ have no solution? For what values c does the linear system $Ax = b$ have infinitely many solutions?

Problem 3.2.21. *The determinant of a triangular matrix, be it diagonal, lower triangular, or upper triangular, is the product of its diagonal entries.*

(a) Show that a triangular matrix is nonsingular if and only if each of its diagonal entries is nonzero.

(b) Compute the determinant of each of the triangular matrices in Problems 3.2.5, 3.2.11, 3.2.14 and 3.2.20.

Why do these results make sense computationally?

3.3 Gaussian Elimination with Partial Pivoting

The 19^{th} century German mathematician and scientist Carl Friedrich Gauss described a process, called Gaussian elimination in his honor, that uses two elementary operations systematically to transform any given linear system into one that is easy to solve. (Actually, the process was supposedly known centuries earlier.)

The two elementary operations are

(a) exchange two equations

(b) subtract a multiple of one equation from any other equation.

Applying either type of elementary operations to a linear system does not change its solution set. So, we may apply as many of these operations as needed, in any order, and the resulting system of linear equations has the same solution set as the original system. To implement the procedure, though, we need a systematic approach in which we apply the operations. In this section we describe the most commonly implemented scheme, Gaussian elimination with partial pivoting (GEPP). Here our "running examples" will all be computed in exact arithmetic. In the next subsection, we will recompute these examples in four significant digit decimal arithmetic to provide some first insight into the effect of rounding error in the solution of linear systems.

3.3.1 Outline of the GEPP Algorithm

For linear systems of order n, GEPP uses n stages to transform the linear system into upper triangular form. At stage k we eliminate variable x_k from all but the first k equations. To achieve this, each stage uses the same two steps. We illustrate the process on the linear system:

$$\begin{array}{rcrcrcl} 1x_1 & + & 2x_2 & + & (-1)x_3 & = & 0 \\ 2x_1 & + & (-1)x_2 & + & 1x_3 & = & 7 \\ (-3)x_1 & + & 1x_2 & + & 2x_3 & = & 3 \end{array}$$

Stage 1. Eliminate x_1 from all but the first equation.

The **exchange step**, exchanges equations so that among the coefficients multiplying x_1 in all of the equations, the coefficient in the first equation has largest magnitude. If there is more than one such coefficient, choose the first. If this coefficient with largest magnitude is nonzero, then it is underlined and called the **pivot** for stage 1, otherwise stage 1 had no pivot and we terminate the elimination algorithm. In this example, the pivot occurs in the third equation, so equations 1 and 3 are exchanged.

$$\begin{aligned} 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \end{aligned}$$

exchange $\downarrow$ step

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ 2x_1 + (-1)x_2 + \quad 1x_3 &= 7 \\ 1x_1 + \quad 2x_2 + (-1)x_3 &= 0 \end{aligned}$$

The **elimination step**, eliminates variable x_1 from all but the first equation. For this example, this involves subtracting $m_{2,1} = \frac{2}{-3}$ times equation 1 from equation 2, and $m_{3,1} = \frac{1}{-3}$ times equation 1 from equation 3. Each of the numbers $m_{i,1}$ is a **multiplier**; the first subscript on $m_{i,1}$ indicates from which equation the multiple of the first equation is subtracted. $m_{i,1}$ is computed as the coefficient of x_1 in equation i divided by the coefficient of x_1 in equation 1 (that is, the pivot).

elimination $\downarrow$ step

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ 0x_1 + (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9 \\ 0x_1 + \quad \tfrac{7}{3}x_2 + (-\tfrac{1}{3})x_3 &= 1 \end{aligned}$$

Stage 2. Eliminate x_2 from all but the first two equations.

Stage 2 repeats the above steps for the variable x_2 on a smaller linear system obtained by removing the first equation from the system at the end of stage 1. This new system involves one fewer unknown (the first stage eliminated variable x_1).

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= 1 \end{aligned}$$

exchange $\downarrow$ step

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= 1 \\ (-\tfrac{1}{3})x_2 + \quad \tfrac{7}{3}x_3 &= 9 \end{aligned}$$

The **exchange step**, exchanges equations so that among the coefficients multiplying x_2 in all of the remaining equations, the second equation has largest magnitude. If this coefficient with largest magnitude is nonzero, then it is underlined and called the **pivot** for stage 2, otherwise stage 2 has no pivot and we terminate. In our example, the pivot occurs in the third equation, so equations 2 and 3 are exchanged.

elimination $\downarrow$ step

$$\begin{aligned} \underline{(-3)}\, x_1 + \quad 1x_2 + \quad 2x_3 &= 3 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= 1 \\ 0x_2 + \quad \tfrac{16}{7}x_3 &= \tfrac{64}{7} \end{aligned}$$

The **elimination step**, eliminates variable x_2 from all below the second equation. For our example, this involves subtracting $m_{3,2} = \frac{-1/3}{7/3} = -\frac{1}{7}$ times equation 2 from equation 3. The **multiplier** $m_{i,2}$ is computed as the coefficient of x_2 in equation i divided by the coefficient of x_2 in equation 2 (that is, the pivot).

This process continues until the last equation involves only one unknown variable, and the transformed linear system is in upper triangular form. The last stage of the algorithm simply involves identifying the coefficient in the last equation as the final pivot element. In our simple example, we are done at stage 3, where we identify the last pivot element by underlining the coefficient for x_3 in the last equation, and obtain the upper triangular linear system

$$\begin{aligned} \underline{(-3)}\, x_1 + 1x_2 + \quad 2x_3 &= 3 \\ \underline{\tfrac{7}{3}}\, x_2 + (-\tfrac{1}{3})x_3 &= 1 \\ \underline{\tfrac{16}{7}}\, x_3 &= \tfrac{64}{7} \end{aligned}$$

GEPP is now finished. When the entries on the diagonal of the final upper triangular linear system are nonzero, as it is in our illustrative example, the linear system is nonsingular and its solution may be determined by backward substitution. (Recommendation: If you are determining a solution by hand in exact arithmetic, you may check that your computed solution is correct by

showing that it satisfies all the equations of the original linear system. If you are determining the solution approximately this check may be unreliable as we will see later.)

The diagonal entries of the upper triangular linear system produced by GEPP play an important role. Specifically, the k^{th} diagonal entry, i.e., the coefficient of x_k in the k^{th} equation, is the pivot for the k^{th} stage of GEPP. So, *the upper triangular linear system produced by GEPP is nonsingular if and only if each stage of GEPP has a pivot*.

To summarize:

- GEPP can always be used to transform a linear system of order n into an upper triangular linear system with the same solution set.
- The k^{th} stage of GEPP begins with a linear system that involves $n-k+1$ equations in the $n-k+1$ unknowns $x_k, x_{k+1}, \cdots, x_n$. The k^{th} stage of GEPP eliminates x_k and ends with a linear system that involves $n-k$ equations in the $n-k$ unknowns $x_{k+1}, x_{k+2}, \cdots, x_n$.
- If GEPP finds a non-zero pivot at every stage, then the final upper triangular linear system is nonsingular. The solution of the original linear system can be determined by applying backward substitution to this upper triangular linear system.
- If GEPP fails to find a non-zero pivot at some stage, then the linear system is singular and the original linear system either has no solution or an infinite number of solutions.

We emphasize that **if GEPP does not find a non-zero pivot at some stage, we can conclude immediately that the original linear system is singular**. Consequently, many GEPP codes simply terminate elimination and return a message that indicates the original linear system is singular.

Example 3.3.1. The previous discussion showed that GEPP transforms the linear system to upper triangular form:

$$\begin{array}{rrrr} 1x_1 + & 2x_2 + & (-1)x_3 = & 0 \\ 2x_1 + & (-1)x_2 + & 1x_3 = & 7 \\ (-3)x_1 + & 1x_2 + & 2x_3 = & 3 \end{array} \quad \rightarrow \quad \cdots \quad \rightarrow \quad \begin{array}{rrrr} \underline{(-3)}\, x_1 + & 1x_2 + & 2x_3 = & 3 \\ & \underline{\frac{7}{3}}\, x_2 + & (-\frac{1}{3})x_3 = & 1 \\ & & \underline{\frac{16}{7}}\, x_3 = & \frac{64}{7} \end{array}$$

Using backward substitution, we find that the solution of this linear system is given by:

$$x_3 = \frac{\frac{64}{7}}{\frac{16}{7}} = 4\,, \quad x_2 = \frac{1 + \frac{1}{3}\cdot 4}{\frac{7}{3}} = 1\,, \quad x_1 = \frac{3 - 1\cdot 1 - 2\cdot 4}{-3} = 2\,.$$

Example 3.3.2. Consider the linear system:

$$\begin{array}{rrrr} 4x_1 + & 6x_2 + & (-10)x_3 = & 0 \\ 2x_1 + & 2x_2 + & 2x_3 = & 6 \\ 1x_1 + & (-1)x_2 + & 4x_3 = & 4 \end{array}$$

Applying GEPP to this example we obtain:

In the first stage, the largest coefficient in magnitude of x_1 is already in the first equation, so no exchange steps are needed. The elimination steps then proceed with the multipliers $m_{2,1} = \frac{2}{4} = 0.5$ and $m_{3,1} = \frac{1}{4} = 0.25$.

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 2x_1 + \quad 2x_2 + \quad 2x_3 &= 6 \\ 1x_1 + (-1)x_2 + \quad 4x_3 &= 4 \end{aligned}$$

$$\downarrow$$

In the second stage, we observe that the largest x_2 coefficient in magnitude in the last two equations occurs in the third equation, so the second and third equations are exchanged. The elimination step then proceeds with the multiplier $m_{3,2} = \frac{-1}{-2.5} = 0.4$.

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \quad (-1)x_2 + \quad 7x_3 &= 6 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \\ 0x_1 + \quad (-1)x_2 + \quad 7x_3 &= 6 \end{aligned}$$

$$\downarrow$$

In the final stage we identify the final pivot entry, and observe that all pivots are non-zero, and thus the linear system is nonsingular and there is a unique solution.

$$\begin{aligned} \underline{4}\,x_1 + \quad 6x_2 + (-10)x_3 &= 0 \\ 0x_1 + \underline{(-2.5)}\,x_2 + \quad 6.5x_3 &= 4 \\ 0x_1 + \quad 0x_2 + \quad \underline{4.4}\,x_3 &= 4.4 \end{aligned}$$

Using backward substitution, we find the solution of the linear system:

$$x_3 = \frac{4.4}{4.4} = 1\,, \quad x_2 = \frac{4 + 6.5 \cdot 1}{-2.5} = 1\,, \quad x_1 = \frac{0 - 6 \cdot 1 + 10 \cdot 1}{4} = 1\,.$$

Example 3.3.3. Consider the linear system:

$$\begin{aligned} 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ 3x_1 + (-6)x_2 + \quad 9x_3 &= 0 \end{aligned}$$

Applying GEPP to this example we obtain:

In the first stage, the largest coefficient in magnitude of x_1 is in the third equation, so we exchange the first and third equations. The elimination steps then proceed with the multipliers $m_{2,1} = \frac{-1}{3}$ and $m_{3,1} = \frac{1}{3}$.

$$\begin{aligned} 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \end{aligned}$$

$$\downarrow$$

$$\begin{aligned} \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \\ (-1)x_1 + \quad 2x_2 + (-1)x_3 &= 1 \\ 1x_1 + (-2)x_2 + (-1)x_3 &= 2 \end{aligned}$$

$$\downarrow$$

In the second stage, we observe that all coefficients multiplying x_2 in the last two equations are zero. We therefore fail to find a non-zero pivot, and conclude that the linear system is singular.

$$\begin{aligned} \underline{3}\,x_1 + (-6)x_2 + \quad 9x_3 &= 0 \\ 0x_1 + \quad 0x_2 + \quad 2x_3 &= 1 \\ 0x_1 + \quad 0x_2 + (-4)x_3 &= 2 \end{aligned}$$

Problem 3.3.1. *Use GEPP followed by backward substitution to solve the linear system of Example 3.1.1. Explicitly display the value of each pivot and each multiplier.*

Problem 3.3.2. *Use GEPP followed by backward substitution to solve the following linear system. Explicitly display the value of each pivot and multiplier.*

$$\begin{aligned} 3x_1 + \quad 0x_2 + 0x_3 + \quad 0x_4 &= \quad 6 \\ 2x_1 + (-3)x_2 + 0x_3 + \quad 0x_4 &= \quad 7 \\ 1x_1 + \quad 0x_2 + 5x_3 + \quad 0x_4 &= -8 \\ 0x_1 + \quad 2x_2 + 4x_3 + (-3)x_4 &= -3 \end{aligned}$$

Problem 3.3.3. *Use GEPP and backward substitution to solve the following linear system. Explicitly display the value of each pivot and multiplier.*

$$\begin{aligned} 2x_1 + x_3 &= 1 \\ x_2 + 4x_3 &= 3 \\ x_1 + 2x_2 &= -2 \end{aligned}$$

Problem 3.3.4. *GEPP provides an efficient way to compute the determinant of any matrix. Recall that the operations used by GEPP are (1) exchange two equations, and (2) subtract a multiple of one equation from another (different) equation. Of these two operations, only the exchange operation changes the value of the determinant of the matrix of coefficients, and then it only changes its sign. In particular, suppose GEPP transforms the coefficient matrix A into the upper triangular coefficient matrix U using m actual exchanges, i.e., exchange steps where an exchange of equations actually occurs. Then*

$$\det(A) = (-1)^m \det(U).$$

(a) Use GEPP to show that

$$\det\left(\begin{bmatrix} 4 & 6 & -10 \\ 2 & 2 & 2 \\ 1 & -1 & 4 \end{bmatrix}\right) = (-1)^1 \det\left(\begin{bmatrix} 4 & 6 & -10 \\ 0 & -2.5 & 6.5 \\ 0 & 0 & 4.4 \end{bmatrix}\right) = -(4)(-2.5)(4.4) = 44$$

Recall from Problem 3.2.21 that the determinant of a triangular matrix is the product of its diagonal entries.

(b) Use GEPP to calculate the determinant of each of the matrices:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 4 & -3 & 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 3 & 4 & 5 \\ -1 & 2 & -2 & 1 \\ 2 & 6 & 3 & 7 \end{bmatrix}$$

3.3.2 The GEPP Algorithm in Inexact Arithmetic

The GEPP algorithm was outlined in the previous subsection using exact arithmetic throughout. In reality, the solution of linear systems is usually computed using DP standard arithmetic and the errors incurred depend on the IEEE DP (binary) representation of the original system and on the effects of rounding error in IEEE DP arithmetic. Since it is difficult to follow binary arithmetic we will resort to decimal arithmetic to simulate the effects of approximate computation. In this subsection, we will use round-to-nearest 4 significant digit decimal arithmetic to solve the problems introduced in the previous subsection.

First, we illustrate the process on the linear system in Example 3.3.1:

$$\begin{aligned} 1.000x_1 + \quad 2.000x_2 + (-1.000)x_3 &= \quad 0 \\ 2.000x_1 + (-1.000)x_2 + \quad 1.000x_3 &= 7.000 \\ (-3.000)x_1 + \quad 1.000x_2 + \quad 2.000x_3 &= 3.000 \end{aligned}$$

Performing the interchange, calculating the mutlipliers $m_{21} = -0.6667$ and $m_{31} = -0.3333$, and eliminating x_1 from the second and third equations we have

$$\begin{aligned} (-3.000)x_1 + \quad 1.000x_2 + \quad 2.000x_3 &= \quad 3.000 \\ 0x_1 + (-0.3333)x_2 + \quad 2.333x_3 &= \quad 9.000 \\ 0x_1 + \quad 2.333x_2 + (-0.3334)x_3 &= 0.9999 \end{aligned}$$

Performing the interchange, calculating the mutliplier $m_{32} = -0.1429$, and eliminating x_2 from the third equation we have

$$\begin{aligned}(-3.000)x_1 + 1.000x_2 + \quad 2.000x_3 &= \ 3.000\\ 0x_1 + 2.333x_2 + (-0.3334)x_3 &= 0.9999\\ 0x_1 + \quad 0x_2 + \quad 2.285x_3 &= \ 9.143\end{aligned}$$

Backsolving we have $x_3 = 4.001$, $x_2 = 1.000$ and $x_1 = 2.001$, a rounding error level perturbation of the answer with exact arithmetic.

If we consider now Example 3.3.2, we see that all the working was exact in four significant digit decimal arithmetic. So, we will get precisely the same answers as before. However, we cannot represent exactly the arithmetic in this example in binary DP arithmetic. So, computationally, in say MATLAB, we would obtain an answer that differed from the exact answer at the level of roundoff error.

Consider next the problem in Example 3.3.3:

$$\begin{aligned}1.000x_1 + (-2.000)x_2 + (-1.000)x_3 &= 2.000\\ (-1.000)x_1 + \quad 2.000x_2 + (-1.000)x_3 &= 1.000\\ 3.000x_1 + (-6.000)x_2 + \quad 9.000x_3 &= \quad 0\end{aligned}$$

Performing the interchange, calculating the mutlipliers $m_{21} = 0.3333$ and $m_{31} = -0.3333$, and eliminating x_1 from the second and third equations we have

$$\begin{aligned}3.000x_1 + (-6.000)x_2 + \quad 9.000x_3 &= \quad 0\\ 0x_1 + \quad 0x_2 + \quad (2.000)x_3 &= 1.000\\ 0x_1 + \quad 0x_2 + (-4.000)x_3 &= 2.000\end{aligned}$$

That is, roundoff does not affect the result and we still observe that the pivot for the next stage is zero hence we must stop.

If, instead, we scale the second equation to give

$$\begin{aligned}1x_1 + (-2)x_2 + (-1)x_3 &= 2\\ (-7)x_1 + \quad 14x_2 + (-7)x_3 &= 7\\ 3x_1 + (-6)x_2 + \quad 9x_3 &= 0\end{aligned}$$

then the first step of GEPP is to interchange the first and second equations to give

$$\begin{aligned}(-7)x_1 + \quad 14x_2 + (-7)x_3 &= 7\\ 1x_1 + (-2)x_2 + (-1)x_3 &= 2\\ 3x_1 + (-6)x_2 + \quad 9x_3 &= 0\end{aligned}$$

Next, we calculate the multipliers $m_{21} = \dfrac{1}{(-7)} = -0.1429$ and $m_{31} = \dfrac{3}{(-7)} = -0.4286$. Eliminating we compute

$$\begin{aligned}(-7)x_1 + \quad 14x_2 + (-7)x_3 &= 7\\ 0x_1 + (-0.001)x_2 + (-2)x_3 &= 3\\ 0x_1 + \quad 0x_2 + \quad 6x_3 &= 3\end{aligned}$$

Observe that the elimination is now complete. The result differs from the exact arithmetic results in just the $(2,2)$ position but this small change is crucial because the resulting system is now nonsingular. When we compute a solution, we obtain, $x_3 = 0.5$, $x_2 = -4000$ and $x_1 = 55990$; the large values possibly giving away that something is wrong. In the next subsection we describe a GEPP algorithm. This algorithm includes a test for singularity that would flag the above problem as singular even though all the pivots are nonzero.

Example 3.3.4. Here is a further example designed to reinforce what we have just seen on the effects of inexact arithmetic. Using exact arithmetic, 2 stages of GEPP transforms the singular linear system:

$$\begin{aligned} 1x_1 + \quad 1x_2 + 1x_3 &= 1 \\ 1x_1 + (-1)x_2 + 2x_3 &= 2 \\ 3x_1 + \quad 1x_2 + 4x_3 &= 4 \end{aligned}$$

into the triangular form:

$$\begin{aligned} 3x_1 + \quad 1x_2 + \quad 4x_3 &= \quad 4 \\ 0x_1 + (-4/3)x_2 + 2/3x_3 &= 2/3 \\ 0x_1 + \quad 0x_2 + \quad 0x_3 &= \quad 0 \end{aligned}$$

which has an infinite number of solutions.

Using GEPP with in four significant digit decimal arithmetic, we compute multipliers $m_{21} = m_{31} = 0.3333$ and eliminating we get

$$\begin{aligned} 3x_1 + \quad 1x_2 + \quad 4x_3 &= \quad 4 \\ 0x_1 + (-1.333)x_2 + \quad 0.667x_3 &= \quad 0.667 \\ 0x_1 + \quad 0.667x_2 + (-0.333)x_3 &= -0.333 \end{aligned}$$

So, $m_{32} = -0.5002$ and eliminating we get

$$\begin{aligned} 3x_1 + \quad 1x_2 + \quad 4x_3 &= \quad 4 \\ 0x_1 + (-1.333)x_2 + \quad 0.667x_3 &= \quad 0.667 \\ 0x_1 + \quad 0x_2 + 0.0006x_3 &= 0.0006 \end{aligned}$$

and backsolving we compute $x_3 = 1$ and $x_2 = x_1 = 0$, *one* solution of the original linear system. Here, we observe some mild effects of loss of significance due to cancellation. Later, in Section 3.3.4, we will see much greater numerical errors due to loss of significance.

Problem 3.3.5. *If the original first equation $x_1 + x_2 + x_3 = 1$ in the example above is replaced by $x_1 + x_2 + x_3 = 2$, then with exact arithmetic 2 stages of GE yields $0x_3 = 1$ as the last equation and there is no solution. In four significant digit decimal arithmetic, a suspiciously large solution is computed. Show this and explain why it happens.*

3.3.3 Implementing the GEPP Algorithm

The version of GEPP discussed in this section is properly called *Gaussian elimination with partial pivoting by rows for size*. The phrase "partial pivoting" refers to the fact that only equations are exchanged in the exchange step. An alternative is "complete pivoting" where both equations and unknowns are exchanged. The phrase "by rows for size" refers to the choice in the exchange step where a nonzero coefficient with largest magnitude is chosen as pivot. Theoretically, Gaussian elimination simply requires that any nonzero number be chosen as pivot. Partial pivoting by rows for size serves two purposes. First, it removes the theoretical problem when, prior to the k^{th} exchange step, the coefficient multiplying x_k in the k^{th} equation is zero. Second, in almost all cases, partial pivoting improves the quality of the solution computed when finite precision, rather than exact, arithmetic is used in the elimination step.

In this subsection we present detailed pseudocode that combines GEPP with backward substitution to solve linear systems of equations. The basic algorithm is:

- for stages $k = 1, 2, \ldots, n-1$
  - find the kth pivot
  - if the kth pivot is zero, quit – the linear system is singular
  - perform row interchanges, if needed
  - compute the multipliers
  - perform the elimination
- for stage n, if the final pivot is zero, quit – the linear system is singular
- perform backward substitution

Before presenting pseudocode for the entire algorithm, we consider implementation details for some of the GEPP steps. Assume that we are given the coefficient matrix A and right hand side vector b of the linear system.

- To find the kth pivot, we need to find the largest of the values $|a_{k,k}|, |a_{k+1,k}|, \ldots, |a_{n,k}|$. This can be done using a simple search:

  $p := k$
  for $i = k + 1$ to n
  if $|a_{i,k}| > |a_{p,k}|$ then $p := i$
  next i

- If the above search finds that $p > k$, then we need to exchange rows. This is fairly simple, though we have to be careful to use "temporary" storage when overwriting the coefficients. (In some languages, such as MATLAB, you can perform the interchanges directly, as we will show in Section 3.6; the temporary storage is created and used in the background and is invisible to the user.)

  if $p > k$ then
  for $j = k$ to n
  $temp := a_{k,j}$
  $a_{k,j} := a_{p,j}$
  $a_{p,j} := temp$
  next j
  $temp := b_k$
  $b_k := b_p$
  $b_p := temp$
  endif

  Notice that when exchanging equations, we must exchange the coefficients in the matrix A and the corresponding values in the right hand side vector b. (Another possibility is to perform the interchanges virtually; that is, to rewrite the algorithm so that we don't physically interchange elements but instead leave them in place but act as if they had been interchanged. This involves using indirect addressing which may be more efficient than physically performing interchanges.)

- Computing the multipliers, $m_{i,k}$, and performing the elimination step can be implemented as:

  for $i = k + 1$ to n
  $m_{i,k} := a_{i,k}/a_{k,k}$
  $a_{i,k} := 0$
  for $j = k + 1$ to n
  $a_{i,j} := a_{i,j} - m_{i,k} * a_{k,j}$
  next j
  $b_i := b_i - m_{i,k} * b_k$
  next i

  Since we know that $a_{i,k}$ should be zero after the elimination step, we do not perform the elimination step on these values, and instead just set them to zero using the instruction $a_{i,k} := 0$. However, we note that since these elements are not involved in determining the solution via backward substitution, this instruction is unnecessary, and has only a "cosmetic" purpose.

Putting these steps together, and including pseudocode for backward substitution, we obtain the pseudocode shown in Fig. 3.7. Note that we implemented the elimination step to update $a_{i,j}$ in a specific order. In particular, the entries in row $k + 1$ are updated first, followed by the entries in row $k + 2$, and so on, until the entries in row n are updated. As with the forward and backward substitution methods, we refer to this as *row-oriented* GEPP, and to be consistent we combine it with

row-oriented backward substitution. It is not difficult to modify the GEPP procedure so that it is column-oriented, and in that case we would combine it with column-oriented backward substitution.

During stage k, the pseudocode in Fig. 3.7 only modifies equations $(k+1)$ through n and the coefficients multiplying $x_k, x_{k+1}, \cdots, x_n$. Of the three fundamental operations: add (or subtract), multiply, and divide, generally add (or subtract) is fastest, multiply is intermediate, and divide is slowest. So, rather than compute the multiplier as $m_{i,k} := a_{i,k}/a_{k,k}$, one might compute the reciprocal $a_{k,k} := 1/a_{k,k}$ outside the i loop and then compute the multiplier as $m_{i,k} := a_{i,k}a_{k,k}$. In stage k, this would replace $n-k$ divisions with 1 division followed by $n-k$ multiplications, which is usually faster. Keeping the reciprocal in $a_{k,k}$ also helps in backward substitution where each divide is replaced by a multiply.

Row-Oriented GEPP with Backward Substitution

Input: matrix $A = [a_{i,j}]$
vector $b = [b_i]$

Output: solution vector $x = [x_i]$

```
for k = 1 to n − 1
    p := k
    for i = k + 1 to n
        if |a_{i,k}| > |a_{p,k}| then p := i
    next i
    if p > k then
        for j = k to n
            temp := a_{k,j}; a_{k,j} := a_{p,j}; a_{p,j} := temp
        next j
        temp := b_k; b_k := b_p; b_p := temp
    endif
    if a_{k,k} = 0 then return (linear system is singular)
    for i = k + 1 to n
        m_{i,k} := a_{i,k}/a_{k,k}
        a_{i,k} := 0
        for j = k + 1 to n
            a_{i,j} := a_{i,j} − m_{i,k} ∗ a_{k,j}
        next j
        b_i := b_i − m_{i,k} ∗ b_k
    next i
next k
if a_{n,n} = 0 then return (linear system is singular)

for i = n downto 1 do
    for j = i + 1 to n
        b_i := b_i − a_{i,j} ∗ x_j
    next j
    x_i := b_i/a_{i,i}
next i
```

Figure 3.7: Pseudocode for *row oriented GEPP and Backward Substitution*

Problem 3.3.6. *In Fig. 3.7, in the second for-j loop (the elimination loop) the code could be logically simplified by recognizing that the effect of $a_{i,k} = 0$ could be achieved by removing this statement and extending the for-j loop so it starts at $j = k$ instead of $j = k + 1$. Why is this NOT a good idea?*

Problem 3.3.7. *In Fig. 3.7, the elimination step is row–oriented. Change the elimination step so it is column–oriented. Hint: You must exchange the order of the for-i and for-j loops.*

Problem 3.3.8. *In Fig. 3.7, show that the elimination step of the k^{th} stage of GEPP requires $n - k$ divisions to compute the multipliers, and $(n - k)^2 + n - k$ multiplications and subtractions to perform the elimination. Conclude that the elimination steps of GEPP requires about*

$$\begin{array}{lllll} \sum_{k=1}^{n-1}(n-k) & = & \sum_{k=1}^{n-1} k & = & \frac{n(n-1)}{2} \sim \frac{n^2}{2} \text{ divisions} \\ \sum_{k=1}^{n-1}(n-k)^2 & = & \sum_{k=1}^{n-1} k^2 & = & \frac{n(n-1)(2n-1)}{6} \sim \frac{n^3}{3} \text{ multiplications and subtractions} \end{array}$$

The notation $f(n) \sim g(n)$ is used when the ratio $\dfrac{f(n)}{g(n)}$ approaches 1 as n becomes large.

Problem 3.3.9. *How does the result in Problem 3.3.8 change if we first compute the reciprocals of the pivots and then use these values in computing the multipliers?*

Problem 3.3.10. *How many comparisons does GEPP use to find pivot elements?*

Problem 3.3.11. *How many assignment statements are needed for all the interchanges in the GEPP algorithm, assuming that at each stage an interchange is required?*

3.3.4 The Role of Interchanges

To illustrate the effect of the choice of pivot, we apply Gaussian elimination to the linear system

$$\begin{aligned} 0.000025x_1 + 1x_2 &= 1 \\ 1x_1 + 1x_2 &= 2 \end{aligned}$$

for which the exact solution is $x_1 = \frac{40000}{39999} \approx 1$ and $x_2 = \frac{39998}{39999} \approx 1$.

What result is obtained with Gaussian elimination using floating–point arithmetic? To make pen-and-paper computation easy, we use round-to-nearest 4 significant digit decimal arithmetic; i.e., the result of every add, subtract, multiply, and divide is rounded to the nearest decimal number with 4 significant digits.

Consider what happens if the exchange step is not used. The multiplier is computed exactly because its value $\frac{1}{0.000025} = 40000$ rounds to itself. The elimination step subtracts 40000 times equation 1 from equation 2. Now 40000 times equation 1 is computed exactly. So, the only rounding error in the elimination step is at the subtraction. There, the coefficient multiplying x_2 in the second equation is $1 - 40000 = -39999$, which rounds to -40000, and the right hand side is $2 - 40000 = -39998$, which also rounds to -40000. So the result is

$$\begin{aligned} 0.000025x_1 + \quad 1x_2 &= \quad 1 \\ 0x_1 + (-40000)x_2 &= -40000 \end{aligned}$$

Backward substitution commits no further rounding errors and produces the approximate solution

$$x_2 = \frac{-40000}{-40000} = 1, \quad x_1 = \frac{1 - x_2}{0.000025} = 0$$

This computed solution differs significantly from the exact solution. Why? Observe that the computed solution has $x_2 = 1$. This is an accurate approximation of its exact value $\frac{39998}{39999}$, in error

by $\frac{39998}{39999} - 1 = -\frac{1}{39999}$. When the approximate value of x_1 is computed as $\frac{1-x_2}{0.000025}$, catastrophic cancellation occurs when the approximate value $x_2 = 1$ is subtracted from 1.

If we include the exchange step, the result of the exchange step is

$$\begin{aligned} 1x_1 + 1x_2 &= 2 \\ 0.000025x_1 + 1x_2 &= 1 \end{aligned}$$

The multiplier $\frac{0.000025}{1} = 0.000025$ is computed exactly, as is 0.000025 times equation 1. The result of the elimination step is

$$\begin{aligned} 1x_1 + 1x_2 &= 2 \\ 0x_1 + 1x_2 &= 1 \end{aligned}$$

because, in the subtract operation, $1 - 0.000025 = 0.999975$ rounds to 1 and $1 - 2 \times 0.000025 = 0.99995$ rounds to 1. Backward substitution commits no further rounding errors and produces an approximate solution

$$x_2 = 1, \quad x_1 = \frac{2 - x_2}{1} = 1$$

that is correct to four significant digits.

The above computation uses round-to-nearest 4 significant digit decimal arithmetic. It leaves open the question of what impact the choice of the number of digits in the above calculation so next we repeat the calculation above using first round-to-nearest 5 significant digit decimal arithmetic then round-to-nearest 3 significant digit decimal arithmetic to highlight the differences that may arise when computing to different accuracies.

Consider what happens if we work in round-to-nearest 5 significant digit decimal arithmetic without exchanges. The multiplier is computed exactly because its value $\frac{1}{0.000025} = 40000$ rounds to itself. The elimination step subtracts 40000 times equation 1 from equation 2. Now, 40000 times equation 1 is computed exactly. So, the only possible rounding error in the elimination step is at the subtraction. There, the coefficient multiplying x_2 in the second equation is $1 - 40000 = -39999$, which is already rounded to five digits, and the right hand side is $2 - 40000 = -39998$, which is again correctly rounded. So the result is

$$\begin{aligned} 0.000025x_1 + \quad 1x_2 &= \quad 1 \\ 0x_1 + (-39999)x_2 &= -39998 \end{aligned}$$

Backward substitution produces the approximate solution

$$x_2 = \frac{-39998}{-39999} = .99997, \quad x_1 = \frac{1 - x_2}{0.000025} = 1.2$$

So, the impact of the extra precision is to compute a less inaccurate result. Next, consider what happens if we permit exchanges and use round-to-nearest 5 significant digit decimal arithmetic. The result of the exchange step is

$$\begin{aligned} 1x_1 + 1x_2 &= 2 \\ 0.000025x_1 + 1x_2 &= 1 \end{aligned}$$

The multiplier $\frac{0.000025}{1} = 0.000025$ is computed exactly, as is 0.000025 times equation 1. The result of the elimination step is

$$\begin{aligned} 1x_1 + \quad 1x_2 \quad &= \quad 2 \\ 0x_1 + 0.99998x_2 &= 0.99995 \end{aligned}$$

because, in the subtract operation, $1 - 0.000025 = 0.999975$ rounds to 0.99998 and $1 - 2 \times 0.000025 = 0.99995$ is correctly rounded. Backward substitution produces an approximate solution

$$x_2 = .99997, \quad x_1 = \frac{2 - x_2}{1} = 1$$

that is correct to five significant digits

Finally, we work in round-to-nearest 3 significant digit decimal arithmetic without using exchanges. The multiplier is computed exactly because its value $\frac{1}{0.000025} = 40000$ rounds to itself. The

elimination step subtracts 40000 times equation 1 from equation 2. Now 40000 times equation 1 is computed exactly. So, the only possible rounding error in the elimination step is at the subtraction. There, the coefficient multiplying x_2 in the second equation is $1 - 40000 = -40000$, to three digits, and the right hand side is $2 - 40000 = -40000$, which is again correctly rounded. So, the result is

$$\begin{aligned} 0.000025x_1 + \quad 1x_2 &= \quad 1 \\ 0x_1 + (-40000)x_2 &= -40000 \end{aligned}$$

Backward substitution produces the approximate solution

$$x_2 = \frac{-40000}{-40000} = 1.0, \quad x_1 = \frac{1 - x_2}{0.000025} = 0.0$$

So, for this example working to three digits without exchanges produces the same result as working to four digits. The reader may verify that working to three digits with exchanges produces the correct result to three digits

Partial pivoting by rows for size is a heuristic. One explanation why this heuristic is generally successful is as follows. Given a list of candidates for the next pivot, those with smaller magnitude are more likely to have been formed by a subtract magnitude computation of larger numbers, so the resulting cancellation might make them less accurate than the other candidates for pivot. The multipliers determined by dividing by such smaller magnitude, inaccurate numbers are therefore larger magnitude, inaccurate numbers. Consequently, elimination may produce large and unexpectedly inaccurate coefficients; in extreme circumstances, these large coefficients may contain little information from the coefficients of the original equations. While the heuristic of partial pivoting by rows for size generally improves the chances that GEPP will produce an accurate answer, it is neither perfect nor always better than any other interchange strategy.

Problem 3.3.12. *Use Gaussian elimination to verify that the exact solution of the linear system*

$$\begin{aligned} 0.000025x_1 + 1x_2 &= 1 \\ 1x_1 + 1x_2 &= 2 \end{aligned}$$

is

$$x_1 = \frac{40000}{39999} \approx 1, \quad x_2 = \frac{39998}{39999} \approx 1$$

Problem 3.3.13. *(Watkins) Carry out Gaussian elimination* without *exchange steps and with row–oriented backward substitution on the linear system:*

$$\begin{aligned} 0.002x_1 + 1.231x_2 + 2.471x_3 &= 3.704 \\ 1.196x_1 + 3.165x_2 + 2.543x_3 &= 6.904 \\ 1.475x_1 + 4.271x_2 + 2.142x_3 &= 7.888 \end{aligned}$$

Use round-to-nearest 4 *significant digit decimal arithmetic. Display each pivot, each multiplier, and the result of each elimination step. Does catastrophic cancellation occur, and if so where? Hint: The exact solution is* $x_1 = 1$, $x_2 = 1$ *and* $x_3 = 1$. *The computed solution with approximate arithmetic and no exchanges is* $x_1 = 4.000$, $x_2 = -1.012$ *and* $x_3 = 2.000$.

Problem 3.3.14. *Carry out GEPP (with exchange steps) and row–oriented backward substitution on the linear system in Problem 3.3.13. Use round-to-nearest* 4 *significant digit decimal arithmetic. Display each pivot, each multiplier, and the result of each elimination step. Does catastrophic cancellation occur, and if so where?*

Problem 3.3.15. *Round the entries in the linear system in Problem 3.3.13 to three significant digits. Now, repeat the calculations in Problems 3.3.13 and 3.3.14 using round-to-nearest* 3 *significant digit decimal arithmetic. What do you observe?*

3.4 Gaussian Elimination and Matrix Factorizations

The concept of matrix factorizations is fundamentally important in the process of numerically solving linear systems, $Ax = b$. The basic idea is to decompose the matrix A into a product of *simply solved systems*, from which the solution of $Ax = b$ can be easily computed. The idea is similar to what we might do when trying to find the roots of a polynomial. For example, the equations

$$x^3 - 6x^2 + 11x - 6 = 0 \quad \text{and} \quad (x-1)(x-2)(x-3) = 0$$

are equivalent, but the factored form is clearly much easier to solve. In general, we cannot solve linear systems so easily (i.e., by inspection), but decomposing A makes solving the linear system $Ax = b$ computationally simpler. Some knowledge of matrix algebra, especially matrix multiplication, is needed to understand the concepts introduced here; a review is given in Section 1.2.

3.4.1 LU Factorization

Suppose we apply Gaussian elimination to a $n \times n$ matrix A without interchanging any rows. For example, for $n = 3$,

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a_{2,2}^{(1)} & a_{2,3}^{(1)} \\ 0 & a_{3,2}^{(1)} & a_{3,3}^{(1)} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a_{2,2}^{(1)} & a_{2,3}^{(1)} \\ 0 & 0 & a_{3,3}^{(2)} \end{bmatrix}$$

$$m_{2,1} = \tfrac{a_{2,1}}{a_{1,1}}, \quad m_{3,1} = \tfrac{a_{3,1}}{a_{1,1}} \qquad\qquad m_{3,2} = \tfrac{a_{3,2}^{(1)}}{a_{2,2}^{(1)}}$$

Here the superscript on the entry $a_{i,j}^{(k)}$ indicates an element of the matrix modified during the kth elimination step. Recall that, in general, the multipliers are computed as

$$m_{i,j} = \frac{\text{element to be eliminated}}{\text{current pivot element}}.$$

If the process does not break down (that is, all the pivot elements, $a_{1,1}, a_{2,2}^{(1)}, a_{3,3}^{(2)}, \ldots$, are nonzero) then the matrix A can be factored as $A = LU$ where

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_{2,1} & 1 & 0 & \cdots & 0 \\ m_{3,1} & m_{3,2} & 1 & & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ m_{n,1} & m_{n,2} & m_{n,3} & \cdots & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} \text{upper triangular matrix} \\ \text{after the elimination is} \\ \text{complete} \end{bmatrix}.$$

Example 3.4.1. Let

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -3 & 2 \\ 3 & 1 & -1 \end{bmatrix}.$$

Using Gaussian elimination without row interchanges, we obtain

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & -5 & -10 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & 0 & -\frac{50}{7} \end{bmatrix}$$

$$m_{2,1} = 2 \quad m_{3,1} = 3 \qquad\qquad m_{3,2} = \tfrac{5}{7}$$

and thus

$$L = \begin{bmatrix} 1 & 0 & 0 \\ m_{2,1} & 1 & 0 \\ m_{3,1} & m_{3,2} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & \frac{5}{7} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -7 & -4 \\ 0 & 0 & -\frac{50}{7} \end{bmatrix}.$$

It is straightforward to verify[1] that $A = LU$.

If we can compute the factorization $A = LU$, then

$$Ax = b \;\Rightarrow\; LUx = b \;\Rightarrow\; Ly = b, \text{ where } Ux = y.$$

So, to solve $Ax = b$:

- Compute the factorization $A = LU$.
- Solve $Ly = b$ using forward substitution.
- Solve $Ux = y$ using backward substitution.

Example 3.4.2. Consider solving the linear system $Ax = b$, where

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 2 \\ 9 \\ -1 \end{bmatrix}$$

Since the LU factorization of A is given, we need only:

- Solve $Ly = b$, or
$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & -2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 9 \\ -1 \end{bmatrix}$$
Using forward substitution, we obtain
$$y_1 = 2$$
$$3y_1 + y_2 = 9 \Rightarrow y_2 = 9 - 3(2) = 3$$
$$2y_1 - 2y_2 + y_3 = -1 \Rightarrow y_3 = -1 - 2(2) + 2(3) = 1$$
- Solve $Ux = y$, or
$$\begin{bmatrix} 1 & 2 & -1 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$
Using backward substitution, we obtain
$$x_3 = 1.$$
$$2x_2 - x_3 = 3 \Rightarrow x_2 = (3 + 1)/2 = 2.$$
$$x_1 + 2x_2 - x_3 = 2 \Rightarrow x_1 = 2 - 2(2) + 1 = -1.$$
Therefore, the solution of $Ax = b$ is given by
$$x = \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}.$$

[1] From Section 1.2, if $A = LU$, then $a_{i,j}$ is obtained by multiplying the ith row of L times the jth column of U.

Problem 3.4.1. *Find the LU factorization of each of the following matrices:*

$$A = \begin{bmatrix} 4 & 2 & 1 & 0 \\ -4 & -6 & 1 & 3 \\ 8 & 16 & -3 & -4 \\ 20 & 10 & 4 & -3 \end{bmatrix}, \qquad B = \begin{bmatrix} 3 & 6 & 1 & 2 \\ -6 & -13 & 0 & 1 \\ 1 & 2 & 1 & 1 \\ -3 & -8 & 1 & 12 \end{bmatrix}$$

Problem 3.4.2. *Suppose the LU factorization of a matrix A is given by:*

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -3 & 2 & -1 \\ 0 & -2 & 5 \\ 0 & 0 & -2 \end{bmatrix}$$

For $b = \begin{bmatrix} -1 \\ -7 \\ -6 \end{bmatrix}$, *solve* $Ax = b$.

Problem 3.4.3. *Suppose*

$$A = \begin{bmatrix} 1 & 3 & -4 \\ 0 & -1 & 5 \\ 2 & 0 & 4 \end{bmatrix} \quad \textit{and} \quad b = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}.$$

(a) Use Gaussian elimination without row interchanges to find the factorization $A = LU$.

(b) Use the factorization of A to solve $Ax = b$.

Problem 3.4.4. *Suppose*

$$A = \begin{bmatrix} 4 & 8 & 12 & -8 \\ -3 & -1 & 1 & -4 \\ 1 & 2 & -3 & 4 \\ 2 & 3 & 2 & 1 \end{bmatrix} \quad \textit{and} \quad b = \begin{bmatrix} 3 \\ 60 \\ 1 \\ 5 \end{bmatrix}.$$

(a) Use Gaussian elimination without row interchanges to find the factorization $A = LU$.

(b) Use the factorization of A to solve $Ax = b$.

3.4.2 $PA = LU$ Factorization

The practical implementation of Gaussian elimination uses partial pivoting to determine if row interchanges are needed. We show that this results in a modification of the LU factorization. Row interchanges can by represented mathematically as multiplication by a *permutation matrix*, obtained by interchanging rows of the identity matrix.

Example 3.4.3. Consider the matrix P obtained by switching the first and third rows of a 4×4 identity matrix:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{matrix} \text{switch} \\ \longrightarrow \\ \text{rows 1 and 3} \end{matrix} \quad P = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiplying any 4×4 matrix on the left by P switches its first and third rows. For example,

$$PA = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & -3 & 0 & 1 \\ 5 & 2 & -4 & 7 \\ 1 & 1 & -1 & -1 \\ 0 & 3 & 8 & -6 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 5 & 2 & -4 & 7 \\ 2 & -3 & 0 & 1 \\ 0 & 3 & 8 & -6 \end{bmatrix}$$

Suppose we apply Gaussian elimination with partial pivoting by rows to reduce A to upper triangular form. Then the matrix factorization computed is not an LU factorization of A, but rather an LU factorization of a *permuted* version of A. That is,

$$PA = LU$$

where P is a permutation matrix representing *all* row interchanges in the order that they are applied. The existence of this factorization is typically proved in advanced treatments of numerical linear algebra, and is therefore not discussed here. However, finding P, L and U is not difficult. In general, we proceed as for the LU factorization, but we keep track of the row interchanges as follows:

- Each time we switch rows of A, we switch corresponding multipliers in L. For example, if at stage 3 we switch rows 3 and 5, then we must also switch the previously computed multipliers $m_{3,k}$ and $m_{5,k}$, $k = 1, 2$.
- Begin with $P = I$. Each time we switch rows of A, we switch corresponding rows of P.

It is possible to implement the computation of the factorization without explicitly constructing P. For example, an index of "pointers" to rows can be used.

If we can compute the factorization $PA = LU$, then

$$Ax = b \Rightarrow PAx = Pb \Rightarrow LUx = Pb \Rightarrow Ly = Pb, \text{ where } Ux = y.$$

Therefore, to solve $Ax = b$:

- Compute the factorization $PA = LU$.
- Permute entries of b to obtain $d = Pb$.
- Solve $Ly = d$ using forward substitution.
- Solve $Ux = y$ using backward substitution.

Example 3.4.4. Consider the matrix

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

To find the $PA = LU$ factorization, we proceed as follows:

A	P	multipliers
$\begin{bmatrix} 1 & 2 & 4 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	nothing yet
$\downarrow$	$\downarrow$	$\downarrow$
$\begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 4 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$	nothing yet
$\downarrow$	$\downarrow$	$\downarrow$
$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{3}{7} & \frac{6}{7} \\ 0 & \frac{6}{7} & \frac{19}{7} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$	$m_{21} = \frac{4}{7} \quad m_{31} = \frac{1}{7}$
$\downarrow$	$\downarrow$	$\downarrow$
$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & \frac{3}{7} & \frac{6}{7} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$	$m_{21} = \frac{1}{7} \quad m_{31} = \frac{4}{7}$
$\downarrow$	$\downarrow$	$\downarrow$
$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & 0 & -\frac{1}{2} \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$	$m_{21} = \frac{1}{7} \quad m_{31} = \frac{4}{7} \quad m_{32} = \frac{1}{2}$

From the information in the final step of the process, we obtain the $PA = LU$ factorization, where

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{7} & 1 & 0 \\ \frac{4}{7} & \frac{1}{2} & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & 0 & -\frac{1}{2} \end{bmatrix}$$

Example 3.4.5. Use the $PA = LU$ factorization of Example 3.4.4 to solve $Ax = b$, where

$$b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Since the $PA = LU$ factorization is given, we need only:

- Obtain $d = Pb = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$
- Solve $Ly = d$, or

$$\begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{7} & 1 & 0 \\ \frac{4}{7} & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}$$

  Using forward substitution, we obtain

$y_1 = 3$
$\frac{1}{7}y_1 + y_2 = 1 \Rightarrow y_2 = 1 - \frac{1}{7}(3) = \frac{4}{7}$
$\frac{4}{7}y_1 + \frac{1}{2}y_2 + y_3 = 2 \Rightarrow y_3 = 2 - \frac{4}{7}(3) - \frac{1}{2}(\frac{4}{7}) = 0$

- Solve $Ux = y$, or

$$\begin{bmatrix} 7 & 8 & 9 \\ 0 & \frac{6}{7} & \frac{19}{7} \\ 0 & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ \frac{4}{7} \\ 0 \end{bmatrix}$$

Using backward substitution, we obtain

$-\frac{1}{2}x_3 = 0 \Rightarrow x_3 = 0.$
$\frac{6}{7}x_2 + \frac{19}{7}x_3 = \frac{4}{7} \Rightarrow x_2 = \frac{7}{6}(\frac{4}{7} - \frac{19}{7}(0)) = \frac{2}{3}.$
$7x_1 + 8x_2 + 9x_3 = 3 \Rightarrow x_1 = \frac{1}{7}(3 - 8(\frac{2}{3}) - 9(0)) = -\frac{1}{3}.$

Therefore, the solution of $Ax = b$ is given by

$$x = \begin{bmatrix} -\frac{1}{3} \\ \frac{2}{3} \\ 0 \end{bmatrix}.$$

Problem 3.4.5. *Use Gaussian elimination with partial pivoting to find the $PA = LU$ factorization of the matrices:*

$$A = \begin{bmatrix} 1 & 3 & -4 \\ 0 & -1 & 5 \\ 2 & 0 & 4 \end{bmatrix}, \; B = \begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 2 \\ -3 & -4 & -11 \end{bmatrix}, \; C = \begin{bmatrix} 2 & -1 & 0 & 3 \\ 0 & -\frac{3}{4} & \frac{1}{2} & 4 \\ 1 & 1 & 1 & -\frac{1}{2} \\ 2 & -\frac{5}{2} & 1 & 13 \end{bmatrix}$$

Problem 3.4.6. *Suppose that* $b = \begin{bmatrix} 3 \\ 60 \\ 1 \\ 5 \end{bmatrix}$, *and suppose that Gaussian elimination with partial pivoting has been used on a matrix A to obtain its $PA = LU$ factorization, where*

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3/4 & 1 & 0 & 0 \\ 1/4 & 0 & 1 & 0 \\ 1/2 & -1/5 & 1/3 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 4 & 8 & 12 & -8 \\ 0 & 5 & 10 & -10 \\ 0 & 0 & -6 & 6 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Use this factorization (do not compute the matrix A) to solve $Ax = b$.

3.5 The Accuracy of Computed Solutions

Methods for determining the accuracy of the computed solution of a linear system are discussed in more advanced courses in numerical linear algebra. However, the following material qualitatively describes some of the factors affecting the accuracy.

Consider a nonsingular linear system $Ax = b$ of order n. The solution process involves two steps: GEPP is used to transform $Ax = b$ into an upper triangular linear system $Ux = c$, then $Ux = c$ is solved by backward substitution (say, row–oriented). We use the phrase "the computed solution" to refer to the solution obtained when these two steps are performed using floating–point arithmetic.

Generally, the solution of the upper triangular linear system $Ux = c$ is determined accurately, so we focus on comparing the solutions of $Ax = b$ and $Ux = c$.

The key to comparing the solutions of $Ax = b$ and $Ux = c$ is to reverse the steps used in GEPP. Specifically, consider the sequence of elementary operations used to transform $Ax = b$ into $Ux = c$. Using exact arithmetic, apply the inverse of each of these elementary operations, in the reverse order, to $Ux = c$. The result is a linear system $A'x = b'$ and, because exact arithmetic was used, it has the same solution as $Ux = c$. One of the nice properties of GEPP is that, generally, A' is "close to" A and b' is "close to" b. This observation is described by saying that GEPP is generally backward **stable**. When GEPP is backward stable, the computed solution of $Ax = b$ is the exact solution of $A'x = b'$ where A' is close to A and b' is close to b.

The next step is to consider what happens to the solution of $Ax = b$ when A and b are changed a little. Linear systems can be placed into two categories with a "grey" area between. One category, consisting of so-called **well–conditioned** linear systems, has the property that *all small changes* in the matrix A and the right hand side b lead to a small change in the solution x of the linear system $Ax = b$. An **ill–conditioned** linear system has the property that *some small changes* in the matrix A and/or the right hand side b lead to a large change in the solution x of the linear system $Ax = b$.

What determines whether a linear system is well–conditioned or ill–conditioned? An important factor is the distance from the matrix A to the "closest" singular matrix. In particular, *the linear system $Ax = b$ is ill–conditioned if the matrix A is close to a singular matrix.*

How do the concepts of well–conditioned and ill–conditioned linear systems apply to the computed solution of $Ax = b$? *Backward error analysis* shows that the approximate solution computed by GEPP of the linear system $Ax = b$ is the exact solution of a related linear system $A'x = b'$. Usually, A' is close to A and b' is close to b. In this case, if $Ax = b$ is well–conditioned, it follows that the computed solution is accurate. On the other hand, if $Ax = b$ is ill–conditioned, even if A' is close to A and b' is close to b, these small differences *may* lead to a large difference between the solution of $Ax = b$ and the solution of $A'x = b'$, and the computed solution *may not* be accurate. In summary, suppose GEPP, when applied to this linear system, is backward stable. If the linear system is well–conditioned, then the computed solution is accurate. On the other hand, if the linear system is ill–conditioned, then the computed solution *may not* be accurate.

A measure of the conditioning of a linear system is the *condition number* of the matrix A of the system. This measure, which is defined more precisely in advanced numerical analysis textbooks, is the product of the "size" of the matrix A and the "size" of its inverse A^{-1}. It is always no smaller than one; it is one for identity matrices. In general the larger the condition number, the more ill–conditioned the matrix and the more likely that the solution of the linear system is inaccurate. To be more precise, if the condition number of A is about 10^p and the machine epsilon, ϵ, is about 10^{-s} then the solution of the linear system $Ax = b$ *may* have no more than about $s - p$ decimal digits accurate. Recall that in SP arithmetic, $s \approx 7$, and in DP arithmetic, $s \approx 16$. Schemes for inexpensively estimating the condition number of any matrix A are available in most high quality numerical software libraries; see Section 3.6.

We conclude by describing two ways of *how not* to attempt to estimate the accuracy of a computed solution. We know that $\det(A) = 0$ when the linear system $Ax = b$ is singular. So, we might assume that the magnitude of $\det(A)$ might be a good indicator of how close the matrix A is to a singular matrix. Unfortunately, this is not always the case. Consider, for example, the two linear systems:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 1.0 \\ 1.5 \end{bmatrix}$$

where the second is obtained from the first by multiplying each of its equations by 0.5. The determinant of the coefficient matrix of the first linear system is 1 and for the second linear system it is 0.125. So, by comparing the magnitude of these determinants, we might believe that GEPP would produce a less accurate solution of the second linear system. However, GEPP produces identical solutions for both linear systems because each linear system is diagonal, so no exchanges or eliminations need be performed. So, this simple example shows that *the magnitude of the determinant is not necessarily a good indicator of how close a coefficient matrix is to the nearest singular matrix.*

Another plausible test is to check how well the approximate solution x determined by GEPP satisfies the original linear system $Ax = b$. Define the *residuals* $r_1, r_2, \cdots, r_n$ associated with the approximate solution $x_1, x_2, \cdots, x_n$ as

$$\begin{aligned} r_1 &= b_1 - a_{1,1}x_1 - a_{1,2}x_2 - \cdots - a_{1,n}x_n \\ r_2 &= b_2 - a_{2,1}x_1 - a_{2,2}x_2 - \cdots - a_{2,n}x_n \\ &\vdots \\ r_n &= b_n - a_{n,1}x_1 - a_{n,2}x_2 - \cdots - a_{n,n}x_n \end{aligned}$$

(Equivalently, in matrix–vector notation we define the residual vector $r = b - Ax$.) Observe that $r_k = 0$ when the k^{th} equation is satisfied exactly, so the magnitudes of the residuals indicate by how much the approximate solution x fails to satisfy the linear system $Ax = b$.

Example 3.5.1. (Watkins) Consider the linear system of equations of order 2:

$$\begin{aligned} 1000x_1 + 999x_2 &= 1999 \\ 999x_1 + 998x_2 &= 1997 \end{aligned}$$

with solution $x_1 = x_2 = 1$ for which $r_1 = r_2 = 0.0$. For the nearby approximate solution $x_1 = 1.01$, $x_2 = 0.99$ the residuals $r_1 = 0.01$, $r_2 = 0.01$, are a predictable size. But, the inaccurate approximate solution $x_1 = 20.97$, $x_2 = -18.99$ gives residuals, $r_1 = 0.01$, $r_2 = -0.01$, also of small magnitude compared to the entries of the approximate solution, to the coefficient matrix or to the right hand side. Indeed, they are of the same size as for the approximate solution $x_1 = 1.01$, $x_2 = 0.99$.

So, inaccurate solutions can give residuals of small magnitude. Indeed, GEPP generally returns an approximate solution whose associated residuals are of small magnitude, even when the solution is inaccurate! So, *small magnitudes of residuals associated with an approximate solution are not a good indicator of the accuracy of that solution.* In contrast, large residuals always correspond to relatively large errors.

By rearranging the equations for the residuals, they may be viewed as perturbations of the right hand side. If the residuals have small magnitude then they are small perturbations of the right hand side. In this case, if the solution corresponding to these residuals is very different from the exact solution (which has zero residuals) then the linear system must be ill–conditioned, because small changes in the right hand side have led to large changes in the solution.

Problem 3.5.1. *Consider a linear system $Ax = b$. When this linear system is placed into the computer's memory, say by reading the coefficient matrix and right hand side from a data file, the entries of A and b must be rounded to floating–point numbers. If the linear system is well–conditioned, and the solution of this "rounded" linear system is computed exactly, will this solution be accurate? Answer the same question but assuming that the linear system $Ax = b$ is ill–conditioned.*

Problem 3.5.2. *Consider the linear system of equation of order 2:*

$$\begin{aligned} 0.780x_1 + 0.563x_2 &= 0.217 \\ 0.913x_1 + 0.659x_2 &= 0.254 \end{aligned}$$

with exact solution $x_1 = 1$, $x_2 = -1$. Consider two approximate solutions: first $x_1 = 0.999$, $x_2 = -1.001$, and second $x_1 = 0.341$, $x_2 = -0.087$. Compute the residuals for these approximate solutions. Is the accuracy of the approximate solutions reflected in the size of the residuals? Is the linear system ill–conditioned?

3.6 Matlab Notes

Software is available for solving linear systems where the coefficient matrices have a variety of structures and properties. We restrict our discussion to "dense" systems of linear equations. (A "dense" system is one where all the coefficients are treated as non-zero values. So, the matrix is considered to have no special structure.) Most of today's best software for solving "dense" systems of linear equations, including that found in MATLAB, was developed in the *LAPACK* project. We describe the main tools (i.e., the backslash operator and the `linsolve` function) provided by MATLAB for solving dense linear systems. First however we develop MATLAB implementations of some of the algorithms discussed in this chapter. These examples build on the introduction in Chapter 1, and are designed to introduce useful MATLAB commands and to teach proper MATLAB programming techniques.

3.6.1 Diagonal Linear Systems

Consider a simple diagonal linear system

$$\begin{array}{c} a_{11}x_1 = b_1 \\ a_{22}x_2 = b_2 \\ \vdots \\ a_{nn}x_n = b_n \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

If the diagonal entries, a_{ii}, are all nonzero, it is trivial to solve for x_i:

$$\begin{array}{c} x_1 = b_1/a_{11} \\ x_2 = b_2/a_{22} \\ \vdots \\ x_n = b_n/a_{nn} \end{array}$$

To write a MATLAB function to solve a diagonal system, we must decide what quantities should be specified as input, and what as output. For example, we could input the matrix A and right hand side vector b, and output the solution of $Ax = b$, as in the code:

```
function x = DiagSolve1(A,b)
%
%      x = DiagSolve1(A, b);
%
% Solve Ax=b, where A is an n-by-n diagonal matrix.
%
n = length(b); x = zeros(n,1);
for i = 1:n
  if A(i,i) == 0
    error('Input matrix is singular')
  end
  x(i) = b(i) / A(i,i);
end
```

We use the MATLAB function `length` to determine the dimension of the linear system, assumed the same as the length of the right hand side vector, and we use the `error` function to print an error message in the command window, and terminate the computation, if the matrix is singular. We can shorten this code, and make it more efficient by using array operations:

```
function x = DiagSolve2(A, b)
%
%     x = DiagSolve2(A, b);
%
% Solve Ax=b, where A is an n-by-n diagonal matrix.
%
d = diag(A);
if any(d == 0)
  error('Input matrix is singular')
end
x = b ./ d;
```

In `DiagSolve2`, we use MATLAB's `diag` function to extract the diagonal entries of `A`, and store them in a column vector `d`. If there is at least one 0 entry in the vector `d`, then `any(d == 0)` returns true, otherwise it returns false. If all diagonal entries are nonzero, the solution is computed using the element-wise division operation, `./` In most cases, if we know the matrix is diagonal, then we can substantially reduce memory requirements by using only a single vector (not a matrix) to store the diagonal elements, as follows:

```
function x = DiagSolve3(d, b)
%
%     x = DiagSolve3(d, b);
%
% Solve Ax=b, where A is an n-by-n diagonal matrix.
%
% Input: d = vector containing diagonal entries of A
%        b = right hand side vector
%
if any(d == 0)
  error('Diagonal matrix defined by input is singular')
end
x = b ./ d;
```

In the algorithms `DiagSolve2` and `DiagSolve3` we do not check if `length(b)` is equal to `length(d)`. If they are not equal, MATLAB will report an array dimension error.

Problem 3.6.1. *Implement the functions* `DiagSolve1`*,* `DiagSolve2`*, and* `DiagSolve3`*, and use them to solve the linear system in Fig. 3.2(a).*

Problem 3.6.2. *Consider the* MATLAB *commands:*

```
n = 200:200:1000;, t = zeros(length(n), 3);
for i = 1:length(n)
  d = rand(n(i),1);, A = diag(d);, x = ones(n(i),1);, b = A*x;
  tic, x1 = DiagSolve1(A,b);, t(i,1) = toc;
  tic, x2 = DiagSolve2(A,b);, t(i,2) = toc;
  tic, x3 = DiagSolve3(d,b);, t(i,3) = toc;
end
disp('-------------------------------------------')
disp('       Timings for DiagSolve functions')
disp('  n     DiagSolve1   DiagSolve2   DiagSolve3')
disp('-------------------------------------------')
```

```
for i = 1:length(n)
  disp(sprintf('%4d    %9.3e    %9.3e    %9.3e',  n(i), t(i,1), t(i,2), t(i,3)))
end
```

Using the MATLAB `help` *and/or* `doc` *commands write a brief explanation of what happens when these commands are executed. Write a script M–file implementing the commands and run the script. Check for consistency by running the script sufficient times so that you have confidence in your results. Describe what you observe from the computed results.*

3.6.2 Triangular Linear Systems

We describe MATLAB implementations of the forward substitution algorithms for lower triangular linear systems. Implementations of backward substitution are left as exercises.

Consider the lower triangular linear system

$$\begin{array}{ll} a_{11}x_1 & = b_1 \\ a_{21}x_1 + a_{22}x_2 & = b_2 \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = b_n \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

We first develop a MATLAB implementation to solve this lower triangular system using the pseudocode for the row-oriented version of forward substitution given in Fig. 3.5:

```
function x = LowerSolve0(A, b)
%
%     x = LowerSolve0(A, b);
%
% Solve Ax=b, where A is an n-by-n lower triangular matrix,
% using row-oriented forward substitution.
%
n = length(b);, x = zeros(n,1);
for i = 1:n
  for j = 1:i-1
    b(i) = b(i) - A(i,j)*x(j);
  end
  x(i) = b(i) / A(i,i);
end
```

This implementation can be improved in several ways. As with the diagonal solve functions, we should include a statement that checks to see if `A(i,i)` is zero. Also, the innermost loop can be replaced with a single MATLAB array operation. Observe that we can write the algorithm as:

for $i = 1 : n$
  $x_i = (b_i - (a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,i-1}x_{i-1}))\,/a_{ii}$
end

Using linear algebra notation, we can write this as:

for $i = 1 : n$

$$x_i = \left(b_i - \begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{i,i-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{i-1} \end{bmatrix} \right) \Big/ a_{ii}$$

end

Recall that, in MATLAB, we can specify entries in a matrix or vector using colon notation. That is,

$$x(1:i-1) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{i-1} \end{bmatrix} \quad \text{and} \quad A(i,1:i-1) = \begin{bmatrix} a_{i,1} & a_{i,2} & \cdots & a_{i,i-1} \end{bmatrix}.$$

So, using MATLAB notation, the algorithm for row-oriented forward substitution can be written:

for $i = 1 : n$
  $x(i) = (b(i) - A(i,1:i-1) * x(1:i-1))/\, A(i,i)$
end

When $i = 1$, MATLAB considers $A(i,1:i-1)$ and $x(1:i-1)$ to be "empty" matrices, and the computation $A(i,1:i-1) * x(1:i-1)$ gives 0. Thus, the algorithm simply computes $x(1) = b(1)/A(1,1)$, as it should. To summarize, a MATLAB function to solve a lower triangular system using row oriented forward substitution could be written:

```
function x = LowerSolve1(A, b)
%
%     x = LowerSolve1(A, b);
%
% Solve Ax=b, where A is an n-by-n lower triangular matrix,
% using row-oriented forward substitution.
%
if any(diag(A) == 0)
  error('Input matrix is singular')
end
n = length(b);, x = zeros(n,1);
for i = 1:n
  x(i) = (b(i) - A(i,1:i-1)*x(1:i-1)) / A(i,i);
end
```

Implementation of column-oriented forward substitution is similar. However, because x_j is computed before b_i is updated, it is not possible to combine the two steps, as in the function `LowerSolve1`. A MATLAB implementation of column-oriented forward substitution could be written:

```
function x = LowerSolve2(A, b)
%
%     x = LowerSolve2(A, b);
%
% Solve Ax=b, where A is an n-by-n lower triangular matrix,
% using column-oriented forward substitution.
%
if any(diag(A) == 0)
  error('Input matrix is singular')
end
n = length(b);, x = zeros(n,1);
for j = 1:n
  x(j) = b(j) / A(j,j);
  b(j+1:n) = b(j+1:n) - A(j+1:n,j)*x(j);
end
```

What is computed by the statement `b(j+1:n) = b(j+1:n) - A(j+1:n,j)*x(j)` when $j = n$? Because there are only n entries in the vectors, MATLAB recognizes `b(n+1:n)` and `A(n+1:n,n)` to be "empty matrices", and skips this part of the computation.

Problem 3.6.3. *Implement the functions* `LowerSolve1` *and* `LowerSolve2`*, and use them to solve the linear system in Fig. 3.2(b).*

Problem 3.6.4. *Consider the following* MATLAB *commands:*

```
n = 200:200:1000;, t = zeros(length(n), 2);
for i = 1:length(n)
  A = tril(rand(n(i)));, x = ones(n(i),1);, b = A*x;
  tic, x1 = LowerSolve1(A,b);, t(i,1) = toc;
  tic, x2 = LowerSolve2(A,b);, t(i,2) = toc;
end
disp('-------------------------------------')
disp('  Timings for LowerSolve functions')
disp('  n     LowerSolve1   LowerSolve2  ')
disp('-------------------------------------')
for i = 1:length(n)
  disp(sprintf('%4d     %9.3e     %9.3e',  n(i), t(i,1), t(i,2)))
end
```

Using the MATLAB `help` *and/or* `doc` *commands write a brief explanation of what happens when these commands are executed. Write a script M–file implementing the commands, run the script, and describe what you observe from the computed results.*

Problem 3.6.5. *Write a* MATLAB *function that solves an upper triangular linear system using row-oriented backward substitution. Test your code using the linear system in Fig. 3.2(c).*

Problem 3.6.6. *Write a* MATLAB *function that solves an upper triangular linear system using column-oriented backward substitution. Test your code using the linear system in Fig. 3.2(c).*

Problem 3.6.7. *Write a script M–file that compares timings using row-oriented and column-oriented backward substitution to solve an upper triangular linear systems. Use the code in Problem 3.6.4 as a template.*

3.6.3 Gaussian Elimination

Next, we consider a MATLAB implementation of Gaussian elimination, using the pseudocode given in Fig. 3.7. First, we explain how to implement the various steps during the k^{th} stage of the algorithm.

- Consider the search for the largest entry in the pivot column. Instead of using a loop, we can use the built-in `max` function. For example, if we use the command

  ```
  [piv, i] = max(abs(A(k:n,k)));
  ```

  then, after execution of this command, `piv` contains the largest entry (in magnitude) in the vector `A(k:n,k)`, and `i` is its location in the vector. Note that if `i = 1`, then the index `p` in Fig. 3.7 is `p = k` and, in general,

  ```
  p = i + k - 1;
  ```

- Once the pivot row, `p`, is known, then the p^{th} row of A is switched with the k^{th} row, and the corresponding entries of b must be switched. This may be implemented using MATLAB's array indexing capabilities:

```
A([k,p],k:n) = A([p,k],k:n);, b([k,p]) = b([p,k]);
```

We might visualize these two statements as

$$\begin{bmatrix} a_{kk} & a_{k,k+1} & \cdots & a_{kn} \\ a_{pk} & a_{p,k+1} & \cdots & a_{pn} \end{bmatrix} := \begin{bmatrix} a_{pk} & a_{p,k+1} & \cdots & a_{pn} \\ a_{kk} & a_{k,k+1} & \cdots & a_{kn} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} b_k \\ b_p \end{bmatrix} := \begin{bmatrix} b_p \\ b_k \end{bmatrix}$$

That is, `b([k,p]) = b([p,k])` instructs MATLAB to replace `b(k)` with the original `b(p)`, and to replace `b(p)` with original `b(k)`. MATLAB's internal memory manager makes appropriate copies of the data so that the original entries are not overwritten before the assignments are completed. Similarly, `A([k,p],k:n) = A([p,k],k:n)` instructs MATLAB to replace the rows specified on the left with the original rows specified on the right.

- The elimination step is straightforward; compute the multipliers, which we store in the strictly lower triangular part of A:

  ```
  A(k+1:n,k) = A(k+1:n,k) / A(k,k);
  ```

  and use array operations to perform the elimination:

  ```
  for i = k+1:n
    A(i,k+1:n) = A(i,k+1:n) - A(i,k)*A(k,k+1:n);, b(i) = b(i) - A(i,k)*b(k);
  end
  ```

- Using array operations, the backward substitution step can be implemented:

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

  The statement `for i = n:-1:1` indicates that the loop runs over values $i = n, n-1, \ldots, 1$; that is, it runs from `i=n` in steps of `-1` until it reaches `i=1`.

- How do we implement a check for singularity? Due to roundoff errors, it is unlikely that any pivots a_{kk} will be exactly zero, and so a statement `if A(k,k) == 0` will usually miss detecting singularity. An alternative is to check if a_{kk} is small in magnitude compared to, say, the largest entry in magnitude in the matrix A:

  ```
  if abs(A(k,k)) < tol
  ```

  where `tol` is computed in an initialization step:

  ```
  tol = eps * max(abs(A(:)));
  ```

  Here, the command `A(:)` reshapes the matrix A into one long vector, and `max(abs(A(:)))` finds the largest entry. Of course, this test also traps matrices which are close to but not exactly singular, which is usually reasonable as by definition they tend to be ill–conditioned.

- Finally, certain other initializations are needed: the dimension, n, space for the solution vector, and for the multipliers. We use the `length`, `zeros`, and `eye` functions:

  ```
  n = length(b);, x = zeros(n,1);, m = eye(n);
  ```

```
function x = gepp(A, b)
%
% Solves Ax=b using Gaussian elimination with partial pivoting
% by rows for size. Initializations:
%
n = length(b);, m = eye(n);, x = zeros(n,1);
tol = eps*max(A(:));
%
% Loop for stages k = 1, 2, ..., n-1
%
for k = 1:n-1
  %
  % Search for pivot entry:
  %
  [piv, psub] = max(abs(A(k:n,k)));, p = psub + k - 1;
  %
  % Exchange current row, k, with pivot row, p:
  %
  A([k,p],k:n) = A([p,k],k:n);, b([k,p]) = b([p,k]);
  %
  % Check to see if A is singular:
  %
  if abs(A(k,k)) < tol
    error('Linear system appears to be singular')
  end
  %
  % Perform the elimination step - row-oriented:
  %
  A(k+1:n,k) = A(k+1:n,k) / A(k,k);
  for i = k+1:n
    A(i,k+1:n) = A(i,k+1:n) - A(i,k)*A(k,k+1:n);,  b(i) = b(i) - A(i,k)*b(k);
  end
end
%
% Check to see if A is singular:
%
if abs(A(n,n)) < tol
  error('Linear system appears to be singular')
end
%
% Solve the upper triangular system by row-oriented backward substitution:
%
for i = n:-1:1
  x(i) = (b(i) - A(i,i+1:n)*x(i+1:n)) / A(i,i);
end
```

The function gepp above implements a function that uses GEPP to solve $Ax = b$.

Problem 3.6.8. *Implement the function* gepp*, and use it to solve the linear systems given in problems 3.3.2 and 3.3.3.*

Problem 3.6.9. *Test* `gepp` *using the linear system*

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

Explain your results.

Problem 3.6.10. *Modify* `gepp`*, replacing the statements* `if abs(A(k,k)) < tol` *and* `if abs(A(n,n)) < tol` *with, respectively,* `if A(k,k) == 0` *and* `if A(n,n) == 0`*. Solve the linear system given in Problem 3.6.9. Why are your results different?*

Problem 3.6.11. *The built-in* MATLAB *function* `hilb` *constructs the Hilbert matrix, whose* (i, j) *entry is* $1/(i + j - 1)$*. Write a script M–file that constructs a series of test problems of the form:*

```
A = hilb(n);, x_true = ones(n,1);, b = A*x_true;
```

The script should use `gepp` *to solve the resulting linear systems, and print a table of results with the following information:*

```
-----------------------------------------------------
  n        error         residual    condition number
-----------------------------------------------------
```

where

- `error` = *relative error* = `norm(x_true - x)/norm(x_true)`
- `residual` = *relative residual error* = `norm(b - A*x)/norm(b)`
- `condition number` = *measure of conditioning* = `cond(A)`

Print a table for $n = 5, 6, \ldots, 13$*. Are the computed residual errors small? What about the relative errors? Is the size of the residual error related to the condition number? What about the relative error? Now run the script with* $n \geq 14$*. What do you observe?*

Problem 3.6.12. *Rewrite the function* `gepp` *so that it uses no array operations. Compare the efficiency (for example, using* `tic` *and* `toc`*) of your function with* `gepp` *on matrices of dimensions* $n = 100, 200, \ldots, 1000$*.*

3.6.4 Built-in Matlab Tools for Linear Systems

An advantage of using a powerful scientific computing environment like MATLAB is that we do not need to write our own implementations of standard algorithms, like Gaussian elimination and triangular solves. MATLAB provides two powerful tools for solving linear systems:

- The *backslash* operator: \
  Given a matrix A and vector b, \ can be used to solve $Ax = b$ with the single command:

  ```
  x = A \ b;
  ```

  MATLAB first checks if the matrix A has a special structure, including diagonal, upper triangular, and lower triangular. If a special structure is recognized (for example, upper triangular), then a special method (for example, column-oriented backward substitution) is used to solve $Ax = b$. If a special structure is not recognized then Gaussian elimination with partial pivoting is used to solve $Ax = b$. During the process of solving the linear system, MATLAB estimates the *reciprocal* of the condition number of A. If A is ill–conditioned, then a warning message is printed along with the estimate, `RCOND`, of the reciprocal of the condition number.

- The function `linsolve`.
  If we know a-priori that A has a special structure recognizable by MATLAB, then we can improve efficiency by avoiding checks on the matrix, and skipping directly to the special solver. This can be especially helpful if, for example, it is known that A is upper triangular. A-priori information on the structure of A can be provided to MATLAB using the `linsolve` function. For more information, see `help linsolve` or `doc linsolve`.

Because the backslash operator is so powerful, we use it almost exclusively to solve general linear systems.

We can compute explicitly the $PA = LU$ factorization using the `lu` function:

```
[L, U, P] = lu(A)
```

Given this factorization, and a vector b, we could solve the $Ax = b$ using the statements:

```
d = P * b;, y = L \ d;, x = U \ y;
```

These statements could be combined into one instruction:

```
x = U \ ( L \ (P * b) );
```

Thus, given a matrix A and vector b we could solve the linear system $Ax = b$ as follows:

```
[L, U, P] = lu(A);
x = U \ ( L \ (P * b) );
```

Note, MATLAB follows the rules of operator precedence thus backslash and `*` are of equal precedence. With operators of equal precedence MATLAB works from the left. So, without parentheses the instruction

```
x = U \ L \ P * b ;
```

would be interpreted as

```
x = ((U \ L) \ P) * b;
```

The cost and accuracy of this approach is essentially the same as using the backslash operator. So when would we prefer to explicitly compute the $PA = LU$ factorization? One situation is when we need to solve several linear systems with the same coefficient matrix, but different right hand side vectors. For large systems it is far more expensive to compute the $PA = LU$ factorization than it is to use forward and backward substitution to solve corresponding lower and upper triangular systems. Thus, if we can compute the factorization just once, and use it for the various different right hand side vectors, we can make a substantial savings. This is illustrated in problem 3.6.18, which involves a relatively small linear system.

Problem 3.6.13. *Use the backslash operator to solve the systems given in problems 3.3.2 and 3.3.3.*

Problem 3.6.14. *Use the backslash operator to solve the linear system*

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}.$$

Explain your results.

Problem 3.6.15. *Repeat problem 3.6.11 using the backslash operator in addition to* `gepp`.

Problem 3.6.16. *Consider the linear system defined by the following* MATLAB *commands:*

```
A = eye(500) + triu(rand(500));, x = ones(500,1);, b = A * x;
```

Does this matrix have a special structure? Suppose we slightly perturb the entries in A:

```
C = A + eps*rand(500);
```

Does the matrix C have a special structure? Execute the following MATLAB *commands:*

```
tic, x1 = A \ b;, toc
tic, x2 = C \ b;, toc
tic, x3 = triu(C) \ b;, toc
```

Are the solutions `x1`*,* `x2` *and* `x3` *good approximations to the exact solution,* `x = ones(500,1)`*? What do you observe about the time required to solve each of the linear systems?*

Problem 3.6.17. *Use the* MATLAB `lu` *function to compute the $PA = LU$ factorization of each of the matrices given in Problem 3.4.5.*

Problem 3.6.18. *Create a script M–file containing the following* MATLAB *statements:*

```
n = 50;, A = rand(n);
tic
for k = 1:n
  b = rand(n,1);, x = A \ b;
end
toc

tic
[L, U, P] = lu(A);
for k = 1:n
  b = rand(n,1);, x = U \ ( L \ (P * b) );
end
toc
```

The dimension of the problem is set to $n = 50$. Experiment with other values of n, such as $n = 100, 150, 200$. What do you observe?

Chapter 4

Curve Fitting

We consider two commonly used methods for curve fitting, interpolation and least squares. For interpolation, we use first polynomials and later splines. Polynomial interpolation may be implemented using a variety of bases. We consider power series, bases associated with the names of Newton and Lagrange, and finally Chebyshev series. We discuss the error arising directly from polynomial interpolation and the (linear algebra) error arising in the solution of the interpolation equations. This error analysis leads us to consider interpolating using piecewise polynomials, specifically splines. Finally, we introduce least squares curve fitting using simple polynomials and later generalize this approach sufficiently to permit other choices of least squares fitting functions; for example, splines, Chebyshev series, or trigonometric series. In a final section we show how to use the MATLAB functions to compute curve fits using many of the ideas introduced here and, indeed, we also consider for the first time fitting trigonometric series for periodic data; this approach uses the fast Fourier transform.

4.1 Polynomial Interpolation

We can approximate a function $f(x)$ by interpolating to the data $\{(x_i, f_i)\}_{i=0}^{N}$ by another (computable) function $p(x)$. (Here, implicitly we assume that the data is obtained by evaluating the function $f(x)$; that is, we assume that $f_i = f(x_i), i = 0, 1, \cdots, N$.)

Definition 4.1.1. *The function $p(x)$ **interpolates** to the data $\{(x_i, f_i)\}_{i=0}^{N}$ if the equations*

$$p(x_i) = f_i, \quad i = 0, 1, \cdots, N$$

are satisfied.

This system of $N + 1$ equations comprise the *interpolating conditions*. Note, the function $f(x)$ that has been evaluated to compute the data *automatically interpolates to its own data.*

Example 4.1.1. Consider the data: $\left(-\frac{\pi}{2}, -1\right)$, $(\pi, 0)$, $\left(\frac{5\pi}{2}, 1\right)$. This data was obtained by evaluating the function $f(x) = \sin(x)$ at $x_0 = -\frac{\pi}{2}$, $x_1 = \pi$ and $x_2 = \frac{5\pi}{2}$. Thus, $f(x) = \sin(x)$ interpolates to the given data. But the linear polynomial

$$p(x) = -\frac{2}{3} + \frac{2}{3\pi}x$$

also interpolates to the given data because $p(-\frac{\pi}{2}) = -1$, $p(\pi) = 0$, and $p(\frac{5\pi}{2}) = 1$. Similarly the cubic polynomial

$$q(x) = \frac{16}{15\pi}x - \frac{8}{5\pi^2}x^2 + \frac{8}{15\pi^3}x^3$$

interpolates to the given data because $q(-\frac{\pi}{2}) = -1$, $q(\pi) = 0$, and $q(\frac{5\pi}{2}) = 1$. The graphs of $f(x)$, $p(x)$ and $q(x)$, along with the interpolating data, are shown in Fig. 4.1. The interpolating conditions imply that these curves pass through the data points.

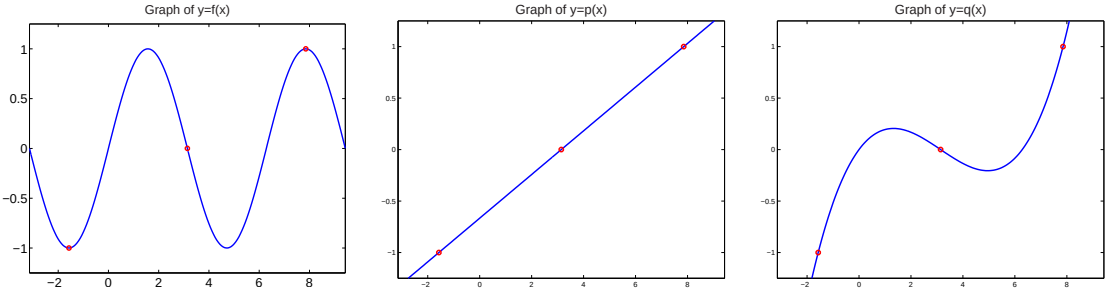

Figure 4.1: Graphs of $f(x) = \sin(x)$, $p(x) = -\frac{2}{3} + \frac{2}{3\pi}x$ and $q(x) = \frac{16}{15\pi}x - \frac{8}{5\pi^2}x^2 + \frac{8}{15\pi^3}x^3$. Each of these functions interpolates to the data $\left(-\frac{\pi}{2}, -1\right)$, $(\pi, 0)$, $\left(\frac{5\pi}{2}, 1\right)$. The data points are indicated by circles on each of the plots.

A common choice for the interpolating function $p(x)$ is a polynomial. Polynomials are chosen because there are efficient methods both for determining and for evaluating them, see, for example, Horner's rule described in Section 6.5. Indeed, they may be evaluated using only adds and multiplies, the most basic computer operations. A polynomial that interpolates to the data is **an interpolating polynomial**. A simple, familiar example of an interpolating polynomial is a straight line, that is a polynomial of degree one, joining two points.

Before we can construct an interpolating polynomial, we need to establish a standard form representation of a polynomial. In addition, we need to define what is meant by the "degree" of a polynomial.

Definition 4.1.2. *A polynomial $p_K(x)$ is of **degree** K if there are constants $c_0, c_1, \cdots, c_K$ for which*

$$p_K(x) = c_0 + c_1 x + \cdots + c_K x^K.$$

*The polynomial $p_K(x)$ is of **exact degree** K if it is of degree K and $c_K \neq 0$.*

Example 4.1.2. $p(x) \equiv 1 + x - 4x^3$ is a polynomial of exact degree 3. It is also a polynomial of degree 4 because we can write $p(x) = 1 + x + 0x^2 - 4x^3 + 0x^4$. Similarly, $p(x)$ is a polynomial of degree $5, 6, 7, \cdots$.

4.1.1 The Power Series Form of The Interpolating Polynomial

Consider first the problem of linear interpolation, that is straight line interpolation. If we have two data points (x_0, f_0) and (x_1, f_1) we can interpolate to this data using the linear polynomial $p_1(x) \equiv a_0 + a_1 x$ by satisfying the pair of linear equations

$$\begin{array}{rcccl} p_1(x_0) & \equiv & a_0 + a_1 x_0 & = & f_0 \\ p_1(x_1) & \equiv & a_0 + a_1 x_1 & = & f_1 \end{array}$$

Solving these equations for the coefficients a_0 and a_1 gives the straight line passing through the two points (x_0, f_0) and (x_1, f_1). These equations have a solution if $x_0 \neq x_1$. If $f_0 = f_1$, the solution is a constant ($a_0 = f_0$ and $a_1 = 0$); that is, it is a polynomial of degree zero,.

Generally, there are an infinite number of polynomials that interpolate to a given set of data. To explain the possibilities we consider the power series form of the complete polynomial (that is, a polynomial where all the powers of x appear)

$$p_M(x) = a_0 + a_1 x + \cdots + a_M x^M$$

of degree M. If the polynomial $p_M(x)$ interpolates to the given data $\{(x_i, f_i)\}_{i=0}^N$, then the interpolating conditions form a linear system of $N+1$ equations

$$\begin{array}{rcccl} p_M(x_0) & \equiv & a_0 + a_1 x_0 + \cdots + a_M x_0^M & = & f_0 \\ p_M(x_1) & \equiv & a_0 + a_1 x_1 + \cdots + a_M x_1^M & = & f_1 \\ & & \vdots & & \\ p_M(x_N) & \equiv & a_0 + a_1 x_N + \cdots + a_M x_N^M & = & f_N \end{array}$$

for the $M+1$ unknown coefficients $a_0, a_1, \ldots, a_M$.

From linear algebra (see Chapter 3), this linear system of equations has a *unique solution* for *every choice* of data values $\{f_i\}_{i=0}^N$ if and only if the system is square (that is, if $M = N$) and it is nonsingular. If $M < N$, there exist choices of the data values $\{f_i\}_{i=0}^N$ for which this linear system has no solution, while for $M > N$ if a solution exists it cannot be unique.

Think of it this way:

1. The complete polynomial $p_M(x)$ of degree M has $M+1$ unknown coefficients,
2. The interpolating conditions comprise $N+1$ equations.
3. So,
   - If $M = N$, there are as many coefficients in the complete polynomial $p_M(x)$ as there are equations obtained from the interpolating conditions,
   - If $M < N$, the number of coefficients is less than the number of data values and we wouldn't have enough coefficients so that we would be able to fit to all the data.
   - If $M > N$, the number of coefficients exceeds the number of data values and we would expect to be able to choose the coefficients in many ways to fit the data.

The square, $M = N$, coefficient matrix of the linear system of interpolating conditions is the Vandermonde matrix

$$V_N \equiv \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^M \\ 1 & x_1 & x_1^2 & \cdots & x_1^M \\ & & \vdots & & \\ 1 & x_N & x_N^2 & \cdots & x_N^M \end{bmatrix}$$

and it can be shown that the determinant of V_N is

$$\det(V_N) = \prod_{i>j} (x_i - x_j)$$

Recall from Problems 3.1.4, 3.2.21, and 3.3.4 that the determinant is a test for singularity, and this is an ideal situation in which to use it. In particular, observe that $\det(V_N) \neq 0$ (that is the matrix V_N is nonsingular) if and only if the nodes $\{x_i\}_{i=0}^N$ are distinct; that is, if and only if $x_i \neq x_j$ whenever $i \neq j$. So, for every choice of data $\{(x_i, f_i)\}_{i=0}^N$, there *exists* a *unique* solution satisfying the interpolation conditions if and only if $M = N$ and the nodes $\{x_i\}_{i=0}^N$ are distinct.

Theorem 4.1.1. (Polynomial Interpolation Uniqueness Theorem) When the nodes $\{x_i\}_{i=0}^N$ are distinct there is a unique polynomial, the *interpolating polynomial* $p_N(x)$, of degree N that interpolates to the data $\{(x_i, f_i)\}_{i=0}^N$.

Though the degree of the interpolating polynomial is N corresponding to the number of data values, its exact degree may be less than N. For example, this happens when the three data points (x_0, f_0), (x_1, f_1) and (x_2, f_2) are collinear; the interpolating polynomial for this data is of degree $N = 2$ but it is of exact degree 1; that is, the coefficient of x^2 turns out to be zero. (In this case, the interpolating polynomial is the straight line on which the data lies.)

Example 4.1.3. Determine the power series form of the quadratic interpolating polynomial $p_2(x) = a_0 + a_1 x + a_2 x^2$ to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$. The interpolating conditions, in the order of the data, are

$$\begin{array}{rcl} a_0 + (-1)a_1 + (1)a_2 & = & 0 \\ a_0 & = & 1 \\ a_0 + (1)a_1 + (1)a_2 & = & 3 \end{array}$$

We may solve this linear system of equations for $a_0 = 1$, $a_1 = \frac{3}{2}$ and $a_2 = \frac{1}{2}$. So, in power series form, the interpolating polynomial is $p_2(x) = 1 + \frac{3}{2}x + \frac{1}{2}x^2$.

Of course, polynomials may be written in many different ways, some more appropriate to a given task than others. For example, when interpolating to the data $\{(x_i, f_i)\}_{i=0}^N$, the Vandermonde system determines the values of the coefficients $a_0, a_1, \cdots, a_N$ in the **power series form**. The number of floating–point computations needed by GEPP to solve the interpolating condition equations grows like N^3 with the degree N of the polynomial (see Chapter 3). This cost can be significantly reduced by exploiting properties of the Vandermonde coefficient matrix. Another way to reduce the cost of determining the interpolating polynomial is to change the way the interpolating polynomial is represented. This is explored in the next two subsections.

Problem 4.1.1. *Show that, when $N = 2$,*

$$\det(V_2) = \det\left(\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix}\right) = \prod_{i>j}(x_i - x_j) = \overbrace{(x_1 - x_0)(x_2 - x_0)}^{j=0;\, i=1,2}\overbrace{(x_2 - x_1)}^{j=1;\, i=2}$$

Problem 4.1.2. *$\omega_{N+1}(x) \equiv (x - x_0)(x - x_1)\cdots(x - x_N)$ is a polynomial of exact degree $N + 1$. If $p_N(x)$ interpolates the data $\{(x_i, f_i)\}_{i=0}^N$, verify that, for any choice of polynomial $q(x)$, the polynomial*

$$p(x) = p_N(x) + \omega_{N+1}(x)q(x)$$

interpolates the same data as $p_N(x)$. Hint: Verify that $p(x)$ satisfies the same interpolating conditions as does $p_N(x)$.

Problem 4.1.3. *Find the power series form of the interpolating polynomial to the data $(1, 2)$, $(3, 3)$ and $(5, 4)$. Check that your computed polynomial does interpolate the data. Hint: For these three data points you will need a complete polynomial that has three coefficients; that is, you will need to interpolate with a quadratic polynomial $p_2(x)$.*

Problem 4.1.4. *Let $p_N(x)$ be the unique interpolating polynomial of degree N for the data $\{(x_i, f_i)\}_{i=0}^N$. Estimate the cost of determining the coefficients of the power series form of $p_N(x)$, assuming that you can set up the coefficient matrix at no cost. Just estimate the cost of solving the linear system via Gaussian Elimination.*

4.1.2 The Newton Form of The Interpolating Polynomial

Consider the problem of determining the interpolating quadratic polynomial for the data (x_0, f_0), (x_1, f_1) and (x_2, f_2). Using the data written in this order, the **Newton form** of the quadratic interpolating polynomial is

$$p_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1)$$

To determine these coefficients b_i simply write down the interpolating conditions:

$$\begin{array}{rcll} p_2(x_0) & \equiv & b_0 & = & f_0 \\ p_2(x_1) & \equiv & b_0 + b_1(x_1 - x_0) & = & f_1 \\ p_2(x_2) & \equiv & b_0 + b_1(x_2 - x_0) + b_2(x_2 - x_0)(x_2 - x_1) & = & f_2 \end{array}$$

The coefficient matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & (x_1 - x_0) & 0 \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) \end{bmatrix}$$

for this linear system is lower triangular. When the nodes $\{x_i\}_{i=0}^{2}$ are distinct, the diagonal entries of this lower triangular matrix are nonzero. Consequently, the linear system has a unique solution that may be determined by forward substitution.

Example 4.1.4. Determine the Newton form of the quadratic interpolating polynomial $p_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1)$ to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$. Taking the data points in their order in the data we have $x_0 = -1$, $x_1 = 0$ and $x_2 = 1$. The interpolating conditions are

$$\begin{array}{lcl} b_0 & = & 0 \\ b_0 + (0 - (-1))b_1 & = & 1 \\ b_0 + (1 - (-1))b_1 + (1 - (-1))(1 - 0)b_2 & = & 3 \end{array}$$

and this lower triangular system may be solved to give $b_0 = 0$, $b_1 = 1$ and $b_2 = \frac{1}{2}$. So, the Newton form of the quadratic interpolating polynomial is $p_2(x) = 0 + 1(x + 1) + \frac{1}{2}(x + 1)(x - 0)$. After rearrangement we observe that this is the same polynomial $p_2(x) = 1 + \frac{3}{2}x + \frac{1}{2}x^2$ as the power series form in Example 4.1.3.

Note, the Polynomial Interpolation Uniqueness theorem tells us that the Newton form of the polynomial in Example 4.1.4 and the power series form in Example 4.1.3 must be the same polynomial. However, we do not need to convert between forms to show that two polynomials are the same. We can use the Polynomial Interpolation Uniqueness theorem as follows. If we have two polynomials of degree N and we want to check if they are different representations of the same polynomial we can evaluate each polynomial at any choice of $(N + 1)$ distinct points. If the values of the two polynomials are the same at all $(N+1)$ points, then the two polynomials interpolate each other and so must be the same polynomial.

Example 4.1.5. We show that the polynomials $p_2(x) = 1 + \frac{3}{2}x + \frac{1}{2}x^2$ and $q_2(x) = 0 + 1(x + 1) + \frac{1}{2}(x + 1)(x - 0)$ are the same. The Polynomial Interpolation Uniqueness theorem tells us that since these polynomials are both of degree two we should check their values at three distinct points. We find that $p_2(1) = 3 = q_2(1)$, $p_2(0) = 1 = q_2(0)$ and $p_2(-1) = 0 = q_2(-1)$, and so $p_2(x) \equiv q_2(x)$.

Example 4.1.6. For the data $(1, 2)$, $(3, 3)$ and $(5, 4)$, using the data points in the order given, the Newton form of the interpolating polynomial is

$$p_2(x) = b_0 + b_1(x - 1) + b_2(x - 1)(x - 3)$$

and the interpolating conditions are

$$\begin{array}{lcll} p_2(1) & \equiv & b_0 & = & 2 \\ p_2(3) & \equiv & b_0 + b_1(3 - 1) & = & 3 \\ p_2(5) & \equiv & b_0 + b_1(5 - 1) + b_2(5 - 1)(5 - 3) & = & 4 \end{array}$$

This lower triangular linear system may be solved by forward substitution giving

$$\begin{array}{lcll} b_0 & & & = & 2 \\ b_1 & = & \dfrac{3 - b_0}{(3 - 1)} & = & \dfrac{1}{2} \\ b_2 & = & \dfrac{4 - b_0 - (5 - 1)b_1}{(5 - 1)(5 - 3)} & = & 0 \end{array}$$

Consequently, the Newton form of the interpolating polynomial is

$$p_2(x) = 2 + \frac{1}{2}(x-1) + 0(x-1)(x-3)$$

Generally, this interpolating polynomial is of degree 2, but its exact degree may be less. Here, rearranging into power series form we have $p_2(x) = \frac{3}{2} + \frac{1}{2}x$ and the exact degree is 1; this happens because the data $(1,2)$, $(3,3)$ and $(5,4)$ are collinear.

The Newton form of the interpolating polynomial is easy to evaluate using *nested multiplication* (which is a generalization of Horner's rule, see Chapter 6). Start with the observation that

$$\begin{aligned} p_2(x) &= b_0 + b_1(x-x_0) + b_2(x-x_0)(x-x_1) \\ &= b_0 + (x-x_0)\{b_1 + b_2(x-x_1)\} \end{aligned}$$

To develop an algorithm to evaluate $p_2(x)$ at a value $x = z$ using nested multiplication, it is helpful to visualize the sequence of nested operations as:

$$\begin{aligned} t_2(z) &= b_2 \\ t_1(z) &= b_1 + (z-x_1)t_2(z) \\ p_2(z) = t_0(z) &= b_0 + (z-x_0)t_1(z) \end{aligned}$$

An extension to degree N of this nested multiplication scheme leads to the pseudocode given in Fig. 4.2. This pseudocode evaluates the Newton form of a polynomial of degree N at a point $x = z$ given the values of the nodes x_i and the coefficients b_i.

Evaluating the Newton Form

Input: nodes x_i
coefficients b_i
scalar z

Output: $p = p_N(z)$

$p := b_N$
for $i = N-1$ downto 0 do
  $p := b_i + (z - x_i) * p$
next i

Figure 4.2: Pseudocode to use nested multiplication to evaluate the Newton form of the interpolating polynomial, $p_N(x) = b_0 + b_1(x-x_0) + \cdots + b_{N-1}(x-x_0)\cdots(x-x_{N-1})$, at $x = z$.

How can the triangular system for the coefficients in the Newton form of the interpolating polynomial be systematically formed and solved? First, note that we may form a sequence of interpolating polynomials as in Table 4.1.

Polynomial	The Data
$p_0(x) = b_0$	(x_0, f_0)
$p_1(x) = b_0 + b_1(x-x_0)$	(x_0, f_0), (x_1, f_1)
$p_2(x) = b_0 + b_1(x-x_0) + b_2(x-x_0)(x-x_1)$	(x_0, f_0), (x_1, f_1), (x_2, f_2)

Table 4.1: Building the Newton form of the interpolating polynomial

So, the Newton equations that determine the coefficients b_0, b_1 and b_2 may be written

$$\begin{array}{lcl} b_0 & = & f_0 \\ p_0(x_1) + b_1(x_1 - x_0) & = & f_1 \\ p_1(x_2) + b_2(x_2 - x_0)(x_2 - x_1) & = & f_2 \end{array}$$

and we conclude that we may solve for the coefficients b_i efficiently as follows

$$\begin{aligned} b_0 &= f_0 \\ b_1 &= \frac{f_1 - p_0(x_1)}{(x_1 - x_0)} \\ b_2 &= \frac{f_2 - p_1(x_2)}{(x_2 - x_0)(x_2 - x_1)} \end{aligned}$$

This process clearly deals with any number of data points. So, if we have data $\{(x_i, f_i)\}_{i=0}^{N}$, we may compute the coefficients in the corresponding Newton polynomial

$$p_N(x) = b_0 + b_1(x - x_0) + \cdots + b_N(x - x_0)(x - x_1)\cdots(x - x_{N-1})$$

from evaluating $b_0 = f_0$ followed by

$$b_k = \frac{f_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})}, \quad k = 1, 2, \ldots, N$$

Indeed, we can easily incorporate additional data values. Suppose we have fitted to the data $\{(x_i, f_i)\}_{i=0}^{N}$ using the above formulas and we wish to add the data (x_{N+1}, f_{N+1}). Then we may simply use the formula

$$b_{N+1} = \frac{f_{N+1} - p_N(x_{N+1})}{(x_{N+1} - x_0)(x_{N+1} - x_1)\cdots(x_{N+1} - x_N)}$$

Fig. 4.3 shows pseudocode that can be used to implement this computational sequence for the Newton form of the interpolating polynomial of degree N given the data $\{(x_i, f_i)\}_{i=0}^{N}$.

Constructing the Newton Form

Input: data (x_k, f_k)
Output: coefficients b_k

```
b_0 := f_0
for k = 1 to N
    num := f_k - p_{k-1}(x_k)
    den := 1
    for j = 0 to k - 1
        den := den * (x_k - x_j)
    next j
    b_k := num/den
next k
```

Figure 4.3: Pseudocode to compute the coefficients, b_i, of the Newton form of the interpolating polynomial, $p_N(x) = b_0 + b_1(x - x_0) + \cdots + b_{N-1}(x - x_0)\cdots(x - x_{N-1})$. This pseudocode requires evaluating $p_{i-1}(x_i)$, which can be done by implementing the pseudocode given in Fig. 4.2.

If the interpolating polynomial is to be evaluated at many points, generally it is best first to determine its Newton form and then to use the nested multiplication scheme to evaluate the interpolating polynomial at each desired point.

Problem 4.1.5. *Use the data in Example 4.1.4 in reverse order (that is, use $x_0 = 1$, $x_1 = 0$ and $x_2 = -1$) to build an alternative quadratic Newton interpolating polynomial. Is this the same polynomial that was derived in Example 4.1.4? Why or why not?*

Problem 4.1.6. *Let $p_N(x)$ be the interpolating polynomial for the data $\{(x_i, f_i)\}_{i=0}^{N}$, and let $p_{N+1}(x)$ be the interpolating polynomial for the data $\{(x_i, f_i)\}_{i=0}^{N+1}$. Show that*

$$p_{N+1}(x) = p_N(x) + b_{N+1}(x - x_0)(x - x_1)\cdots(x - x_N)$$

What is the value of b_{N+1}?

Problem 4.1.7. *In Example 4.1.4 we showed how to form the quadratic Newton polynomial $p_2(x) = 0 + 1(x + 1) + \frac{1}{2}(x + 1)(x - 0)$ that interpolates to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$. Starting from this quadratic Newton interpolating polynomial build the cubic Newton interpolating polynomial to the data $(-1, 0)$, $(0, 1)$, $(1, 3)$ and $(2, 4)$.*

Problem 4.1.8. *Let*

$$p_N(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \cdots + b_N(x - x_0)(x - x_1)\cdots(x - x_{N-1})$$

be the Newton interpolating polynomial for the data $\{(x_i, f_i)\}_{i=0}^{N}$. Write down the coefficient matrix for the linear system of equations for interpolating to the data with the polynomial $p_N(x)$.

Problem 4.1.9. *Let the nodes $\{x_i\}_{i=0}^{2}$ be given. A complete quadratic $p_2(x)$ can be written in power series form or Newton form:*

$$p_2(x) = \left\{ \begin{array}{ll} a_0 + a_1 x + a_2 x^2 & \textit{power series form} \\ b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) & \textit{Newton form} \end{array} \right\}$$

By expanding the Newton form into a power series in x, verify that the coefficients b_0, b_1 and b_2 are related to the coefficients a_0, a_1 and a_2 via the equations

$$\begin{bmatrix} 1 & -x_0 & x_0 x_1 \\ 0 & 1 & -(x_0 + x_1) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}$$

Describe how you could use these equations to determine the a_i's given the values of the b_i's? Describe how you could use these equations to determine the b_i's given the values of the a_i's?

Problem 4.1.10. *Write a code segment to determine the value of the derivative $p_2'(x)$ at a point x of the Newton form of the quadratic interpolating polynomial $p_2(x)$. (Hint: $p_2(x)$ can be evaluated via nested multiplication just as a general polynomial can be evaluated by Horner's rule. Review how $p_2'(x)$ is computed via Horner's rule in Chapter 6.)*

Problem 4.1.11. *Determine the Newton form of the interpolating (cubic) polynomial to the data $(0, 1)$, $(-1, 0)$, $(1, 2)$ and $(2, 0)$. Determine the Newton form of the interpolating (cubic) polynomial to the same data written in reverse order. By converting both interpolating polynomials to power series form show that they are the same. Alternatively, show they are the same by evaluating the two polynomials at four distinct points of your choice. Finally, explain why the Polynomial Interpolation Uniqueness Theorem tells you directly that these two polynomials must be the same.*

Problem 4.1.12. *Determine the Newton form of the (quartic) interpolating polynomial to the data $(0, 1)$, $(-1, 2)$, $(1, 0)$, $(2, -1)$ and $(-2, 3)$.*

Problem 4.1.13. *Let $p_N(x)$ be the interpolating polynomial for the data $\{(x_i, f_i)\}_{i=0}^N$. Determine the number of adds (subtracts), multiplies, and divides required to determine the coefficients of the Newton form of $p_N(x)$. Hint: The coefficient $b_0 = f_0$, so it costs nothing to determine b_0. Now, recall that*

$$b_k = \frac{f_k - p_{k-1}(x_k)}{(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1})}$$

Problem 4.1.14. *Let $p_N(x)$ be the unique polynomial interpolating to the data $\{(x_i, f_i)\}_{i=0}^N$. Given its coefficients, determine the number of adds (or, equivalently, subtracts) and multiplies required to evaluate the Newton form of $p_N(x)$ at one value of x by nested multiplication. Hint: The nested multiplication scheme for evaluating the polynomial $p_N(x)$ has the form*

$$\begin{aligned} t_N &= b_N \\ t_{N-1} &= b_{N-1} + (x - x_{N-1})t_N \\ t_{N-2} &= b_{N-2} + (x - x_{N-2})t_{N-1} \\ &\vdots \\ t_0 &= b_0 + (x - x_0)t_1 \end{aligned}$$

4.1.3 The Lagrange Form of The Interpolating Polynomial

Consider the data $\{(x_i, f_i)\}_{i=0}^2$. The *Lagrange form* of the quadratic polynomial interpolating to this data may be written:

$$p_2(x) \equiv f_0 \cdot \ell_0(x) + f_1 \cdot \ell_1(x) + f_2 \cdot \ell_2(x)$$

We construct each *basis polynomial* $\ell_i(x)$ so that it is quadratic and so that it satisfies

$$\ell_i(x_j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{4.1}$$

then clearly

$$\begin{aligned} p_2(x_0) &= 1 \cdot f_0 + 0 \cdot f_1 + 0 \cdot f_2 &&= f_0 \\ p_2(x_1) &= 0 \cdot f_0 + 1 \cdot f_1 + 0 \cdot f_2 &&= f_1 \\ p_2(x_2) &= 0 \cdot f_0 + 0 \cdot f_1 + 1 \cdot f_2 &&= f_2 \end{aligned}$$

This property of the basis functions may be achieved using the following construction:

$$\ell_0(x) = \frac{\text{product of linear factors for each node except } x_0}{\text{numerator evaluated at } x = x_0} = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}$$

$$\ell_1(x) = \frac{\text{product of linear factors for each node except } x_1}{\text{numerator evaluated at } x = x_1} = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}$$

$$\ell_2(x) = \frac{\text{product of linear factors for each node except } x_2}{\text{numerator evaluated at } x = x_2} = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

Notice that these basis polynomials, $\ell_i(x)$, satisfy the condition given in equation (4.1), namely:

$$\begin{aligned} &\ell_0(x_0) = 1, \quad \ell_0(x_1) = 0, \quad \ell_0(x_2) = 0 \\ &\ell_1(x_0) = 0, \quad \ell_1(x_1) = 1, \quad \ell_1(x_2) = 0 \\ &\ell_2(x_0) = 0, \quad \ell_2(x_1) = 0, \quad \ell_2(x_2) = 1 \end{aligned}$$

Example 4.1.7. For the data $(1,2)$, $(3,3)$ and $(5,4)$ the Lagrange form of the interpolating polynomial is

$$\begin{aligned} p_2(x) &= (2)\frac{(x-3)(x-5)}{(\text{numerator at } x=1)} + (3)\frac{(x-1)(x-5)}{(\text{numerator at } x=3)} + (4)\frac{(x-1)(x-3)}{(\text{numerator at } x=5)} \\ &= (2)\frac{(x-3)(x-5)}{(1-3)(1-5)} + (3)\frac{(x-1)(x-5)}{(3-1)(3-5)} + (4)\frac{(x-1)(x-3)}{(5-1)(5-3)} \end{aligned}$$

The polynomials multiplying the data values (2), (3) and (4), respectively, are *quadratic basis functions.*

More generally, consider the polynomial $p_N(x)$ of degree N interpolating to the data $\{(x_i, f_i)\}_{i=0}^N$:

$$p_N(x) = \sum_{i=0}^{N} f_i \cdot \ell_i(x)$$

where the **Lagrange basis polynomials** $\ell_k(x)$ are polynomials of degree N and have the property

$$\ell_k(x_j) = \begin{cases} 1, & k = j \\ 0, & k \neq j \end{cases}$$

so that clearly

$$p_N(x_i) = f_i, \ i = 0, 1, \cdots, N$$

The basis polynomials may be defined by

$$\ell_k(x) \equiv \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_N)}{\text{numerator evaluated at } x = x_k} = \prod_{j=1, j\neq k}^{N} \frac{(x-x_j)}{(x_k-x_j)}$$

Algebraically, the basis function $\ell_k(x)$ is a fraction whose numerator is the product of the linear factors $(x - x_i)$ associated with each of the nodes x_i except x_k, and whose denominator is the value of its numerator at the node x_k.

Pseudocode that can be used to evaluate the Lagrange form of the interpolating polynomial is given in Fig. 4.4. That is, given the data $\{(x_i, f_i)\}_{i=0}^N$ and the value z, the pseudocode in Fig. 4.4 returns the value of the interpolating polynomial $p_N(z)$.

Unlike when using the Newton form of the interpolating polynomial, the Lagrange form has no coefficients whose values must be determined. In one sense, Lagrange form provides an explicit solution of the interpolating conditions. However, the Lagrange form of the interpolating polynomial can be more expensive to evaluate than either the power form or the Newton form (see Problem 4.1.20).

In summary, the Newton form of the interpolating polynomial is attractive because it is easy to

- Determine the coefficients.
- Evaluate the polynomial at specified values via nested multiplication.
- Extend the polynomial to incorporate additional interpolation points and data.

The Lagrange form of the interpolating polynomial

- is useful theoretically because it does not require solving a linear system, and
- explicitly shows how each data value f_i affects the overall interpolating polynomial.

Problem 4.1.15. *Determine the Lagrange form of the interpolating polynomial to the data* $(-1,0)$, $(0,1)$ *and* $(1,3)$. *Is it the same polynomial as in Example 4.1.4? Why or why not?*

Evaluating the Lagrange Form

Input: data (x_i, f_i)
scalar z

Output: $p = p_N(z)$

```
p := 0
for k = 0 to N
    ℓ_num := 1
    ℓ_den := 1
    for j = 0 to k − 1
        ℓ_num := ℓ_num ∗ (z − x_j)
        ℓ_den := ℓ_den ∗ (x_k − x_j)
    next j
    for j = k + 1 to N
        ℓ_num := ℓ_num ∗ (z − x_j)
        ℓ_den := ℓ_den ∗ (x_k − x_j)
    next j
    ℓ := ℓ_num/ℓ_den
    p = p + f_k ∗ ℓ
next k
```

Figure 4.4: Pseudocode to evaluate the Lagrange form of the interpolating polynomial, $p_N(x) = f_0 \cdot \ell_0(x) + f_1 \cdot \ell_1(x) + \cdots + f_N \cdot \ell_N(x)$, at $x = z$.

Problem 4.1.16. *Show that* $p_2(x) = f_0 \cdot l_0(x) + f_1 \cdot l_1(x) + f_2 \cdot l_2(x)$ *interpolates to the data* $\{(x_i, f_i)\}_{i=0}^2$.

Problem 4.1.17. *For the data* $(1, 2)$, $(3, 3)$ *and* $(5, 4)$ *in Example 4.1.7, check that the Lagrange form of the interpolating polynomial agrees precisely with the power series form* $\frac{3}{2} + \frac{1}{2}x$ *fitting to the same data, as for the Newton form of the interpolating polynomial.*

Problem 4.1.18. *Determine the Lagrange form of the interpolating polynomial for the data* $(0, 1)$, $(-1, 0)$, $(1, 2)$ *and* $(2, 0)$. *Check that you have determined the same polynomial as the Newton form of the interpolating polynomial for the same data, see Problem 4.1.11.*

Problem 4.1.19. *Determine the Lagrange form of the interpolating polynomial for the data* $(0, 1)$, $(-1, 2)$, $(1, 0)$, $(2, -1)$ *and* $(-2, 3)$. *Check that you have determined the same polynomial as the Newton form of the interpolating polynomial for the same data, see Problem 4.1.12.*

Problem 4.1.20. *Let* $p_N(x)$ *be the Lagrange form of the interpolating polynomial for the data* $\{(x_i, f_i)\}_{i=0}^N$. *Determine the number of additions (subtractions), multiplications, and divisions that the code segments in Fig. 4.4 use when evaluating the Lagrange form of* $p_N(x)$ *at one value of* $x = z$.

How does the cost of using the Lagrange form of the interpolating polynomial for m *different evaluation points* x *compare with the cost of using the Newton form for the same task? (See Problem 4.1.14.)*

Problem 4.1.21. *In this problem we consider "cubic Hermite interpolation" where the function and its first derivative are interpolated at the points* $x = 0$ *and* $x = 1$*:*

1. *For the function* $\phi(x) \equiv (1 + 2x)(1 - x)^2$ *show that* $\phi(0) = 1, \phi(1) = 0, \phi'(0) = 0, \phi'(1) = 0$
2. *For the function* $\psi(x) \equiv x(1 - x)^2$ *show that* $\psi(0) = 0, \psi(1) = 0, \psi'(0) = 1, \psi'(1) = 0$

3. *For the cubic Hermite interpolating polynomial* $P(x) \equiv f(0)\phi(x) + f'(0)\psi(x) + f(1)\phi(1-x) - f'(1)\psi(1-x)$ *show that* $P(0) = f(0), P'(0) = f'(0), P(1) = f(1), P'(1) = f'(1)$

4.1.4 Chebyshev Polynomials and Series

Next, we consider the Chebyshev polynomials, $T_j(x)$, and we explain why Chebyshev series provide a better way to represent polynomials $p_N(x)$ than do power series (that is, why a Chebyshev polynomial basis often provides a better representation for the polynomials than a power series basis). This representation only works if all the data are in the interval $[-1, 1]$ but it is easy to transform any finite interval $[a, b]$ onto the interval $[-1, 1]$ as we see later. Rather than discuss transformations to put all the data in the interval $[-1, 1]$, for simplicity we assume the data is already in place.

The j^{th} Chebyshev polynomial $T_j(x)$ is defined by

$$T_j(x) \equiv \cos(j \arccos(x)), \quad j = 0, 1, 2, \cdots .$$

Note, this definition only works on the interval $x \in [-1, 1]$, because that is the domain of the function $\arccos(x)$. Now, let us compute the first few Chebyshev polynomials (and convince ourselves that they are polynomials):

$$\begin{aligned} T_0(x) &= \cos(0 \arccos(x)) = 1 \\ T_1(x) &= \cos(1 \arccos(x)) = x \\ T_2(x) &= \cos(2 \arccos(x)) = 2[\cos(\arccos(x))]^2 - 1 = 2x^2 - 1 \end{aligned}$$

We have used the double angle formula, $\cos 2\theta = 2 \cdot \cos^2 \theta - 1$, to derive $T_2(x)$, and we use its extension, the addition formula $\cos(\alpha) + \cos(\beta) = 2\cos\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right)$, to derive the higher degree Chebyshev polynomials. We have

$$\begin{aligned} T_{j+1}(x) + T_{j-1}(x) &= \cos[(j+1)\arccos(x)] + \cos[(j-1)\arccos(x)] \\ &= \cos[j \cdot \arccos(x) + \arccos(x)] + \cos[j \cdot \arccos(x) - \arccos(x)] \\ &= 2\cos[j \cdot \arccos(x)]\cos[\arccos(x)] \\ &= 2x\,T_j(x) \end{aligned}$$

So, starting from $T_0(x) = 1$ and $T_1(x) = x$, the Chebyshev polynomials may be defined by the recurrence relation

$$T_{j+1}(x) = 2xT_j(x) - T_{j-1}(x),\ j = 1, 2, \ldots$$

By this construction we see that the Chebyshev polynomial $T_N(x)$ has exact degree N and that if N is even (odd) then $T_N(x)$ involves only even (odd) powers of x; see Problems 4.1.24 and 4.1.25.

A **Chebyshev series** of order N has the form:

$$b_0T_0(x) + b_1T_1(x) + \cdots + b_NT_N(x)$$

for a given set of coefficients $b_0, b_1, \cdots, b_N$. It is a polynomial of degree N; see Problem 4.1.26. The first eight Chebyshev polynomials written in power series form, and the first eight powers of x written in Chebyshev series form, are given in Table 4.2. This table illustrates the fact that every polynomial can be written either in power series form or in Chebyshev series form.

It is easy to compute the zeros of the first few Chebyshev polynomials manually. You'll find that $T_1(x)$ has a zero at $x = 0$, $T_2(x)$ has zeros at $x = \pm\frac{1}{\sqrt{2}}$, $T_3(x)$ has zeros at $x = 0, \pm\frac{\sqrt{3}}{2}$ etc. What you'll observe is that all the zeros are real and are in the interval $(-1, 1)$, and that the zeros of $T_n(x)$ interlace those of $T_{n+1}(x)$; that is, between each pair of zeros of $T_{n+1}(x)$ is zero of $T_n(x)$. In fact, all these properties are seen easily using the general formula for the zeros of the Chebyshev polynomial $T_n(x)$:

$$x_i = \cos\left(\frac{2i-1}{2n}\pi\right),\ i = 1, 2, \ldots, n$$

$T_0(x)$	$=$	1	$T_0(x)$	$=$	1
$T_1(x)$	$=$	x	$T_1(x)$	$=$	x
$T_2(x)$	$=$	$2x^2 - 1$	$\frac{1}{2}\{T_2(x) + T_0(x)\}$	$=$	x^2
$T_3(x)$	$=$	$4x^3 - 3x$	$\frac{1}{4}\{T_3(x) + 3T_1(x)\}$	$=$	x^3
$T_4(x)$	$=$	$8x^4 - 8x^2 + 1$	$\frac{1}{8}\{T_4(x) + 4T_2(x) + 3T_0(x)\}$	$=$	x^4
$T_5(x)$	$=$	$16x^5 - 20x^3 + 5x$	$\frac{1}{16}\{T_5(x) + 5T_3(x) + 10T_1(x)\}$	$=$	x^5
$T_6(x)$	$=$	$32x^6 - 48x^4 + 18x^2 - 1$	$\frac{1}{32}\{T_6(x) + 6T_4(x) + 15T_2(x) + 10T_0(x)\}$	$=$	x^6
$T_7(x)$	$=$	$64x^7 - 112x^5 + 56x^3 - 7x$	$\frac{1}{64}\{T_7(x) + 7T_5(x) + 21T_3(x) + 35T_1(x)\}$	$=$	x^7

Table 4.2: Chebyshev polynomial conversion table

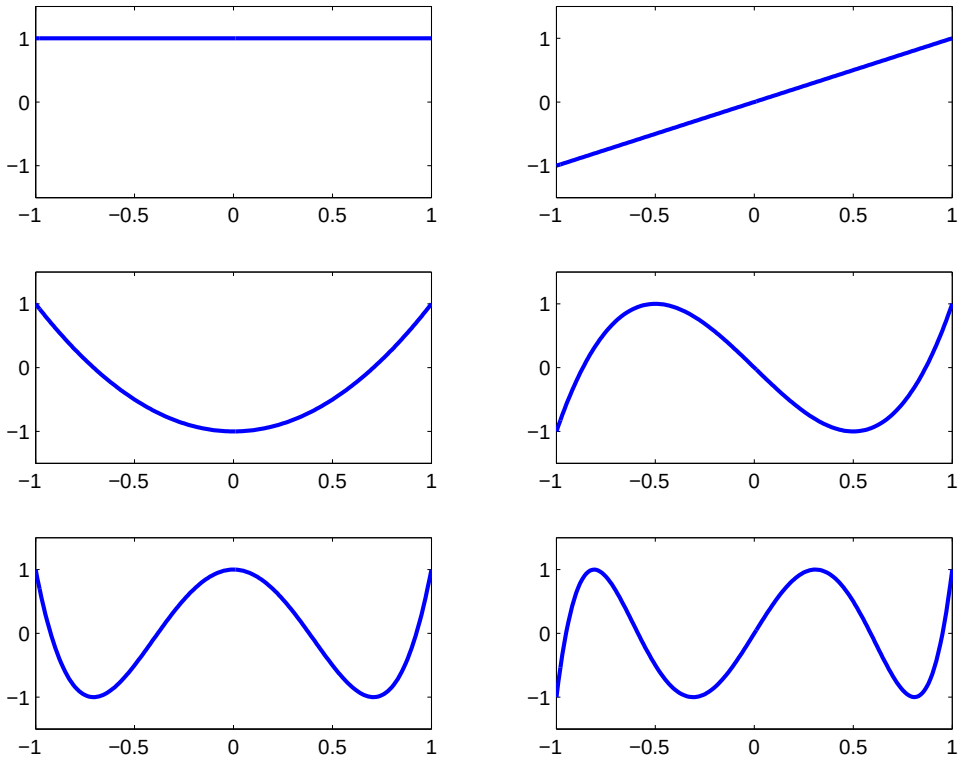

Figure 4.5: The Chebyshev polynomials $T_0(x)$ top left; $T_1(x)$ top right; $T_2(x)$ middle left; $T_3(x)$ middle right; $T_4(x)$ bottom left; $T_5(x)$ bottom right.

which are so-called Chebyshev points. These points are important when discussing error on polynomial interpolation; see Section 4.2.

One advantage of using a Chebyshev polynomial $T_j(x) = \cos(j \arccos(x))$ over using the corresponding power term x^j of the same degree to represent data for values $x \in [-1, +1]$ is that $T_j(x)$ oscillates j times between $x = -1$ and $x = +1$, see Fig. 4.5. Also, as j increases the first and last zero crossings for $T_j(x)$ get progressively closer to, but never reach, the interval endpoints $x = -1$ and $x = +1$. This is illustrated in Fig. 4.6 and Fig. 4.7, where we plot some of the Chebyshev and power series bases on the interval $[0, 1]$, with transformed variable $\bar{x} = 2x - 1$. That is, Fig. 4.6 shows $T_3(\bar{x})$, $T_4(\bar{x})$ and $T_5(\bar{x})$ for $x \in [0, 1]$, and Fig. 4.7 shows the corresponding powers $\bar{x}^3$, $\bar{x}^4$ and $\bar{x}^5$. Observe that the graphs of the Chebyshev polynomials are quite distinct while those of the powers of x are quite closely grouped. When we can barely discern the difference of two functions graphically it is likely that the computer will have difficulty discerning the difference numerically.

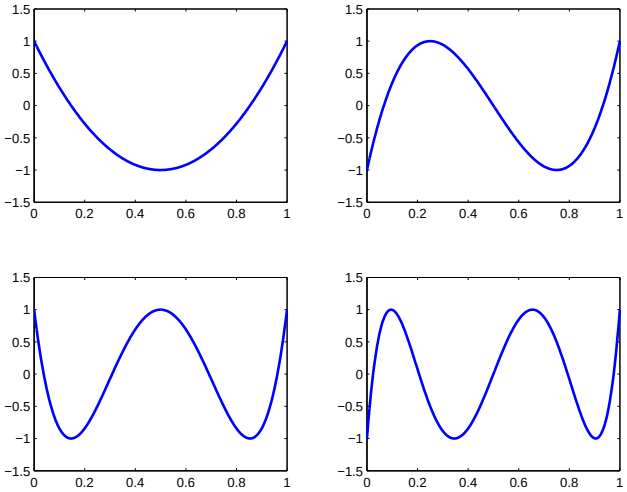

Figure 4.6: The shifted Chebyshev polynomials $T_2(\bar{x})$ top left; $T_3(\bar{x})$ top right; $T_4(\bar{x})$ bottom left, $T_5(\bar{x})$ bottom right.

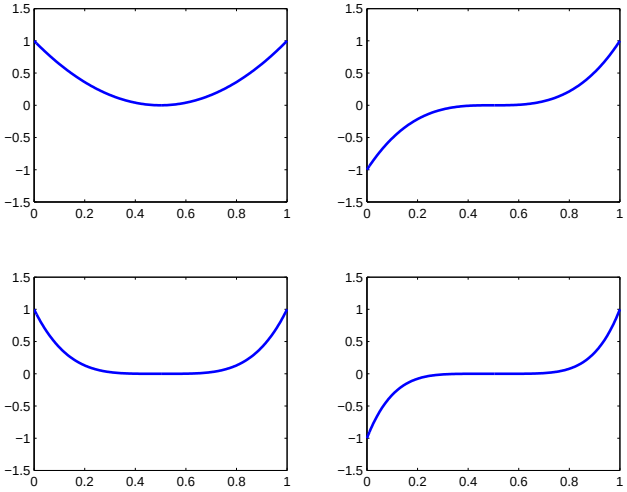

Figure 4.7: The powers $\bar{x}^2$ top left; $\bar{x}^3$ top right; $\bar{x}^4$ bottom left, $\bar{x}^5$ bottom right.

Suppose we want to construct an interpolating polynomial of degree N using a Chebyshev series $p_N(x) = \sum_{j=0}^{N} b_j T_j(x)$. Given the data $(x_i, f_i)_{i=0}^{N}$, where all the $x_i \in [-1, 1]$, we need to build and solve the linear system

$$p_N(x_i) \equiv \sum_{j=0}^{N} b_j T_j(x_i) = f_i, \quad i = 0, 1, \ldots, N$$

This is simply a set of $(N+1)$ equations in the $(N+1)$ unknowns, the coefficients b_j, $j = 0, 1, \ldots, N$. To evaluate $T_j(x_i)$, $j = 0, 1, \ldots, N$ in the i^{th} equation we simply use the recurrence relation $T_{j+1}(x_i) = 2x_i T_j(x_i) - T_{j-1}(x_i)$, $j = 1, 2, \ldots, N-1$ with $T_0(x_i) = 1, T_1(x_i) = x_i$. The resulting linear system can then be solved using, for example, the techniques discussed in Chapter 3. By the polynomial interpolation uniqueness theorem, we compute the same polynomial (written in a different form) as when using each of the previous interpolation methods on the same data.

The algorithm most often used to evaluate a Chebyshev series is a recurrence that is similar to Horner's scheme for power series (see Chapter 6). Indeed, the power series form and the corresponding Chebyshev series form of a given polynomial can be evaluated at essentially the same arithmetic cost using these recurrences.

Problem 4.1.22. *In this problem you are to exploit the Chebyshev recurrence relation.*

1. *Use the Chebyshev recurrence relation to reproduce Table 4.2.*
2. *Use the Chebyshev recurrence relation and Table 4.2 to compute $T_8(x)$ as a power series in x.*
3. *Use Table 4.2 and the formula for $T_8(x)$ to compute x^8 as a Chebyshev series.*

Problem 4.1.23. *Use Table 4.2 to*

1. *Write $3T_0(x) - 2T_1(x) + 5T_2(x)$ as a polynomial in power series form.*
2. *Write the polynomial $4 - 3x + 5x^2$ as a Chebyshev series involving $T_0(x)$, $T_1(x)$ and $T_2(x)$.*

Problem 4.1.24. *Prove that the n^{th} Chebyshev polynomial $T_N(x)$ is a polynomial of degree N.*

Problem 4.1.25. *Prove the following results that are easy to observe in Table 4.2.*

1. *The even indexed Chebyshev polynomials $T_{2n}(x)$ all have power series representations in terms of even powers of x only.*
2. *The odd powers x^{2n+1} have Chebyshev series representations involving only odd indexed Chebyshev polynomials $T_{2s+1}(x)$.*

Problem 4.1.26. *Prove that the Chebyshev series*

$$b_0 T_0(x) + b_1 T_1(x) + \cdots + b_N T_N(x)$$

is a polynomial of degree N.

Problem 4.1.27. *Show that the* Chebyshev points $x_i = \cos\left(\frac{2i-1}{2N}\pi\right)$, $i = 1, 2, \ldots, N$ *are the complete set of zeros of $T_N(x)$*

Problem 4.1.28. *Show that the points $x_i = \cos\left(\frac{i}{N}\pi\right)$, $i = 1, 2, \ldots, N-1$ are the complete set of extrema of $T_N(x)$.*

4.2 The Error in Polynomial Interpolation

Let $p_N(x)$ be the polynomial of degree N interpolating to the data $\{(x_i, f_i = f(x_i))\}_{i=0}^N$. How accurately does the polynomial $p_N(x)$ approximate the data function $f(x)$ at any point x? Mathematically the answer to this question is the same for all forms of the polynomial of a given degree interpolating the same data. However, as we see later, there are differences in accuracy in practice, but these are related not to the accuracy of polynomial interpolation but to the accuracy of the implementation.

Let the evaluation point x and all the interpolation points $\{x_i\}_{i=0}^N$ lie in a closed interval $[a, b]$. An advanced course in numerical analysis shows, using Rolle's Theorem (see Problem 4.2.1) repeatedly, that the error expression may be written as

$$f(x) - p_N(x) = \frac{\omega_{N+1}(x)}{(N+1)!} f^{(N+1)}(\xi_x)$$

where $\omega_{N+1}(x) \equiv (x - x_0)(x - x_1) \cdots (x - x_N)$ and ξ_x is some (unknown) point in the interval $[a, b]$. The precise location of the point ξ_x depends on the function $f(x)$, the point x at which the interpolating polynomial is to be evaluated, and the data points $\{x_i\}_{i=0}^N$. The term $f^{(N+1)}(\xi_x)$ is the $(N+1)^{\text{st}}$ derivative of $f(x)$ evaluated at the point $x = \xi_x$. Some of the intrinsic properties of the interpolation error are:

1. For any value of i, the error is zero when $x = x_i$ because $w_{N+1}(x_i) = 0$ (the interpolating conditions).

2. The error is zero when the data f_i are measurements of a polynomial $f(x)$ of exact degree N because then the $(N+1)^{\text{st}}$ derivative $f^{(N+1)}(x)$ is identically zero. This is simply a statement of the uniqueness theorem of polynomial interpolation.

Taking absolute values in the interpolation error expression and maximizing both sides of the resulting inequality over $x \in [a, b]$, we obtain the **polynomial interpolation error bound**

$$\max_{x\in[a,b]} |f(x) - p_N(x)| \leq \max_{x\in[a,b]} |\omega_{N+1}(x)| \cdot \frac{\max\limits_{z\in[a,b]} |f^{(N+1)}(z)|}{(N+1)!} \tag{4.2}$$

To make this error bound small we must make either the term $\max\limits_{x\in[a,b]} \dfrac{|f^{(N+1)}(x)|}{(N+1)!}$ or the term $\max\limits_{x\in[a,b]} |\omega_{N+1}(x)|$, or both, small. Generally, we have little information about the function $f(x)$ or its derivatives, even about their sizes, and in any case we cannot change them to minimize the bound.

In the absence of information about $f(x)$ or its derivatives, we might aim to choose the nodes $\{x_i\}_{i=0}^N$ so that $|\omega_{N+1}(x)|$ is small throughout $[a, b]$. The plots in Fig. 4.8 illustrate that generally equally spaced nodes defined by $x_i = a + \dfrac{i}{N}(b-a)$, $i = 0, 1, \ldots, N$ are not a good choice, as for them the polynomial $\omega_{N+1}(x)$ oscillates with zeros at the nodes as anticipated, but with the amplitudes of the oscillations growing as the evaluation point x approaches either endpoint of the interval $[a, b]$. Of course, data is usually measured at equally spaced points and this is a matter over which you will have little or no control. So, though equally spaced points are a bad choice in theory, they may be our only choice!

The right plot in Fig. 4.8 depicts the polynomial $\omega_{N+1}(x)$ for points x slightly outside the interval $[a, b]$ where $\omega_{N+1}(x)$ grows like x^{N+1}. Evaluating the polynomial $p_N(x)$ for points x outside the interval $[a, b]$ is *extrapolation*. We can see from this plot that if we evaluate the polynomial $p_N(x)$ outside the interval $[a, b]$, then the error component $\omega_{N+1}(x)$ can be very large, indicating that extrapolation should be avoided whenever possible; this is true for all choices of interpolation points. This observation implies that we need to know the application intended for the interpolating function before we measure the data, a fact frequently ignored when designing an experiment. We may not be able to avoid extrapolation but we certainly need to be aware of its dangers.

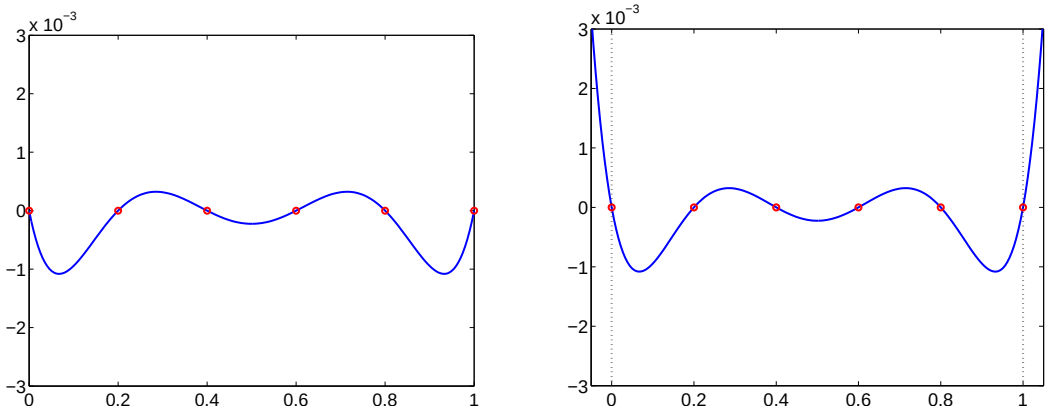

Figure 4.8: Plot of the polynomial $\omega_{N+1}(x)$ for equally spaced nodes $x_i \in [0,1]$, and $N = 5$. The equally spaced nodes are denoted by circles. The left plot shows $\omega_{N+1}(x)$ on the interval $[0,1]$, and the right plot shows $\omega_{N+1}(x)$ on the interval $[-0.05, 1.05]$.

For the data in Fig. 4.9, we use the Chebyshev points for the interval $[a,b] = [0,1]$:

$$x_i = \frac{b+a}{2} - \frac{b-a}{2}\cos\left(\frac{2i+1}{2N+2}\pi\right) \quad i = 0, 1, \cdots, N.$$

Note that the Chebyshev points do not include the endpoints a or b and that all the Chebyshev points lie inside the open interval (a,b). For the interval $[a,b] = [-1,+1]$, the Chebyshev points are the zeros of the Chebyshev polynomial $T_{N+1}(x)$, hence the function $\omega_{N+1}(x)$ is a scaled version of $T_{N+1}(x)$ (see Section 4.1.4 for the definition of the Chebyshev polynomials and the Chebyshev points).

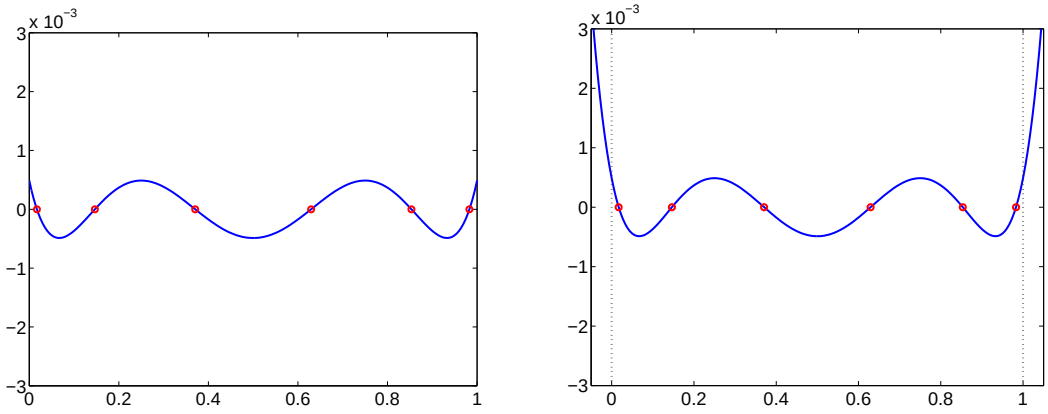

Figure 4.9: Plot of the polynomial $\omega_{N+1}(x)$ for Chebyshev nodes $x_i \in [0,1]$, and $N = 5$. The Chebyshev nodes are denoted by circles. The left plot shows $\omega_{N+1}(x)$ on the interval $[0,1]$, and the right plot shows $\omega_{N+1}(x)$ on the interval $[-0.05, 1.05]$

With the Chebyshev points as nodes, the maximum value of the polynomial $|\omega_{N+1}(x)|$ on $[a,b] = [0,1]$ is less than half its value in Fig. 4.8. As N increases, the improvement in the size of $|\omega_{N+1}(x)|$ when using the Chebyshev points rather than equally spaced points is even greater. Indeed, for polynomials of degree 20 or less, interpolation to the data measured at the Chebyshev points gives a maximum error not greater than twice the smallest possible maximum error (known as the minimax

error) taken over all polynomial fits to the data (not just using interpolation). However, there are penalties:

1. As with equally space points, extrapolation should be avoided because the error component $\omega_{N+1}(x)$ can be very large for points outside the interval $[a, b]$ (see the right plot in Fig. 4.9. Indeed, extrapolation using the interpolating function based on data measured at Chebyshev points can be even more disastrous than extrapolation based on the same number of equally spaced data points.

2. It may be difficult to obtain the data f_i measured at the Chebyshev points.

Figs. 4.8 — 4.9 suggest that to choose $p_N(x)$ so that it will accurately approximate all reasonable choices of $f(x)$ on an interval $[a, b]$, we should

1. Choose the nodes $\{x_i\}_{i=0}^N$ so that, for all likely evaluation points x, $\min(x_i) \le x \le \max(x_i)$.

2. If possible, choose the nodes as the Chebyshev points. If this is not possible, attempt to choose them so they are denser close to the endpoints of the interval $[a, b]$.

Problem 4.2.1. *Prove* Rolle's Theorem*: Let $f(x)$ be continuous and differentiable on the closed interval $[a, b]$ and let $f(a) = f(b) = 0$, then there exists a point $c \in (a, b)$ such that $f'(c) = 0$.*

Problem 4.2.2. *Let $x = x_0 + sh$ and $x_i = x_0 + ih$, show that $w_{n+1}(x) = h^{N+1}s(s-1)\cdots(s-N)$.*

Problem 4.2.3. *Determine the following bound for straight line interpolation (that is with a polynomial of degree one, $p_1(x)$) to the data $(a, f(a))$ and $(b, f(b))$*

$$\max_{x\in[a,b]} |f(x) - p_1(x)| \le \frac{1}{8}|b-a|^2 \max_{z\in[a,b]} |f^{(2)}(z)|$$

[Hint: You need to use the bound (4.2) for this special case, and use standard calculus techniques to find $\max_{x\in[a,b]} |\omega_{N+1}(x)|$.]

Problem 4.2.4. *Let $f(x)$ be a polynomial of degree $N+1$ for which $f(x_i) = 0$, $i = 0, 1, \cdots, N$. Show that the interpolation error expression reduces to $f(x) = A\, w_{N+1}(x)$ for some constant A. Explain why this result is to be expected.*

Problem 4.2.5. *For $N = 5$, $N = 10$ and $N = 15$ plot, on one graph, the three polynomials $\omega_{N+1}(x)$ for $x \in [0, 1]$ defined using equally spaced nodes. For the same values of N plot, on one graph, the three polynomials $\omega_{N+1}(x)$ for $x \in [0, 1]$ defined using Chebyshev nodes. Compare the three maximum values of $|\omega_{N+1}(x)|$ on the interval $[0, 1]$ for each of these two graphs. Also, compare the corresponding maximum values in the two graphs.*

Runge's Example

To use polynomial interpolation to obtain an accurate estimate of $f(x)$ on a fixed interval $[a, b]$, it is natural to think that increasing the number of interpolating points in $[a, b]$, and hence increasing the degree of the polynomial, will reduce the error in the polynomial interpolation to $f(x)$. In fact, for some functions $f(x)$ this approach may worsen the accuracy of the interpolating polynomial. When this is the case, the effect is usually greatest for equally spaced interpolation points. At least a part of the problem arises from the term $\max_{x\in[a,b]} \dfrac{|f^{(N+1)}(x)|}{(N+1)!}$ in the expression for interpolation error. This term may grow, or at least not decay, as N increases, even though the denominator $(N+1)!$ grows very quickly as N increases.

Consider the function $f(x) = \dfrac{1}{1+x^2}$ on the interval $[-5, 5]$. Fig. 4.10 shows a plot of interpolating polynomials of degree $N = 10$ for this function using 11 (i.e., $N + 1$) equally spaced nodes and 11 Chebyshev nodes. Also shown in the figure is the error $f(x) - p_N(x)$ for each interpolating polynomial. Note that the behavior of the error for the equally spaced points clearly mimics the behavior of $\omega_{N+1}(x)$ in Fig. 4.8. For the same number of Chebyshev points the error is much smaller and more evenly distributed on the interval $[-5, 5]$ but it still mimics $\omega_{N+1}(x)$ for this placement of points.

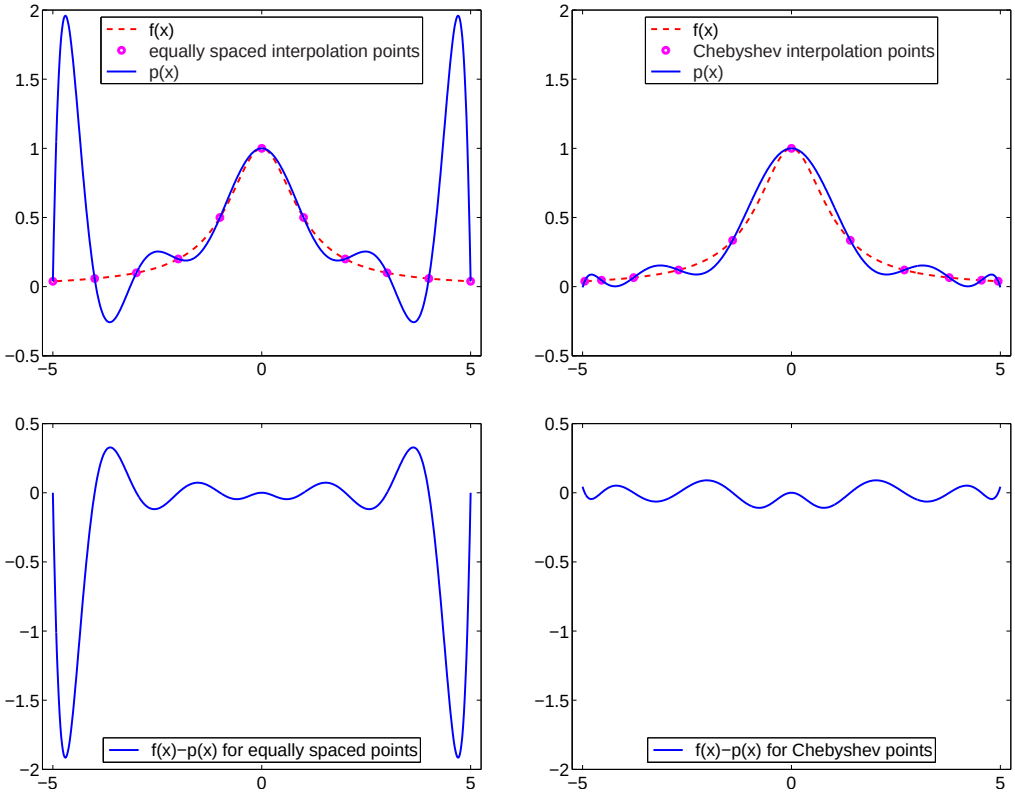

Figure 4.10: Interpolating polynomials for $f(x) = \dfrac{1}{1+x^2}$ using 11 equally spaced points (top left) and 11 Chebyshev points (top right) on the interval $[-5, 5]$. The associated error functions $f(x) - p_{10}(x)$ are shown in the bottom two plots.

Problem 4.2.6. Runge's example. *Consider the function* $f(x) = \dfrac{1}{1+x^2}$ *on the interval* $[-5, +5]$.

1. *For* $N = 5$, $N = 10$ *and* $N = 15$ *plot the error* $e(x) = p_N(x) - f(x)$ *where the polynomial* $p_N(x)$ *is computed by interpolating at the* $N+1$ *equally spaced nodes,* $x_i = a + \dfrac{i}{N}(b-a), i = 0, 1, \cdots, N$.
2. *Repeat Part 1 but interpolate at the* $N + 1$ *Chebyshev nodes shifted to the interval* $[-5, 5]$.

Create a table listing the approximate maximum absolute error and its location in $[a, b]$ *as a function of* N*; there will be two columns one for each of Parts 1 and 2. [Hint: For simplicity, compute the interpolating polynomial* $p_N(x)$ *using the Lagrange form. To compute the approximate maximum error you will need to sample the error between each pair of nodes (recall, the error is zero at the nodes) — sampling at the midpoints of the intervals between the nodes will give you a sufficiently accurate estimate.]*

Problem 4.2.7. *Repeat the previous problem with data taken from* $f(x) = e^x$ *instead of* $f(x) = \frac{1}{1+x^2}$. *What major differences do you observe in the behavior of the error?*

The Effect of the Method of Representation of the Interpolating Polynomial

A second source of error in polynomial interpolation is that from the computation of the polynomial interpolating function and its evaluation. It is "well-known" that large Vandermonde systems tend to be ill–conditioned, but we see errors of different sizes for all our four fitting techniques (Vandermonde, Newton, Lagrange and Chebyshev) and the Newton approach (in our implementation) turns out to be even more ill–conditioned than the Vandermonde as the number of interpolation points increases. To illustrate this fact we present results from interpolating to e^x with $n = 10, 20, \ldots, 100$ equally spaced points in the interval $[-1, 1]$ so we are interpolating with polynomials of degrees $N = 9, 19, \ldots, 99$, respectively. In Table 4.3, we show the maximum absolute errors (over 201 equally spaced points in $[-1, 1]$) of this experiment. We don't know the size of the interpolation error as the error shown is a combination of interpolation error, computation of fit error and evaluation error, though we can be fairly confident that the errors for $n = 10$ are dominated by interpolation error since the errors are all approximately the same. The interpolation errors then decrease and at $n = 20$ the error is clearly dominated by calculation errors (since all the errors are different). For $n > 20$ it is our supposition that calculation errors dominate, and in any case, for each value of n the interpolation error cannot be larger than the smallest of the errors shown. Clearly, for large enough n using the Chebyshev series is best, and for small n it doesn't matter much which method is used. Surprisingly, the Lagrange interpolating function for which we do not solve a system of linear equations but which involves an expensive evaluation has very large errors when n is large, but Newton is always worst. A plot of the errors shows that in all cases the errors are largest close to the ends of the interval of interpolation.

n	Vandermonde	Newton	Lagrange	Chebyshev
10	3.837398e-009	3.837397e-009	3.837398e-009	3.837399e-009
20	3.783640e-013	1.100786e-013	1.179390e-012	1.857403e-013
30	6.497576e-010	7.048900e-011	7.481358e-010	2.436179e-011
40	5.791111e-008	1.385682e-008	8.337930e-007	3.987289e-008
50	3.855115e-005	2.178791e-005	2.021784e-004	9.303255e-005
60	1.603913e-001	4.659020e-002	9.780899e-002	2.822994e-003
70	1.326015e+003	2.905506e+004	2.923504e+002	1.934900e+000
80	8.751649e+005	3.160393e+010	3.784625e+004	4.930093e+001
90	2.280809e+010	7.175811e+015	6.391371e+007	1.914712e+000
100	2.133532e+012	4.297183e+022	1.122058e+011	1.904017e+000

Table 4.3: Errors fitting e^x using n equally spaced data points on the interval $[-1, 1]$.

We repeated the experiment using Chebyshev points; the results are shown in Table 4.4. Here, the interpolation error is smaller, as is to be anticipated using Chebyshev points, and the computation error does not show until $n = 50$ for Newton and hardly at all for the Lagrange and Chebyshev fits. When the errors are large they are largest near the ends of the interval of interpolation but overall they are far closer to being evenly distributed across the interval than in the equally spaced points case.

The size of the errors reported above depends, to some extent, on the way that the interpolation function is coded. In each case, we coded the function in the "obvious" way (using the formulas presented in this text). But, using these formulas is not necessarily the best way to reduce the errors. It is clear that in the case of the Newton form the choice of the order of the interpolation points determines the diagonal values in the lower triangular matrix, and hence its degree of ill–conditioning. In our implementation of all the forms, and specifically the Newton form, we ordered

n	Vandermonde	Newton	Lagrange	Chebyshev
10	6.027099e-010	6.027103e-010	6.027103e-010	6.027103e-010
20	8.881784e-016	1.332268e-015	3.552714e-015	1.554312e-015
30	8.881784e-016	9.992007e-016	2.664535e-015	1.332268e-015
40	1.110223e-015	1.110223e-015	3.552714e-015	1.332268e-015
50	7.105427e-015	1.532131e-009	3.996803e-015	1.776357e-015
60	5.494938e-012	1.224753e-004	5.773160e-015	1.332268e-015
70	1.460477e-009	1.502797e+000	4.884981e-015	2.164935e-015
80	2.291664e-006	3.292080e+005	4.440892e-015	1.332268e-015
90	2.758752e-004	3.140739e+011	5.773160e-015	1.332268e-015
100	1.422977e+000	1.389857e+018	8.881784e-015	1.776357e-015

Table 4.4: Errors fitting e^x using n Chebyshev data points in $[-1, 1]$.

the points in increasing order of value, that is starting from the left. It is less apparent what, if any, substantial effect changing the order of the points would have in the Vandermonde and Lagrange cases. It is however possible to compute these polynomials in many different ways. For example, the Lagrange form can be computed using "barycentric" formulas. Consider the standard Lagrange form $p_N(x) = \sum_{i=0}^{N} l_i(x) f_i$ where $l_i(x) = \prod_{j=0, j\neq i}^{N} \left(\frac{x - x_j}{x_i - x_j} \right)$. It is easy to see that we can rewrite this in the barycentric form $p_N(x) = l(x) \sum_{i=0}^{N} \frac{\omega_i}{x - x_i} f_i$ where $l(x) = \prod_{i=0}^{N} (x - x_i)$ and $\omega_j = \frac{1}{\prod_{k\neq j}(x_j - x_k)}$ which is less expensive for evaluation. It is also somewhat more accurate but not by enough to be even close to competitive with using the Chebyshev polynomial basis.

Problem 4.2.8. *Produce the equivalent of Tables 4.3 and 4.4 for Runge's function defined on the interval* $[-1, 1]$. *You should expect relatively similar behavior to that exhibited in the two tables.*

Problem 4.2.9. *Derive the barycentric form of the Lagrange interpolation formula from the standard form. For what values of* x *is the barycentric form indeterminate (requires computing a "zero over zero"), and how would you avoid this problem in practice*

4.3 Polynomial Splines

Now, we consider techniques designed to reduce the problems that arise when data are *interpolated* by a *single polynomial*. The first technique *interpolates* the data by a *collection of low degree polynomials* rather than by a single high degree polynomial. Another technique outlined in Section 4.4, *approximates* but not necessarily interpolates, the data by least squares fitting a *single low degree polynomial*.

Generally, by reducing the size of the interpolation error bound we reduce the actual error. Since the term $\left|\omega_{N+1}(x)\right|$ in the bound is the product of $N + 1$ linear factors $|x - x_i|$, each the distance between two points that both lie in $[a, b]$, we have $|x - x_i| \leq |b - a|$ and so

$$\max_{x\in[a,b]} |f(x) - p_N(x)| \leq |b - a|^{N+1} \cdot \frac{\max_{x\in[a,b]} |f^{(N+1)}(x)|}{(N + 1)!}$$

This (larger) bound suggests that we can make the error as small as we wish by freezing the value of N and then reducing the size of $|b - a|$. We still need an approximation over the original interval, so we use a *piecewise polynomial approximation*: the original interval is divided into non-overlapping subintervals and a different polynomial fit of the data is used on each subinterval.

4.3.1 Linear Polynomial Splines

A simple piecewise polynomial fit is the linear interpolating spline. For data $\{(x_i, f_i)\}_{i=0}^{N}$, where

$$a = x_0 < x_1 \cdots < x_N = b, \quad h \equiv \max_i |x_i - x_{i-1}|,$$

the **linear spline** $S_{1,N}(x)$ is a continuous function that interpolates to the data and is constructed from linear functions that we identify as two–point interpolating polynomials:

$$S_{1,N}(x) = \begin{cases} f_0 \dfrac{x - x_1}{x_0 - x_1} & + & f_1 \dfrac{x - x_0}{x_1 - x_0} & \text{when } x \in [x_0, x_1] \\ f_1 \dfrac{x - x_2}{x_1 - x_2} & + & f_2 \dfrac{x - x_1}{x_2 - x_1} & \text{when } x \in [x_1, x_2] \\ & \vdots & & \\ f_{N-1} \dfrac{x - x_N}{x_{N-1} - x_N} & + & f_N \dfrac{x - x_{N-1}}{x_N - x_{N-1}} & \text{when } x \in [x_{N-1}, x_N] \end{cases}$$

From the bound on the error for polynomial interpolation, in the case of an interpolating polynomial of degree one,

$$\begin{aligned} \max_{z \in [x_{i-1}, x_i]} |f(z) - S_{1,N}(z)| &\leq \frac{|x_i - x_{i-1}|^2}{8} \cdot \max_{x \in [x_{i-1}, x_i]} |f^{(2)}(x)| \\ &\leq \frac{h^2}{8} \cdot \max_{x \in [a,b]} |f^{(2)}(x)| \end{aligned}$$

(See Problem 4.2.3.) That is, the bound on the maximum absolute error behaves like h^2 as the maximum interval length $h \to 0$. Suppose that the nodes are chosen to be equally spaced in $[a, b]$, so that $x_i = a + ih$, $i = 0, 1, \cdots, N$, where $h \equiv \dfrac{b-a}{N}$. As the number of points N increases, the error in using $S_{1,N}(z)$ as an approximation to $f(z)$ tends to zero like $\dfrac{1}{N^2}$.

Example 4.3.1. We construct the linear spline to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$:

$$S_{1,2}(x) = \begin{cases} 0 \cdot \dfrac{x - 0}{(-1) - 0} & + & 1 \cdot \dfrac{x - (-1)}{0 - (-1)} & \text{when } x \in [-1, 0] \\ 1 \cdot \dfrac{x - 1}{0 - 1} & + & 3 \cdot \dfrac{x - 0}{1 - 0} & \text{when } x \in [0, 1] \end{cases}$$

An alternative (but equivalent) method for representing a linear spline uses a **linear B–spline basis**, $L_i(x), i = 0, 1, \cdots, N$, chosen so that $L_i(x_j) = 0$ for all $j \neq i$ and $L_i(x_i) = 1$. Here, each $L_i(x)$ is a "roof" shaped function with the apex of the roof at $(x_i, 1)$ and the span on the interval $[x_{i-1}, x_{i+1}]$, and with $L_i(x) \equiv 0$ outside $[x_{i-1}, x_{i+1}]$. That is,

$$L_0(x) = \begin{cases} \dfrac{x - x_1}{x_0 - x_1} & \text{when} \quad x \in [x_0, x_1] \\ 0 & \text{for all other } x, \end{cases}$$

$$L_i(x) = \begin{cases} \dfrac{x - x_{i-1}}{x_i - x_{i-1}} & \text{when} \quad x \in [x_{i-1}, x_i] \\ \dfrac{x - x_{i+1}}{x_i - x_{i+1}} & \text{when} \quad x \in [x_i, x_{i+1}] \\ 0 & \text{for all other } x \end{cases} \qquad i = 1, 2, \ldots, N-1,$$

and

$$L_N(x) = \begin{cases} \dfrac{x - x_{N-1}}{x_N - x_{N-1}} & \text{when} \quad x \in [x_{N-1}, x_N] \\ 0 & \text{for all other } x \end{cases}$$

In terms of the linear B–spline basis we can write

$$S_{1,N}(x) = \sum_{i=0}^{N} L_i(x) \cdot f_i$$

Example 4.3.2. We construct the linear spline to the data $(-1, 0)$, $(0, 1)$ and $(1, 3)$ using the linear B–spline basis. First we construct the basis:

$$\begin{aligned} L_0(x) &= \begin{cases} \dfrac{x-0}{(-1)-0}, & x \in [-1, 0] \\ 0, & x \in [0, 1] \end{cases} \\ L_1(x) &= \begin{cases} \dfrac{x-(-1)}{0-(-1)}, & x \in [-1, 0] \\ \dfrac{x-1}{0-1}, & x \in [0, 1] \end{cases} \\ L_2(x) &= \begin{cases} 0, & x \in [-1, 0] \\ \dfrac{x-0}{1-0}, & x \in [0, 1] \end{cases} \end{aligned}$$

which are shown in Fig. 4.11. Notice that each $L_i(x)$ is a "roof" shaped function. The linear spline interpolating to the data is then given by

$$S_{1,2}(x) = 0 \cdot L_0(x) + 1 \cdot L_1(x) + 3 \cdot L_2(x)$$

Problem 4.3.1. *Let $S_{1,N}(x)$ be a linear spline.*

1. *Show that $S_{1,N}(x)$ is continuous and that it interpolates the data $\{(x_i, f_i)\}_{i=0}^{N}$.*
2. *At the interior nodes x_i, $i = 1, 2, \cdots, N-1$, show that $S'_{1,N}(x_i^-) = \dfrac{f_i - f_{i-1}}{x_i - x_{i-1}}$ and $S'_{1,N}(x_i^+) = \dfrac{f_{i+1} - f_i}{x_{i+1} - x_i}$*
3. *Show that, in general, $S'_{1,N}(x)$ is discontinuous at the interior nodes.*
4. *Under what circumstances would $S_{1,N}(x)$ have a continuous derivative at $x = x_i$?*
5. *Determine the linear spline $S_{1,3}(x)$ that interpolates to the data $(0, 1)$, $(1, 2)$, $(3, 3)$ and $(5, 4)$. Is $S'_{1,3}(x)$ discontinuous at $x = 1$? At $x = 2$? At $x = 3$?*

Problem 4.3.2. *Given the data $(0, 1)$, $(1, 2)$, $(3, 3)$ and $(5, 4)$, write down the linear B–spline basis functions $L_i(x)$, $i = 0, 1, 2, 3$ and the sum representing $S_{1,3}(x)$. Show that $S_{1,3}(x)$ is the same linear spline that was described in Problem 4.3.1. Using this basis function representation of the linear spline, evaluate the linear spline at $x = 1$ and at $x = 2$.*

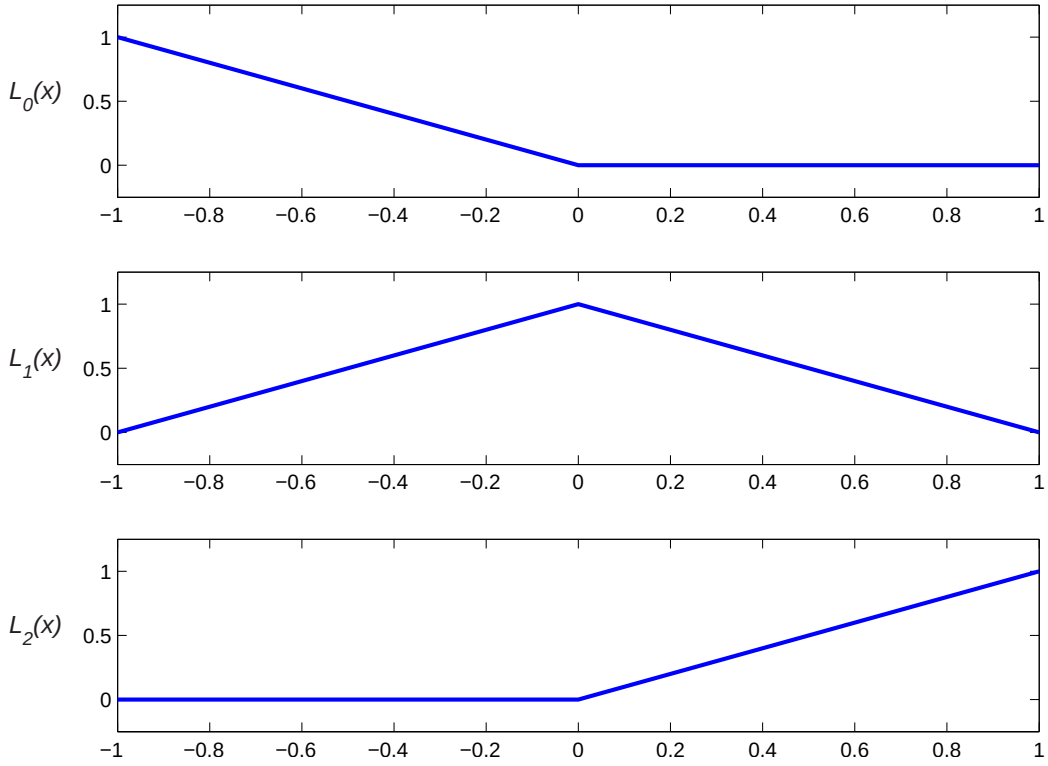

Figure 4.11: Linear B–spline basis functions for the data $(-1, 0), (0, 1)$ and $(1, 3)$. Each $L_i(x)$ is a "roof" shaped function with the apex of the roof at $(x_i, 1)$.

4.3.2 Cubic Polynomial Splines

Linear splines suffer from a major limitation: the derivative of a linear spline is generally discontinuous at each interior node, x_i. To derive a piecewise polynomial approximation with a continuous derivative requires that we use polynomial pieces of higher degree and constrain the pieces to make the curve smoother.

Before the days of Computer Aided Design, a (mechanical) spline, for example a flexible piece of wood, hard rubber, or metal, was used to help draw curves. To use a mechanical spline, pins were placed at a judicious selection of points along a curve in a design, then the spline was bent so that it touched each of these pins. Clearly, with this construction the spline *interpolates* the curve at these pins and could be used to reproduce the curve in other drawings[1]. The locations of the pins are called *knots*. We can change the shape of the curve defined by the spline by adjusting the location of the knots. For example, to interpolate to the data $\{(x_i, f_i)\}$ we can place knots at each of the nodes x_i. This produces a special interpolating cubic spline.

To derive a mathematical model of this mechanical spline, suppose the data is $\{(x_i, f_i)\}_{i=0}^{N}$ where, as for linear splines, $x_0 < x_1 < \cdots < x_N$. The shape of a mechanical spline suggests the curve between the pins is approximately a cubic polynomial. So, we model the mechanical spline by a mathematical cubic spline — a special piecewise cubic approximation. Mathematically, a cubic spline $S_{3,N}(x)$ is a C^2 (that is, twice continuously differentiable) piecewise cubic polynomial. This means that

- $S_{3,N}(x)$ is piecewise cubic; that is, between consecutive knots x_i

$$S_{3,N}(x) = \begin{cases} p_1(x) &= a_{1,0} + a_{1,1}x + a_{1,2}x^2 + a_{1,3}x^3 & x \in [x_0, x_1] \\ p_2(x) &= a_{2,0} + a_{2,1}x + a_{2,2}x^2 + a_{2,3}x^3 & x \in [x_1, x_2] \\ p_3(x) &= a_{3,0} + a_{3,1}x + a_{3,2}x^2 + a_{3,3}x^3 & x \in [x_2, x_3] \\ & \vdots & \\ p_N(x) &= a_{N,0} + a_{N,1}x + a_{N,2}x^2 + a_{N,3}x^3 & x \in [x_{N-1}, x_N] \end{cases}$$

  where $a_{i,0}$, $a_{i,1}$, $a_{i,2}$ and $a_{i,3}$ are the coefficients in the power series representation of the i^{th} cubic piece of $S_{3,N}(x)$. (Note: The approximation changes from one cubic polynomial piece to the next at the **knots** x_i.)

- $S_{3,N}(x)$ is C^2 (read C two); that is, $S_{3,N}(x)$ *is continuous and has continuous first and second derivatives* everywhere in the interval $[x_0, x_N]$ (and particularly at the knots).

To be an **interpolating cubic spline** we must have, in addition,

- $S_{3,N}(x)$ interpolates the data; that is,

$$S_{3,N}(x_i) = f_i, \quad i = 0, 1, \cdots, N$$

(Note: the points of interpolation $\{x_i\}_{i=0}^{N}$ are called **nodes** (pins), and we have *chosen* them to coincide with the knots.) For the mechanical spline, the knots where $S_{3,N}(x)$ changes shape and the nodes where $S_{3,N}(x)$ interpolates are the same. For the mathematical spline, it is traditional to place the knots at the nodes, as in the definition of $S_{3,N}(x)$. However, this placement is a choice and not a necessity.

Within each interval (x_{i-1}, x_i) the corresponding cubic polynomial $p_i(x)$ is continuous and has continuous derivatives of all orders. Therefore, $S_{3,N}(x)$ or one of its derivatives can be discontinuous

[1] Splines were used frequently to trace the plan of an airplane wing. A master template was chosen, placed on the material forming the rib of the wing, and critical points on the template were transferred to the material. After removing the template, the curve defining the shape of the wing was "filled-in" using a mechanical spline passing through the critical points.

only at a knot. For example, consider the following illustration of what happens at the knot x_i.

For $x_{i-1} < x < x_i$, $S_{3,N}(x)$ has value $p_i(x) = a_{i,0} + a_{i,1}x + a_{i,2}x^2 + a_{i,3}x^3$	For $x_i < x < x_{i+1}$, $S_{3,N}(x)$ has value $p_{i+1}(x) = a_{i+1,0} + a_{i+1,1}x + a_{i+1,2}x^2 + a_{i+1,3}x^3$
Value of $S_{3,N}(x)$ as $x \to x_i^-$	Value of $S_{3,N}(x)$ as $x \to x_i^+$
$p_i(x_i)$	$p_{i+1}(x_i)$
$p_i'(x_i)$	$p_{i+1}'(x_i)$
$p_i''(x_i)$	$p_{i+1}''(x_i)$
$p_i'''(x_i)$	$p_{i+1}'''(x_i)$

(Here, $x \to x_i^+$ means take the limit as $x \to x_i$ from above.) Observe that the function $S_{3,N}(x)$ has two cubic pieces incident to the interior knot x_i; to the left of x_i it is the cubic $p_i(x)$ while to the right it is the cubic $p_{i+1}(x)$. Thus, a necessary and sufficient condition for $S_{3,N}(x)$ to be continuous and have continuous first and second derivatives is for these two cubic polynomials incident at the interior knot to match in value, and in first and second derivative values. So, we have a set of Smoothness Conditions; that is, at each interior knot:

$$p_i'(x_i) = p_{i+1}'(x_i), \;\; p_i''(x_i) = p_{i+1}''(x_i), \quad i = 1, 2, \cdots, N-1$$

In addition, to interpolate the data we have a set of Interpolation Conditions; that is, on the i^{th} interval:

$$p_i(x_{i-1}) = f_{i-1}, \;\; p_i(x_i) = f_i, \quad i = 1, 2, \cdots, N$$

This way of writing the interpolation conditions also forces $S_{3,N}(x)$ to be continuous at the knots.

Each of the N cubic pieces has four unknown coefficients, so our description of the function $S_{3,N}(x)$ involves $4N$ unknown coefficients. Interpolation imposes $2N$ linear constraints on the coefficients, and assuring continuous first and second derivatives imposes $2(N-1)$ additional linear constraints. (A linear constraint is a linear equation that must be satisfied by the coefficients of the polynomial pieces.) Therefore, there are a total of $4N - 2 = 2N + 2(N-1)$ linear constraints on the $4N$ unknown coefficients. In order to have the same number of equations as unknowns, we need 2 more (linear) constraints and the whole set of constraints must be linearly independent.

Natural Boundary Conditions

A little thought about the mechanical spline as it is forced to touch the pins indicates why two constraints are missing. What happens to the spline before it touches the first pin and after it touches the last? If you twist the spline at its ends you find that its shape changes. A natural condition is to let the spline rest freely without stress or tension at the first and last knot, that is don't twist it at the ends. Such a spline has "minimal energy". Mathematically, this condition is expressed as the Natural Spline Condition:

$$p_1''(x_0) = 0, \;\; p_N''(x_N) = 0$$

The so-called **natural spline** results when these conditions are used as the 2 missing linear constraints.

Despite its comforting name and easily understood physical origin, the natural spline is seldom used since it does not deliver an accurate approximation $S_{3,N}(x)$ near the ends of the interval $[x_0, x_N]$. This may be anticipated from the fact that we are forcing a zero value on the second derivative when this is not normally the value of the second derivative of the function that the data measures. A natural cubic spline is built up from cubic polynomials, so it is reasonable to expect that if the data is measured from a cubic polynomial then the natural cubic spline will reproduce the cubic polynomial. However, for example, if the data are measured from the function $f(x) = x^2$ then the natural spline $S_{3,N}(x) \neq f(x)$; the function $f(x) = x^2$ has nonzero second derivatives at the

nodes x_0 and x_N where value of the second derivative of the natural cubic spline $S_{3,N}(x)$ is zero by definition. In fact the error behaves like $O(h^2)$ where h is the largest distance between interpolation points. Since it can be shown that the best possible error behavior is $O(h^4)$, the accuracy of the natural cubic spline leaves something to be desired.

Second Derivative Conditions

To clear up the inaccuracy problem associated with the natural spline conditions we could replace them with the correct second derivative values

$$p_1''(x_0) = f''(x_0), \;\; p_N''(x_N) = f''(x_N).$$

These second derivatives of the data are not usually available but they can be replaced by sufficiently accurate approximations. If exact values or sufficiently accurate approximations are used then the resulting spline will be as accurate as possible for a cubic spline; that is the error in the spline will behave like $O(h^4)$ where h is the largest distance between interpolation points. (Approximations to the second derivative may be obtained by using polynomial interpolation. That is, two separate sets of data values near each of the endpoints of the interval $[x_0, x_N]$ are used to construct two interpolating polynomials. Then the two interpolating polynomials are each twice differentiated and the resulting twice differentiated polynomials are evaluated at the corresponding endpoints to approximate the values of $f''(x_0)$ and $f''(x_N)$.)

First Derivative (Slope) Conditions

Another choice of boundary conditions which delivers the full $O(h^4)$ accuracy of cubic splines is to use the correct fist derivative values

$$p_1'(x_0) = f'(x_0), \;\; p_N'(x_N) = f'(x_N).$$

If we do not have access to the derivative of f, we can approximate it in a similar way to that described above for the second derivative.

Not-a-knot Conditions

A simpler, and accurate, spline may be determined by replacing the boundary conditions with the so-called **not-a-knot** conditions. Recall, at each knot, the spline $S_{3,N}(x)$ changes from one cubic to the next. The idea of the not-a-knot conditions is *not to change* cubic polynomials as one crosses both the first and the last *interior* nodes, x_1 and x_{N-1}. [Then, x_1 and x_{N-1} are no longer knots!] These conditions are expressed mathematically as the Not-a-Knot Conditions

$$p_1'''(x_1) = p_2'''(x_1), \;\; p_{N-1}'''(x_{N-1}) = p_N'''(x_{N-1}).$$

By construction, the first two pieces, $p_1(x)$ and $p_2(x)$, of the cubic spline $S_{3,N}(x)$ agree in value, as well as in first and second derivative at x_1. If $p_1(x)$ and $p_2(x)$ also satisfy the not-a-knot condition at x_1, it follows that $p_1(x) \equiv p_2(x)$; that is, x_1 is no longer a knot. The accuracy of this approach is also $O(h^4)$ but the error may still be quite large near the ends of the interval.

Cubic Spline Accuracy

For each way of supplying the additional linear constraints discussed above, the system of $4N$ linear constraints has a unique solution as long as the knots are distinct. So, the cubic spline interpolating function constructed using any one of the natural, the correct endpoint first or second derivative value, an approximated endpoint first or second derivative value, or the not-a-knot conditions is unique.

This uniqueness result permits an estimate of the error associated with approximations by cubic splines. From the error bound for polynomial interpolation, for a cubic polynomial $p_3(x)$ interpolating at data points in the interval $[a, b]$, we have

$$\max_{x\in[x_{i-1},x_i]} |f(x) - p_3(x)| \leq Ch^4 \cdot \max_{x\in[a,b]} |f^{(4)}(x)|$$

where C is a constant and $h = \max_i |x_i - x_{i-1}|$. We might anticipate that the error associated with approximation by a cubic spline behave like h^4 for h small, as for an interpolating cubic polynomial. However, the maximum absolute error associated with the natural cubic spline approximation behaves like h^2 as $h \to 0$. In contrast, the maximum absolute error for a cubic spline based on correct endpoint first or second derivative values or on the not-a-knot conditions behaves like h^4. Unlike the natural cubic spline, the correct first and second derivative value and not-a-knot cubic splines reproduce cubic polynomials. That is, in both these cases, $S_{3,N} \equiv f$ on the interval $[a, b]$ whenever the data values are measured from a cubic polynomial f. This reproducibility property is a necessary condition for the error in the cubic spline $S_{3,N}$ approximation to a general function f to behave like h^4.

B–splines

Codes that work with cubic splines do not use the power series representation of $S_{3,N}(x)$. Rather, often they represent the spline as a linear combination of cubic **B–splines**; this approach is similar to using a linear combination of the linear B–spline roof basis functions L_i to represent a linear spline. B–splines have **compact support**, that is they are non-zero only inside a set of contiguous subintervals just like the linear spline roof basis functions. So, the linear B–spline basis function, L_i, has support (is non-zero) over just the two contiguous intervals which combined make up the interval $[x_{i-1}, x_{i+1}]$, whereas the corresponding cubic B–spline basis function, B_i, has support (is non-zero) over four contiguous intervals which combined make up the interval $[x_{i-2}, x_{i+2}]$.

Construction of a B–spline

Assume the points x_i are equally spaced with spacing h. We'll construct $B_p(x)$ the cubic B–spline centered on x_p. We know already that $B_p(x)$ is a cubic spline that is identically zero outside the interval $[x_p-2h, x_p+2h]$ and has knots at x_p-2h, x_p-h, x_p, x_p+h, and x_p+2h. We'll normalize it at x_p by requiring $B_p(x_p) = 1$. So on the interval $[x_p-2h, x_p-h]$ we can choose $B_p(x) = A(x-[x_p-2h])^3$ where A is a constant to be determined later. This is continuous and has continuous first and second derivatives matching the zero function at the knot $x_p - 2h$. Similarly on the interval $[x_p + h, x_p + 2h]$ we can choose $B_p(x) = -A(x - [x_p + 2h])^3$ where A will turn out to be the same constant by symmetry. Now, we need $B_p(x)$ to be continuous and have continuous first and second derivatives at the knot $x_p - h$. This is achieved by choosing $B_p(x) = A(x - [x_p - 2h])^3 + B(x - [x_p - h])^3$ on the interval $[x_p - h, x_p]$ and similarly $B_p(x) = -A(x - [x_p + 2h])^3 - B(x - [x_p + h])^3$ on the interval $[x_p, x_p + h]$ where again symmetry ensures the same constants. Now, all we need to do is to fix up the constants A and B to give the required properties at the knot $x = x_p$. Continuity and the requirement $B_p(x_p) = 1$ give

$$A(x_p - [x_p - 2h])^3 + B(x_p - [x_p - h])^3 = -A(x_p - [x_p + 2h])^3 - B(x_p - [x_p + h])^3 = 1$$

that is

$$8h^3A + h^3B = -8(-h)^3A - (-h)^3B = 1$$

which gives one equation, $8A + B = \frac{1}{h^3}$, for the two constants A and B. Now first derivative continuity at the knot $x = x_p$ gives

$$3A(x_p - [x_p - 2h])^2 + 3B(x_p - [x_p - h])^2 = -3A(x_p - [x_p + 2h])^2 - 3B(x_p - [x_p + h])^2$$

After cancelation, this reduces to $4A + B = -4A - B$. The second derivative continuity condition gives an automatically satisfied identity. So, solving we have $B = -4A$. Hence $A = \frac{1}{4h^3}$ and

$B = -\frac{1}{h^3}$. So,

$$B_p(x) = \begin{cases} 0, & x < x_p - 2h \\ \frac{1}{4h^3}(x - [x_p - 2h])^3, & x_p - 2h \le x < x_p - h \\ \frac{1}{4h^3}(x - [x_p - 2h])^3 - \frac{1}{h^3}(x - [x_p - h])^3, & x_p - h \le x < x_p \\ -\frac{1}{4h^3}(x - [x_p + 2h])^3 + \frac{1}{h^3}(x - [x_p + h])^3, & x_p \le x < x_p + h \\ -\frac{1}{4h^3}(x - [x_p + 2h])^3, & x_p + h \le x < x_p + 2h \\ 0, & x \ge x_p + 2h \end{cases}$$

Interpolation using Cubic B–splines

Suppose we have data $\{(x_i, f_i)\}_{i=0}^n$ and the points x_i are equally spaced so that $x_i = x_0 + ih$. Define the "exterior" equally spaced points x_{-i}, x_{n+i}, $i = 1, 2, 3$ then these are all the points we need to define the B–splines $B_i(x)$, $i = -1, 0, \ldots, n+1$. This is the B–spline basis; that is, the set of all B–splines which are nonzero in the interval $[x_0, x_n]$. We seek a B–spline interpolating function of the form $S_n(x) = \sum_{i=-1}^{n+1} a_i B_i(x)$. The interpolation conditions give

$$\sum_{i=-1}^{n+1} a_i B_i(x_j) = f_j,\ j = 0, 1, \ldots, n$$

which simplifies to

$$a_{j-1}B_{j-1}(x_j) + a_j B_j(x_j) + a_{j+1}B_{j+1}(x_j) = f_j,\ j = 0, 1, \ldots, n$$

as all other terms in the sum are zero at $x = x_j$. Now, by definition $B_j(x_j) = 1$, and we compute $B_{j-1}(x_j) = B_{j+1}(x_j) = \frac{1}{4}$ by evaluating the above expression for the B–spline, giving the equations

$$\begin{aligned} \tfrac{1}{4}a_{-1} + a_0 + \tfrac{1}{4}a_1 &= f_0 \\ \tfrac{1}{4}a_0 + a_1 + \tfrac{1}{4}a_2 &= f_1 \\ &\vdots \\ \tfrac{1}{4}a_{n-1} + a_n + \tfrac{1}{4}a_{n+1} &= f_n \end{aligned}$$

These are $n+1$ equations in the $n+3$ unknowns a_j, $j = -1, 0, \ldots, n+1$. The additional equations come from applying the boundary conditions. For example, if we apply the natural spline conditions $S''(x_0) = S''(x_n) = 0$ we get the two additional equations

$$\begin{aligned} \tfrac{3}{2h^2}a_{-1} - \tfrac{3}{h^2}a_0 + \tfrac{3}{2h^2}a_1 &= 0 \\ \tfrac{3}{2h^2}a_{n-1} - \tfrac{3}{h^2}a_n + \tfrac{3}{2h^2}a_{n+1} &= 0 \end{aligned}$$

The full set of $n+3$ linear equations may be solved by Gaussian elimination but we can simplify the equations. Taking the first of the additional equations and the first of the previous set together we get $a_0 = \frac{2}{3}f_0$; similarly, from the last equations we find $a_n = \frac{2}{3}f_n$. So the set of linear equations reduces to

$$\begin{aligned} a_1 + \tfrac{1}{4}a_2 &= f_1 - \tfrac{1}{6}f_0 \\ \tfrac{1}{4}a_1 + a_2 + \tfrac{1}{4}a_3 &= f_2 \\ \tfrac{1}{4}a_2 + a_3 + \tfrac{1}{4}a_4 &= f_3 \\ &\vdots \\ \tfrac{1}{4}a_{n-3} + a_{n-2} + \tfrac{1}{4}a_{n-1} &= f_{n-2} \\ \tfrac{1}{4}a_{n-2} + a_{n-1} &= f_{n-1} - \tfrac{1}{6}f_n \end{aligned}$$

The coefficient matrix of this linear system is

$$\begin{bmatrix} 1 & \frac{1}{4} & & & \\ \frac{1}{4} & 1 & \frac{1}{4} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{1}{4} & 1 & \frac{1}{4} \\ & & & \frac{1}{4} & 1 \end{bmatrix}$$

Matrices of this structure are called tridiagonal and this particular tridiagonal matrix is of a special type known as positive definite. For this type of matrix interchanges are not needed for the stability of Gaussian elimination. When interchanges are not needed for a tridiagonal system, Gaussian elimination reduces to a particularly simple algorithm. After we have solved the linear equations for $a_1, a_2, \ldots, a_{n-1}$ we can compute the values of a_{-1} and a_{n+1} from the "additional" equations above.

Problem 4.3.3. *Let $r(x) = r_0 + r_1x + r_2x^2 + r_3x^3$ and $s(x) = s_0 + s_1x + s_2x^2 + s_3x^3$ be cubic polynomials in x. Suppose that the value, first, second, and third derivatives of $r(x)$ and $s(x)$ agree at some point $x = a$. Show that $r_0 = s_0$, $r_1 = s_1$, $r_2 = s_2$, and $r_3 = s_3$, i.e., $r(x)$ and $s(x)$ are the same cubic. [Note: This is another form of the polynomial uniqueness theorem.]*

Problem 4.3.4. *Write down the equations determining the coefficients of the not-a-knot cubic spline interpolating to the data $(0, 1)$, $(1, 0)$, $(2, 3)$ and $(3, 2)$. Just four equations are sufficient. Why?*

Problem 4.3.5. *Let the knots be at the integers, i.e. $x_i = i$, so $B_0(x)$ has support on the interval $[-2, +2]$. Construct $B_0(x)$ so that it is a cubic spline normalized so that $B_0(0) = 1$. [Hint: Since $B_0(x)$ is a cubic spline it must be piecewise cubic and it must be continuous and have continuous first and second derivatives at all the knots, and particularly at the knots $-2, -1, 0, +1, +2$.]*

Problem 4.3.6. *In the derivation of the linear system for B–spline interpolation replace the equations corresponding to the natural boundary conditions by equations corresponding to (a) exact second derivative conditions and (b) knot-a-knot conditions. In both cases use these equations to eliminate the coefficients a_{-1} and a_{n+1} and write down the structure of the resulting linear system.*

Problem 4.3.7. *Is the following function $S(x)$ a cubic spline? Why or why not?*

$$S(x) = \begin{cases} 0, & x < 0 \\ x^3, & 0 \le x < 1 \\ x^3 + (x-1)^3, & 1 \le x < 2 \\ -(x-3)^3 - (x-4)^3, & 2 \le x < 3 \\ -(x-4)^3, & 3 \le x < 4 \\ 0, & 4 \le x \end{cases}$$

4.3.3 Monotonic Piecewise Cubic Polynomials

Another approach to interpolation with piecewise cubic polynomials is, in addition to interpolating the data, to attempt to preserve a qualitative property exhibited by the data. For example, the piecewise cubic polynomial could be chosen in such a way that it is monotonic in the intervals between successive monotonic data values. In the simplest case where the data values are nondecreasing (or non-increasing) throughout the interval then the fitted function is forced to have the same property at all points in the interval. The *piecewise cubic Hermite interpolating polynomial* (PCHIP) achieves monotonicity by choosing the values of the derivative at the data points so that the function is non-decreasing, that is with derivative nonnegative (or is non-increasing, that is derivative non-positive) throughout the interval. The resulting piecewise cubic polynomial will usually have discontinuous second derivatives at the data points and will hence be less smooth and a little less accurate for

smooth functions than the cubic spline interpolating the same data. However, forcing monotonicity when it is present in the data has the effect of preventing oscillations and overshoot which can occur with a spline. For an illustration of this, see Examples 4.5.10 and 4.5.11 in Section 4.5.3.

It would be wrong, though, to think that because we don't enforce monotonicity directly when constructing a cubic spline that we don't get it in practice. When the data points are sufficiently close (that is, h is sufficiently small) and the function represented is smooth, the cubic spline is a very accurate approximation to the function that the data represents and any properties, such as monotonicity, are preserved.

4.4 Least Squares Fitting

In previous sections we determined an approximation of $f(x)$ by interpolating to the data $\{(x_i, f_i)\}_{i=0}^{N}$. An alternative to approximation via interpolation is approximation via a least squares fit.

Let $q_M(x)$ be a polynomial of degree M. Observe that $q_M(x_r) - f_r$ is the error in accepting $q_M(x_r)$ as an approximation to f_r. So, the sum of the squares of these errors

$$\sigma(q_M) \equiv \sum_{r=0}^{N} \{q_M(x_r) - f_r\}^2$$

gives a measure of how well $q_M(x)$ fits $f(x)$. The idea is that the smaller the value of $\sigma(q_M)$, the closer the polynomial $q_M(x)$ fits the data.

We say $p_M(x)$ is a least squares polynomial of degree M if $p_M(x)$ is a polynomial of degree M with the property that

$$\sigma(p_M) \leq \sigma(q_M)$$

for all polynomials $q_M(x)$ of degree M; usually we only have equality if $q_M(x) \equiv p_M(x)$. For simplicity, we write σ_M in place of $\sigma(p_M)$. As shown in an advanced course in numerical analysis, if the points $\{x_r\}$ are distinct and if $N \geq M$ there is one and only one least squares polynomial of degree M for this data, so we say $p_M(x)$ is the least squares polynomial of degree M. So, the polynomial $p_M(x)$ that produces the smallest value σ_M yields the least squares fit of the data.

While $p_M(x)$ produces the closest fit of the data in the least squares sense, it may not produce a very useful fit. For, consider the case $M = N$ then the least squares fit $p_N(x)$ is the same as the interpolating polynomial; see Problem 4.4.1. We have seen already that the interpolating polynomial can be a poor fit in the sense of having a large and highly oscillatory error. So, a close fit in a least squares sense does not necessarily imply a very good fit and, in some cases, the closer the fit is to an interpolating function the less useful it might be.

Since the least squares criterion relaxes the fitting condition from interpolation to a weaker condition on the coefficients of the polynomial, we need fewer coefficients (that is, a lower degree polynomial) in the representation. For the problem to be well–posed, it is sufficient that all the data points be distinct and that $M \leq N$.

Example 4.4.1. How is $p_M(x)$ determined for polynomials of each degree M? Consider the example:

- data:

i	x_i	f_i
0	1	2
1	3	4
2	4	3
3	5	1

- least squares fit to this data by a straight line: $p_1(x) = a_0 + a_1 x$; that is, the coefficients a_0 and a_1 are to be determined.

We have

$$\sigma_1 = \sum_{r=0}^{3} \{p_1(x_r) - f_r\}^2 = \sum_{r=0}^{3} \{a_0 + a_1 x_r - f_r\}^2$$

Multivariate calculus provides a technique to identify the values of a_0 and a_1 that make σ_1 smallest; for a minimum of σ_1, the unknowns a_0 and a_1 must satisfy the linear equations

$$\begin{aligned} \frac{\partial}{\partial a_0}\sigma_1 &\equiv \sum_{r=0}^{3} 2\{a_0 + a_1 x_r - f_r\} &= 2(4a_0 + 13a_1 - 10) &= 0 \\ \frac{\partial}{\partial a_1}\sigma_1 &\equiv \sum_{r=0}^{3} 2x_r\{a_0 + a_1 x_r - f_r\} &= 2(13a_0 + 51a_1 - 31) &= 0 \end{aligned}$$

that is, in matrix form after canceling the 2's throughout,

$$\begin{bmatrix} 4 & 13 \\ 13 & 51 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 10 \\ 31 \end{bmatrix}$$

Gaussian elimination followed by backward substitution computes the solution

$$a_0 = \frac{107}{35} \simeq 3.057$$

$$a_1 = -\frac{6}{35} \simeq -0.171$$

Substituting a_0 and a_1 gives the minimum value of $\sigma_1 = \dfrac{166}{35}$.

Generally, if we consider fitting data using a polynomial written in power series form

$$p_M(x) = a_0 + a_1 x + \cdots + a_M x^M$$

then σ_M is quadratic in the unknown coefficients $a_0, a_1, \cdots, a_M$. For data $\{(x_i, f_i)\}_{i=0}^{N}$ we have

$$\begin{aligned} \sigma_M &= \sum_{r=0}^{N} \{p_M(x_r) - f_r\}^2 \\ &= \{p_M(x_0) - f_0\}^2 + \{p_M(x_1) - f_1\}^2 + \cdots + \{p_M(x_N) - f_N\}^2 \end{aligned}$$

The coefficients $a_0, a_1, \cdots, a_M$ are determined by solving the linear system

$$\begin{aligned} \frac{\partial}{\partial a_0}\sigma_M &= 0 \\ \frac{\partial}{\partial a_1}\sigma_M &= 0 \\ &\vdots \\ \frac{\partial}{\partial a_M}\sigma_M &= 0 \end{aligned}$$

For each value $j = 0, 1, \cdots, M$, the linear equation $\dfrac{\partial}{\partial a_j}\sigma_M = 0$ is formed as follows. Observe that

$\dfrac{\partial}{\partial a_j} p_M(x_r) = x_r^j$ so, by the chain rule,

$$\begin{aligned}
\frac{\partial}{\partial a_j}\sigma_M &= \frac{\partial}{\partial a_j}\left[\{f_0 - p_M(x_0)\}^2 + \{f_1 - p_M(x_1)\}^2 + \cdots + \{f_N - p_M(x_N)\}^2\right] \\
&= 2\left[\{p_M(x_0) - f_0\}\frac{\partial p_M(x_0)}{\partial a_j} + \{p_M(x_1) - f_1\}\frac{\partial p_M(x_1)}{\partial a_j} + \cdots + \{p_M(x_N) - f_N\}\frac{\partial p_M(x_N)}{\partial a_j}\right] \\
&= 2\left[\{p_M(x_0) - f_0\}x_0^j + \{p_M(x_1) - f_1\}x_1^j + \cdots + \{p_M(x_N) - f_N\}x_N^j\right] \\
&= 2\sum_{r=0}^{N}\{p_M(x_r) - f_r\}x_r^j \\
&= 2\sum_{r=0}^{N} x_r^j\, p_M(x_r) - 2\sum_{r=0}^{N} f_r\, x_r^j.
\end{aligned}$$

Substituting for the polynomial $p_M(x_r)$ the power series form leads to

$$\begin{aligned}
\frac{\partial}{\partial a_j}\sigma_M &= 2\left(\textstyle\sum_{r=0}^{N} x_r^j\left\{a_0 + a_1 x_r + \cdots + a_M x_r^M\right\} - \sum_{r=0}^{N} f_r\, x_r^j\right) \\
&= 2\left(a_0\textstyle\sum_{r=0}^{N} x_r^j + a_1\sum_{r=0}^{N} x_r^{j+1} + \cdots + a_M\sum_{r=0}^{N} x_r^{j+M} - \sum_{r=0}^{N} f_r\, x_r^j\right)
\end{aligned}$$

Therefore, $\dfrac{\partial}{\partial a_j}\sigma_M = 0$ may be rewritten as the **Normal equations**:

$$a_0\sum_{r=0}^{N} x_r^j + a_1\sum_{r=0}^{N} x_r^{j+1} + \cdots + a_M\sum_{r=0}^{N} x_r^{j+M} = \sum_{r=0}^{N} f_r\, x_r^j, \quad j = 0, 1, \cdots, M$$

In matrix form the Normal equations may be written

$$\begin{bmatrix} \sum_{r=0}^{N} 1 & \sum_{r=0}^{N} x_r & \cdots & \sum_{r=0}^{N} x_r^M \\ \sum_{r=0}^{N} x_r & \sum_{r=0}^{N} x_r^2 & \cdots & \sum_{r=0}^{N} x_r^{M+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{r=0}^{N} x_r^M & \sum_{r=0}^{N} x_r^{M+1} & \cdots & \sum_{r=0}^{N} x_r^{2M} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{bmatrix} = \begin{bmatrix} \sum_{r=0}^{N} f_r \\ \sum_{r=0}^{N} f_r x_r \\ \vdots \\ \sum_{r=0}^{N} f_r x_r^M \end{bmatrix}$$

The coefficient matrix of the Normal equations has special properties (it is both symmetric and positive definite). These properties permit the use of an accurate, efficient version of Gaussian elimination which exploits these properties, without the need for partial pivoting by rows for size.

Example 4.4.2. To compute a straight line fit $a_0 + a_1 x$ to the data $\{(x_i, f_i)\}_{i=0}^{N}$ we set $M = 1$ in the Normal equations to give

$$\begin{aligned}
a_0\sum_{r=0}^{N} 1 + a_1\sum_{r=0}^{N} x_r &= \sum_{r=0}^{N} f_r \\
a_0\sum_{r=0}^{N} x_r + a_1\sum_{r=0}^{N} x_r^2 &= \sum_{r=0}^{N} f_r x_r
\end{aligned}$$

Substituting the data

i	x_i	f_i
0	1	2
1	3	4
2	4	3
3	5	1

from Example 4.4.1 we have the Normal equations

$$\begin{aligned} 4a_0 + 13a_1 &= 10 \\ 13a_0 + 51a_1 &= 31 \end{aligned}$$

which gives the same result as in Example 4.4.1.

Example 4.4.3. To compute a quadratic fit $a_0 + a_1 x + a_2 x^2$ to the data $\{(x_i, f_i)\}_{i=0}^{N}$ we set $M = 2$ in the Normal equations to give

$$\begin{aligned} a_0 \sum_{r=0}^{N} 1 + a_1 \sum_{r=0}^{N} x_r + a_2 \sum_{r=0}^{N} x_r^2 &= \sum_{r=0}^{N} f_r \\ a_0 \sum_{r=0}^{N} x_r + a_1 \sum_{r=0}^{N} x_r^2 + a_2 \sum_{r=0}^{N} x_r^3 &= \sum_{r=0}^{N} f_r x_r \\ a_0 \sum_{r=0}^{N} x_r^2 + a_1 \sum_{r=0}^{N} x_r^3 + a_2 \sum_{r=0}^{N} x_r^4 &= \sum_{r=0}^{N} f_r x_r^2 \end{aligned}$$

The least squares formulation permits more general functions $f_M(x)$ than simply polynomials, but the unknown coefficients in $f_M(x)$ must still occur linearly. The most general form is

$$f_M(x) = a_0\phi_0(x) + a_1\phi_1(x) + \cdots + a_M\phi_M(x) = \sum_{r=0}^{M} a_r\phi_r(x)$$

with a linearly independent basis $\{\phi_r(x)\}_{r=0}^{M}$. By analogy with the power series case, the linear system of Normal equations is

$$a_0 \sum_{r=0}^{N} \phi_0(x_r)\phi_j(x_r) + a_1 \sum_{r=0}^{N} \phi_1(x_r)\phi_j(x_r) + \cdots + a_M \sum_{r=0}^{N} \phi_M(x_r)\phi_j(x_r) = \sum_{r=0}^{N} f_r\phi_j(x_r)$$

for $j = 0, 1, \cdots, M$. The Normal equations are

$$\begin{bmatrix} \sum_{r=0}^{N} \phi_0(x_r)^2 & \sum_{r=0}^{N} \phi_0(x_r)\phi_1(x_r) & \cdots & \sum_{r=0}^{N} \phi_0(x_r)\phi_M(x_r) \\ \sum_{r=0}^{N} \phi_1(x_r)\phi_0(x_r) & \sum_{r=0}^{N} \phi_1(x_r)^2 & \cdots & \sum_{r=0}^{N} \phi_1(x_r)\phi_M(x_r) \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{r=0}^{N} \phi_M(x_r)\phi_0(x_r) & \sum_{r=0}^{N} \phi_M(x_r)\phi_1(x_r) & \cdots & \sum_{r=0}^{N} \phi_M(x_r)^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_M \end{bmatrix} = \begin{bmatrix} \sum_{r=0}^{N} f_r\phi_0(x_r) \\ \sum_{r=0}^{N} f_r\phi_1(x_r) \\ \vdots \\ \sum_{r=0}^{N} f_r\phi_M(x_r) \end{bmatrix}$$

The coefficient matrix of this linear system is symmetric and potentially ill–conditioned. In particular, the basis functions ϕ_j could be, for example, a linear polynomial spline basis, a cubic polynomial B–spline basis, Chebyshev polynomials in a Chebyshev series fit, or a set of linearly independent trigonometric functions.

The coefficient matrix of the Normal equations is usually reasonably well-conditioned when the number, M, of functions being fitted is small. The ill–conditioning grows with the number of functions. To avoid the possibility of ill–conditioning the Normal equations are not usually formed but instead a stable QR factorization of a related matrix is employed to compute the least squares solution directly. Further discussion of this approach is beyond the scope of this book. QR factorization is the approach taken by MATLAB's `polyfit` function.

Problem 4.4.1. *Consider the data* $\{(x_i, f_i)\}_{i=0}^{N}$. *Argue why the interpolating polynomial of degree* N *is also the least squares polynomial of degree* N. *Hint: What is the value of* $\sigma(q_N)$ *when* $q_N(x)$ *is the interpolating polynomial?*

Problem 4.4.2. *Show that* $\sigma_0 \geq \sigma_1 \geq \cdots \geq \sigma_N = 0$ *for any set of data* $\{(x_i, f_i)\}_{i=0}^{N}$. *Hint: The proof follows from the concept of minimization.*

Problem 4.4.3. *Find the least squares constant fit* $p_0(x) = a_0$ *to the data in Example 4.4.1. Plot the data, and both the constant and the linear least squares fits on one graph.*

Problem 4.4.4. *Find the least squares linear fit* $p_1(x) = a_0 + a_1 x$ *to the following data. Explain why you believe your answer is correct?*

i	x_i	f_i
0	1	1
1	3	1
2	4	1
3	5	1
4	7	1

Problem 4.4.5. *Find the least squares quadratic polynomial fits* $p_2(x) = a_0 + a_1 x + a_2 x^2$ *to each of the data sets:*

i	x_i	f_i
0	-2	6
1	-1	3
2	0	1
3	1	3
4	2	6

and

i	x_i	f_i
0	-2	-5
1	-1	-3
2	0	0
3	1	3
4	2	5

Problem 4.4.6. *Use the chain rule to derive the Normal equations for the general basis functions* $\{\phi_j\}_{j=0}^{M}$.

Problem 4.4.7. *Write down the Normal equations for the following choice of basis functions:* $\phi_0(x) = 1$, $\phi_1(x) = \sin(\pi x)$ *and* $\phi_2(x) = \cos(\pi x)$. *Find the coefficients* a_0, a_1 *and* a_2 *for a least squares fit to the data*

i	x_i	f_i
0	-1	-5
1	-0.5	-3
2	0	0
3	0.5	3
4	1	5

Problem 4.4.8. *Write down the Normal equations for the following choice of basis functions:* $\phi_0(x) = T_0(x)$, $\phi_1(x) = T_1(x)$ *and* $\phi_2(x) = T_2(x)$. *Find the coefficients* a_0, a_1 *and* a_2 *for a least squares fit to the data in the Problem 4.4.7.*

4.5 Matlab Notes

MATLAB has several functions designed specifically for manipulating polynomials, and for curve fitting. Some functions that are relevant to the topics discussed in this chapter include:

`polyfit`	used to create the coefficients of an interpolating or least squares polynomial
`polyval`	used to evaluate a polynomial
`spline`	used to create, and evaluate, an interpolatory cubic spline
`pchip`	used to create, and evaluate, a piecewise cubic Hermite interpolating polynomial
`interp1`	used to create, and evaluate, a variety of piecewise interpolating polynomials, including linear and cubic splines, and piecewise cubic Hermite polynomials
`ppval`	used to evaluate a piecewise polynomial, such as given by `spline`, `pchip` or `interp1`

In general, these functions assume a canonical representation of the power series form of a polynomial to be

$$p(x) = a_1 x^N + a_2 x^{N-1} + \cdots + a_N x + a_{N+1}.$$

Note that this is slightly different than the notation used in Section 4.1, but in either case, all that is needed to represent the polynomial is a vector of coefficients. Using MATLAB's canonical form, the vector representing $p(x)$ is:

$$a = \left[\; a_1; \quad a_2; \quad \cdots \quad a_N; \quad a_{N+1} \; \right].$$

Note that although a could be a row or a column vector, we will generally use column vectors.

Example 4.5.1. Consider the polynomial $p(x) = 7x^3 - x^2 + 1.5x - 3$. Then the vector of coefficients that represents this polynomial is given by:

```
>> a = [7; -1; 1.5; -3];
```

Similarly, the vector of coefficients that represents the polynomial $p(x) = 7x^5 - x^4 + 1.5x^2 - 3x$ is given by:

```
>> a = [7; -1; 0; 1.5; -3; 0];
```

Notice that it is important to explicitly include any zero coefficients when constructing the vector a.

4.5.1 Polynomial Interpolation

In Section 4.1, we constructed interpolating polynomials using three different forms: power series, Newton and Lagrange forms. MATLAB's main tool for polynomial interpolation, `polyfit`, uses the power series form. To understand how this function is implemented, suppose we are given data points

$$(x_1, f_1),\ (x_2, f_2),\ \ldots, (x_{N+1}, f_{N+1}).$$

Recall that to find the power series form of the (degree N) polynomial that interpolates this data, we need to find the coefficients, a_i, of the polynomial

$$p(x) = a_1 x^N + a_2 x^{N-1} + \cdots + a_N x + a_{N+1}$$

such that $p(x_i) = f_i$. That is,

$$\begin{aligned}
p(x_1) = f_1 \quad &\Rightarrow \quad a_1 x_1^N + a_2 x_1^{N-1} + \cdots + a_N x_1 + a_{N+1} = f_1 \\
p(x_2) = f_2 \quad &\Rightarrow \quad a_1 x_2^N + a_2 x_2^{N-1} + \cdots + a_N x_2 + a_{N+1} = f_2 \\
&\vdots \\
p(x_N) = f_N \quad &\Rightarrow \quad a_1 x_N^N + a_2 x_N^{N-1} + \cdots + a_N x_N + a_{N+1} = f_N \\
p(x_{N+1}) = f_{N+1} \quad &\Rightarrow \quad a_1 x_{N+1}^N + a_2 x_{N+1}^{N-1} + \cdots + a_N x_{N+1} + a_{N+1} = f_{N+1}
\end{aligned}$$

or, more precisely, we need to solve the linear system $Va = f$:

$$\begin{bmatrix} x_1^N & x_1^{N-1} & \cdots & x_1 & 1 \\ x_2^N & x_2^{N-1} & \cdots & x_2 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_N^N & x_N^{N-1} & \cdots & x_N & 1 \\ x_{N+1}^N & x_{N+1}^{N-1} & \cdots & x_{N+1} & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \\ a_{N+1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \\ f_{N+1} \end{bmatrix}.$$

In order to write a MATLAB function implementing this approach, we need to:

- Define vectors containing the given data:

  $\texttt{x} = \left[\; x_1; \quad x_2; \quad \cdots \quad x_{N+1} \; \right]$
  $\texttt{f} = \left[\; f_1; \quad f_2; \quad \cdots \quad f_{N+1} \; \right]$

- Let $n = \text{length}(x) = N + 1$.

- Construct the $n \times n$ matrix, V, which can be done one column at a time using the vector `x`:

  ```
  jth column of V = V(:, j) = x .^ (n-j)
  ```

- Solve the linear system, $Va = f$, using MATLAB's backslash operator:

  ```
  a = V \ f
  ```

Putting these steps together, we obtain the following function:

```
function a = InterpPow1(x, f)
%
%      a = InterpPow1(x, f);
%
% Construct the coefficients of a power series representation of the
% polynomial that interpolates the data points (x_i, f_i):
%
%   p = a(1)*x^N + a(2)*x^(N-1) + ... + a(N)*x + a(N+1)
%
n = length(x);
V = zeros(n, n);
for j = 1:n
  V(:, j) = x .^ (n-j);
end
a = V \ f;
```

We remark that MATLAB provides a function, `vander`, that can be used to construct the matrix V from a given vector x. Using `vander` in place of the first five lines of code in `InterpPow1`, we obtain the following function:

```
function a = InterpPow(x, f)
%
%      a = InterpPow(x, f);
%
% Construct the coefficients of a power series representation of the
% polynomial that interpolates the data points (x_i, f_i):
%
%   p = a(1)*x^N + a(2)*x^(N-1) + ... + a(N)*x + a(N+1)
%
V = vander(x);
a = V \ f;
```

Problem 4.5.1. *Implement* `InterpPow`*, and use it to find the power series form of the polynomial that interpolates the data* $(-1, 0)$, $(0, 1)$, $(1, 3)$. *Compare the results with that found in Example 4.1.3.*

Problem 4.5.2. *Implement* `InterpPow`*, and use it to find the power series form of the polynomial that interpolates the data* $(1, 2)$, $(3, 3)$, $(5, 4)$. *Compare the results with what you computed by hand in Problem 4.1.3.*

The built-in MATLAB function, `polyfit`, essentially uses the approach outlined above to construct an interpolating polynomial. The basic usage of `polyfit` is:

```
a = polyfit(x, f, N)
```

where `N` is the degree of the interpolating polynomial. In general, provided the x_i values are distinct, `N = length(x) - 1 = length(f) - 1`. As we see later, `polyfit` can be used for polynomial least squares data fitting by choosing a different (usually smaller) value for `N`.

Once the coefficients are computed, we may want to plot the resulting interpolating polynomial. Recall that to plot any function, including a polynomial, we must first evaluate it at many (e.g., 200) points. MATLAB provides a built-in function, `polyval`, that can be used to evaluate polynomials:

```
y = polyval(a, x);
```

Note that `polyval` requires the first input to be a vector containing the coefficients of the polynomial, and the second input a vector containing the values at which the polynomial is to be evaluated. At this point, though, we should be sure to distinguish between the (relatively few) "data points" used to construct the interpolating polynomial, and the (relatively many) "evaluation points" used for plotting. An example will help to clarify the procedure.

Example 4.5.2. Consider the data points $(-1, 0)$, $(0, 1)$, $(1, 3)$. First, plot the data using (red) circles, and set the axis to an appropriate scale:

```
x_data = [-1 0 1];
f_data = [0 1 3];
plot(x_data, f_data, 'ro')
axis([-2, 2, -1, 6])
```

Now we can construct and plot the polynomial that interpolates this data using the following set of MATLAB statements:

```
hold on
a = polyfit(x_data, f_data, length(x_data)-1);
x = linspace(-2,2,200);
y = polyval(a, x);
plot(x, y)
```

By including labels on the axes, and a legend (see Section 1.3.2):

```
xlabel('x')
ylabel('y')
legend('Data points','Interpolating polynomial', 'Location', 'NW')
```

we obtain the plot shown in Fig. 4.12. Note that we use different vectors to distinguish between the given data (`x_data` and `f_data`) and the set of points `x` and values `y` used to evaluate and plot the resulting interpolating polynomial.

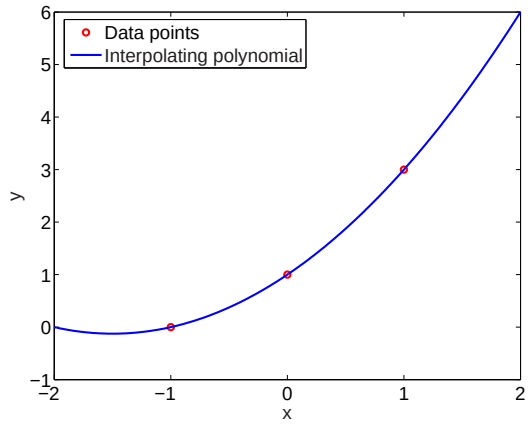

Figure 4.12: Plot generated by the MATLAB code in Example 4.5.2.

Although the power series approach usually works well for a small set of data points, one difficulty that can arise, especially when attempting to interpolate a large set of data, is that the matrix V may be very ill–conditioned. Recall that an ill–conditioned matrix is close to singular, in which case large errors can occur when solving $Va = f$. Thus, if V is ill–conditioned, the computed polynomial coefficients may be inaccurate. MATLAB's `polyfit` function checks V and displays a warning message if it detects it is ill–conditioned. In this case, we can try the alternative calling sequence of `polyfit` and `polyval`:

```
[a, s, mu] = polyfit(x_data, f_data, length(x_data)-1);
y = polyval(a, x, s, mu);
```

This forces `polyfit` to first scale `x_data` (using its mean and standard deviation) before constructing the matrix V and solving the corresponding linear system. This scaling usually results in a matrix that is better-conditioned.

Example 4.5.3. Consider the following set of data, obtained from the National Weather Service, `http://www.srh.noaa.gov/fwd`, which shows average high and low temperatures, total precipitation, and the number of clear days for each month in 2003 for Dallas-Fort Worth, Texas.

Monthly Weather Data, 2003
Dallas - Fort Worth, Texas

Month	1	2	3	4	5	6	7	8	9	10	11	12
Avg. High	54.4	54.6	67.1	78.3	85.3	88.7	96.9	97.6	84.1	80.1	68.8	61.1
Avg. Low	33.0	36.6	45.2	55.6	65.6	69.3	75.7	75.8	64.9	57.4	50.0	38.2
Precip.	0.22	3.07	0.85	1.90	2.53	5.17	0.08	1.85	3.99	0.78	3.15	0.96
Clear Days	15	6	10	11	4	9	13	10	11	13	7	18

Suppose we attempt to fit an interpolating polynomial to the average high temperatures. Our first attempt might use the following set of MATLAB commands:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
a = polyfit(x_data, f_data, 11);
x = linspace(1, 12, 200);
y = polyval(a, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

If we run these commands in MATLAB, then a warning message is printed in the command window indicating that V is ill–conditioned. If we replace the two lines containing `polyfit` and `polyval` with:

```
[a, s, mu] = polyfit(x_data, f_data, 11);
y = polyval(a, x, s, mu);
```

the warning no longer occurs. The resulting plot is shown in Fig. 4.13 (we also made use of the MATLAB commands `axis, legend, xlabel` and `ylabel`).

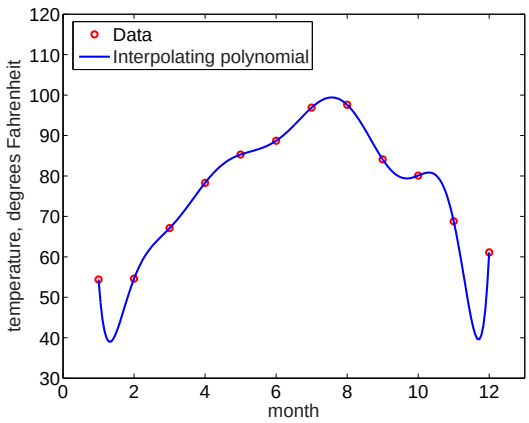

Figure 4.13: Interpolating polynomial from Example 4.5.3

Notice that the polynomial does not appear to provide a good model of the monthly temperature changes between months 1 and 2, and between months 11 and 12. This is a mild example of the more serious problem, discussed in Section 4.2, of excessive oscillation of the interpolating polynomial. A more extreme illustration of this is the following example.

Example 4.5.4. Suppose we wish to construct an interpolating polynomial approximation of the function $f(x) = \sin(x + \sin 2x)$ on the interval $[-\frac{\pi}{2}, \frac{3\pi}{2}]$. The following MATLAB code can be used to construct an interpolating polynomial approximation of $f(x)$ using 11 equally spaced points:

```
f = @(x) sin(x+sin(2*x));
x_data = linspace(-pi/2, 3*pi/2, 11);
f_data = f(x_data);
a = polyfit(x_data, f_data, 10);
```

A plot of the resulting polynomial is shown on the left of Fig. 4.14. Notice that, as with Runge's example (Example 4.2), the interpolating polynomial has severe oscillations near the end points of the interval. However, because we have an explicit function that can be evaluated, instead of using equally spaced points, we can choose to use the Chebyshev points. In MATLAB these points can be generated as follows:

```
c = -pi/2; d = 3*pi/2;
N = 10; I = 0:N;
x_data = (c+d)/2 - (d-c)*cos((2*I+1)*pi/(2*N+2))/2;
```

Notice that `I` is a vector containing the integers $0, 1, \ldots, 9$, and that we avoid using a loop to create `x_data` by making use of MATLAB's ability to operate on vectors. Using this `x_data`, and the corresponding `f_data = f(x_data)` to construct the interpolating polynomial, we can create the plot shown on the right of Fig. 4.14. We observe from this example that much better approximations can be obtained by using the Chebyshev points instead of equally spaced points.

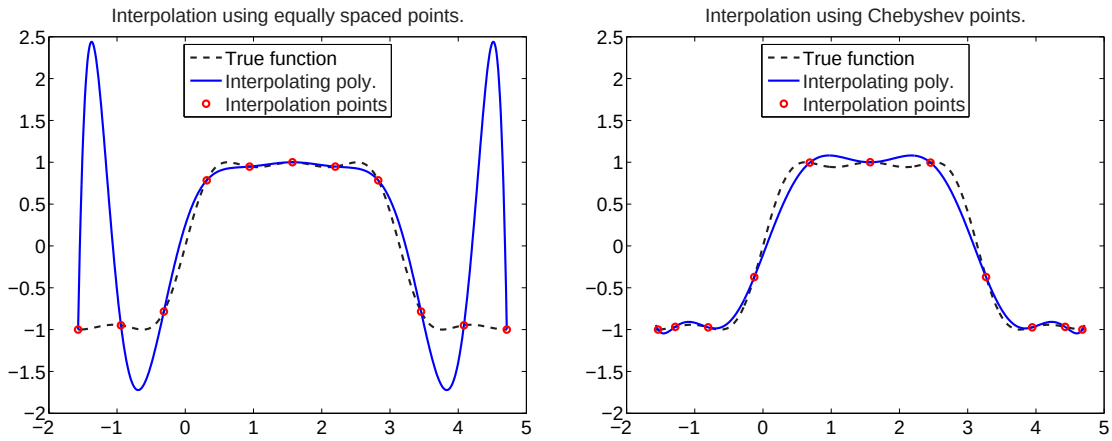

Figure 4.14: Interpolating polynomials for $f(x) = \sin(x + \sin 2x)$ on the interval $[-\frac{\pi}{2}, \frac{3\pi}{2}]$. The plot on the left uses 11 equally spaced points, and the plot on the right uses 11 Chebyshev points to generate the interpolating polynomial.

Problem 4.5.3. *Consider the data* $(0, 1)$, $(1, 2)$, $(3, 3)$ *and* $(5, 4)$. *Construct a plot that contains the data (as circles) and the polynomial of degree 3 that interpolates this data. You should use the* `axis` *command to make the plot look sufficiently nice.*

Problem 4.5.4. *Consider the data* $(1, 1)$, $(3, 1)$, $(4, 1)$, $(5, 1)$ *and* $(7, 1)$. *Construct a plot that contains the data (as circles) and the polynomial of degree 4 that interpolates this data. You should use the* `axis` *command to make the plot look sufficiently nice. Do you believe the curve is a good representation of the data?*

Problem 4.5.5. *Construct plots that contain the data (as circles) and interpolating polynomials for*

the following sets of data:

x_i	f_i
-2	6
-1	3
0	1
1	3
2	6

and

x_i	f_i
-2	-5
-1	-3
0	0
1	3
2	5

Problem 4.5.6. *Construct interpolating polynomials through all four sets of weather data given in Example 4.5.3. Use subplot to show all four plots in the same figure, and use the* `title, xlabel` *and* `ylabel` *commands to document the plots. Each plot should show the data points as circles on the corresponding curves.*

Problem 4.5.7. *Consider the function given in Example 4.5.4. Write a* MATLAB *script M–file to create the plots shown in Fig. 4.14. Experiment with using more points to construct the interpolating polynomial, starting with 11, 12, ... At what point does* MATLAB *print a warning that the polynomial is badly conditioned (both for equally spaced and Chebyshev points)? Does centering and scaling improve the results?*

4.5.2 Chebyshev Polynomials and Series

MATLAB does not provide specific functions to construct a Chebyshev representation of the interpolating polynomial. However, it is not difficult to write our own MATLAB functions similar to `polyfit`, `spline` and `pchip`. Recall that if $x \in [-1, 1]$, the jth Chebyshev polynomial, $T_j(x)$, is defined as

$$T_j(x) = \cos(j \arccos(x)),\ j = 0, 1, 2, ...$$

We can easily plot Chebyshev polynomials with just a few MATLAB commands.

Example 4.5.5. The following set of MATLAB commands can be used to plot the 5th Chebyshev polynomial.

```
x = linspace(-1, 1, 200);
y = cos( 5*acos(x) );
plot(x, y)
```

Recall that the Chebyshev form of the interpolating polynomial (for $x \in [-1, 1]$) is

$$p(x) = b_0 T_0(x) + b_1 T_1(x) + \cdots + b_N T_N(x).$$

Suppose we are given data (x_i, f_i), $i = 0, 1, \ldots, N$, and that $-1 \le x_i \le 1$. To construct Chebyshev form of the interpolating polynomial, we need to determine the coefficients $b_0, b_1, \ldots, b_N$ such that $p(x_i) = f_i$. That is,

$$\begin{aligned}
p(x_0) = f_0 \quad &\Rightarrow \quad b_0 T_0(x_0) + b_1 T_1(x_0) + \cdots + b_N T_N(x_0) = f_1 \\
p(x_1) = f_1 \quad &\Rightarrow \quad b_0 T_0(x_1) + b_1 T_1(x_1) + \cdots + b_N T_N(x_1) = f_2 \\
&\ \ \vdots \\
p(x_N) = f_N \quad &\Rightarrow \quad b_0 T_0(x_N) + b_1 T_1(x_N) + \cdots + b_N T_N(x_N) = f_N
\end{aligned}$$

or, more precisely, we need to solve the linear system $Tb = f$:

$$\begin{bmatrix} T_0(x_0) & T_1(x_0) & \cdots & T_N(x_0) \\ T_0(x_1) & T_1(x_1) & \cdots & T_N(x_1) \\ \vdots & \vdots & & \vdots \\ T_0(x_N) & T_1(x_N) & \cdots & T_N(x_N) \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_N \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_N \end{bmatrix}.$$

If the data x_i are not all contained in the interval $[-1, 1]$, then we must perform a variable transformation. For example,

$$\bar{x}_j = -\frac{x_j - x_{mx}}{x_{mn} - x_{mx}} + \frac{x_j - x_{mn}}{x_{mx} - x_{mn}} = \frac{2x_j - x_{mx} - x_{mn}}{x_{mx} - x_{mn}},$$

where $x_{mx} = \max(x_i)$ and $x_{mn} = \min(x_i)$. Notice that when $x_j = x_{mn}$, $\bar{x} = -1$ and when $x_j = x_{mx}$, $\bar{x} = 1$, and therefore $-1 \leq \bar{x}_j \leq 1$. The matrix T should then be constructed using $\bar{x}_j$. In this case, we need to use the same variable transformation when we evaluate the resulting Chebyshev form of the interpolating polynomial. For this reason, we write a function (such as `spline` and `pchip`) that can be used to construct **and** evaluate the interpolating polynomial.

The basic steps of our function can be outlined as follows:

- The input should be three vectors,

  `x_data` = vector containing given data, x_j.

  `f_data` = vector containing given data, f_j.

  `x` = vector containing points at which the polynomial is to be evaluated. Note that the values in this vector should be contained in the interval $[x_{mn}, x_{mx}]$.

- Perform a variable transformation on the entries in `x_data` to obtain $\bar{x}_j$, $-1 \leq \bar{x}_j \leq 1$, and define a new vector containing the transformed data:

  `x_data` = $\left[\ \bar{x}_0;\quad \bar{x}_1;\quad \cdots\quad \bar{x}_N\ \right]$

- Let $n =$ `length(x_data)` $= N + 1$.

- Construct the $n \times n$ matrix, T one column at a time using the recurrence relation for generating the Chebyshev polynomials and the (transformed) vector `x_data`:

  ```
  1st column of T = T(:,1) = ones(n,1)
  2nd column of T = T(:,2) = x_data
  ```

  and for $j = 3, 4, \ldots, n$,

  ```
  j^{\small th} column of T = T(:, j) = 2*x_data .* T(:,j-1) - T(:,j-2)
  ```

- Compute the coefficients using MATLAB's backslash operator:

  ```
  b = T \ f_data
  ```

- Perform the variable transformation of the entries in the vector `x`.

- Evaluate the polynomial at the transformed points.

Putting these steps together, we obtain the following function:

```
function y = chebfit(x_data, f_data, x)
%
%      y = chebfit(x_data, f_data, x);
%
% Construct and evaluate a Chebyshev representation of the
% polynomial that interpolates the data points (x_i, f_i):
%
%         p = b(1)*T_0(x) + b(1)*T_1(x) + ... + b(n)T_N(x)
%
% where n = N+1, and T_j(x) = cos(j*acos(x)) is the j^{\small th} Chebyshev
% polynomial.
%
n = length(x_data);
xmax = max(x_data);
xmin = min(x_data);
xx_data = (2*x_data - xmax - xmin)/(xmax - xmin);
T = zeros(n, n);
T(:,1) = ones(n,1);
T(:,2) = xx_data;
for j = 3:n
  T(:,j) = 2*xx_data.*T(:,j-1) - T(:,j-2);
end
b = T \ f_data;
xx = (2*x - xmax - xmin)/(xmax - xmin);
y = zeros(size(x));
for j = 1:n
  y = y + b(j)*cos( (j-1)*acos(xx) );
end
```

Example 4.5.6. Consider again the average high temperatures from Example 4.5.3. Using `chebfit`, for example with the following MATLAB commands,

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = chebfit(x_data, f_data, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

we obtain the plot shown in Fig. 4.15. As with previous examples, we also used the MATLAB commands `axis, legend, xlabel` and `ylabel` to make the plot look a bit nicer. Because the polynomial that interpolates this data is unique, it should not be surprising that the plot obtained using `chebfit` looks identical to that obtained using `polyfit`. (When there is a large amount of data, the plots may differ slightly due to the effects of computational error and the possible difference in conditioning of the linear systems.)

Problem 4.5.8. *Write a* MATLAB *script M–file that will generate a figure containing the jth Chebyshev polynomials, $j = 1, 3, 5, 7$. Use* `subplot` *to put all plots in one figure, and* `title` *to document the various plots.*

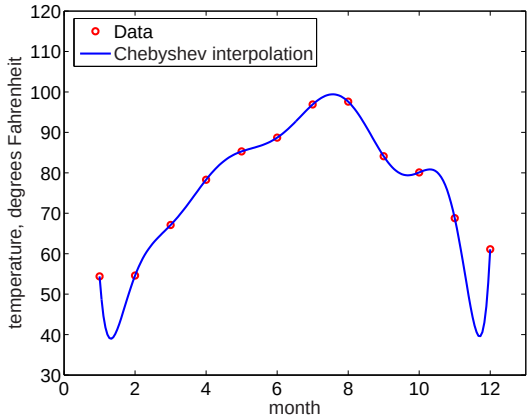

Figure 4.15: Interpolating polynomial or average high temperature data given in Example 4.5.3 using the Chebyshev form.

Problem 4.5.9. *Consider the data* $(0,1)$, $(1,2)$, $(3,3)$ *and* $(5,4)$. *Construct a plot that contains the data (as circles) and the polynomial of degree 3 that interpolates this data using the function* `chebfit`. *You should use the* `axis` *command to make the plot look sufficiently nice.*

Problem 4.5.10. *Consider the data* $(1,1)$, $(3,1)$, $(4,1)$, $(5,1)$ *and* $(7,1)$. *Construct a plot that contains the data (as circles) and the polynomial of degree 4 that interpolates this data using the function* `chebfit` *You should use the* `axis` *command to make the plot look sufficiently nice. Do you believe the curve is a good representation of the data?*

Problem 4.5.11. *Construct plots that contain the data (as circles) and interpolating polynomials, using the function* `chebfit`, *for the following sets of data:*

x_i	f_i
-2	6
-1	3
0	1
1	3
2	6

and

x_i	f_i
-2	-5
-1	-3
0	0
1	3
2	5

Problem 4.5.12. *Construct interpolating polynomials, using the function* `chebfit`, *through all four sets of weather data given in Example 4.5.3. Use subplot to show all four plots in the same figure, and use the* `title, xlabel` *and* `ylabel` *commands to document the plots. Each plot should show the data points as circles on the corresponding curves.*

4.5.3 Polynomial Splines

Polynomial splines help to avoid excessive oscillations by fitting the data using a collection of low degree polynomials. We've actually already used *linear polynomial splines* to connect data via MATLAB's plot command. But we can create this linear spline more explicitly using the `interp1` function. The basic calling syntax is given by:

```
y = interp1(x_data, f_data, x);
```

where

- `x_data` and `f_data` are vectors containing the given data points,
- `x` is a vector containing values at which the linear spline is to be evaluated (e.g., for plotting the spline), and
- `y` is a vector containing $S(x)$ values.

The following example illustrates how to use `interp1` to construct, evaluate, and plot a linear spline interpolating function.

Example 4.5.7. Consider the average high temperatures from Example 4.5.3. The following set of MATLAB commands can be used to construct a linear spline interpolating function of this data:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = interp1(x_data, f_data, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

The resulting plot is shown in Fig. 4.16. Note that we also used the MATLAB commands `axis`, `legend`, `xlabel` and `ylabel` to make the plot look a bit nicer.

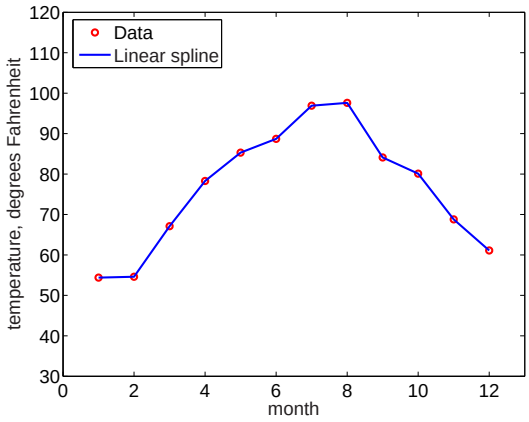

Figure 4.16: Linear polynomial spline interpolating function for average high temperature data given in Example 4.5.3

We can obtain a smoother curve by using a higher degree spline. The primary MATLAB function for this purpose, `spline`, constructs a cubic spline interpolating function. The basic calling syntax, which uses, by default, not-a-knot end conditions, is as follows:

```
y = spline(x_data, f_data, x);
```

where `x_data`, `f_data`, `x` and `y` are defined as for `interp1`. The following example illustrates how to use `spline` to construct, evaluate, and plot a cubic spline interpolating function.

Example 4.5.8. Consider again the average high temperatures from Example 4.5.3. The following set of MATLAB commands can be used to construct a cubic spline interpolating function of this data:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = spline(x_data, f_data, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

The resulting plot is shown in Fig. 4.17. Note that we also used the MATLAB commands `axis, legend, xlabel, ylabel` and `title` to make the plot look a bit nicer.

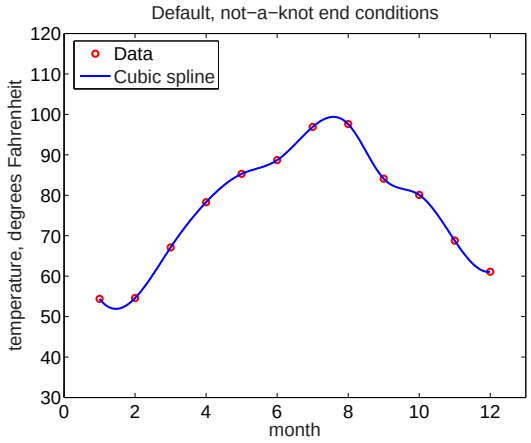

Figure 4.17: Cubic polynomial spline interpolating function, using not-a-knot end conditions, for average high temperature data given in Example 4.5.3

It should be noted that other end conditions can be used, if they can be defined by the slope of the spline at the end points. In this case, the desired slope values at the end points are attached to the beginning and end of the vector `f_data`. This is illustrated in the next example.

Example 4.5.9. Consider again the average high temperatures from Example 4.5.3. The so-called *clamped* end conditions assume the slope of the spline is 0 at the end points. The following set of MATLAB commands can be used to construct a cubic spline interpolating function with *clamped* end conditions:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = spline(x_data, [0, f_data, 0], x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

Observe that, when calling spline, we replaced `f_data` with the vector `[0, f_data, 0]`. The first and last values in this augmented vector specify the desired slope of the spline (in this case zero) at the end points. The resulting plot is shown in Fig. 4.18 Note that we also used the MATLAB commands `axis, legend, xlabel, ylabel` and `title` to make the plot look a bit nicer.

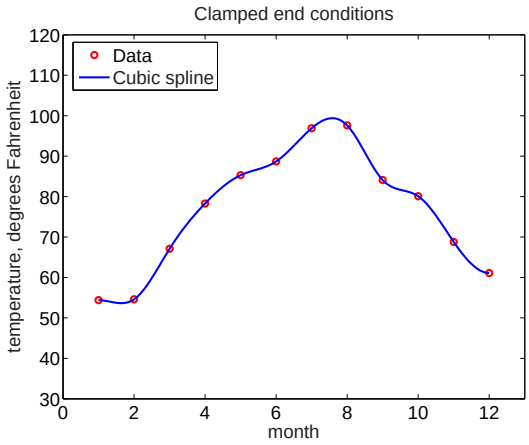

Figure 4.18: Cubic polynomial spline interpolating function, using clamped end conditions, for average high temperature data given in Example 4.5.3

In all of the previous examples using `interp1` and `spline`, we construct **and** evaluate the spline in one command. Recall that for polynomial interpolation we used `polyfit` and `polyval` in a two step process. It is possible to use a two step process with both linear and cubic splines as well. For example, when using `spline`, we can execute the following commands:

```
S = spline(x_data, f_data);
y = ppval(S, x);
```

where `x` is a set of values at which the spline is to be evaluated. If we specify only two input arguments to `spline`, it returns a *structure array* that defines the piecewise cubic polynomial, $S(x)$, and the function `ppval` is then used to evaluate $S(x)$.

In general, if all we want to do is evaluate and/or plot $S(x)$, it is not necessary to use this two step process. However, there may be situations for which we must access explicitly the polynomial pieces. For example, we could find the maximum and minimum (i.e., critical points) of $S(x)$ by explicitly computing $S'(x)$. Another example is given in Section 5.3 where the cubic spline is used to integrate tabular data.

We end this subsection by mentioning that MATLAB has another function, `pchip`, that can be used to construct a piecewise *Hermite* cubic interpolating polynomial, $H(x)$. In interpolation, the name *Hermite* is usually attached to techniques that use specific slope information in the construction of the polynomial pieces. Although `pchip` constructs a piecewise cubic polynomial, strictly speaking, it is not a spline because the second derivative may be discontinuous at the knots. However, the first derivative is continuous. The slope information used to construct $H(x)$ is determined from the data. In particular, if there are intervals where the data are monotonic, then so is $H(x)$, and at points where the data has a local extremum, so does $H(x)$. We can use `pchip` exactly as we use `spline`; that is, either one call to construct and evaluate:

```
y = pchip(x_data, f_data, x);
```

or via a two step approach to construct and then evaluate:

```
H = pchip(x_data, f_data);
y = ppval(H, x);
```

where `x` is a set of values at which $H(x)$ is to be evaluated.

Example 4.5.10. Because the piecewise cubic polynomial constructed by `pchip` will usually have discontinuous second derivatives at the data points, it will be less smooth and a little less accurate for

smooth functions than the cubic spline interpolating the same data. However, forcing monotonicity when it is present in the data has the effect of preventing oscillations and overshoot which can occur with a spline. To illustrate the point we consider a somewhat pathological example of non-smooth data from the MATLAB Help pages. The following MATLAB code fits first a cubic spline then a piecewise cubic monotonic polynomial to the data, which is monotonic.

```
x_data = -3:3;
f_data = [-1 -1 -1 0 1 1 1];
x = linspace(-3, 3, 200);
s = spline(x,y,t);
p = pchip(x,y,t);
plot(x_data, f_data, 'ro')
hold on
plot(x, s, 'k--')
plot(x, p, 'b-')
legend('Data', 'Cubic spline', 'PCHIP interpolation', 'Location', 'NW')
```

The resulting plot is given in Fig. 4.19. We observe both oscillations and overshoot in the cubic spline fit and monotonic behavior in the piecewise cubic monotonic polynomial fit.

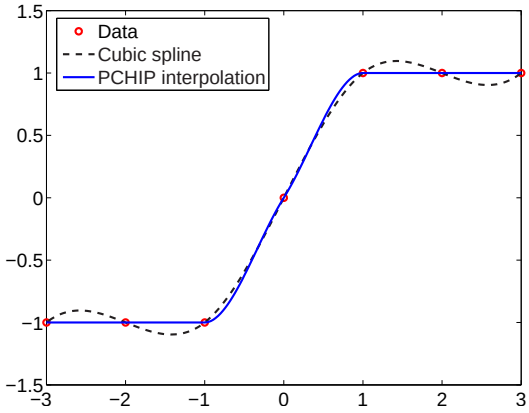

Figure 4.19: Comparison of fitting a monotonic piecewise cubic polynomial to monotonic data (using MATLAB's `pchip` function) and to a cubic spline (using MATLAB's `spline` function).

Example 4.5.11. Consider again the average high temperatures from Example 4.5.3. The following set of MATLAB commands can be used to construct a piecewise cubic Hermite interpolating polynomial through the given data:

```
x_data = 1:12;
f_data = [54.4 54.6 67.1 78.3 85.3 88.7 96.9 97.6 84.1 80.1 68.8 61.1];
x = linspace(1, 12, 200);
y = pchip(x_data, f_data, x);
plot(x_data, f_data, 'ro')
hold on
plot(x, y)
```

The resulting plot is shown in Fig. 4.20 As with previous examples, we also used the MATLAB commands `axis, legend, xlabel` and `ylabel` to make the plot more understandable. Observe that fitting the data using `pchip` avoids oscillations and overshoot that can occur with the cubic spline fit, and that the monotonic behavior of the data is preserved.

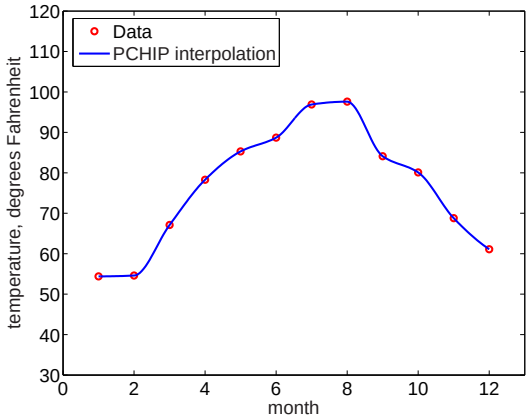

Figure 4.20: Piecewise cubic Hermite interpolating polynomial, for average high temperature data given in Example 4.5.3

Notice that the curve generated by `pchip` is smoother than the piecewise linear spline, but because `pchip` does not guarantee continuity of the second derivative, it is not as smooth as the cubic spline interpolating function. In addition, the flat part on the left, between the months January and February, is probably not an accurate representation of the actual temperatures for this time period. Similarly, the maximum temperature at the August data point (which is enforced by the `pchip` construction) may not be accurate; more likely is that the maximum temperature occurs some time between July and August. We mention these things to emphasize that it is often extremely difficult to produce an accurate, physically reasonable fit to data.

Problem 4.5.13. *Consider the data* $(0,1)$, $(1,2)$, $(3,3)$ *and* $(5,4)$. *Use subplot to make a figure with 4 plots: a linear spline, a cubic spline with not-a-knot end conditions, a cubic spline with clamped end conditions, and a piecewise cubic Hermite interpolating polynomial fit of the data. Each plot should also show the data points as circles on the curve, and should have a title indicating which of the four methods was used.*

Problem 4.5.14. *Consider the following sets of data:*

x_i	f_i
-2	6
-1	3
0	1
1	3
2	6

and

x_i	f_i
-2	-5
-1	-3
0	0
1	3
2	5

Make two figures, each with four plots: a linear spline, a cubic spline with not-a-knot end conditions, a cubic spline with clamped end conditions, and a piecewise cubic Hermite interpolating polynomial fit of the data. Each plot should also show the data points as circles on the curve, and should have a title indicating which of the four methods was used.

Problem 4.5.15. *Consider the four sets of weather data given in Example 4.5.3. Make four figures, each with four plots: a linear spline, a cubic spline with not-a-knot end conditions, a cubic spline with clamped end conditions, and a piecewise cubic Hermite interpolating polynomial fit of the data. Each plot should also show the data points as circles on the curve, and should have a title indicating which of the four methods was used.*

Problem 4.5.16. *Notice that there is an obvious "wobble" at the left end of the plot given in Fig. 4.17. Repeat the previous problem, but this time shift the data so that the year begins with*

April. Does this help eliminate the wobble? Does shifting the data have an effect on any of the other plots?

4.5.4 Interpolating to Periodic Data

The methods considered thus far and those to be considered later are designed to fit to general data. When the data has a specific property it behoves us to exploit that property. So, in the frequently occurring case where the data is thought to represent a periodic function we should fit it using a periodic function, particularly with a trigonometric series where the frequency of the terms is designed to fit the frequency of the data.

Given a set of data that is supposed to represent a periodic function sampled at equal spacing across the period of the function, MATLAB's function `interpft` for periodic data produces interpolated values to this data on a finer user specified equally spaced mesh. First, `interpft` uses a discrete Fourier transform internally to produce a trigonometric series interpolating the data. Then, it uses an inverse discrete Fourier transform to evaluate the trigonometric series on the finer mesh, over the same interval. (At no stage does the user see the trigonometric series.)

Example 4.5.12. Consider sampling $f(x) = \sin(\pi x)$ at $x = i/10$, $i = 0, 1, \ldots, 9$. The data is assumed periodic so we do not evaluate $f(x)$ for $i = 10$. We use `interpft` to interpolate to this data at half the mesh size and plot the results using the commands:

```
x_data = (0:1:9)/10;
f_data = sin(pi*x_data);
x = (0:1:19)/20;
y = interpft(f_data, 20);
plot(x_data, f_data, 'ro')
hold on
plot(x, y, '+')
```

The plot is given on the left in Figure 4.21 where the interpolated and data values coincide at the original mesh points.

If we use instead the nonperiodic function $f(x) = \sin(\pi x^2)$ at the same points as follows

```
x_data = (0:1:9)/10;
f_data = sin(pi*(x.^2));
x = (0:1:19)/20;
y = interpft(f_data, 20);
plot(x_data, f_data, 'ro')
hold on
plot(x, y, '+')
```

we obtain the plot on the right in Figure 4.21. Note the behavior near $x = 0$. The plotted function is periodic. The negative value arises from fitting a periodic function to data arising from a nonperiodic function.

4.5.5 Least Squares

Using least squares to compute a polynomial that approximately fits given data is very similar to the interpolation problem. MATLAB implementations can be developed by mimicking what was done in Section 4.5.1, except we replace the interpolation condition $p(x_i) = f_i$ with $p(x_i) \approx f_i$. For example, suppose we want to find a polynomial of degree M that approximates the $N+1$ data points (x_i, f_i), $i = 1, 2, \ldots N+1$. If we use the power series form, then we end up with an $(N+1) \times (M+1)$ linear system

$$Va \approx f.$$

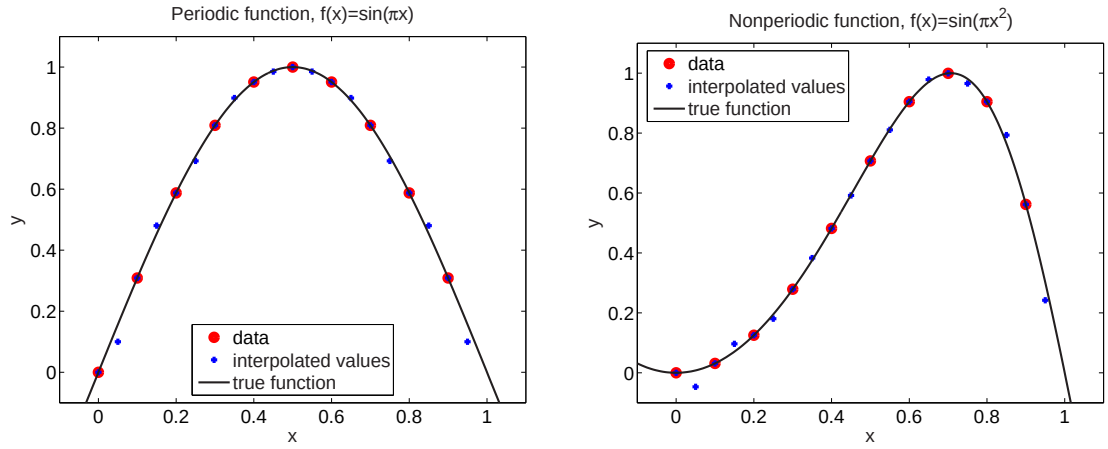

Figure 4.21: Trigonometric fit to data using MATLAB's `interpft` function. The left plot fits to periodic data, and the right plot fits to non-periodic data.

The Normal equations approach to construct the least squares fit of the data is simply the $(M+1)\times(M+1)$ linear system

$$V^T V a = V^T f.$$

Example 4.5.13. Consider the data

x_i	1	3	4	5
f_i	2	4	3	1

Suppose we want to find a least squares fit to this data by a line: $p(x) = a_0 + a_1 x$. Then using the conditions $p(x_i) \approx f_i$ we obtain

$$\begin{array}{lll} p(x_0) \approx f_0 & \Rightarrow & a_0 + a_1 \approx 2 \\ p(x_1) \approx f_1 & \Rightarrow & a_0 + 3a_1 \approx 4 \\ p(x_2) \approx f_2 & \Rightarrow & a_0 + 4a_1 \approx 3 \\ p(x_3) \approx f_3 & \Rightarrow & a_0 + 5a_1 \approx 1 \end{array} \quad \Rightarrow \quad \begin{bmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \approx \begin{bmatrix} 2 \\ 4 \\ 3 \\ 1 \end{bmatrix}.$$

To find a least squares solution, we solve the Normal equations

$$V^T V = V^T f \quad \Rightarrow \quad \begin{bmatrix} 4 & 23 \\ 13 & 51 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} 10 \\ 31 \end{bmatrix}$$

Notice that this linear system is equivalent to what was obtained in Example 4.4.1.

MATLAB's backslash operator can be used to compute least squares approximations of linear systems $Va \approx f$ using the simple command

```
a = V \ f
```

In general, when the backslash operator is used, MATLAB checks first to see if the matrix is square (number of rows = number of columns). If the matrix is square, it will use Gaussian elimination with partial pivoting by rows to solve $Va = f$. If the matrix is not square, it will compute a least squares solution, which is equivalent to solving the Normal equations. MATLAB does not use the Normal equations, but instead a generally more accurate approach based on a QR decomposition of the rectangular matrix V. The details are beyond the scope of this book, but the important point is

that methods used for polynomial interpolation can be used to construct polynomial least squares fit to data. Thus, the main built-in MATLAB functions for polynomial least squares data fitting are `polyfit` and `polyval`. In particular, we can construct the coefficients of the polynomial using

```
a = polyfit(x_data, f_data, M)
```

where M is the degree of the polynomial. In the case of interpolation, `M = N = length(x_data)-1`, but in least squares, `M` is generally less than `N`.

Example 4.5.14. Consider the data

x_i	1	3	4	5
f_i	2	4	3	1

To find a least squares fit of this data by a line (i.e., polynomial of degree $M = 1$), we use the MATLAB commands:

```
x_data = [1 3 4 5];
f_data = [2 4 3 1];
a = polyfit(x_data, f_data, 1);
```

As with interpolating polynomials, we can use `polyfit` to evaluate the polynomial, and `plot` to generate a figure:

```
x = linspace(1, 5, 200);
y = polyval(a, x);
plot(x_data, f_data, 'ro');
hold on
plot(x, y)
```

If we wanted to construct a least squares fit of the data by a quadratic polynomial, we simply replace the `polyfit` statement with:

```
a = polyfit(x_data, f_data, 2);
```

Fig. 4.22 shows the data, and the linear and quadratic polynomial least squares fits.

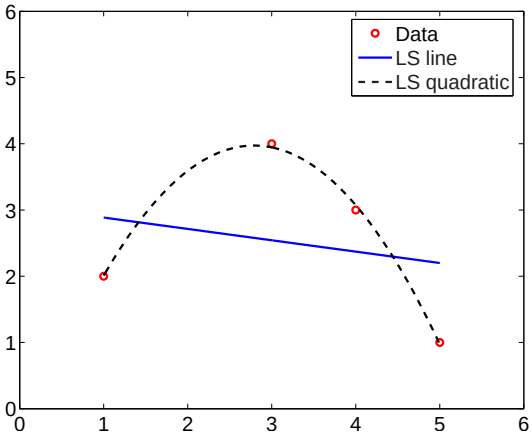

Figure 4.22: Linear and quadratic polynomial least squares fit of data given in Example 4.5.14

Note that it is possible to modify `chebfit` so that it can compute a least squares fit of the data; see Problem 4.5.17.

It may be the case that functions other than polynomials are more appropriate to use in data fitting applications. The next two examples illustrate how to use least squares to fit data to more general functions.

Example 4.5.15. Suppose it is known that data collected from an experiment, (x_i, f_i), can be represented well by a sinusoidal function of the form

$$g(x) = a_1 + a_2 \sin(x) + a_3 \cos(x).$$

To find the coefficients, a_1, a_2, a_3, so that $g(x)$ is a least squares fit of the data, we use the criteria $g(x_i) \approx f_i$. That is,

$$\begin{array}{rcl} g(x_1) \approx f_1 & \Rightarrow & a_1 + a_2 \sin(x_1) + a_3 \cos(x_1) \approx f_1 \\ g(x_2) \approx f_2 & \Rightarrow & a_1 + a_2 \sin(x_2) + a_3 \cos(x_2) \approx f_2 \\ & \vdots & \\ g(x_n) \approx f_n & \Rightarrow & a_1 + a_2 \sin(x_n) + a_3 \cos(x_n) \approx f_n \end{array}$$

which can be written in matrix-vector form as

$$\begin{bmatrix} 1 & \sin(x_1) & \cos(x_1) \\ 1 & \sin(x_2) & \cos(x_2) \\ \vdots & \vdots & \vdots \\ 1 & \sin(x_n) & \cos(x_n) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \approx \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}.$$

In MATLAB, the least squares solution can then be found using the backslash operator which, assuming more than 3 data points, uses a QR decomposition of the matrix. A MATLAB function to compute the coefficients could be written as follows:

```
function a = sinfit(x_data, f_data)
%
%        a = sinfit(x_data, f_data);
%
%  Given a set of data, (x_i, f_i), this function computes the
%  coefficients of
%       g(x) = a(1) + a(2)*sin(x) + a(3)*cos(x)
%  that best fits the data using least squares.
%
n = length(x_data);
W = [ones(n, 1), sin(x_data), cos(x_data)];
a = W \ f_data;
```

Example 4.5.16. Suppose it is known that data collected from an experiment, (x_i, f_i), can be represented well by an exponential function of the form

$$g(x) = a_1 e^{a_2 x}.$$

To find the coefficients, a_1, a_2, so that $g(x)$ is a least squares fit of the data, we use the criteria $g(x_i) \approx f_i$. The difficulty in this problem is that $g(x)$ is not linear in its coefficients. But we can

use logarithms, and their properties, to rewrite the problem as follows:

$$\begin{aligned} g(x) &= a_1 e^{a_2 x} \\ \ln(g(x)) &= \ln\left(a_1 e^{a_2 x}\right) \\ &= \ln(a_1) + a_2 x \\ &= \hat{a}_1 + a_2 x, \quad \text{where } \hat{a}_1 = \ln(a_1), \text{ or } a_1 = e^{\hat{a}_1}. \end{aligned}$$

With this transformation, we would like $\ln(g(x_i)) \approx \ln(f_i)$, or

$$\begin{aligned} \ln(g(x_1)) \approx \ln(f_1) &\Rightarrow \hat{a}_1 + a_2 x_1 \approx \ln(f_1) \\ \ln(g(x_2)) \approx \ln(f_2) &\Rightarrow \hat{a}_1 + a_2 x_2 \approx \ln(f_2) \\ &\vdots \\ \ln(g(x_n)) \approx \ln(f_n) &\Rightarrow \hat{a}_1 + a_2 x_n \approx \ln(f_n) \end{aligned}$$

which can be written in matrix-vector form as

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} \hat{a}_1 \\ a_2 \end{bmatrix} \approx \begin{bmatrix} \ln(f_1) \\ \ln(f_2) \\ \vdots \\ \ln(f_n) \end{bmatrix}.$$

In MATLAB, the least squares solution can then be found using the backslash operator. A MATLAB function to compute the coefficients could be written as follows:

```
function a = expfit(x_data, f_data)
%
%        a = expfit(x_data, f_data);
%
%  Given a set of data, (x_i, f_i), this function computes the
%  coefficients of
%       g(x) = a(1)*exp(a(2)*x)
%  that best fits the data using least squares.
%
n = length(x_data);
W = [ones(n, 1), x_data];
a = W \ log(f_data);
a(1) = exp(a(1));
```

Here we use the built-in MATLAB functions `log` and `exp` to compute the natural logarithm and the natural exponential, respectively.

Problem 4.5.17. *Modify the function* `chebfit` *so that it computes a least squares fit to the data. The modified function should have an additional input value specifying the degree of the polynomial. Use the data in Example 4.5.14 to test your function. In particular, produce a plot similar to the one given in Fig. 4.22.*

Chapter 5

Differentiation and Integration

Two fundamental problems of univariate Calculus are Differentiation and Integration. In a first Calculus course we learn systematic approaches to Differentiation that can be used to compute almost all derivatives of interest, based on limits and on the rules of differentiation, specifically the chain rule. For Integration we learn how to "invert" differentiation (find antiderivatives) and how to compute definite integrals from their definition (through Riemann sums). We discuss first techniques for computing derivatives all based on ideas from Calculus then techniques for computing definite integrals. Finally, we discuss the facilities in MATLAB for differentiation and integration.

5.1 Differentiation

In Calculus we learned that one definition of the derivative function $f'(x)$ of a differentiable function $f(x)$ is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

For this definition to be valid we need that $f(x)$ be defined in an interval centered on the point x, as the parameter h can be either positive or negative in this definition. If we assume a little more, for example that the function f also has a second derivative at x, then we can show using Taylor series that for h sufficiently small

$$f'(x) - \frac{f(x+h) - f(x)}{h} \approx \frac{h}{2} f''(x)$$

so as $h \to 0$ the difference quotient tends to the derivative like $O(h)$.

This forms the basis for the best known and simplest method for approximating a derivative numerically: if we want to compute the value of $f'(a)$, we approximate it using the one-sided difference $\dfrac{f(a+h) - f(a)}{h}$ for a chosen value of h. So, how do we choose h assuming that we can calculate $f(a+h)$ for any value of h? The limit definition seems to imply that we can choose it sufficiently small to achieve whatever accuracy of the computed $f'(a)$ that we desire. In practice, we are limited by the effects of catastrophic cancelation. When h is small we are proposing to calculate two values of $f(x)$ at close arguments hence guaranteeing that almost always there is heavy cancelation in which many significant digits are lost. Then, we propose to divide the result by a very small number hence grossly magnifying the effect. In practice, it turns out that values of h much smaller than about $\sqrt{\epsilon}$ cannot be used as, then, roundoff effects will dominate.

Example 5.1.1. We use the expression $f'(1) \approx \dfrac{f(1+h) - f(1)}{h}$ to approximate the derivative of $f(x) = x^4$ at $x = 1$. We choose the sequence $h = 10^{-n}$, $n = 0, 1, \ldots, 16$; note that the square root of machine precision is $\sqrt{\epsilon_{\text{DP}}} \approx 10^{-8}$. The results are displayed in the second column of table 5.1.

Observe that until rounding error begins to dominate, at about $n = 9$, the error is reduced by about a factor of ten from one row to the next just as is the step size, h; that is, the error behaves linearly (at first order) with h.

n	First Order	Second Order
0	15	8
1	4.64100000000001	4.04000000000000
2	4.06040100000000	4.00040000000000
3	4.00600400099949	4.00000399999972
4	4.00060003999947	4.00000003999923
5	4.00006000043085	4.00000000040368
6	4.00000599976025	3.99999999994849
7	4.00000060185590	4.00000000011502
8	4.00000004230350	4.00000000344569
9	4.00000033096148	4.00000010891688
10	4.00000033096148	4.00000033096148
11	4.00000033096148	4.00000033096148
12	4.00035560232936	4.00013355772444
13	3.99680288865056	3.99902333469981
14	3.99680288865056	3.99680288865056
15	4.44089209850063	4.21884749357559
16	0	2.22044604925031

Table 5.1: Derivative of x^4 at $x = 1$ by first and second order differences

The last column of table 5.1 is generated using central differences. Assuming that the function $f(x)$ has continuous third derivatives in a neighborhood of the point x, it is easy to show using Taylor series that

$$f(x+h) - f(x-h) \approx 2hf'(x) + \frac{h^3}{3}f'''(x)$$

so the error in the central difference approximation

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

behaves like h^2. We see the appropriate second order behavior in table 5.1. As we divide the step size by a factor of ten the error is divided by a factor of one hundred, until roundoff error dominates. this happens earlier than for the first order approximation because the actual error is much smaller so easier to dominate.

Problem 5.1.1. *Use Taylor series to show that* $f'(x) - \dfrac{f(x+h) - f(x)}{h} \approx \dfrac{h}{2}f''(x)$.

Problem 5.1.2. *Use Taylor series to show that central difference approximation* $f'(x) \approx \dfrac{f(x+h) - f(x-h)}{2h}$ *is second order in* h.

Problem 5.1.3. *Reproduce table 5.1. Produce a similar table for the derivative of* $f(x) = e^x$ *at* $x = 1$. *When does roundoff error begin to dominate?*

In reality we would normally calculate the derivative of a given function directly (using symbolic differentiation or automatic differentiation, see section 5.3.1). Where we need numerical differentiation is when we need an approximation to the derivative of a function which is tabulated. Suppose

we have data $\{(x_i, f_i)\}_{i=0}^N$ where we assume that the points $x_i = x_0 + ih$ are equally spaced and we assume that $f_i = f(x_i), i = 0, 1 \ldots, N$ for some unknown smooth function $f(x)$. The aim is to compute an approximation $f_i' \approx f'(x_i), i = 0, 1 \ldots, N$. The simplest approach is to use the second order centered difference $f_i' \approx \dfrac{f_{i+1} - f_{i-1}}{2h}, i = 1, 2, \ldots, N-1$ but that leaves open the question of how to produce an accurate approximation to the derivatives at the endpoints f_0' and f_N'.

How can we deal with variably spaced points x_i (and, incidentally, with derivatives at endpoints)? The simplest way to compute an approximation to f_i' is to fit a quadratic polynomial $p_2(x)$ to the data $(x_{i-1}, f_{i-1}), (x_i, f_i), (x_{i+1}, f_{i+1})$ then differentiate it and evaluate at x_i. (For f_0' proceed similarly but interpolate the data points $(x_0, f_0), (x_1, f_1), (x_2, f_2)$, differentiate, then evaluate at x_0.) Write $p_2(x) = A(x-x_i)^2 + B(x-x_i) + C$ then $f_i' \approx p_2'(x_i) = B$. Interpolating $p_2(x_j) = f_j, j = i-1, i, i+1$, and defining $h_{i+1} = x_{i+1} - x_i$ and $h_i = x_i - x_{i-1}$ we find that

$$f_i' \approx p_2'(x_i) = B = \frac{h_i^2(f_{i+1} - f_i) + h_{i+1}^2(f_i - f_{i-1})}{h_i h_{i+1}(h_{i+1} + h_i)}$$

Note, $B = \dfrac{f_{i+1} - f_{i-1}}{x_{i+1} - x_{i-1}}$ when $h_i = h_{i+1}$. (The MATLAB function `gradient` uses this central difference approximation except at the endpoints where a one-sided difference is used.) Note, this central difference is a second order approximation to the derivative at the point midway between x_{i-1} and x_{i+1}, and this is the point x_i only when the points are equally spaced. Otherwise, to leading order the error behaves like $(h_{i+1} - h_i)$.

Problem 5.1.4. *Show that* $B = \dfrac{h_i^2(f_{i+1} - f_i) + h_{i+1}^2(f_i - f_{i-1})}{h_i h_{i+1}(h_{i+1} + h_i)}$ *in the formula for* $p_2(x) = A(x - x_i)^2 + B(x - x_i) + C$.

Problem 5.1.5. *Show, using Taylor series, that* $f'(x_i) - \dfrac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} = \dfrac{(h_{i+1} - h_i)}{2} f''(x_i)$. *to leading order.*

So, using quadratic interpolation gives a disappointing result for unequally spaced meshes. It is tempting to increase the degree of polynomial interpolation, by including additional points, to achieve a more accurate result. But, recall that high degree polynomial interpolation is potentially inaccurate and taking derivatives increases the inaccuracy. We prefer to use cubic spline interpolation to the data and to differentiate the result to give a derivative at any point in the interval defined by the data. Unlike the methods described above this is a global method; that is, we can only write the derivative in terms of all the data. If $S_{3,N}(x)$ is the not-a-knot cubic spline interpolating the data $\{(x_i, f_i)\}_{i=0}^N$, then the interpolation error behaves like $\max\limits_{i=1,\ldots,N} h_i^4$ everywhere in the interval. The error in the derivative $S_{3,N}'(x)$ behaves like $\max\limits_{i=1,\ldots,N} h_i^3$.

5.2 Integration

Consider the definite integral

$$I(f) \equiv \int_a^b f(x)\, dx$$

Assume that the function $f(x)$ is continuous on the closed interval $[a, b]$, so that the integral $I(f)$ exists. We develop efficient methods for computing approximations to the integral using only values of the integrand $f(x)$ at points $x \in [a, b]$. We'll study methods for finding integration rules, of which the midpoint rule is our first example, and for finding the error associated with these rules. We'll also consider composite versions of these rules (obtained by dividing the interval $[a, b]$ into subintervals and using the basic rule on each) and we'll study the errors associated with them. Finally we'll see

how to choose the subintervals in the composite rules adaptively to make the error as small as we need.

We need the property of the definite integral $I(f)$ that it is a linear functional[1], that is

$$I(\alpha f(x) + \beta g(x)) = \alpha I(f(x)) + \beta I(g(x))$$

for any constants α and β and any functions $f(x)$ and $g(x)$ for which the integrals $I(f(x))$ and $I(g(x))$ exist. In integral notation this equation provides the standard linearity result

$$\int_a^b (\alpha f(x) + \beta g(x))\, dx = \alpha \int_a^b f(x)\, dx + \beta \int_a^b g(x)\, dx$$

for integrals.

5.2.1 Integrals and the Midpoint Rule

To approximate the integral $I(f)$ we can integrate exactly piecewise polynomial approximations of $f(x)$ on the interval $[a, b]$. As in Figure 5.1, partition the interval $[a, b]$ into N subintervals $[x_{i-1}, x_i]$, $i = 1, 2, \cdots, N$, where $a = x_0 < x_1 < \cdots < x_N = b$ and use the **additivity property of integrals**:

$$I(f) = \int_{x_0}^{x_1} f(x)\, dx + \int_{x_1}^{x_2} f(x)\, dx + \cdots + \int_{x_{N-1}}^{x_N} f(x)\, dx$$

The first, and simplest, approximation to $f(x)$ that we consider is a piecewise constant on a

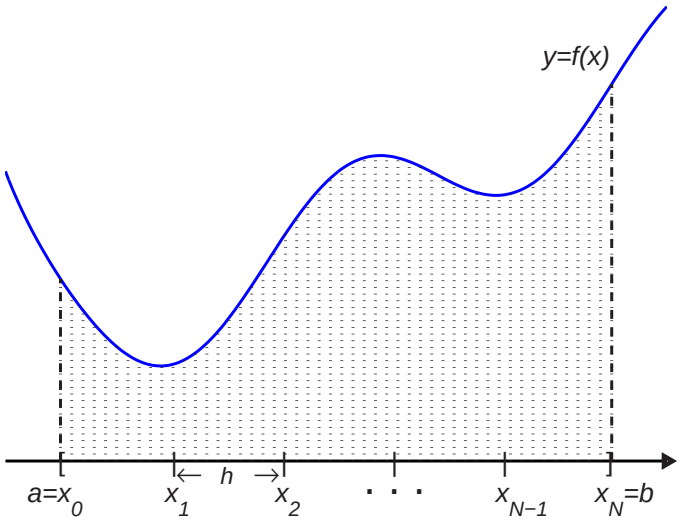

Figure 5.1: Partition of $[a, b]$ into N equal pieces of length h

subinterval. On the interval $[x_{i-1}, x_i]$, we approximate $f(x)$ by its value at the midpoint, $\dfrac{x_{i-1} + x_i}{2}$, of the interval. This turns out to be a pretty good choice. We have

$$\int_{x_{i-1}}^{x_i} f(x)\, dx \approx \int_{x_{i-1}}^{x_i} f\left(\frac{x_{i-1} + x_i}{2}\right) dx = (x_i - x_{i-1}) f\left(\frac{x_{i-1} + x_i}{2}\right)$$

[1] A functional maps functions to numbers. A definite integral provides a classic example of a functional; this functional assigns to each function a number that is the definite integral of that function.

which is the **midpoint rule**; the quantity $(x_i - x_{i-1})f\left(\frac{x_{i-1}+x_i}{2}\right)$ is the area of the rectangle of width $(x_i - x_{i-1})$ and height $f\left(\frac{x_{i-1}+x_i}{2}\right)$, see Figure 5.2. Hence, using the additivity property of definite integrals we obtain the **composite midpoint rule**:

$$\begin{aligned} I(f) &= \int_{x_0}^{x_1} f(x)\,dx + \int_{x_1}^{x_2} f(x)\,dx + \cdots + \int_{x_{N-1}}^{x_N} f(x)\,dx \\ &\approx (x_1 - x_0)f\left(\frac{x_0+x_1}{2}\right) + (x_2 - x_1)f\left(\frac{x_1+x_2}{2}\right) + \cdots + (x_N - x_{N-1})f\left(\frac{x_{N-1}+x_N}{2}\right) \\ &= \sum_{i=1}^{N}(x_i - x_{i-1})f\left(\frac{x_{i-1}+x_i}{2}\right) \end{aligned}$$

In the case when the points x_i are equally spaced, that is, $x_i \equiv a + ih$, $i = 0, 1, \cdots, N$, and $h = x_i - x_{i-1} \equiv \frac{b-a}{N}$, this formula reduces to

$$\begin{aligned} I(f) &= \int_{x_0}^{x_1} f(x)\,dx + \int_{x_1}^{x_2} f(x)\,dx + \cdots + \int_{x_{N-1}}^{x_N} f(x)\,dx \\ &\approx hf\left(x_{\frac{1}{2}}\right) + hf\left(x_{\frac{3}{2}}\right) + \cdots + hf\left(x_{\frac{2N-1}{2}}\right) \\ &= h\sum_{i=1}^{N} f\left(x_{\frac{2i-1}{2}}\right) \\ &\equiv R_{CM}(f,h) \end{aligned}$$

the composite midpoint rule, where for any real value s we have $x_s = a + sh$.

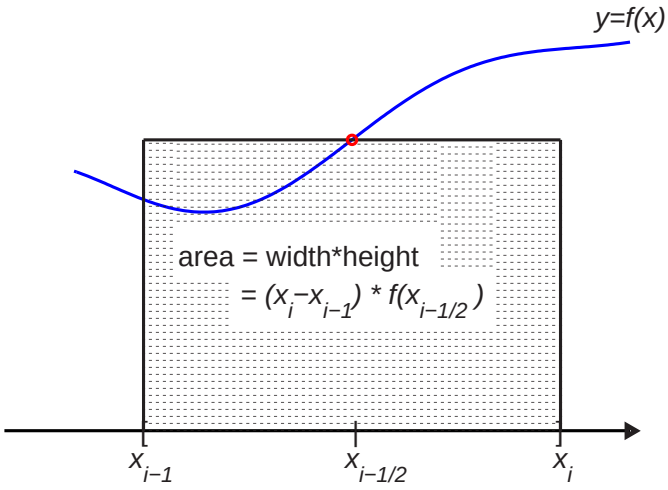

Figure 5.2: Geometry of the Midpoint Rule. The midpoint of the interval $[x_{i-1}, x_i]$ is $x_{i-\frac{1}{2}} \equiv \frac{x_{i-1}+x_i}{2}$.

5.2.2 Quadrature Rules

Generally, a **quadrature rule**[2] (such as the midpoint rule for any interval of integration) has the form

$$R(f) \equiv \sum_{i=0}^{N} w_i f(x_i)$$

[2]The term "quadrature" refers to a mathematical process that constructs a square whose area equals the area under a given curve. The problem of "squaring a circle" is an example of an ancient quadrature problem; construct a square whose area equals that of a given circle.

for given **points** $x_0 < x_1 < \cdots < x_N$ and **weights** $w_0, w_1, \cdots, w_N$.

Similarly to an integral, a quadrature rule $R(f)$ is a linear functional, that is we have a linearity rule:

$$R(\alpha f(x) + \beta g(x)) = \alpha R(f(x)) + \beta R(g(x))$$

or, in summation notation,

$$\sum_{i=0}^{N} w_i(\alpha f(x_i) + \beta g(x_i)) = \alpha \sum_{i=0}^{N} w_i f(x_i) + \beta \sum_{i=0}^{N} w_i g(x_i)$$

To approximate a definite integral $I(f)$ where we don't know an antiderivative for $f(x)$, a good choice is to integrate a simpler function $q(x)$ whose antiderivative we do know and that approximates the function $f(x)$ well. From the linearity of the functional $I(f)$, we have

$$I(q) = I(f + [q - f]) = I(f) + I(q - f)$$

So that the error in approximating the true integral $I(f)$ by the integral of the approximation $I(q)$ can be expressed as

$$I(q) - I(f) = I(q - f);$$

that is, the error in approximating the definite integral of the function $f(x)$ by using the definite integral of the approximating function $q(x)$ is the definite integral of the error in approximating $f(x)$ by $q(x)$. If $q(x)$ approximates $f(x)$ well, that is if the error $q(x) - f(x)$ is in some sense small, then the error $I(q - f)$ in the integral $I(q)$ approximating $I(f)$ will also be small, because the integral of a small function is always relatively small; see Problem 5.2.1.

Example 5.2.1. Consider the definite integral $I(f) = \int_c^d f(x)\,dx$. Interpolation can be used to determine polynomials $q(x)$ that approximate the function $f(x)$ on $[c, d]$. As we have seen, the choice of the function $q_0(x)$ as a polynomial of degree $N = 0$ (that is, a constant approximation) interpolating the function $f(x)$ at the midpoint $x = \dfrac{c+d}{2}$ of the interval $[c, d]$ gives the midpoint rule.

Example 5.2.2. As another example, consider the function $q_1(x)$ which is the polynomial of degree one (that is, the straight line) that interpolates the function $f(x)$ at the integration interval endpoints $x = c$ and $x = d$; that is, the polynomial $q_1(x)$ is chosen so that the interpolating conditions

$$\begin{aligned} q_1(c) &= f(c) \\ q_1(d) &= f(d) \end{aligned}$$

are satisfied. The Lagrange form of the interpolating straight line is

$$q_1(x) = \ell_1(x)f(c) + \ell_2(x)f(d)$$

where the Lagrange basis functions are

$$\ell_1(x) = \frac{x-d}{c-d}$$

$$\ell_2(x) = \frac{x-c}{d-c}$$

Because the function values $f(c)$ and $f(d)$ are constants, due to linearity we obtain

$$I(q_1) = I(\ell_1)f(c) + I(\ell_2)f(d),$$

and since $I(\ell_1) = I(\ell_2) = \dfrac{d-c}{2}$, we have

$$I(q_1) = w_1 f(c) + w_2 f(d)$$

where $w_1 = w_2 = \dfrac{d-c}{2}$. This is the **trapezoidal** rule

$$R_T(f) \equiv \frac{d-c}{2} f(c) + \frac{d-c}{2} f(d) = \frac{d-c}{2}[f(c) + f(d)]$$

Problem 5.2.1. *Consider approximating* $\int_a^b f(x)\,dx$ *by* $\int_a^b q(x)\,dx$ *where* $\max_{x\in[a,b]} |f(x)-q(x)| = \epsilon$. *Show that* $|\int_a^b f(x)\,dx - \int_a^b q(x)\,dx| \le \epsilon|b-a|$.

Problem 5.2.2. *For the basis functions* $\ell_1(x) = \dfrac{x-d}{c-d}$ *and* $\ell_2(x) = \dfrac{x-c}{d-c}$ *show that* $I(\ell_1) = I(\ell_2) = \dfrac{d-c}{2}$ *in two separate ways:*

(1) algebraically, by direct integration over the interval $[c,d]$,

(2) geometrically, by calculating the areas under the graphs of $\ell_1(x)$ *and* $\ell_2(x)$ *on the interval* $[c,d]$.

Problem 5.2.3. *Plot* $q_1(x) = \ell_1(x)f(c) + \ell_2(x)f(d)$ *and show that the area under the graph of* $q_1(x)$ *over the interval* $[c,d]$ *is the area of a trapezoid. Hence, compute its value.*

Problem 5.2.4. *Proceeding analogously to the derivation of the composite midpoint rule to derive the composite trapezoidal rule.*

Error in the Trapezoidal and Composite Trapezoidal Rules

Next, we need to see precisely how the error in approximating a definite integral $I(f)$ depends on the integrand $f(x)$. We use the trapezoidal rule to develop this analysis. We need the Integral Mean Value Theorem: If the functions $g(x)$ and $w(x)$ are continuous on the closed interval $[a,b]$ and $w(x)$ is nonnegative on the open interval (a,b), then for some point $\eta \in (a,b)$:

$$\int_a^b w(x)g(x)\,dx = \left\{\int_a^b w(x)\,dx\right\} g(\eta)$$

If the second derivative $f''(x)$ exists and is continuous on $[c,d]$, the error in the trapezoidal rule is

$$\begin{aligned} I(f) - R_T(f) &= I(f) - I(q_1) \\ &= I(f - q_1) \\ &= \int_c^d \{\text{the error in linear interpolation}\}\,dx \\ &= \int_c^d \frac{(x-c)(x-d)}{2} f''(\xi_x)\,dx \end{aligned}$$

where ξ_x is an unknown point in the interval $[c,d]$ whose location depends both on the integrand $f(x)$ and on the value of x. Here we have used the formula for the error in polynomial interpolation developed in Chapter 4. Now, since the function $-(x-c)(x-d) > 0$ for all $x \in (c,d)$ we can apply the Integral Mean Value Theorem. Thus, we obtain

$$\begin{aligned} I(f) - R_T(f) &= -\int_c^d \frac{(x-c)(d-x)}{2} f''(\xi_x)\,dx \\ &= -\frac{(d-c)^3}{12} f''(\eta) \end{aligned}$$

where η is an (unknown) point located in the open interval (c, d). Note, the point η is necessarily unknown or else the formula $I(f) = R(f) + \dfrac{(d-c)^3}{12} f''(\eta)$ could be used to evaluate *any* integral $I(f)$ exactly from explicit evaluations of $f(x)$ and of its second derivative $f''(x)$. However in the example that follows we see one way to determine the point η for some functions $f(x)$. Note that though we can determine the point η in this case, in most cases we cannot. Also, note that there may be more than one such point η.

Example 5.2.3. For $f(x) = \dfrac{(x-c)^3}{6}$ explicitly determine each of the values $f''(x)$, $R_T(f)$ and $I(f)$. Use these values to determine the "unknown" point $\eta \in (c, d)$ so that the equation

$$I(f) - R_T(f) = -\frac{(d-c)^3}{12} f''(\eta)$$

is satisfied.

Solution: For $f(x) = \dfrac{(x-c)^3}{6}$ we have $f''(x) = (x-c)$ and

$$\begin{aligned} R_T(f) &= \frac{(d-c)}{2}\left(0 + \frac{(d-c)^3}{6}\right) = \frac{(d-c)^4}{12} \\ I(f) &= \int_c^d f(x)\,dx = \frac{(d-c)^4}{24} \end{aligned}$$

Now, substituting these values in expression for the error in $R_T(f)$, we obtain

$$-\frac{1}{24}(d-c)^4 = I(f) - R_T(f) = -\frac{(d-c)^3}{12} f''(\eta) = -\frac{(d-c)^3}{12}(\eta - c)$$

so that

$$\eta - c = \frac{d-c}{2}$$

and solving this equation for η yields the explicit value

$$\eta = c + \frac{d-c}{2} = \frac{d+c}{2}$$

that is, the midpoint of the interval $[c, d]$.

Now, we develop a composite quadrature formula for $\int_a^b f(x)\,dx$ using the trapezoidal rule. Recall, $h = x_i - x_{i-1}$ and approximate $\int_{x_{i-1}}^{x_i} f(x)\,dx$ by the trapezoidal rule

$$\int_{x_{i-1}}^{x_i} f(x)\,dx \approx \frac{(x_i - x_{i-1})}{2}[f(x_{i-1}) + f(x_i)] = \frac{h}{2}[f(x_{i-1}) + f(x_i)] = R_T(f)$$

So, the error is

$$I(f) - R_T(f) = \int_{x_{i-1}}^{x_i} f(x)\,dx - \frac{h}{2}[f(x_{i-1}) + f(x_i)] = -\frac{h^3}{12} f''(\eta_i)$$

where η_i is an unknown point in (x_{i-1}, x_i), and hence in (a, b). So, we have

$$I(f) \equiv \int_a^b f(x)\,dx = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)\,dx \approx \sum_{i=1}^{N} \frac{h}{2}[f(x_{i-1}) + f(x_i)] \equiv R_{CT}(f, h)$$

which is the **composite trapezoidal rule**.

Next, we will need the Generalized Mean Value Theorem for Sums. We'll begin with a simple example.

Example 5.2.4. (Mean Value Theorem for Sums) Show that if the function $f(x)$ is continuous on the interval $[a,b]$, if the weights w_0 and w_1 are nonnegative numbers with $w_0 + w_1 > 0$, and if the points x_0 and x_1 both lie in $[a,b]$, then there is a point $\eta \in [a,b]$ for which

$$w_0 f(x_0) + w_1 f(x_1) = \{w_0 + w_1\} f(\eta)$$

Solution: Let $m = \min_{x\in[a,b]} f(x)$ and $M = \max_{x\in[a,b]} f(x)$. These extrema exist because the function $f(x)$ is continuous on the closed interval $[a,b]$. Since the weights w_0 and w_1 are nonnegative, we have

$$\{w_0 + w_1\}\, m \le w_0 f(x_0) + w_1 f(x_1) \le \{w_0 + w_1\} M$$

Because $w_0 + w_1 > 0$, we can divide throughout by this factor to give

$$m \le \frac{w_0 f(x_0) + w_1 f(x_1)}{w_0 + w_1} \le M$$

which is a weighted average of the function values $f(x_1)$ and $f(x_2)$. Because the function $f(x)$ is continuous on $[a,b]$, the Intermediate Value Theorem (see Problem 6.1.1) states that there is a point $\eta \in [a,b]$ for which

$$f(\eta) = \frac{w_0 f(x_0) + w_1 f(x_1)}{w_0 + w_1}$$

which gives the required result.

A generalization of the above argument shows that if the function $g(x)$ is continuous on the interval $[a,b]$, the weights $\{w_i\}_{i=0}^N$ are all nonnegative numbers with $\sum_{i=0}^N w_i > 0$, and the points $\{x_i\}_{i=0}^N$ all lie in $[a,b]$, then for some point $\eta \in [a,b]$:

$$\sum_{i=0}^{N} w_i g(x_i) = \left\{ \sum_{i=0}^{N} w_i \right\} g(\eta)$$

This result, the Generalized Mean Value Theorem for Sums, may be proved directly or via a simple induction argument using the result of Example 5.2.4.

Now, assuming that $f''(x)$ is continuous on the interval $[a,b]$, the error in the composite trapezoidal rule is

$$\begin{aligned}
I(f) - R_{CT}(f) &= \int_a^b f(x)\,dx - \sum_{i=1}^{N} \frac{h}{2}[f(x_{i-1}) + f(x_i)] \\
&= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)\,dx - \sum_{i=1}^{N} \frac{h}{2}[f(x_{i-1}) + f(x_i)] \\
&= \sum_{i=1}^{N} \left(\int_{x_{i-l}}^{x_i} f(x)\,dx - \frac{h}{2}[f(x_{i-1}) + f(x_i)] \right) \\
&= -\frac{h^3}{12} \sum_{i=1}^{N} f''(\eta_i) \\
&= -\frac{h^3}{12} N f''(\eta)
\end{aligned}$$

for some unknown point $\eta \in (a,b)$. Here, we have used the Generalized Mean Value Theorem for Sums. Now, $Nh = (b-a)$, so this last expression reduces to

$$I(f) - R_{CT}(f) = -\frac{h^2}{12}(b-a) f''(\eta)$$

So, as $N \to \infty$ and $h \to 0$ simultaneously in such a way that the value $Nh = (b-a)$ is fixed, the error in the composite trapezoidal rule decreases like h^2.

Problem 5.2.5. *For $f(x) = \dfrac{(x-c)^2}{2}$, verify that the formula for the error in $R_T(f)$ in Example 5.2.3 gives the correct result. That is, calculate the error as the difference between the rule for this integrand $f(x)$ and the integral for this integrand, and show that you get the same expression as you do by evaluating the error expression for this integrand.*

Problem 5.2.6. *Argue that the trapezoidal rule is exact when used to find the area under any straight line. Use two approaches:*

(1) Use the polynomial interpolation uniqueness theorem to determine the error explicitly.

(2) Exploit the fact that the error expression depends on the second derivative $f''(x)$.

Problem 5.2.7. *Compute $\int_0^1 (x^2 + x - 1)\,dx$ and approximate it using the trapezoidal rule. Hence, compute the error exactly and also compute it from the error expression.*

Problem 5.2.8. *This problem asks you to repeat the steps in the error analysis of the composite trapezoidal rule outlined in this section.*

(1) Use the formula for the error in polynomial interpolation to show that the error in the trapezoidal rule is

$$\int_c^d f(x)\,dx - \frac{(d-c)}{2}[f(c) + f(d)] = -\frac{1}{2}\int_c^d w_1(x) f''(\xi_x)\,dx$$

for some point $\xi_x \in (c,d)$, where $w_1(x) = (x-c)(d-x)$

(2) Use the Mean Value Theorem for integrals to show that, for some point $\eta \in (c,d)$, we have

$$\int_c^d w_1(x) f''(x)\,dx = f''(\eta)\int_c^d w_1(x)\,dx$$

(3) Finally, show that $\displaystyle\int_c^d w_1(x)\,dx = \frac{(d-c)^3}{6}$

Problem 5.2.9. *By similar arguments to those used to develop the error in the composite trapezoidal rule, show that for the integral $I(f) = \int_a^b f(x)\,dx$ and step size $h = \frac{b-a}{N}$, the error in the composite midpoint rule $R_{CM}(f)$ is given by*

$$I(f) - R_{CM}(f) = \frac{h^2}{24}(b-a) f''(\eta)$$

for some unknown point $\eta \in (a,b)$. [Hint: Recall, for the integral $I(f) = \int_c^d f(x)\,dx$ the midpoint rule is $R_M(f) = (d-c) f\left(\frac{c+d}{2}\right)$. You should use the fact that the error is $I(f) - R_M(f) = \frac{(d-c)^3}{24} f''(\tau)$ for some unknown point $\tau \in (c,d)$.]

Interpolatory Quadrature

Rather than add points in the interval $[a,b]$ by making composite versions of simple rules such as the midpoint and trapezoidal rules, we may also generalize these rules by adding more interpolation points and using a higher degree interpolating polynomial. Let $x_0 < x_1 < \cdots < x_N$ and let $q_N(x)$ be the polynomial of degree N interpolating the data $\{(x_i, f(x_i))\}_{i=0}^N$. The Lagrange form of the interpolating polynomial is

$$q_N(x) = \sum_{i=0}^N f(x_i)\ell_i(x)$$

where the Lagrange basis functions $\ell_i(x)$ are defined in Chapter 4. Exploiting linearity, we have

$$I(f) \approx I(q_N) = I(\sum_{i=0}^{N} f(x_i)\ell_i(x)) = \sum_{i=0}^{N} I(\ell_i(x))f(x_i) = \sum_{i=0}^{N} w_i f(x_i) \equiv R(f)$$

where the weights

$$w_i = I(\ell_i(x)) \equiv \int_a^b \ell_i(x)\,dx$$

$R(f)$ is an **interpolatory quadrature rule**. When the nodes x_i are equally spaced in $[a, b]$, so $x_i = a + ih$ where $h \equiv \dfrac{b-a}{N}$, we obtain the Newton–Cotes rules. In particular, the *closed* $(N+1)$–point Newton–Cotes rule has the points $x_0, x_1, \cdots, x_N$ as points; note, this list *includes* the interval endpoints $x_0 = a$ and $x_N = b$. The *open* $(N-1)$–point Newton–Cotes rule has the points $x_1, x_2, \cdots, x_{N-1}$ as points; note, this list *excludes* the endpoints $x_0 = a$ and $x_N = b$.

The open 1–point Newton–Cotes rule is the midpoint rule, the closed 2–point Newton–Cotes rule is the trapezoidal rule. The closed 3–point Newton–Cotes rule is **Simpson's rule**:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

The closed 4–point Newton–Cotes rule is Weddle's rule, see Problem 5.2.17.

Recall that $R(f) = I(q)$, so $I(f) - R(f) = I(f) - I(q) = I(f - q)$. Hence, $R(f)$ cannot accurately approximate $I(f)$ when $I(f - q)$ is large, which can occur only when $f - q$ is large. This can happen when using many equally spaced interpolation points, see Runge's example in Section 4.2. Integrating the polynomial interpolant used there to approximate $f(x) = \dfrac{1}{1+x^2}$ would correspond to using an 11–point closed Newton–Cotes rule to approximate $\int_{-5}^{5} f(x)\,dx$, with the possibility of a resulting large error.

However, $f - q$ can be large yet $I(f - q)$ be zero. Consider the error in the midpoint and Simpson's rules. The midpoint rule is derived by integrating a constant interpolating the data $\left(\dfrac{(a+b)}{2}, f\left(\dfrac{(a+b)}{2}\right)\right)$. This interpolant is exact only for constants, so we would anticipate that the error would be zero only for constant integrands because only then does $f - q \equiv 0$. But, like the trapezoidal rule, the midpoint rule turns out to be exact for all straight lines, see Problem 5.2.13. How can a polynomial approximation $q(x)$ of $f(x)$ that is exact for only constants yield a quadrature rule that is also exact for all straight lines? Similarly, Simpson's rule may be derived by integrating a quadratic interpolating function to the data $(a, f(a))$, $\left(\dfrac{(a+b)}{2}, f\left(\dfrac{(a+b)}{2}\right)\right)$, $(b, f(b))$, see Problem 5.2.12. This quadratic interpolating function is exact for all functions f that are quadratic polynomials, yet the quadrature rule derived from integrating this interpolating function turns out to be exact for all cubic polynomials, see Problem 5.2.13. How can a polynomial approximation q of f that is exact for only quadratic polynomials yield a quadrature rule that is also exact for all cubic polynomials? In both of these cases $I(f - q) = 0$ when $f - q$ is not identically zero and, indeed, some values of $f(x) - q(x)$ are potentially large.

These rules exhibit a form of *superconvergence*; that is, the rules integrate exactly all polynomials of a certain higher degree than is to be anticipated from their construction. Indeed, all Newton–Cotes rules (closed or open) with an odd number of points exhibit this type of superconvergence; that is, they each integrate exactly all polynomials of degree one higher than the degree of the polynomial integrated to derive the rule. Gaussian quadrature rules, to be discussed in Section 5.2.4, yield the ultimate in superconvergent quadrature rules, integrating polynomials of degree almost twice the number of points.

Problem 5.2.10. *In general, an interpolatory quadrature rule based on $N + 1$ nodes integrates exactly all polynomials of degree N. Why?*

Problem 5.2.11. *What is the highest degree general polynomial that*

(1) the $(N+1)$–point closed Newton–Cotes rules will integrate exactly?

(2) the $(N-1)$–point open Newton–Cotes rules will integrate exactly?

Problem 5.2.12. *Derive Simpson's rule for $\int_{-1}^{1} f(x)\,dx$ by constructing and integrating a quadratic interpolating polynomial to the data $(-1, f(-1))$, $(0, f(0))$ and $(1, f(1))$.*

Problem 5.2.13. *Consider the midpoint and Simpson's rules for the interval $[-1, 1]$. Show that*

- *the midpoint rule is exact for all straight lines*
- *Simpson's rule is exact for all cubic polynomials*

Degree of Precision and the Method of Undetermined Coefficients

Here we present an alternative, but related, way to derive quadrature rules and a theorem which simplifies determining the error in some rules.

Degree of Precision

Definition 5.2.1. Degree of precision (DOP). *The rule $R(f) = \sum_{i=0}^{N} w_i f(x_i)$ approximating the definite integral $I(f) = \int_a^b f(x)\,dx$ has DOP $= m$ if $\int_a^b f(x)\,dx = \sum_{i=0}^{N} w_i f(x_i)$ whenever $f(x)$ is a polynomial of degree at most m, but $\int_a^b f(x)\,dx \neq \sum_{i=0}^{N} w_i f(x_i)$ for some polynomial $f(x)$ of degree $m+1$.*

The following theorem gives an equivalent test for the DOP as the above definition.

Theorem 5.2.1. The rule $R(f) = \sum_{i=0}^{N} w_i f(x_i)$ approximating the definite integral $I(f) = \int_a^b f(x)\,dx$ has DOP $= m$ if $\int_a^b x^r\,dx = \sum_{i=0}^{N} w_i x_i^r$ for $r = 0, 1, \cdots, m$, but $\int_a^b x^{m+1}\,dx \neq \sum_{i=0}^{N} w_i x_i^{m+1}$.

From a practical point of view, if a quadrature rule $R_{hi}(f)$ has a higher DOP than another rule $R_{lo}(f)$, then $R_{hi}(f)$ is generally considered more accurate than $R_{lo}(f)$ because it integrates exactly higher degree polynomials and hence potentially integrates exactly more accurate polynomial approximations to f. (In practice, $R_{lo}(f)$ is sometimes more accurate than $R_{hi}(f)$, but for most integrands $f(x)$ the rule $R_{hi}(f)$ will be more accurate than the rule $R_{lo}(f)$.) These considerations will be important in our discussion of adaptive integration in Section 5.2.3.

The DOP concept may be used to derive quadrature rules directly using the *Method of Undetermined Coefficients*. With the points $x_i, i = 0, 1, \cdots, N$, given, consider the rule $I(f) = \int_{-1}^{1} f(x)\,dx \approx \sum_{i=0}^{N} w_i f(x_i) = R(f)$. Note that we have chosen a special (canonical) interval $[-1, 1]$ here. The weights (the undetermined coefficients), $w_i, i = 0, 1, \cdots, N$, are chosen to maximize the DOP, by solving the following *equations of precision* (starting from the first and leaving out no equations)

$$\begin{aligned}
\int_{-1}^{1} 1\,dx &= \sum_{i=0}^{N} w_i 1 \\
\int_{-1}^{1} x\,dx &= \sum_{i=0}^{N} w_i x_i \\
&\vdots \\
\int_{-1}^{1} x^m\,dx &= \sum_{i=0}^{N} w_i x_i^m
\end{aligned}$$

for the weights w_i. Suppose each of these equations is satisfied, but the next equation is not satisfied; that is,

$$\int_{-1}^{1} x^{m+1}\,dx \neq \sum_{i=0}^{N} w_i x_i^{m+1}\,.$$

Then the DOP corresponds to the last power of x for which we succeeded in satisfying the corresponding equation of precision, so DOP$= m$.

Example 5.2.5. Suppose that the quadrature rule

$$R(f) = w_0 f(-1) + w_1 f(0) + w_2 f(+1)$$

estimates the integral $I(f) \equiv \int_{-1}^{1} f(x)\,dx$. What choice of the weights w_0, w_1 and w_2 maximizes the DOP of the rule?
Solution: Create a table listing the values of $I(x^m)$ and $R(x^m)$ for $m = 0, 1, 2, \cdots$.

m	$I(x^m)$	$R(x^m)$
0	2	$w_0 + w_1 + w_2$
1	0	$-w_0 + w_2$
2	$\frac{2}{3}$	$w_0 + w_2$
3	0	$-w_0 + w_2$
4	$\frac{2}{5}$	$w_0 + w_2$

To determine the three free parameters, w_0, w_1 and w_2, we solve the first three equations of precision (to give us DOP≥ 2). That is we solve $I(x^m) = R(x^m)$ for $m = 0, 1, 2$:

$$\begin{aligned} 2 &= w_0 + w_1 + w_2 \\ 0 &= -w_0 + w_2 \\ \frac{2}{3} &= w_0 + w_2 \end{aligned}$$

These three equations have the unique solution (corresponding to Simpson's rule):

$$w_0 = w_2 = \frac{1}{3}, w_1 = \frac{4}{3}.$$

However, this rule has DOP $= 3$ not DOP $= 2$ because for this choice of weights $I(x^3) = R(x^3)$ too; that is, the first four equations of precision are satisfied. (Indeed, $I(x^m) = R(x^m) = 0$ for all odd powers $m \geq 0$.) DOP$= 3$ because if DOP$= 4$ then the equations $w_0 + w_2 = \frac{2}{3}$ and $w_0 + w_2 = \frac{2}{5}$ would both be satisfied, a clear contradiction.

Problem 5.2.14. *Use the linearity of integrals and quadrature rules to prove that the two definitions of DOP are equivalent.*

Problem 5.2.15. *For the integral $\int_{-1}^{1} f(x)\,dx$, find the DOP of each of the trapezoidal, midpoint and Simpson's rule.*

Problem 5.2.16. *Derive the midpoint and trapezoidal rules by fixing the points x_i and computing the weights w_i to maximize the DOP by the method of undetermined coefficients. What is the DOP in each case?*

Problem 5.2.17. *The four–point closed Newton–Cotes rule is also known as* Weddle's rule *or the* $\frac{3}{8}$th's rule. *For the interval* $[-1,1]$ *use the equally spaced points* $x_0 = -1$, $x_1 = -\frac{1}{3}$, $x_2 = +\frac{1}{3}$ *and* $x_3 = +1$ *and determine the weights* w_0, w_1, w_2 *and* w_3 *to maximize the DOP of the rule. What is the DOP?*

Problem 5.2.18. *The two–point open Newton–Cotes rule uses points* x_1 *and* x_2 *of Problem 5.2.17. Determine the weights* w_1 *and* w_2 *to maximize the DOP of the rule. What is the DOP?*

Peano's Theorem and the Error in Integration

The next theorem, due to Peano, relates the error in using a quadrature rule to the DOP of the rule.

Theorem 5.2.2. *Peano's theorem.* Let the rule $R(f) \equiv \sum_{i=0}^{N} w_i f(x_i)$ approximating the integral $I(f) = \int_a^b f(x)\,dx$ have DOP$= m$. Suppose that the integrand $f(x)$, and its first $m+1$ derivatives $f'(x), f''(x), \cdots, f^{(m+1)}(x)$ exist and are continuous on the closed interval $[a,b]$. Then, there exists a function $K(x)$, the Peano kernel, that does not depend on the integrand $f(x)$ nor on its derivatives, for which

$$I(f) - R(f) = \int_a^b K(x) f^{(m+1)}(x)\,dx$$

When the Peano kernel $K(x)$ does not change sign on the interval of integration, using the Integral Mean Value Theorem it can be shown that there is a simpler relation

$$I(f) - R(f) = \kappa f^{(m+1)}(\eta)$$

where η is some unknown point in (a,b). In this relation, *the Peano constant* κ does not depend on the integrand f nor on its derivatives. The kernel $K(x)$ does not change sign on the interval of integration for the trapezoidal, midpoint or Simpson rules. Also, it does not change sign for the Gauss and Lobatto rules to be met later.

To calculate κ, we may use this relation directly with the special choice of integrand $f(x) = x^{m+1}$, as in the example that follows. To see this, observe that since κ is a constant we need to substitute for f in the relation a function whose $(m+1)^{\text{st}}$ derivative is a constant whatever the value of η. The only possible choice is a polynomial of exact degree $m+1$. We make the simplest such choice, $f(x) = x^{m+1}$. Observe that if we have just checked the DOP of the rule we will already have calculated m, $I(x^{m+1})$ and $R(x^{m+1})$.

Example 5.2.6. Calculate the Peano constant κ for Simpson's rule $R(f) = \frac{1}{3}\{f(-1) + 4f(0) + f(1)\}$ estimating the integral $I(f) = \int_{-1}^{1} f(x)\,dx$.
Solution: Simpson's rule has DOP $= 3$, so Peano's theorem tells us that

$$I(f) - R(f) = \int_{-1}^{1} K(x) f^{(4)}(x)\,dx = \kappa f^{(4)}(\eta)$$

Choose the integrand $f(x) = x^4$ so that the fourth derivative $f^{(4)}(x) = 4! = 24$ no longer involves x. From Example 5.2.5, $I(x^4) = \frac{2}{5}$ and $R(x^4) = \frac{2}{3}$, so $\frac{2}{5} - \frac{2}{3} = 24\kappa$; that is, $\kappa = -\frac{1}{90}$.

A higher DOP for a quadrature rule is associated with a larger number of integrand evaluations, but as the DOP increases the rule becomes more accurate. So, not surprisingly, greater accuracy comes at higher cost. The appropriate balance between cost and accuracy is problem dependent, but most modern, general purpose software (with the notable exception of MATLABś `quad`) uses

integration rules of a higher DOP than any we have encountered so far. However, this software does not use the high DOP Newton–Cotes rules (that is, those with many points) because the weights, w_i, of these rules oscillate in sign for large numbers of points N. This oscillation in the weights can lead to (catastrophic) cancelation when summing the rule for smooth slowly changing integrands hence resulting in a significant loss of numerical accuracy.

We end this section with a reworking of Example 5.2.6 for an interval of integration of length h. This way we can observe the behavior of the error as $h \to 0$. This is of use when considering composite rules made up of rules on N intervals each of length h as for the composite trapezoidal and midpoint rules. Also this analysis enables us to understand the effect of bisecting intervals in the globally adaptive integration algorithm to be described in the next section.

Example 5.2.7. Use the DOP approach to determine Simpson's rule and the Peano constant, for approximating the integral $I(f) = \int_{-\frac{h}{2}}^{\frac{h}{2}} f(x)\,dx$.

Solution: Simpson's rule for approximating $I(f)$ has the form $R(f) = w_0 f(-\frac{h}{2}) + w_1 f(0) + w_2 f(\frac{h}{2})$. The usual table is

m	$I(x^m)$	$R(x^m)$
0	h	$w_0 + w_1 + w_2$
1	0	$(-w_0 + w_2)\frac{h}{2}$
2	$\frac{h^3}{12}$	$(w_0 + w_2)\frac{h^2}{4}$
3	0	$(-w_0 + w_2)\frac{h^3}{8}$
4	$\frac{h^5}{80}$	$(w_0 + w_2)\frac{h^4}{16}$

The first three equations of precision

$$\begin{aligned} h &= w_0 + w_1 + w_2 \\ 0 &= -w_0 + w_2 \\ \frac{h}{3} &= w_0 + w_2 \end{aligned}$$

have the unique solution

$$w_0 = w_2 = \frac{h}{6}, \quad w_1 = \frac{4h}{6}$$

Simpson's rule has DOP = 3 not DOP = 2 because $I(x^3) = R(x^3) = 0$. Peano's theorem tells us that the error

$$I(f) - R(f) = \kappa f^{(4)}(\eta)$$

To determine Peano's constant κ consider the special choice $f(x) = x^4$. From the table above, $I(x^4) = \frac{h^5}{80}$ and $R(x^4) = \left(\frac{h}{3}\right)\frac{h^4}{16} = \frac{h^5}{48}$, so $\frac{h^5}{80} - \frac{h^5}{48} = 24\kappa$; that is, $\kappa = -\frac{h^5}{2880}$.

Problem 5.2.19. *Calculate Peano's constant κ for the trapezoidal rule approximating the integral $I(f) = \int_a^b f(x)\,dx$. [Consider the special choice of integrand $f(x) = x^{m+1}$ where m is the DOP of the trapezoidal rule.]*

Problem 5.2.20. *Calculate Peano's constant κ for the midpoint rule when used to approximate $I(f) = \int_{-h}^{h} f(x)\,dx$.*

Problem 5.2.21. *For any quadrature rule with DOP $= m$ approximating $\int_{-\frac{h}{2}}^{\frac{h}{2}} f(x)\,dx$ and with Peano kernel that does not change sign on $[-\frac{h}{2}, \frac{h}{2}]$, show that the error has the form $\bar{\kappa} h^{m+2} f^{(m+1)}(\eta)$ where $\bar{\kappa}$ is a constant independent of the integrand $f(x)$ and of h. [Hint: We can assume the error has the form $\kappa f^{(m+1)}(\eta)$ then we show that κ always has a factor h^{m+2}.]*

Problem 5.2.22. *Write down a composite Simpson rule for the integral $\int_a^b f(x)\,dx$ where Simpson's rule is to be used on each subinterval $[x_{i-1}, x_i]$ of the interval $[a, b]$. Derive the error term for the composite rule. [Hint: Use the Generalized Mean Value Theorem for sums and the analysis of the error in Simpson's rule in Example 5.2.7.]*

5.2.3 Adaptive Integration

In a course on the calculus of one variable, we meet limits of Riemann sums as a method for defining definite integrals. The composite midpoint, trapezoidal and Simpson's rules are all Riemann sums, as are the Gauss and Lobatto rules that we will meet later.

Thinking in terms of the composite midpoint rule, if we let $h \to 0$, by exploiting the fact that the rule is a Riemann sum we can show that the integral exists, assuming only that the function f is continuous on the interval of integration. Also, this observation shows that as long as the function f is continuous on the interval of integration, using the composite midpoint rule with a sufficiently small step size h will give an accurate approximation to the integral. But, it gives us no information as to the rate of convergence of the composite midpoint rule to the integral as $h \to 0$. In contrast, from the expression for the error in the composite midpoint rule we know that if the second derivative of the integrand $f''(x)$ is continuous on the interval of integration then the composite midpoint rule converges to the integral at a rate proportional to h^2 as $h \to 0$. So, additional smoothness in the integrand $f(x)$ ensures a reasonable rate of convergence. Indeed, if we only know that the first derivative of the integrand $f'(x)$ is continuous on the interval of integration (that is, we have no information about the second derivative), we can use a form Peano's theorem to show that the composite midpoint rule converges to the integral at a rate proportional at least to h as $h \to 0$. However, more smoothness (e.g. continuous third derivative of the integrand) does not permit us to improve on the $O(h^2)$ convergence rate.

For difficult integrals, that is for integrals where the integrand is not slowly varying everywhere on the interval of integration, it is tempting to use a composite quadrature rule with subintervals of equal length h, since we know, from the discussion above, that, for the composite quadrature rules we have seen, making $h \to 0$ also forces the error to zero. However, this approach is usually very inefficient. Subintervals sufficiently short that the quadrature rule will integrate accurately the integrand where it is worst behaved are usually far too short for those parts of the integration interval $[a, b]$ where the integrand is relatively well behaved; that is, the error on these latter subintervals will be very small and determining the integral this unnecessarily accurately on these subintervals can be very inefficient, "wasting" many integrand evaluations. So, most modern, general purpose codes for integration are **adaptive**. That is, they adapt to the integrand's behavior the length of the subintervals on which the quadrature rules are applied, hence aiming to use a small number of subintervals of varying and appropriate lengths, and so not to "waste" integrand evaluations. The two basic types of adaptive algorithms are *global* and *local*. To illustrate the use of quadrature rules and to identify what else we need to study, we outline a **global adaptive integration algorithm**.

Let the problem be to estimate the integral $I(f, T) = \int_T f(x)\,dx$ where T is the interval of integration. Suppose we have quadrature rules $R_1(f, T^*)$ and $R_2(f, T^*)$ that each approximate the integral $I(f, T^*)$ for any specified subinterval T^* of T. Furthermore, assume that the rule R_2 is "more accurate" than the rule R_1. By "more accurate" we mean one of the following:

- The DOP of the rule R_2 is greater than the DOP of the rule R_1. Generally this implies that the error for R_2 is expected to be smaller than the error for R_1 for most integrands on most intervals of integration.
- The rule R_2 is the same as R_1 but it is applied on the two halves of the interval. So, if the DOP of R_1 is p the expected error in R_2 on half the interval is a factor of $\left(\frac{1}{2}\right)^{p+2}$ smaller than for R_1. Since we expect approximately the same error on each half interval, the error on both half intervals combined will be a factor of $2\left(\frac{1}{2}\right)^{p+2}$ smaller than for R_1. For example, if we use Simpson's rule $p = 3$ so the error in Simpson's rule at half the step is expected to be a factor of $2\frac{1}{32} = \frac{1}{16}$ smaller than for the basic rule, and only two extra integrand evaluations are needed to evaluate it (the other three are involved from the basic rule).

Traditionally, the error in using the rule $R_1(f, T^*)$ to approximate the integral $I(f, T^*)$ is estimated by

$$E(f, T^*) \equiv |I(f, T^*) - R_1(f, T^*)| \approx |R_2(f, T^*) - R_1(f, T^*)|$$

where latter approximation is usually justified by arguing that the rule $R_2(f, T^*)$ is a very accurate approximation of the integral $I(f, T^*)$. We approximate the integral $\int_a^b f(x)\,dx$ to a specified accuracy tolerance *tol*; that is, we want $E(f) \leq tol$ where $E(f)$ is our estimate of the global error in the integration.

At any stage of the algorithm the original interval T has been divided into a number of subintervals $T_i, i = 1, 2, \cdots, n$. Suppose that the rules $R_1(f, T_i)$ computed so far have been accumulated in $R(f)$ and the corresponding accumulated error estimates are held in $E(f)$. In Figure 5.3, we show the case $n = 5$. Assume that $E(f) > tol$; if $E(f) < tol$ the algorithm will terminate. For

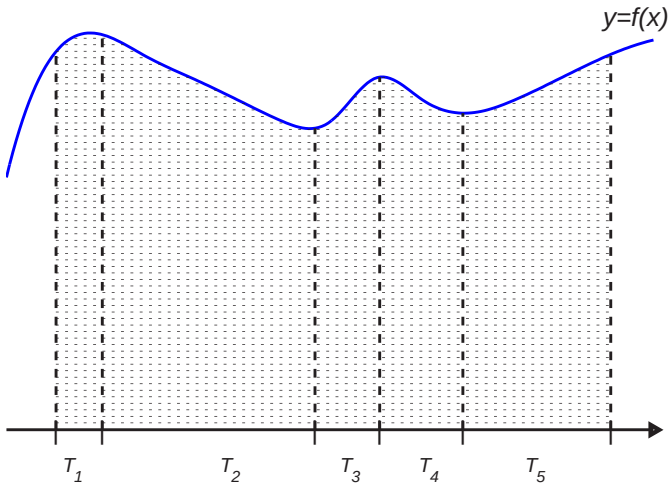

Figure 5.3: One step of a global adaptive integration strategy.

the sake of argument, we suppose that the error estimate with largest magnitude (out of the set of error estimates E_1, E_2, E_3, E_4 and E_5) is E_2 corresponding to the subinterval T_2. Then, we bisect the interval T_2 into two equal subintervals denoted T_{2L} (for Left) and T_{2R} (for Right). Next, we estimate the integral and the magnitude of the error on these subintervals. Finally, we modify the estimates of the overall definite integral and the error as appropriate. Thus, before the bisection of interval T_2 we have

$$\begin{aligned} R(f) &= R_1 + R_2 + R_3 + R_4 + R_5 \\ E(f) &= E_1 + E_2 + E_3 + E_4 + E_5 \end{aligned}$$

and after the bisection of T_2 we have (remember, := is the programming language assignment operation, not mathematical equality)

$$\begin{array}{rcrcl} R(f) & := & R(f) - R_2 + (R_{2L} + R_{2R}) & = & R_1 + R_{2L} + R_{2R} + R_3 + R_4 + R_5 \\ E(f) & := & E(f) - E_2 + (E_{2L} + E_{2R}) & = & E_1 + E_{2L} + E_{2R} + E_3 + E_4 + E_5 \end{array}$$

Next, we must check whether $E(f) \leq tol$. If it is, we are done; if it is not, we again search for the subinterval T_i of T with the largest error estimate E_i in magnitude and proceed as above. In Figure 5.4, we present an implementation of the approach described above. For simplicity, in this implementation the subintervals T_{2L} and T_{2R} in the example above would be called $T(2)$ and $T(last)$, respectively.

Global Adaptive Integration

Input: integrand, $f(x)$
endpoints, a, b
accuracy tolerance, tol

Output: integral approximation, $R(f) \approx \int_a^b f(x)\,dx$
error estimate, $E(f)$

$T(1) := T = [a, b]; last := 1$
Compute $R1(f, T(1)), R2(f, T(1))$
$R(f) = R(f, T(1)) := R1(f, T(1))$
$E(f) = E(f, T(1)) := |R2(f, T(1)) - R1(f, T(1))|$
while $E(f) > tol$ do
  Find the index i of the largest error estimate $E(f, T(i))$
  $Rold := R(f, T(i)); Eold := E(f, T(i))$
  $last := last + 1$
  $T(last) :=$ the first half of the interval $T(i)$
  $T(i) :=$ the second half of the interval $T(i)$
  Compute $R(f, T(i)), R(f, T(last)), E(f, T(i)), E(f, T(last))$
  $R(f) := R(f) + (R(f, T(i)) + R(f, T(last)) - Rold)$
  $E(f) := E(f) + (E(f, T(i)) + E(f, T(last)) - Eold)$
end while

Figure 5.4: Pseudocode *Global Adaptive Integration*.

In the algorithm in Figure 5.4 we have made a small change from the description given above. We have arranged to make the subtraction of the values from the discarded intervals and the addition of the values from the newly created intervals together so that at each iteration we are adding a (potentially small) correction to the current approximation to the integral and the associated error. This way we avoid possible catastrophic cancelation when computing the integral and error.

The global adaptive integration algorithm in Figure 5.4 terminates for all integrands $f(x)$ that are sufficiently smooth. When the algorithm terminates, $R(f)$ approximates the integral $I(f)$ and $E(f)$ estimates the error. The estimate, but not the actual error, is guaranteed to be smaller than the accuracy tolerance tol. Here, tol is assumed to be greater than ϵ_{DP}; if this is not the case then the accuracy requirement cannot be met. To choose rules R_1 and R_2 valid for any interval T^* that might arise in the adaptive integration algorithm, first we choose rules R_1 and R_2 appropriate for a canonical interval, then we transform the rules from the canonical interval to the interval T^*.

Problem 5.2.23. *Show that the composite midpoint rule $R_{CM}(f, h)$ is a Riemann sum.*

Problem 5.2.24. *In the example above, argue why the new value of the error estimate $E(f)$ is normally expected to be smaller than the previous value of $E(f)$? [Hint: Argue that, if the DOP's of the integration rules R_1 and R_2 are both greater than one, then $E_{2L} + E_{2R} < E_2$ usually.]*

Problem 5.2.25. *Under the assumptions of Problem 5.2.24, argue that the globally adaptive integration algorithm outlined in this section will normally terminate.*

5.2.4 Gauss and Lobatto Rules

High order integration rules used widely in adaptive quadrature are mainly of Gaussian type. Here, we describe two such classes of rules used in various software packages. First, we discuss the Gauss rules which are open, and their extension to the (open) Gauss-Kronrod rules used for estimating the error. Then, we discuss the Lobatto rules which are closed, that is they include the endpoints of the range of integration.

Gauss Rules

The most popular choices for rules for adaptive integration are the Gauss rules, for which the canonical interval is $[-1, +1]$; here, $I(f) \equiv \int_{-1}^{+1} f(x)\,dx \approx \sum_{i=0}^{N} w_i f(x_i) \equiv R(f)$ where all of the weights w_i, $i = 0, 1, \cdots, N$, and all of the points x_i, $i = 0, 1, \cdots, N$, are chosen to maximize the DOP. In Problems 5.2.26 — 5.2.28, you are asked to find the weights and points in some simple Gauss rules. The equations for the weights and points are nonlinear though not difficult to solve. In practice, Gauss rules with much larger numbers of points than in these problems are generally used. Deriving these practical rules by maximizing the DOP and solving the resulting large systems of nonlinear equations would be difficult, and tedious. Fortunately there are systematic approaches to deriving these rules based on a more sophisticated mathematical theory and the values of the weights and points have been tabulated for values N as large as are ever likely to be needed in practice.

Let us now consider the properties of the Gauss rules. For each value of $N \geq 0$, for the $(N+1)$–point Gauss rule we have:

(1) All the weights $w_i > 0$.

(2) All the points $x_i \in (-1, +1)$.

(3) **Symmetry** – The points x_i are placed symmetrically around the origin and the weights w_i are correspondingly symmetric. For N odd, the points satisfy $x_0 = -x_N, x_1 = -x_{N-1}$ etc. and the weights satisfy $w_0 = w_N, w_1 = w_{N-1}$ etc. For N even, the points and weights satisfy the same relations as for N odd, plus $x_{N/2} = 0$.

(4) The points $\overline{x}_i$ of the N–point Gauss rule interlace the points x_i of the $(N+1)$–point Gauss rule: $-1 < x_0 < \overline{x}_0 < x_1 < \overline{x}_1 < x_2 < \cdots < x_{N-1} < \overline{x}_{N-1} < x_N < +1$.

(5) The Gauss rules are interpolatory quadrature rules; that is, after the points $x_0, x_1, \cdots, x_N$ have been determined, then the weights $w_0, w_1, \cdots, w_N$ may be computed by integrating over the interval $[-1, +1]$ the polynomial of degree N, $q_N(x)$, that interpolates the integrand $f(x)$ at the points $x_0, x_1, \cdots, x_N$.

(6) The DOP of the $(N+1)$–point Gauss rule is $2N + 1$.

If the points $x_0, x_1, \cdots, x_N$ had been fixed arbitrarily, then, by analogy with those rules we have derived previously, with $N+1$ free weights we would expect DOP $= N$. But for the Gauss quadrature rules the points are chosen to increase the DOP to $2N+1$. (This shouldn't be a surprise: the Gauss quadrature rule has a total of $2(N+1)$ unknowns, taking the weights and the points together, and it is plausible that they can be chosen to solve all the $2(N+1)$ equations of precision $R(x^k) = I(x^k)$ for $k = 0, 1, \cdots, 2N+1$.)

So, this describes how to choose the rule R_1. Typically, values of N in the range 4 to 30 are used in real-life applications. Given that the rule R_1 has been chosen to be a $(N+1)$–point Gauss rule with DOP $= 2N+1$, a sensible choice for the rule R_2 is the $(N+2)$–point Gauss rule which has DOP $= 2(N+2)+1 = (2N+1)+2$; that is, its DOP is 2 higher than for the corresponding Gauss $(N+1)$–point rule. Also, the interlacing property is important because it guarantees a good "sampling of the integration interval" by the points that the quadrature rule and the error estimate together provide.

But, for the same expense as evaluating the integrand $f(x)$ at an additional $N+2$ points to compute the Gauss $(N+2)$–point rule, we can achieve DOP $= 3N+2$ in R_2 by using a Gauss–Kronrod (GK) rule. Without going into details, this GK rule reuses the integrand evaluations from the Gauss $(N+1)$–point rule but with a new set of weights and adds $N+2$ new points and related weights. Then, the $N+2$ unknown points and the $(N+1)+(N+2) = 2N+3$ weights are used to maximize the DOP of the GK rule. The properties of the GK rules are very similar to those of the Gauss rules, including: all the points lie in the integration interval $(-1,+1)$, all the weights are positive, and there is an interlacing property of the points of the $(N+1)$–point Gauss rule with the $N+2$ new points of the corresponding $(2N+3)$–point GK rule.

Problem 5.2.26. *You are to derive Gauss rules by solving the equations of precision*

(1) Show that the Gauss one–point rule is the midpoint rule.

(2) By the symmetry properties the Gauss two–point rule has the simplified form:

$$\int_{-1}^{1} f(x)\,dx \approx w_0 f(x_0) + w_0 f(-x_0)$$

Find w_0 and x_0 to maximize the DOP. Find the Peano constant κ.

Problem 5.2.27. *By the symmetry properties the Gauss three–point rule has the simplified form:*

$$\int_{-1}^{1} f(x)\,dx \approx w_0 f(x_0) + w_1 f(0) + w_0 f(-x_0)$$

Compute the values w_0, w_1 and x_0 to maximize the DOP. Find the Peano constant κ.

Problem 5.2.28. *What is the simplified form of the Gauss four–point rule implied by the symmetry properties? How many unknowns are there? Find the unknowns by maximizing the DOP.*

Problem 5.2.29. *Why do we want all the points in the Gauss rules to lie in the interval $(-1,+1)$? Why do we want all the weights in the Gauss rules to be positive? [Hint: Since DOP > 0, we must satisfy the first equation of precision $\sum_{i=0}^{N} w_i = 2$.]*

Problem 5.2.30. *The Gauss one–point rule is the midpoint rule. Show that the corresponding Gauss–Kronrod three-point rule is the same as the Gauss three–point rule.*

Lobatto Rules

Another popular choices of rules for adaptive integration are the Lobatto rules, for which the canonical interval is again $[-1,+1]$. These rules are close relatives of the Gauss rules. The (N+1)-point rule has the form $I(f) \equiv \int_{-1}^{+1} f(x)\,dx \approx w_0 f(-1) + \sum_{i=1}^{N-1} w_i f(x_i) + w_N f(1) \equiv R(f)$ where the all weights w_i, $i = 0, 1, \cdots, N$, and all the unknown points x_i, $i = 1, 2, \cdots, N-1$, are chosen to maximize the DOP. The points ± 1 are included since the *endpoints* of the interval of integration are always used in Lobatto integration. In Problems 5.2.31— 5.2.33, you are asked to find the weights and points in some simple Lobatto rules. In reality, Lobatto rules with larger numbers of points than in these problems are often used.

Let us now consider the properties of the Lobatto rules. For each value of $N \geq 0$, for the $(N+1)$–point Lobatto rule we have:

(1) All the weights $w_i > 0$.

(2) The points $x_i \in (-1, +1), 1 = 1, 2, \ldots, N-1$.

(3) **Symmetry** – The points x_i are placed symmetrically around the origin and the weights w_i are correspondingly symmetric. For N odd, the points satisfy $x_1 = -x_{N-1}, x_2 = -x_{N-2}$ etc. and the weights satisfy $w_0 = w_N, w_1 = w_{N-1}, w_2 = w_{N-2}$ etc. For N even, the points and weights satisfy the same relations as for N odd, plus $x_{N/2} = 0$.

(4) The points $\overline{x}_i$ of the N–point Lobatto rule interlace the points x_i of the $(N+1)$–point Lobatto rule: $-1 < x_1 < \overline{x}_1 < x_2 < \overline{x}_2 < x_3 < \cdots < x_{N-2} < \overline{x}_{N-2} < x_{N-1} < +1$. Note that the points ± 1 are points for all the Lobatto rules.

(5) The Lobatto rules are interpolatory quadrature rules; that is, after the points $x_1, x_2, \cdots, x_{N-1}$ have been determined, then the weights $w_0, w_1, \cdots, w_N$ may be computed by integrating over the interval $[-1, +1]$ the polynomial of degree N, $q_N(x)$, that interpolates the integrand $f(x)$ at the points $-1, x_1, x_2 \cdots, x_{N-1}, 1$.

(6) The DOP of the $(N+1)$–point Lobatto rule is $2N-1$.

If the points $x_1, x_2, \cdots, x_{N-1}$ had been fixed arbitrarily, then, by analogy with those rules we have derived previously, with $N+1$ free weights we would expect DOP$= N$, or in some (symmetric) cases DOP$=N+1$. But in the Lobatto quadrature rules the points are chosen to increase the DOP to $2N-1$. (This shouldn't be a surprise: the Lobatto $(N+1)$–point quadrature rule has a total of $2N$ unknowns, taking the weights and the points together, and it is plausible that they can be chosen to solve all the $2N$ equations of precision $R(x^k) = I(x^k)$ for $k = 0, 1, \cdots, 2N-1$.)

So, if we are using Lobatto rules for adaptive integration this describes how to choose the rule R_1. Typically, values of N in the range 4 to 10 are used in real-life applications. For the rule R_2, given that the rule R_1 has been chosen to be an $(N+1)$–point Lobatto rule with DOP $= 2N-1$, a sensible choice is the $(N+2)$–point Lobatto rule which has DOP $= 2(N+1) - 1 = (2N-1) + 2$; that is, its DOP is 2 higher than for the corresponding Lobatto $(N+1)$–point rule. The resulting interlacing property is important because it guarantees a good "sampling of the integration interval" by the points that the quadrature rule and the error estimate together provide. Also, there are two endpoints in common, so the total number of points to compute the integral and error estimate is $(N+1) + (N+2) - 2 = 2N+1$. In addition, it is possible to organize the computation so that integrand evaluations at the ends of subintervals are shared across intervals hence effectively reducing the number of evaluations for the integral and error estimate to $2N$ per interval.

Problem 5.2.31. *You are to derive Lobatto rules by solving the equations of precision*

(1) Show that the Lobatto three–point rule is Simpson's rule.

(2) By the symmetry properties the Lobatto four–point rule has the simplified form:

$$\int_{-1}^{1} f(x)\,dx \approx w_0 f(-1) + w_1 f(-x_1) + w_1 f(x_1) + w_0 f(1)$$

Find w_0, w_1 and x_1 to maximize the DOP. Find the Peano constant κ for each rule.

Problem 5.2.32. *By the symmetry properties the Lobatto five–point rule has the simplified form:*

$$\int_{-1}^{1} f(x)\,dx \approx w_0 f((-1) + w_1 f(-x_1) + w_2 f(0) + w_1 f(x_1) + w_0 f(1)$$

Compute the values w_0, w_1, w_2 and x_1 to maximize the DOP. Find the Peano constant κ for this rule.

Problem 5.2.33. *What is the simplified form of the Lobatto six–point rule implied by the symmetry properties? How many unknowns are there? Find the unknowns by maximizing the DOP.*

Problem 5.2.34. *Why do we want all the points in the Lobatto rules to lie in the interval* $(-1, +1)$*? Why do we want all the weights in the Lobatto rules to be positive? [Hint: Since DOP* > 0*, we must satisfy the first equation of precision* $\sum_{i=0}^{N} w_i = 2$*.]*

Problem 5.2.35. *Explain how you know (without deriving the formula) that the Lobatto four– point rule is not Weddle's (four–point) closed Newton-Cotes rule.*

Comparing Gauss and Lobatto rules

In terms of degree of precision the Gauss N–point rule is comparable to the Lobatto $(N+1)$–point rule; the DOP $= 2N + 1$ in each case. If we consider the number of points where we must evaluate the integrand (for an amount of accuracy delivered) as the measure of efficiency then it seems that the Gauss rules are slightly the more efficient.

However, in the context of adaptive quadrature we observe that the point corresponding to $x = 1$ in one interval is the same as the point corresponding to $x = -1$ in the next. So, assuming we keep the values of the integrand at the interval endpoints after we have evaluated them and reuse them where appropriate then the cost of evaluating a Lobatto rule is reduced by about one integrand evaluation giving about the same cost as the Gauss rule with the same DOP. A further saving for the Lobatto rules arises in computing the error estimates as the integrand evaluations at the interval endpoints have already been calculated and if we reuse them in the Lobatto $(N+2)$–point rule the cost of the Lobatto error estimate is one less integrand evaluation than the for the corresponding Gauss rule. So, overall an efficient computation with the Lobatto rules seems likely to be at least as efficient as one with the Gauss rules.

An alternative approach to estimating the error in the Lobatto rules is to use a Lobatto-Kronrod rule. So, for a Lobatto $(N+1)$–point rule we construct a Lobatto-Kronrod $(2N+1)$–point rule reusing all the points in the Lobatto rule and adding N interior points symmetrically interlacing the interior points of the Lobatto rule. The cost of this is the same as using the Lobatto $(N+2)$–point rule but the DOP is higher.

There are other factors involved:

- We have discussed using Gauss and Lobatto rules of the same degree of precision. This implies that the rules behave similarly as the step size tends to zero but the actual error depends also on the size of the Peano constant. For example, the Gauss two-point rule and Simpson's rule (the Lobatto three–point rule) have the same degree of precision but the Peano constant for the Gauss two–point rule is $1/135$ and that for Simpson's rule (the Lobatto three–point rule) is $1/90$ hence the Gauss two–point rule is about 50% more accurate on the same interval. The sizes of the Peano constants for other Gauss and Lobatto rules of the same DOP similarly favor the Gauss rules.

- Suppose we want to compute the integral

$$\int_0^1 \frac{\cos x}{\sqrt{x}}\, dx$$

  which exists even though there is a singular point at $x = 0$. If we use a Gauss rule in our adaptive quadrature scheme it will never need the integrand value at $x = 0$, and will, with some effort, compute an accurate value. It takes some effort because the adaptive scheme will subdivide the interval a number of times near $x = 0$ before it computes the answer sufficiently accurately to meet the user's error tolerance. In contrast, using a Lobatto rule in the adaptive algorithm will fail immediately.

There are ways round the latter difficulty which may be superior even for the Gauss rules. One, relatively simple, approach is to expand the integrand in a series about the singularity then integrate the series over a short interval near the singularity and the original integrand by your

favorite adaptive scheme elsewhere. For example, in the case of $\int_0^1 \frac{\cos x}{\sqrt{x}}\,dx$, we could expand $\cos x$ in Taylor series about $x = 0$, then split the interval as follows:

$$\int_0^1 \frac{\cos x}{\sqrt{x}}\,dx = \int_0^\delta \frac{\cos x}{\sqrt{x}}\,dx + \int_\delta^1 \frac{\cos x}{\sqrt{x}}\,dx \approx \int_0^\delta \frac{1 - x^2/2}{\sqrt{x}}\,dx + \int_\delta^1 \frac{\cos x}{\sqrt{x}}\,dx$$

then choose δ sufficiently small so the series is accurate enough but not so small that the second integral is too irregular near the point $x = \delta$. Then the first integral may be calculated using simple Calculus and the second by your favorite adaptive scheme.

Problem 5.2.36. *Compare the accuracies of the following pairs of rules*

- *The Gauss three-point rule and the Lobatto four point rule*
- *The Gauss four–point rule and the Lobatto five–point rule*

Problem 5.2.37. *How accurate is the expression* $(1 - x^2/2)$ *as an approximation to* $\cos x$ *on the interval* $[0, \delta]$*? How accurate would be the Taylor series with one more non-zero term? Calculate the approximation* $\int_0^\delta \frac{1-x^2/2}{\sqrt{x}}\,dx$ *and the corresponding approximation using one more non-zero term in the Taylor series. How small must* δ *be for these approximations to agree to four significant digits? [Hint: The Taylor series for* $\cos x$ *is an alternating series.*

5.2.5 Transformation from a Canonical Interval

When describing adaptive integration it was assumed that we have available quadrature rules for any interval of integration. However, almost all our derivations and discussions of quadrature rules have been for canonical intervals such as $[-1, 1]$ and $[-h, h]$. So, we consider how to transform a quadrature rule for a canonical interval of integration, chosen here as $[-1, 1]$, to a quadrature rule on a general interval of integration $[a, b]$.

The standard change of variable formula of integral calculus using the transformation $x = g(t)$ is

$$\int_{g(c)}^{g(d)} f(x)\,dx = \int_c^d f(g(t))g'(t)\,dt$$

One possible approach is to choose the transformation $g(t)$ as the straight line interpolating the points $(-1, a)$ and $(+1, b)$; that is, using the Lagrange form of the interpolating polynomial,

$$g(t) = a\frac{t - (+1)}{(-1) - (+1)} + b\frac{t - (-1)}{(+1) - (-1)} = \frac{(b - a)}{2}t + \frac{(b + a)}{2}\,.$$

The transformation $g(t)$ maps the canonical interval onto the interval of interest.

For this transformation $g(t)$ we have $g'(t) = \dfrac{(b - a)}{2}$ and the change of variable formula reads

$$\int_a^b f(x)\,dx = \frac{(b - a)}{2}\int_{-1}^{+1} f\left(\frac{(b - a)}{2}t + \frac{(b + a)}{2}\right)\,dt$$

Now, assume that for the canonical interval $[-1, 1]$ we have a quadrature rule

$$I^*(h) \equiv \int_{-1}^{+1} h(t)\,dt \approx \sum_{i=0}^{N} w_i^* h(t_i^*) \equiv R^*(h)$$

then, substituting for $h(t) = f(g(t))$, we have

$$\begin{aligned} I(f) \equiv \int_a^b f(x)\,dx &= \frac{(b-a)}{2}\int_{-1}^{+1} f\left(\frac{(b-a)}{2}t + \frac{(b+a)}{2}\right)\,dt \\ &= \frac{(b-a)}{2} I^*(f\circ g) \approx \frac{(b-a)}{2} R^*(f\circ g) \\ &= \frac{(b-a)}{2}\sum_{i=0}^{N} w_i^* f\left(\frac{(b-a)}{2}t_i^* + \frac{(b+a)}{2}\right) = \sum_{i=0}^{N} w_i f(x_i) \equiv R(f) \end{aligned}$$

where in $R(f)$ the weights $w_i = \dfrac{(b-a)}{2}w_i^*$ and the points $x_i = \dfrac{(b-a)}{2}t_i^* + \dfrac{(b+a)}{2}$.

The DOP of the transformed quadrature rule is the same as the DOP of the canonical quadrature rule. The error term for the canonical quadrature rule may be transformed using $g(t)$ to obtain the error term for the transformed quadrature rule. However, this error term can more easily be determined by applying Peano's theorem directly.

Problem 5.2.38. *Here we see how to use the transformation presented in this section to use rules derived on a canonical interval for integration on a general interval*

(1) Transform the Gauss 2–point quadrature rule given for the canonical interval $[-1,+1]$ to the interval $[a,b]$. Show that the DOP of the transformed quadrature rule is 3 and determine Peano's constant.

(2) Repeat (1) for the Gauss 3–point quadrature rule.

Problem 5.2.39. *Transform the Gauss 2–point quadrature rule for the canonical interval $[-h,+h]$ to the interval $[a,a+h]$. Show that the DOP of the transformed quadrature rule is 3.*

5.3 Matlab Notes

5.3.1 Differentiation

Symbolic Differentiation

When we need to find the derivative $f'(x)$ of a given function $f(x)$ then usually our starting point should be the MATLAB symbolic algebra package which is a part of Student MATLAB. (The package is derived from the Maple symbolic package.)

MATLAB "knows" the derivative of all the mathematical functions that it can represent and evaluate. These functions include the trigonometric, exponential, logarithmic, and hyperbolic functions and their inverses, and many other less familiar functions. It uses the "rules" of differentiation (product, quotient, chain rule etc.) to compute the derivative of combinations of functions.

For example, the sequence of MATLAB instructions

```
syms z
f = atan(3*z);
diff(f)
```

produces the result

```
ans =

3/(1+9*z^2)
```

which you can verify is correct. Observe that you need to indicate to MATLAB that you are using symbolic variables, by specifying them in the `syms` line. Without this specification MATLAB would not know which of its two `diff` functions to apply.

The more complicated sequence

```
syms z
f = sin(4*cos(3*z))/exp(5*(1+z^2));
diff(f)
```

produces the (correct) result

```
ans =

-12*cos(4*cos(3*z))*sin(3*z)/exp(5+5*z^2)-10*sin(4*cos(3*z))/exp(5+5*z^2)*z
```

MATLAB has the ability to "simplify" the result, which is taken to mean to represent the answer using as short a string of symbols as possible. So, the MATLAB statements

```
syms z
f = sin(4*cos(3*z))/exp(5*(1+z^2));
simplify(diff(f))
```

computes the "simplified" result

```
ans =

-2*(6*cos(4*cos(3*z))*sin(3*z)+5*sin(4*cos(3*z))*z)*exp(-5-5*z^2)
```

Though this simplification seems trivial, MATLAB went through a long sequence of steps and tried a variety of "tricks" to arrive at it. To see what MATLAB tried replace `simplify` by `simple`.

In cases where the "function" that we wish to differentiate is represented by a program (as is often the case in real world applications), it has become standard practice recently to use *automatic differentiation*. This process is not available directly in MATLAB though there are several packages that implement versions of it. These packages work by parsing the program then using the chain and other differentiation rules automatically to produce either numerical values of a derivative or a program to evaluate the derivative.

Numerical Differentiation

In those cases where we need to resort to numerical differentiation we can use the MATLAB function `gradient` which is designed for approximating the gradient of a multivariate function from a table of values, but can be used in our one dimensional case.

Example 5.3.1. Evaluating $\sin(x)$ on the interval $[0, \frac{\pi}{2}]$ using an equispaced mesh with mesh size $h = \pi/20$ and calculating the derivative (gradient) and the error at each mesh point, we obtain the results in Table 5.2 using the MATLAB code

```
N = 10;
h = pi/(2 * N);
x = 0 : h: pi/2;
f = sin(x);
fx = gradient(f, h);
err = fx - cos(x);
for i = 1 : N + 1
    disp(sprintf('%d %d %d', x(i), fx(i), err(i)))
end
```

Observe the lower accuracy at the right end point due to using first order differences there as against second order differences at internal points. (This problem persists for smaller step sizes.) The second argument in `gradient` is the step size. The step size may be replaced by a vector of locations for an unequally spaced mesh.

x	gradient	error
0	9.958927e-001	-4.107265e-003
1.570796e-001	9.836316e-001	-4.056698e-003
3.141593e-001	9.471503e-001	-3.906241e-003
4.712389e-001	8.873469e-001	-3.659600e-003
6.283185e-001	8.056941e-001	-3.322847e-003
7.853982e-001	7.042025e-001	-2.904275e-003
9.424778e-001	5.853711e-001	-2.414190e-003
1.099557e+000	4.521258e-001	-1.864659e-003
1.256637e+000	3.077478e-001	-1.269215e-003
1.413717e+000	1.557919e-001	-6.425178e-004
1.570796e+000	7.837846e-002	7.837846e-002

Table 5.2: Using `gradient` to approximate the derivative of $\sin x$

Alternatively, we can fit with a cubic spline and differentiate the resulting function. This requires us to use the function `unmkpp` to determine the coefficients of the cubic polynomial pieces. To be specific, given a set of data points, (x_i, f_i), the following MATLAB commands compute the spline and its coefficients:

```
S = spline(x, f);
[x, a, N, k] = unmkpp(S);
```

Here, `N` is the number of polynomial pieces, and `a` is an `N`×`k` array whose ith row contains the coefficients of the ith polynomial. That is, if

$$S_{3,N}(x) = \begin{cases} a_{11}(x-x_0)^3 + a_{12}(x-x_0)^2 + a_{13}(x-x_0) + a_{14} & x_0 \le x \le x_1 \\ a_{21}(x-x_1)^3 + a_{22}(x-x_1)^2 + a_{23}(x-x_1) + a_{24} & x_1 \le x \le x_2 \\ \vdots & \vdots \\ a_{N1}(x-x_{N-1})^3 + a_{N2}(x-x_{N-1})^2 + a_{N3}(x-x_{N-1}) + a_{N4} & x_{N-1} \le x \le x_N \end{cases}$$

then `unmkpp` returns the array

$$a = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N1} & a_{N2} & a_{N3} & a_{N4} \end{bmatrix}.$$

Example 5.3.2. Evaluating $\sin(x)$ on the interval $[0, \frac{\pi}{2}]$ using an equispaced mesh with mesh size $h = \pi/20$ and calculating the derivative and the error at each mesh point using the MATLAB function `spline`, we obtain the results in Table 5.3 using the MATLAB code

```
N = 10;
h = pi/(2 * N);
x = 0 : h : pi/2;
f = sin(x);
```

```
S = spline(x, f);
[x, a] = unmkpp(S);
for i = 1 : N
    der(i) = a(i, 3);
end
der(N + 1) = 3*a(N, 1)* h^2 + 2*a(N, 2)*h + a(N, 3);
err = der - cos(x);
for i = 1 : N + 1
    disp(sprintf('%d %d %d', x(i), der(i), err(i)))
end
```

Note the special treatment of the last point which is only at the *end* of an interval. Observe is Table 5.3 that the errors using `spline` are far smaller than using `gradient` on the same mesh. Also note that the errors near the end of the interval are larger than in the interior, even though they are of the same order of accuracy.

x	spline derivative	error
0	1.000105e+000	1.054555e-004
1.570796e-001	9.876558e-001	-3.251156e-005
3.141593e-001	9.510611e-001	4.570561e-006
4.712389e-001	8.910015e-001	-5.048342e-006
6.283185e-001	8.090146e-001	-2.437651e-006
7.853982e-001	7.071052e-001	-1.599607e-006
9.424778e-001	5.877798e-001	-5.496783e-006
1.099557e+000	4.540022e-001	1.167249e-005
1.256637e+000	3.089666e-001	-5.039545e-005
1.413717e+000	1.566181e-001	1.836456e-004
1.570796e+000	-6.873579e-004	-6.873579e-004

Table 5.3: Using `spline` to approximate the derivative of $\sin x$

Problem 5.3.1. *Tabulate the function* $\ln x$ *at integer values on the interval* $[1, 10]$ *From your tabulated values approximate the derivative of* $\ln x$ *at the points* $1, 2 \ldots, 10$ *using the* MATLAB *function* `gradient`. *Tabulate the approximate derivatives and their errors. Also, calculate the approximate derivative by fitting to the data with the* MATLAB *function* `spline` *and differentiating the result. Again, tabulate the approximate derivatives and their errors. Which approach delivers the more accurate approximations to the derivative?*

5.3.2 Integration

Symbolic Integration

Consider first computing antiderivatives of integrable functions. In Calculus we learned first how to compute antiderivatives of a number of simple functions, for example powers, trigonometric and exponential functions then we learned a number of techniques to extend the range of functions for which we could compute antiderivatives. These techniques included substitution, integration by parts and partial fractions. In each of these cases our approach was to reduce the computation of an antiderivative to looking up a linear combination of well-known antiderivatives. This is essentially the technique used by MATLAB's symbolic integrator `int`. Given a function, it attempts to transform the function into a linear combination of functions each of which is in MATLAB's lookup table of

antiderivatives. The difference between your attempts to find antiderivatives and MATLAB's is that MATLAB's `int` knows many more transformations and has a much longer lookup table than do you.

Consider the following MATLAB code for computing the antiderivatives of four integrands.

```
syms z
int(sin(z) * cos(z))
int(sin(4*z) * cos(3*z))
int(sin(z)^2 * cos(z)^3)
int((1/(1-z)) * (1/(1 + z + z^2)))
int((sin(z)/(1 - z)) * (1/(1 + z + z^2)))
int((sin(z)/(1 - z)) * (1/(1 + cos(z^2) + z^2)))
```

This code produces the following somewhat edited output.

```
ans =
1/2*sin(z)^2

ans =
-1/14*cos(7*z)-1/2*cos(z)

ans =
-1/5*sin(z)*cos(z)^4+1/15*cos(z)^2*sin(z)+2/15*sin(z)

ans =
-1/3*log(z-1)+1/6*log(z^2+z+1)+1/3*3^(1/2)*atan(1/3*(2*z+1)*3^(1/2))

ans =
-1/3*sinint(z-1)*cos(1)-1/3*cosint(z-1)*sin(1)-1/9*i*(3/2+1/2*i*3^(1/2))*3^(1/2)*
(sinint(z+1/2-1/2*i*3^(1/2))*cos(1/2-1/2*i*3^(1/2))-cosint(z+1/2-1/2*i*3^(1/2))*
sin(1/2-1/2*i*3^(1/2)))+1/9*i*(3/2-1/2*i*3^(1/2))*3^(1/2)*(sinint(z+1/2+1/2*i*3^(1/2))*
cos(1/2+1/2*i*3^(1/2))-cosint(z+1/2+1/2*i*3^(1/2))*sin(1/2+1/2*i*3^(1/2)))

Warning: Explicit integral could not be found.
> In sym.int at 58
  In symbint at 7
ans =
int(sin(z)/(1-z)/(1+cos(z^2)+z^2),z)
```

the first antidifferentiation used integration by parts followed by a trigonometric identity, the second used trigonometric identities followed by substitution and lookup, the third used trigonometric identities followed by integration by parts, and the fourth used partial fractions. Observe that the difference between the fourth and fifth integrands is the introduction of a `sin(z)` in the numerator but the effect on the result is dramatic. You will observe that the result is complex and involves two MATLAB functions `sinint` and `cosint`. These stand for the Sine Integral and the Cosine Integral, respectively, and they are "well-known" functions that are in MATLAB's lookup tables and whose values MATLAB can evaluate. The sixth integrand is a modification of the fifth, replacing a z by a $\cos(z^2)$. This is a function for which MATLAB cannot find the antiderivative, a warning is given and the line of code is repeated.

Consider next the definite integral. We take the set of integrands above and integrate the first three over $[0, 1]$ and the last three over $[2, 3]$ (all these integrals exist). The code is

```
syms z
int(sin(z) * cos(z), 0, 1)
int(sin(4*z) * cos(3*z), 0, 1)
int(sin(z)^2 * cos(z)^3, 0, 1)
int((1/(1 - z)) * (1/(1 + z + z^2)), 2, 3)
```

```
int((sin(z)/(1 - z)) * (1/(1 + z + z^2)), 2, 3)
int((sin(z)/(1 - z)) * (1/(1 + cos(z^2) + z^2)), 2, 3)
```

and the (edited) results are

```
ans =
1/2*sin(1)^2

ans =
-1/14*cos(7)-1/2*cos(1)+4/7

ans =
-1/5*sin(1)*cos(1)^4+1/15*cos(1)^2*sin(1)+2/15*sin(1)

ans =
-1/3*log(2)+1/6*log(13)+1/3*3^(1/2)*atan(7/3*3^(1/2))-1/6*log(7)-1/3*3^(1/2)*atan(5/3*3^(1

ans =
The value of this integral has been omitted by the authors.
The expression takes over twelve lines to reproduce.

Warning: Explicit integral could not be found.
> In sym.int at 58
  In symbint at 13

ans =
int(sin(z)/(1-z)/(1+cos(z^2)+z^2),z = 2, 3)
```

Of course, often you want a numerical value when you evaluate a definite integral. For the above integrals the values can be calculated as follows

```
a = int(sin(z) * cos(z), 0, 1); double(a)
a = int(sin(4*z) * cos(3*z), 0, 1); double(a)
a = int(sin(z)^2 * cos(z)^3, 0, 1); double(a)
a = int((1/(1 - z)) * (1/(1 + z + z^2)), 2, 2); double(a)
a = int((sin(z)/(1 - z)) * (1/(1 + z + z^2)), 2, 3); double(a)
a = int((sin(z)/(1-z)) * (1/(1+cos(z^2) +z^2)), 2, 3); double(a)
```

and the results are

```
ans =
   0.35403670913679

ans =
   0.24742725746998

ans =
   0.11423042636636

ans =
     0

ans =
 -0.04945332203255 - 0.00000000000000i

Warning: Explicit integral could not be found.
```

```
> In sym.int at 58
  In symbint at 19

ans =
  -0.06665303913176
```

The first five of these values were calculated by arithmetically evaluating the expressions above. Note that the result for the fifth integral is real (as it must be for a real function integrated over a real interval) even though the antiderivative is complex (the imaginary parts cancel to give zero). The sixth integral is computed differently. As we have already seen, MATLAB cannot find the antiderivative. It produces a numerical value by resorting to a numerical integration scheme such as those described in the next section. In fact, all these values may be calculated simply by using numerical integration schemes.

Finally, let us consider the improper integrals represented by the following code

```
int(1/z, 0, 1)
int(1/sqrt(z), 0, 1)
int(1/z, -1, 1)
int(1/z, 1, Inf)
int(1/z^2, 1, Inf)
```

where `Inf` is the MATLAB function representing infinity. These give

```
ans =
Inf

ans =
2

ans =
NaN

ans =
Inf

ans =
1
```

The first, third and fourth integrals are not convergent whereas the second and fifth are convergent. It is interesting that MATLAB spots that there is a problem with the third integral – it does more than simply substitute the limits of integration into an antiderivative.

Problem 5.3.2. *Use the* MATLAB *function* `int` *to compute the definite integrals*

(a) $\displaystyle\int_1^2 \frac{\ln x}{1+x}\,dx$ *(b)* $\displaystyle\int_0^3 \frac{1}{1+t^2+t^4}\,dt$

(c) $\displaystyle\int_0^{1/2} \sin\left(e^{t/2}\right)\,dt$ *(d)* $\displaystyle\int_0^4 \sqrt{1+\sqrt{x}}\,dx$

Numerical Integration

When attempting to compute approximations of

$$I(f) = \int_a^b f(x)\,dx\,, \tag{5.1}$$

three basic situations can occur:

1. $f(x)$ is a fairly simple, known function.
2. $f(x)$ is a known function, but is complicated by the fact that it contains various parameters that can change depending on the application.
3. $f(x)$ is not known explicitly. Instead, only a table of $(x_i, f(x_i))$ values is known.

In this section we describe how to use MATLAB for each of these situations. We also discuss how to handle certain special situations, such as infinite limits of integration, and integrand singularities.

Explicitly Known, Simple Integrand

Suppose the integrand, $f(x)$ is explicitly known. In this case, MATLAB provides two functions for definite integration, `quad` and `quadl`. The function `quad` implements adaptive integration using a low order method, Simpson's rule of DOP= 3, for both the integral and the error estimate. The function `quadl` implements adaptive integration using a higher order method, the Lobatto four–point rule of DOP= 5, with an eight-point Lobatto-Kronrod rule of DOP= 9 for error estimation. Since both types of rules evaluate the integral at the endpoints of the interval of integration, it appears that the MATLAB numerical integration codes, `quad` and `quadl` are not safe for use for evaluating integrals with interval endpoint singularities (recall the discussion in Section 5.2.4). However, the codes are implemented in such a way as not to fail catastrophically and often compute quite accurate results in these cases; see Section 5.3.2.

The basic usage of `quad` and `quadl` is

```
I = quad(fun, a, b);
I = quadl(fun, a, b);
```

In order to use these, we must first define the function that is to be integrated. This can be achieved in several ways, including anonymous functions and function M–files. We illustrate how to use each of these with `quad`, for simple integrands, in the following example.

Example 5.3.3. Consider the simple example

$$\int_0^{\pi} (2\sin(x) - 4\cos(x))\, dx$$

Here are two ways to define $f(x) = 2\sin(x) - 4\cos(x)$, and how they can be used with `quad` to compute an approximate integral.

- Since the integrand is a fairly simple function, it can be easily defined as an anonymous function, with the result and the endpoints of the interval of integration then passed to `quad`:

```
f = @(x) 2*sin(x) - 4*cos(x);
I = quad(f, 0, pi)
```

- A second option is to write a function M–file that evaluates the integrand, such as:

```
function f = MyFun( x )
%
%  This function evaluates f(x) = 2*sin(x) - 4*cos(x)
%
%  Input:  x = vector of x-values at which f(x) is to be evaluated.
%
%  Output: f = vector containing the f(x) values.
%
f = 2*sin(x) - 4*cos(x);
```

Then we can use a "function handle" to tell `quad` about `MyFun`

```
I = quad(@MyFun, 0, pi)
```

When the integrand is a simple function, like the previous example, the anonymous approach is usually easiest to use. For more complicated integrands, function M–files may be necessary. It is important to note that, regardless of the approach used, evaluation of the integrand must be vectorized; that is, it must be able to compute $f(x)$ at a vector of x-values as required by the functions `quad` and `quadl`.

Example 5.3.4. Suppose we want to compute an approximation of

$$\int_0^{\pi} x \sin(x)\, dx$$

Since the integrand in this case is very simple, we define $f(x)$ as an anonymous function. Our first attempt might be to use the statement:

```
f = @(x) x*sin(x);
```

This is a legal MATLAB statement, however we can only use it to compute $f(x)$ for scalar values x. For example, the statements

```
f(0)
f(pi/2)
f(pi)
```

will execute without error, and compute accurate approximations of the true values 0, $\frac{\pi}{2}$, and 0, respectively. However, if we try to execute the statements

```
x = [0 pi/2 pi];
f(x)
```

then an error will occur, stating that "inner matrix dimensions must agree". The same error will occur if we try to use `quad` or `quadl` with this anonymous function because, for reasons of efficiency, the software wants the integrand values at all the points in the rule simultaneously and so the computation `x*sin(x)` must be vectorized. A vectorized implementation is given by:

```
f = @(x) x.*sin(x);
```

In this case, we can compute $f(x)$ for vectors of x-values, and use `quad` and `quadl`, as in the previous example, without error.

Example 5.3.5. Suppose we want to compute an approximation of

$$\int_0^{2\pi} (\cos x)^2 (\sin x)^2\, dx$$

The following MATLAB statements can be used to approximate this integral:

```
f = @(x) ( (cos(x)).^2 ) .* ( sin(x).^2 );
I = quad(f, 0, 2*pi)
```

Note that we must use vector operations (dot multiply and dot exponentiation) in the definition of the integrand.

Example 5.3.6. Suppose we want to compute an approximation of

$$\int_1^e \frac{1}{x\,(1+(\ln x)^2)}\,dx$$

The following MATLAB statements can be used to approximate this integral:

```
f = @(x) 1 ./ (x .* (1 + log(x).^2));
I = quad(f, 1, exp(1))
```

Notice again that we must use vector operations (dot divide, dot multiply and dot exponentiation) in the definition of the integrand. Note, the MATLAB `log` function is the natural logarithm, and the `exp` function is the natural exponential. Thus, `exp(1)` computes $e^1 = e$.

As with most MATLAB functions, it is possible to provide more information to, and get more information from `quad` and `quadl`. For example, the basic calling sequence

```
I = quad(f, a, b)
```

uses a default absolute error tolerance of 10^{-6}. We can override this default value by using the calling sequence

```
I = quad(f, a, b, tol)
```

where a value for `tol` must be predefined.

The amount of work required by `quad` or `quadl` is determined by the total number of integrand evaluations that must be calculated. This information can be obtained by using the calling sequence:

```
[I, nevals] = quad(f, a, b)
```

In this case, `I` contains the approximation of the integral, and `nevals` is the total number of integrand evaluations needed to compute `I`.

Problem 5.3.3. *Use* `quad` *and* `quadl` *to compute approximations to the given integrals.*

(a) $\displaystyle\int_1^2 \frac{\ln x}{1+x}\,dx$ *(b)* $\displaystyle\int_0^3 \frac{1}{1+t^2+t^4}\,dt$

(c) $\displaystyle\int_0^{1/2} \sin\left(e^{t/2}\right)\,dt$ *(d)* $\displaystyle\int_0^4 \sqrt{1+\sqrt{x}}\,dx$

In each case, report on the number of function evaluations required by each method. Check your results against the exact values of these integrals that you computed in Problem 5.3.2.

Explicitly Known, Complicated Integrand

Suppose the integrand, $f(x)$, is known, but that it contains one or more parameters that may change. Such a situation is best illustrated with an example.

A random variable X has a *continuous distribution* if there exists a nonnegative function, f, such that for any interval $[a, b]$,

$$\text{Prob}(a \le X \le b) = \int_a^b f(x)\,dx\,.$$

The function $f(x)$ is called the *probability density function* (PDF). An important PDF is the *normal* (sometimes called a *Gaussian* or *bell curve*) distribution, which is defined as

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)\,,$$

where μ is the mean and σ^2 is the variance.

To compute probabilities for this distribution, we need to calculate integrals where the integrand depends not only on the variable x, but also on the parameters μ and σ. We might try to write a function M–file that has input parameters x, μ and σ, such as:

```
function f = NormalPDF(x, mu, sigma)
%
% Compute f(x) for normal PDF, with mean mu and standard deviation sigma.
%
c = 1 / (sigma*sqrt(2*pi));
d = (x - mu).^2 / (2*sigma^2);
f = c * exp( -d );
```

However, if we try to use `quad` to compute probabilities, say Prob$(-1 \leq X \leq 1)$:

```
p = quad(@NormalPDF, -1, 1)
```

then MATLAB will print an error message complaining that `mu` and `sigma` have not been defined. The difficulty here is that we would like to be able to specify values for `mu` and `sigma` without editing the function `NormalPDF`. The easiest way to do this is through *nested functions*; that is, a function nested inside another function M–file. For example, consider the following function M–file:

```
function p = TestProbs(a, b, mu, sigma)
%
%    p = TestProbs(a, b, mu, sigma)
%
% Compute a <= prob(X) <= b for normal distribution
% with mean mu and standard deviation sigma.
%
p = quad(@NormalPDF, a, b);

  function f = NormalPDF(x)
  %
  % Compute f(x) for normal PDF, with mean mu, standard deviation sigma.
  %
  c = 1 / (sigma*sqrt(2*pi));
  d = (x - mu).^2 / (2*sigma^2);
  f = c * exp( -d );
  end

end
```

Nested functions have access to variables in their parent function. Thus, since `mu` and `sigma` are defined as input to `TestProbs`, they can also be used in the nested function `NormalPDF` without having to pass them as input. Note that both the nested function `NormalPDF`, and its parent function, `TestProbs`, must be closed with `end` statements[3].

Nested functions are new in MATLAB 7; for additional examples, read the `doc` pages for `quad` and `quadl`.

[3]Actually, any function M–file in MATLAB can be closed with an `end` statement, but it is not always necessary; it is for nested functions.

Problem 5.3.4. *Suppose a set of test scores are distributed normally with mean* $\mu = 78$ *and standard deviation* $\sigma = 10$.

(a) Plot the probability density function, including marks on the x-axis denoting the mean, and $\mu \pm 2 * \sigma$.

(b) Implement `TestProbs` *and use it to answer the following questions:*

What percentage of the scores are between 0 and 100? Does this make sense?

What percentage of the scores are between 0 and 60?

What percentage of scores are between 90 and 100?

What percentage of scores are within 2 standard deviations of the mean?

Problem 5.3.5. *Modify* `TestProbs` *so that it can accept as input vectors* `mus` *and* `sigmas` *(which have the same number of entries), and returns as output a vector* `p` *containing the probabilities* $Prob(a \leq X \leq b)$ *corresponding to* `mus(i)` *and* `sigmas(i)`, $i = 1, 2, ...$

Problem 5.3.6. *The intensity of light with wavelength* λ *travelling through a diffraction grating with* n *slits at an angle* θ *is given by*

$$I(\theta) = n^2 \frac{\sin^2 k}{k^2}, \quad where \quad k = \frac{\pi n d \sin\theta}{\lambda},$$

and d *is the distance between adjacent slits. A helium-neon laser with wavelength* $\lambda = 632.8 * 10^{-9} m$ *is emitting a narrow band of light, given by* $-10^{-6} < \theta < 10^{-6}$, *through a grating with* $10,000$ *slits spaced* $10^{-4} m$ *apart. Estimate the total light intensity,*

$$\int_{-10^{-6}}^{10^{-6}} I(\theta)\ d\theta$$

emerging from the grating. You should write a function that can compute $I(\theta)$ *for general values of* n, d *and* λ, *and which uses a nested function to evaluate the integrand (similar to the approach used in* `TestProbs`*).*

Integration of Tabular Data

Suppose that we do not know the integrand explicitly, but we can obtain function values at certain discrete points. In this case, instead of using `quad` or `quadl`, we could first fit a curve through the data, and then integrate the resulting curve. We can use any of the methods discussed in Chapter 4, but the most common approach is to use a piecewise polynomial, such as a spline. That is,

- Suppose we are given a set of data points, (x_i, f_i).
- Fit a spline, $S(x)$, through this data. Keep in mind that $S(x)$ is a piecewise polynomial on several intervals. In particular,

$$S(x) = \begin{cases} p_1(x) & x_0 \leq x \leq x_1 \\ p_2(x) & x_1 \leq x \leq x_2 \\ \vdots & \vdots \\ p_N(x) & x_{N-1} \leq x \leq x_N \end{cases}$$

- Then

$$\int_a^b f(x)\,dx \approx \int_a^b S(x)\,dx = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} S(x)\,dx = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} p_i(x)\,dx$$

Since each $p_i(x)$ is a polynomial, we can compute the integrals by hand. For example, if we were to use a linear spline, then the data are connected by straight lines, and

$$\int_{x_{i-1}}^{x_i} p_i(x)\,dx = \text{ area under a line } = \text{ area of a trapezoid.}$$

This approach is equivalent to using the composite trapezoidal rule. MATLAB provides a function that can be used to integrate tabular data using the composite trapezoidal rule, called `trapz`. Its basic usage is illustrated by:

```
I = trapz(x, y)
```

where `x` and `y` are vectors containing the points and the data values, respectively. Note that `trapz` does not estimate the error, and thus should not be used without using another method to provide an error estimate.

Instead of a linear spline, we might consider using a cubic spline interpolating function. In this case, each $p_i(x)$ is a cubic polynomial, which can be integrated exactly by any integration rule with $DOP \geq 3$. We will use Simpson's rule to obtain the following function to integrate tabular data:

```
function I = quad3(x, f)
%
%         I = quad3(x, f);
%
%  This function integrates tabular data using cubic splines.
%
%  Input:  x, f  - vectors containing the tabular data
%  Output:     I  - approximation of the integral
%
S = spline(x, f);
N = length(x);
I = 0;
for i = 1:N-1
  h = x(i+1) - x(i);
  I = I + (h/6)*(f(i) + 4 * ppval(S, (x(i) + x(i+1))/2) + f(i+1));
end
```

The code `quad3` has been written this way for clarity. It could be made more efficient by taking advantage of the repeated function values (at the subinterval ends) in the sum and by vectorizing the call to `ppval` over all the subinterval midpoints. This neat approach exploiting the properties of integration rules allows us to avoid the more direct, but longwinded, approach of extracting the cubic polynomial form of the spline on each subinterval using the MATLAB function `unmkpp` then integrating the resulting polynomials directly.

We can use our function `quad3` to provide an error estimate for `trapz`. Generally, we would expect `quad3` to be the more accurate as it uses cubic splines in contrast to the linear spline used by `trapz`. If we want an error estimate for `quad3`, we need another rule of at least the same order of accuracy, for example that discussed in Problem 5.3.7.

Example 5.3.7. The *dye dilution method* is used to measure cardiac output. Dye is injected into the right atrium and flows through the heart into the aorta. A probe inserted into the aorta measures the concentration of the dye leaving the heart. Let $c(t)$ be the concentration of the dye at time t. Then the cardiac output is given by

$$F = \frac{A}{\int_0^T c(t)\,dt},$$

where A is the amount of dye initially injected, and T is the time elapsed.

In this problem, $c(t)$ is not known explicitly; we can only obtain measurements at fixed times. Thus, we obtain a set of tabular data, (t_i, c_i). In order to compute F, we can use either MATLAB's `trapz` function, or our own `quad3` to approximate the integral.

For example, suppose a 5-mg bolus dye is injected into a right atrium. The concentration of the dye (in milligrams per liter) is measured in the aorta at one-second intervals. The collected data is shown in the table:

t	0	1	2	3	4	5	6	7	8	9	10
$c(t)$	0	0.4	2.8	6.5	9.8	8.9	6.1	4.0	2.3	1.1	0

Using `trapz` and `quad3` we can compute approximations of F as follows:

```
t = 0:10;
c = [0 0.4 2.8 6.5 9.8 8.9 6.1 4.0 2.3 1.1 0];
A = 5;
F1 = A / trapz(t, c)
F3 = A / quad3(t, c)
```

We see that $F1 \approx 0.1193$L/s and $F3 \approx 0.1192$L/s. These results would seem to imply that we can be fairly confident of the first three digits in both results. Note however that the data is only given to two digits. Assuming the data is accurate to the digits shown, you should be cautious about assuming significantly more accuracy in the integral than you observe in the data.

Problem 5.3.7. *Modify the* MATLAB *function* `quad3` *so that it uses* `pchip` *instead of* `spline`. *Why is the modification simple? Use the modified function to integrate the tabular data in Example 5.3.7. Does the result that you obtain increase your confidence that the value of the integral is 0.119L/s to three digits? Why or why not?*

Improper Integrals

In calculus, the term *improper integral* is used for situations where either $\pm\infty$ is one of the limits of integration, or if the integrand, $f(x)$, is not defined at a point in the interval (i.e., $f(x)$ has a *singularity*).

Example 5.3.8. Since the function $f(x) = \dfrac{1}{\sqrt{x}}$ is not defined at $x = 0$, we say the integrand for

$$\int_0^1 \frac{1}{\sqrt{x}}\,dx$$

is singular at the endpoint $x = 0$ (that is, it has an endpoint singularity).

The function $f(x) = \dfrac{\sin x}{x}$ is also not defined at $x = 0$, so the integrand of

$$\int_{-1}^1 \frac{\sin x}{x}\,dx$$

has a singularity within the interval of integration (i.e., not at an endpoint).

An example of an infinite interval of integration is

$$\int_1^\infty \frac{1}{x^2}\,dx$$

All of the integrals in this example can be computed using calculus techniques, and have well-defined solution.

The MATLAB functions `quad` and `quadl` can generally be used to compute approximations of these types of improper integrals.

Endpoint Singularities

Although the basic algorithms implemented by `quad` and `quadl` are based on *closed* quadrature rules, the actual implementation includes a check for endpoint singularities, and slightly shifts the endpoints if a singularity is detected. It is therefore possible to use `quad` and `quadl` directly on problems with endpoint singularities. A divide by zero warning will be displayed in the MATLAB command window, but it is not a fatal error, and a quite accurate result is generally computed.

Example 5.3.9. The following MATLAB statements

```
f = @(x) 1 ./ sqrt(x);
I = quad(f, 0, 1)
```

compute the approximation $\int_0^1 \frac{1}{\sqrt{x}}\,dx \approx 2.00001$ and give a warning that a "Divide by zero" has been encountered. The value of the computed integral is in error by one in the fifth digit even though a (default) absolute accuracy, `tol`$= 10^{-6}$, was requested; that is, `quad` did not quite deliver the error requested.

Interior Singularities

If the singularity is not at an endpoint, but is within the interval of integration, then `quad` and `quadl` may not find the singularity during the computation, and thus they may compute an approximation of the integral without incurring a fatal error. However, because the quadrature points are chosen adaptively, it is possible that a fatal divide by zero error will be encountered, causing `quad` and `quadl` to fail.

Example 5.3.10. If we attempt to use `quad` directly to compute an approximation of the convergent integral $\int_{-0.5}^1 \frac{\sin x}{x}\,dx$,

```
f = @(x) sin(x) ./ x;
I = quad(f, -0.5, 1)
```

then the (removable) singularity at $x = 0$ is not encountered during the computation, and it is found that $I = 1.43919048307848$ which is correct to nine digits.

On the other hand if we try the same direct approach to compute $\int_{-1}^1 \frac{\sin x}{x}\,dx$,

```
f = @(x) sin(x) ./ x;
I = quad(f, -1, 1)
```

then the singularity is encountered precisely when the algorithm used by `quad` halves the interval of integration, and I is computed to be NaN (not a number) even though the integral is convergent. The strategy used by `quad` and `quadl` at interval endpoints of checking for singularities and moving the endpoint when one is encountered is not used at interior points.

If an integral contains an interior singularity, it is safest, and almost essential, to split the interval of integration at the singular point, and calculate the sum of the resulting two integrals. Because `quad` and `quadl` can handle endpoint singularities quite well, the individual integrals can then be

computed fairly safely. Note, the singularity here is "removable" as $\lim_{x\to 0} \dfrac{\sin x}{x} = 1$ so we could easily avoid the problem by defining the integrand to have value one at $x = 0$.

Example 5.3.11. An approximation of the integral $\displaystyle\int_{-0.5}^{1} \frac{\sin x}{x}\, dx$ should be computed using:

```
f = @(x) sin(x) ./ x;
I = quad(f, -0.5, 0) + quad(f, 0, 1)
```

giving the value 1.43919048811714 which has ten correct digits and agrees with the value above in Example 5.3.10 to more than the default absolute accuracy request of `tol = 1.0e-6`. Similarly, an approximation of the integral $\displaystyle\int_{-1}^{1} \frac{\sin x}{x}\, dx$ should be computed using:

```
f = @(x) sin(x) ./ x;
I = quad(f, -1, 0) + quad(f, 0, 1)
```

which gives the value 1.89216614015307. This integral is convergent and this is the correct value to nine digits.

Infinite Limits of Integration

There are a variety of techniques that can be used to compute approximations of integrals with infinite limits. One approach that often works well is to use a substitution such as $x = \dfrac{1}{t}$.

Example 5.3.12. Consider the integral $\displaystyle\int_{1}^{\infty} \frac{x^3}{x^5+2}\, dx$. If we use the substitution $x = \dfrac{1}{t}$, we obtain

$$\int_{1}^{\infty} \frac{x^3}{x^5+2}\, dx = \int_{1}^{0} \frac{(1/t)^3}{(1/t)^5+2}\left(\frac{-1}{t^2}\right) dt = \int_{0}^{1} \frac{1}{1+2t^5}\, dt,$$

which can be easily integrated using the MATLAB statements:

```
f = @(t) 1 ./ (1 + 2*t.^5);
I = quad(f, 0, 1)
```

giving the value 0.826798, to six digits.

Note that for some problems a substitution such as $x = \dfrac{1}{t}$ or $x = \dfrac{1}{t^2}$ might cause the transformed integral to have an endpoint singularity, but this usually is not a problem since `quad` and `quadl` can work fairly safely for such situations.

A well-used approach to dealing with integrals with infinite limits is to truncate the interval then use a finite interval of integration method.

Example 5.3.13. Continuing the previous example, we could approximate "infinity" by 10 and use the MATLAB script

```
f = @(x) (x.^3) ./ (x.^5 + 2);
I = quad(f, 0, 10)
```

which gives 0.726799, not a very good approximation. If, instead we approximate "infinity" by 100 we get 0.816798, and by 1000 we get 0.825798. This is very slowly convergent integral, so this slow convergence of the integrals over truncated intervals is to be expected.

To make the integral converge faster we can "subtract out the singularity".

Example 5.3.14. Continuing the previous example, we could subtract out the singularity $\frac{1}{x^2}$, which is the limit of the integrand as $x \to \infty$. This gives

$$\int_1^\infty \frac{x^3}{x^5+2}\,dx = \int_1^\infty \frac{1}{x^2}\,dx - \int_1^\infty \frac{2}{x^2(x^5+2)}\,dx = 1 - \int_1^\infty \frac{2}{x^2(x^5+2)}\,dx$$

We then compute the integral using MATLAB script

```
f = @(x) (2) ./ x.^2.*(x.^5 + 2);
I = 1 - quad(f, 0, 10)
```

which gives 0.826798 to six digits. This time approximating infinity by ten gives a much more accurate result as by subtracting out the singularity we have produced a much faster converging integral.

Problem 5.3.8. *Determine the singular points for each of the following integrands, and compute approximations of each of the integrals.*

(a) $\displaystyle\int_0^1 \frac{e^{-x}}{\sqrt{1-x}}\,dx$ *(b)* $\displaystyle\int_4^5 \frac{1}{(5-t)^{2/5}}\,dt$ *(c)* $\displaystyle\int_0^1 \frac{1}{\sqrt{x}(1+x)}\,dx$

Problem 5.3.9. *Use each of the* MATLAB *functions* `quad` *and* `quadl` *to compute the integral* $\displaystyle\int_0^1 \frac{1}{\sqrt{x}}\,dx$ *for the absolute error tolerances* $tol = 10^{-i}$, $i = 1, 2, \ldots, 12$ *in turn. Tabulate your results and their absolute errors. What do you observe from your tabulated results? What else did you observe when running your* MATLAB *script.*

Problem 5.3.10. *An approximation of the integral* $\displaystyle\int_{-0.5}^1 \frac{e^x - 1}{x}\,dx$ *is to be computed using:*

```
f = @(x) (exp(x)-1) ./ x;
I = quad(f, -0.5, 1)
```

and of the integral $\displaystyle\int_{-1}^1 \frac{e^x - x}{x}\,dx$ *similarly. What do you observe? There is a singularity. Is it removable? Can you split the range of integration appropriately?*

Repeat your experiment using `quadl` *in place of* `quad`.

To five digits what is the correct value of both integrals? Why?

Problem 5.3.11. *Consider the integral* $\displaystyle\int_1^7 \frac{\sin(x-4)}{(x-4)}\,dx$. *At what value* x *is the integrand singular? Attempt to use* `quad` *and* `quadl` *directly to compute an approximation of this integral. Now, split the interval of integration into two parts at the singular point and repeat your experiment. Comment on your results.*

Problem 5.3.12. *Use the substitution* $x = \frac{1}{t}$ *followed by a call to* `quad` *to compute approximations of the following integrals:*

(a) $\displaystyle\int_1^\infty \frac{\sin x}{x^2}\,dx$ *(b)* $\displaystyle\int_1^\infty \frac{2}{\sqrt{x}(1+x)}\,dx$ *(c)* $\displaystyle\int_{-\infty}^3 \frac{1}{x^2+9}\,dx$

Problem 5.3.13. *Compute an approximation of the integral*

$$\int_0^\infty \frac{1}{\sqrt{x}(1+x)}\,dx$$

Problem 5.3.14. *In a single* MATLAB *program use each of the functions* `quad` *and* `quadl` *to compute all three of the definite integrals:*

1. $\displaystyle\int_0^8 [\, e^{-3x} - \cos(5\pi x)\,]\,dx$
2. $\displaystyle\int_{-1}^2 \left(\left| x - \frac{1}{\sqrt{3}} \right| + \left| x + \frac{1}{\sqrt{2}} \right| \right) dx$
3. $\displaystyle\int_0^1 x^{-\frac{2}{3}}\,dx$

setting the error tolerance to give first and absolute error of `tol=1.0e-6` *and then* `tol=1.0e-12` *In each case keep track of the number of integrand evaluations used to compute the integral. Do you get consistent results with the different tolerances and integrators? Can you explain the warnings that* MATLAB *issues?*

Chapter 6

Root Finding

We were introduced to the concept of finding roots of a function in basic algebra courses. For example, we learned the quadratic formula so that we could factor quadratic polynomials, and we found x-intercepts provided key points when we sketched graphs. In many practical applications, though, we are presented with problems involving very complicated functions for which basic algebraic techniques cannot be used, and we must resort to computational methods. In this chapter we consider the problem of computing roots of a general nonlinear function of one variable. We describe and analyze several techniques, including the well known Newton iteration. We show how the Newton iteration can be applied to the rather special, but important, problem of calculating square roots. We also discuss how to calculate real roots in the special case when the nonlinear function is a polynomial of arbitrary degree. The chapter ends with a discussion of implementation details, and we describe how to use some of MATLAB's built-in root finding functions.

6.1 Roots and Fixed Points

A **root** of a function f is a point, $x = x_*$, where the equation

$$f(x_*) = 0$$

is satisfied. A point $x = x_*$ is a **simple root** if the following properties are satisfied:

- $f(x_*) = 0$ (that is, x_* is a root of $f(x)$),
- the function $f'(x)$ exists everywhere on some open interval containing x_*, and
- $f'(x_*) \neq 0$.

Geometrically these conditions imply that, at a simple root x_*, the graph of $f(x)$ crosses the x-axis. However this geometric interpretation can be deceiving because it is possible to have a function $f(x)$ whose graph crosses the x-axis at a point x_*, but also have $f'(x_*) = 0$. In such a situation x_* is *not* a simple root of $f(x)$; see Example 6.1.2.

Example 6.1.1. The point $x_* = 1$ is a root of each of the functions

$$f(x) = x - 1, \quad f(x) = \ln(x), \quad f(x) = (x-1)^2, \quad \text{and} \quad f(x) = \sqrt{x-1}.$$

It appears from the graphs of these functions, shown in Fig. 6.1, that $x_* = 1$ is a simple root for $f(x) = x - 1$ and $f(x) = \ln(x)$, but to be sure we must check that the above three conditions are satisfied. In the case of $f(x) = \ln(x)$, we see that

$$\begin{aligned} &f(1) = \ln(1) = 0 \\ &f'(x) = \tfrac{1}{x} \quad \text{exists on, for example, the open interval } (\tfrac{1}{2}, \tfrac{3}{2}) \\ &f'(1) = 1 \neq 0. \end{aligned}$$

Thus $x_* = 1$ is a simple root for $f(x) = \ln(x)$.

In the case of $f(x) = (x-1)^2$, we see that $f'(1) = 0$, so $x_* = 1$ is not a simple root for this function. (A similar situation occurs for $(x-1)^n$ for any even integer $n > 0$.) In the case of $f(x) = \sqrt{x-1}$, the function $f'(x)$ does not exist for $x \leq 1$, so $x_* = 1$ is also not a simple root for this function.

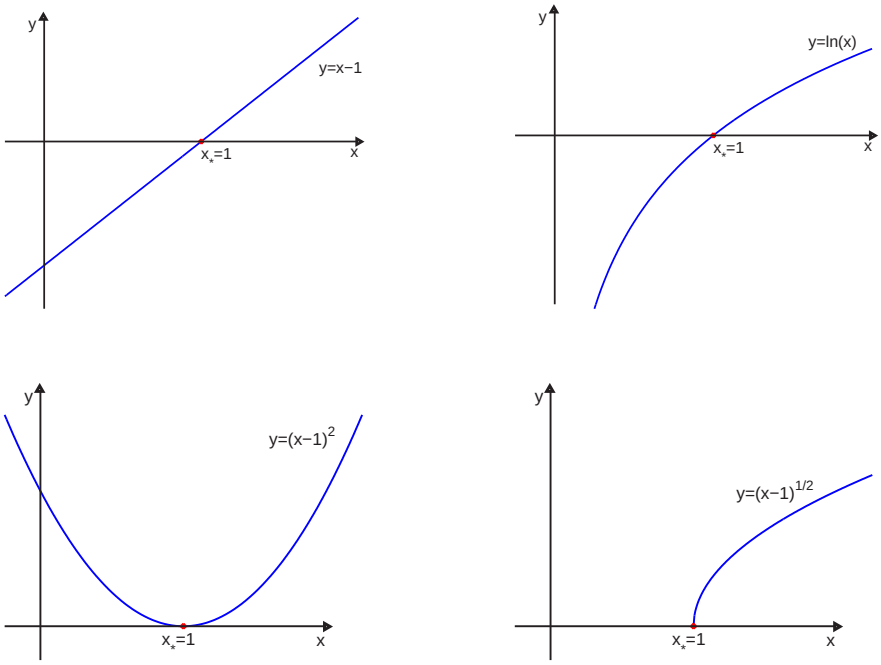

Figure 6.1: Graphs of the functions given in Example 6.1.1. $x_* = 1$ is a simple root of $f(x) = x - 1$ and $f(x) = \ln(x)$. It is a root, but it is *not* simple, for $f(x) = (x-1)^2$ and $f(x) = \sqrt{x-1}$.

Example 6.1.2. The graph of the function $f(x) = (x-1)^3$ is shown in Fig. 6.2. Observe that the graph crosses the x-axis at $x_* = 1$. However, $f'(x) = 3(x-1)^2$, and so $f'(x_*) = f'(1) = 0$. Thus, although $x_* = 1$ is root of this function, it is *not* a simple root. In fact, a similar situation occurs for $f(x) = (x-1)^n$ for any odd integer $n > 1$.

Since $f(x) = (x-1)^3$ crosses the $x-$axis, many methods for simple roots work also for this function though, in some cases, more slowly.)

Typically, the first issue that needs to be addressed when attempting to compute a simple root is to make an initial estimate of its value. One way to do this is to apply one of the most important theorems from Calculus, the Intermediate Value Theorem (see Problem 6.1.1). This theorem states that if a function f is continuous on the closed interval $[c, d]$, and if $f(a) \cdot f(b) < 0$, then there is a point $x_* \in (c, d)$ where $f(x_*) = 0$. That is, if the function f is continuous and changes sign on an interval, then it must have at least one root in that interval. We exploit this property in Section 6.3.3 when we discuss the bisection method for finding roots, but in this section we begin with a related problem. (Of course, we can use our graphing calculator and its zoom feature for obtaining a good approximation to a root for functions that are simple enough to graph accurately.)

Root finding problems often occur in **fixed–point** form; that is, the problem is to find a fixed–point $x = x_*$ that satisfies the equation

$$x_* = g(x_*).$$

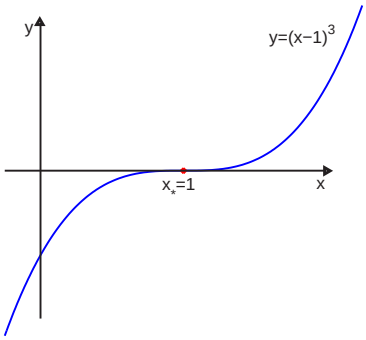

Figure 6.2: Graph of the function, $f(x) = (x-1)^3$, discussed in Example 6.1.2. In this case the graph crosses the x-axis at the root $x_* = 1$, but it is *not* a simple root because $f'(x_*) = f'(1) = 0$.

This type of problem sometimes arises directly when finding an equilibrium of some process. We can convert a problem written in fixed–point form, $x = g(x)$, to the standard form of solving the equation $f(x) = 0$ simply by defining $f(x) \equiv x - g(x) = 0$. Then, we know that the root is simple as long as $f'(x_*) = 1 - g'(x_*) \neq 0$. However, first we consider this problem in its natural fixed–point form.

Example 6.1.3. Consider the equation

$$x = 5 - \frac{3}{x+2}.$$

This is in fixed–point form $x = g(x)$, where $g(x) = 5 - \dfrac{3}{x+2}$. Geometrically, the fixed–points are the values x_* where the graphs of $y = x$ and $y = g(x)$ intersect. Fig. 6.3 shows a plot of the graphs of $y = x$ and $y = g(x)$ for this example. Notice that there are two locations where the graphs intersect, and thus there are two fixed–points.

We can convert the fixed–point form to the standard form $f(x) = 0$:

$$f(x) = x - g(x) = x - 5 + \frac{3}{x+2} = 0.$$

For this particular example it is easy to algebraically manipulate the equations to find the fixed–points of $x = g(x)$, or equivalently, the roots of $f(x) = 0$. In particular,

$$x - 5 + \frac{3}{x+2} = 0 \quad \Rightarrow \quad x^2 - 3x - 7 = 0,$$

and using the quadratic equation we find that the fixed–points are $x_* = \dfrac{3 \pm \sqrt{37}}{2}$.

Problem 6.1.1. The Intermediate Value Theorem. *Let the function f be continuous on the closed interval $[a, b]$. Show that for any number v between the values $f(a)$ and $f(b)$ there exists at least one point $c \in (a, b)$ such that $f(c) = v$.*

Problem 6.1.2. *Consider the function $f(x) = x - e^{-x}$. Use the Intermediate Value Theorem (see Problem 6.1.1) to prove that the equation $f(x) = 0$ has a solution $x_* \in (0, 1)$.*

Problem 6.1.3. The Mean Value Theorem. *Let the function f be continuous and differentiable on the closed interval $[a, b]$. Show that there exists a point $c \in (a, b)$ such that*

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

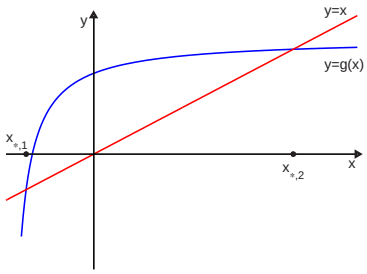

Figure 6.3: Graphs of $y = x$ and $y = g(x)$ for the function $g(x)$ discussed in Example 6.1.3. In this case there are two fixed–points, which are denoted as $x_{*,1}$ and $x_{*,2}$.

Problem 6.1.4. *Consider the function $f(x) = x - e^{-x}$ on the interval $[a, b] = [0, 1]$. Use the Mean Value Theorem (see Problem 6.1.3) to find a point $c \in (0, 1)$ where*

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

.

Problem 6.1.5. *Consider the functions $f(x) = (x - 1)(x + 2)^2$ and $F(x) = x\sqrt{x + 2}$.*

1. *What are the roots of f?*
2. *What are the simple roots of f?*
3. *What are the roots of F?*
4. *What are the simple roots of F?*

Problem 6.1.6. *Consider the equation $x = \cos\left(\frac{\pi}{2}x\right)$. Convert this equation into the standard form $f(x) = 0$ and hence show that it has a solution $x_* \in (0, 1)$ radians. Is this solution simple? [Hint: Use the Intermediate Value Theorem (see Problem 6.1.1) on $f(x)$.]*

6.2 Fixed–Point Iteration

A commonly used technique for computing a solution x_* of an equation $x = g(x)$, and hence a root of $f(x) = x - g(x) = 0$, is to define a sequence $\{x_n\}_{n=0}^{\infty}$ of approximations of x_* according to the iteration

$$x_{n+1} = g(x_n), \quad n = 0, 1, \ldots$$

Here x_0 is an initial estimate of x_* and g is *the fixed–point iteration function*. If the function g is continuous and the sequence $\{x_n\}_{n=0}^{\infty}$ converges to x_*, then x_* satisfies $x_* = g(x_*)$. In practice, to find a good starting point x_0 for the fixed–point iteration we may either:

- Estimate x_* from a graph of $f(x) = x - g(x)$. That is, we attempt to estimate where the graph of $f(x)$ crosses the x-axis.
- Alternatively, we estimate the intersection point of the graphs of $y = x$ and $y = g(x)$.

Unfortunately, even if we begin with a very accurate initial estimate, the fixed–point iteration does not always converge. Before we present mathematical arguments that explain the convergence properties of the fixed–point iteration, we try to develop an understanding of the convergence behavior through a series of examples.

6.2.1 Examples of Fixed–Point Iteration

In this subsection we explore the convergence behavior of the fixed–point iteration through a series of examples.

Example 6.2.1. Suppose we want to find $x = x_*$ to satisfy the following equivalent equations:

Fixed-point form: $x = g(x)$	Root form: $f(x) = x - g(x) = 0$
$x = e^{-x}$	$x - e^{-x} = 0$

The graph of $f(x) = x - e^{-x}$ is shown in Fig. 6.4.

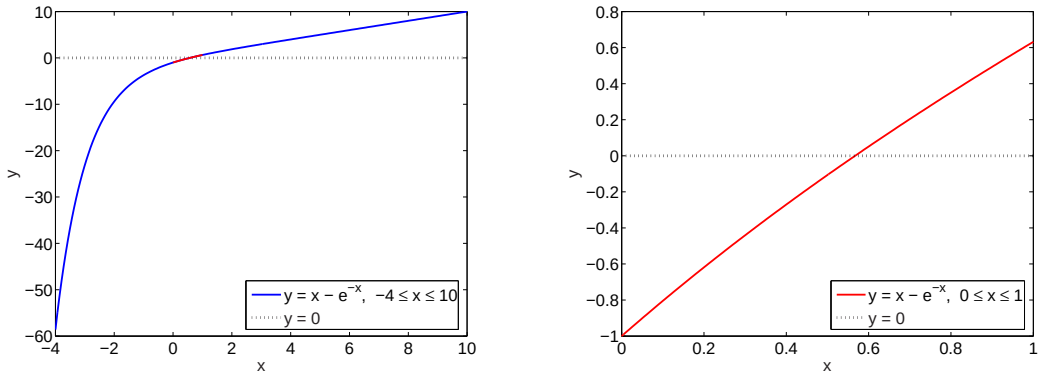

Figure 6.4: Graph of $f(x) = x - e^{-x}$. The left plot shows the graph on the interval $-4 \le x \le 10$, and the right plot zooms in on the region where the graph of f crosses the x-axis (the line $y = 0$).

(a) From the Intermediate Value Theorem (Problem 6.1.1) there is a root of the function f in the interval $(0, 1)$ because:

- $f(x)$ is continuous on $[0, 1]$, and
- $f(0) = -1 < 0$ and $f(1) = 1 - e^{-1} > 0$, and so $f(0) \cdot f(1) < 0$.

(b) From the graph of the derivative $f'(x) = 1 + e^{-x}$, shown in Fig. 6.5, we observe that this root is simple because $f'(x)$ is continuous and positive for all $x \in [0, 1]$.

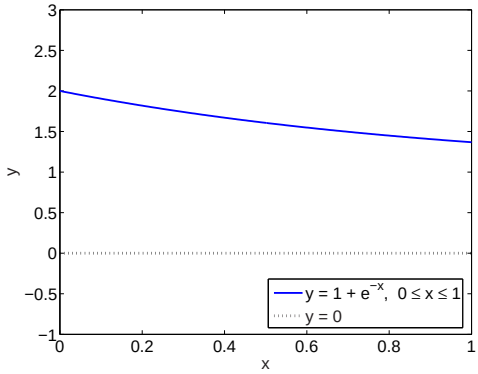

Figure 6.5: Graph of $f'(x) = 1 + e^{-x}$. Notice that f' is continuous and positive for $0 \le x \le 1$.

(c) From the graph of $f(x)$ shown in Fig. 6.4, we see that $x_0 = 0.5$ is a reasonable estimate of a root of $f(x) = 0$, and hence a reasonable estimate of a fixed–point of $x = g(x)$.

(d) Alternatively, the value x_* we seek can be characterized as the intersection point of the line $y = x$ and the curve $y = g(x) \equiv e^{-x}$ as in Fig. 6.6. From this graph we again see that $x_0 = 0.5$ is a reasonable initial estimate of x_*.

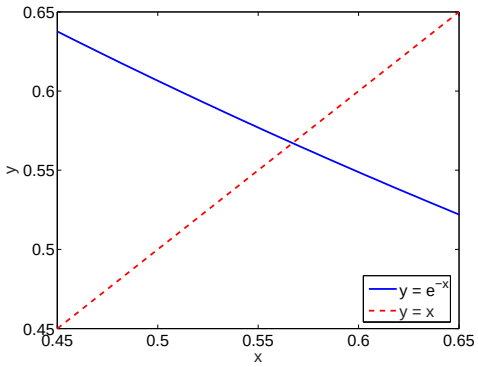

Figure 6.6: Graphs of $y = e^{-x}$ and $y = x$. The point where these graphs intersect is a fixed–point of $x = e^{-x}$, and hence a root of $x - e^{-x} = 0$.

(e) Of course we are relying rather too much on graphical evidence here. As we saw in Section 2.3.1 graphical evidence can be misleading. This is especially true of the evidence provided by low resolution devices such as graphing calculators. But we can easily prove there is a unique fixed–point of the equation $x = e^{-x}$. The curve $y = x$ is strictly monotonic increasing and continuous everywhere and the curve $y = e^{-x}$ is strictly monotonic decreasing and continuous everywhere. We have already seen these curves intersect so there is at least one fixed–point, and the monotonicity properties show that the fixed–point is simple and unique.

(f) If we apply the fixed–point iteration, $x_{n+1} = g(x_n)$, to this problem, we obtain:

$$\begin{array}{lllllll}
x_0 & = & & & & & 0.5000 \\
x_1 & = & g(x_0) & = & e^{-x_0} & \approx & 0.6065 \\
x_2 & = & g(x_1) & = & e^{-x_1} & \approx & 0.5452 \\
x_3 & = & g(x_2) & = & e^{-x_2} & \approx & 0.5797 \\
x_4 & = & g(x_3) & = & e^{-x_3} & \approx & 0.5601 \\
x_5 & = & g(x_4) & = & e^{-x_4} & \approx & 0.5712 \\
x_6 & = & g(x_5) & = & e^{-x_5} & \approx & 0.5649 \\
x_7 & = & g(x_6) & = & e^{-x_6} & \approx & 0.5684 \\
x_8 & = & g(x_7) & = & e^{-x_7} & \approx & 0.5664 \\
x_9 & = & g(x_8) & = & e^{-x_8} & \approx & 0.5676 \\
x_{10} & = & g(x_9) & = & e^{-x_9} & \approx & 0.5669 \\
x_{11} & = & g(x_{10}) & = & e^{-x_{10}} & \approx & 0.5673 \\
x_{12} & = & g(x_{11}) & = & e^{-x_{11}} & \approx & 0.5671
\end{array}$$

These computations were performed using double precision arithmetic, but we show only the first four digits after the decimal point. The values x_n appear to be converging to $x_* = 0.567\cdots$, and this could be confirmed by computing additional iterations.

Example 6.2.2. It is often the case that a fixed–point problem does not have a unique form. Consider the fixed–point problem from the previous example, $x = e^{-x}$. If we take natural logarithms

of both sides of this equation and rearrange we obtain $x = -\ln(x)$ as an alternative fixed–point equation. Although both these equations have the same fixed-points, the corresponding fixed–point iterations, $x_{n+1} = g(x_n)$, behave very differently. Table 6.1 lists the iterates computed using each of the fixed–point iteration functions $g(x) = e^{-x}$ and $g(x) = -\ln(x)$. In each case we used $x_0 = 0.5$ as an initial estimate of x_*. The computations were performed using double precision arithmetic, but we show only the first four digits following the decimal point. The asterisks in the table are used to indicate that the logarithm of a negative number (e.g., $x_5 = \ln(x_4) \approx \ln(-0.0037)$) is not defined when using real arithmetic.

	Iteration Function	
	$x_{n+1} = e^{-x_n}$	$x_{n+1} = -\ln(x_n)$
n	x_n	x_n
0	0.5000	0.5000
1	0.6065	0.6931
2	0.5452	0.3665
3	0.5797	1.0037
4	0.5601	-0.0037
5	0.5712	******
6	0.5649	******
7	0.5684	******
8	0.5664	******
9	0.5676	******
10	0.5669	******
11	0.5673	******
12	0.5671	******

Table 6.1: Computing the root of $x - e^{-x} = 0$ by fixed–point iteration. The asterisks indicate values that are not defined when using only real arithmetic. Note, if complex arithmetic is used as in MATLAB, these values are defined and are complex numbers.

As we saw in the previous example, the iterates computed using the iteration function $g(x) = e^{-x}$ converge to $x_* = 0.5671\cdots$. But here we see that the iterates computed using the iteration function $g(x) = -\ln(x)$ diverge. Why should these two fixed–point iterations behave so differently? The difficulty is not that we started from $x_0 = 0.5$. We would observe similar behavior starting from any point sufficiently close to $x_* = 0.5671\cdots$.

Recall that the fixed–point can be characterized by the point where the curves $y = x$ and $y = g(x)$ intersect. The top two plots in Fig. 6.7 show graphs of these intersecting curves for the two fixed–point functions $g(x) = e^{-x}$ and $g(x) = -\ln(x)$. In the bottom two plots of this same figure, we show, geometrically, how the fixed–point iteration proceeds:

$$x_0 \to g(x_0) \to x_1 \to g(x_1) \to x_2 \to g(x_2) \to \cdots$$

We observe:

- In the case of $g(x) = e^{-x}$, near the intersection point, the slope of $g(x)$ is relatively flat, and each evaluation $g(x_n)$ brings us closer to the intersection point.
- In the case of $g(x) = -\ln(x)$, near the intersection point, the slope of $g(x)$ is relatively steep, and each evaluation $g(x_n)$ pushes us away from the intersection point.

In general, if the slope satisfies $|g'(x)| < 1$ near $x = x_*$, and if the initial estimate x_0 is "close enough" to x_*, then we can expect the fixed–point iteration to converge. Moreover, by looking at these plots we might suspect that very flat slopes (that is, $|g'(x)| \approx 0$) result in faster convergence,

and less flat slopes (that is, $|g'(x)| \lessapprox 1$) result in slower convergence. In the next subsection we provide mathematical arguments to confirm these geometrical observations.

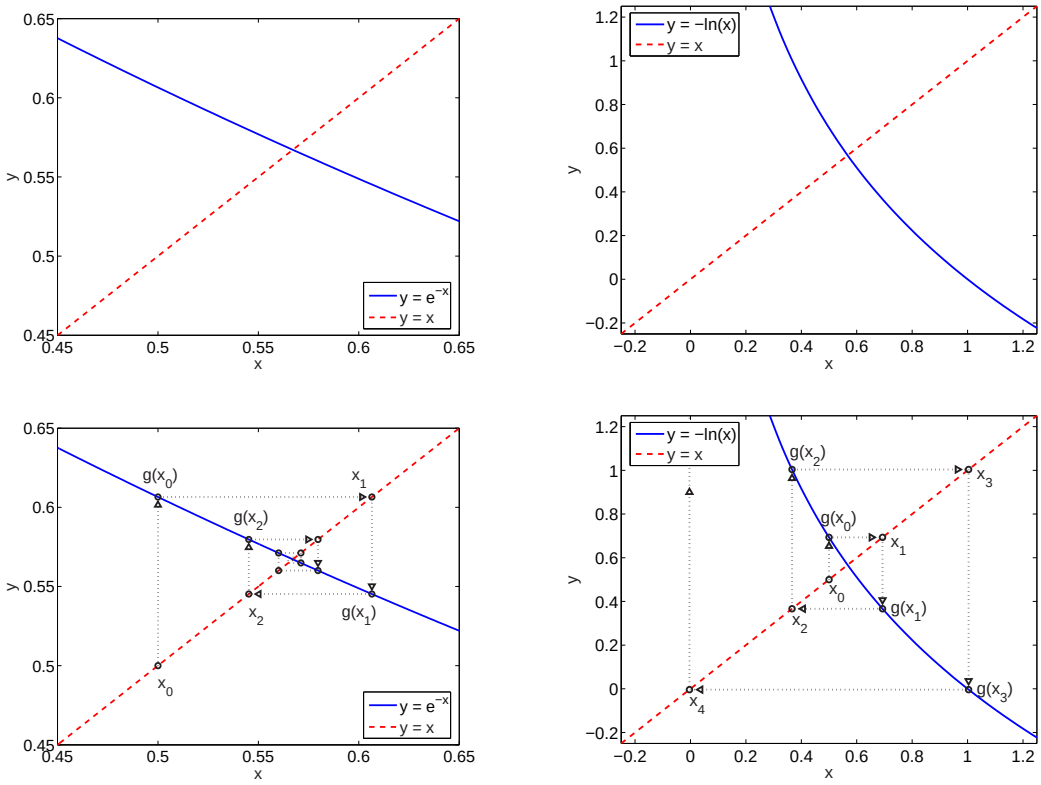

Figure 6.7: Graphs of $y = g(x)$ and $y = x$, where $g(x) = e^{-x}$ (left) and $g(x) = -\ln(x)$ (right). The point where these graphs intersect is the desired fixed–point. The bottom two plots show how the fixed–point iteration proceeds: $x_0 \to g(x_0) \to x_1 \to g(x_1) \to x_2 \to g(x_2) \to \cdots$ In the case of $g(x) = e^{-x}$ these values converge toward the fixed–point, but in the case of $g(x) = -\ln(x)$ these values move away from the fixed–point.

Example 6.2.3. Consider the cubic polynomial $f(x) = x^3 - x - 1$, whose three roots comprise one real root and a complex conjugate pair of roots. From the graph of f, shown in Fig. 6.8, we see that the one real root is $x_* \approx 1.3$. In Section 6.5 we look at the specific problem of computing roots of a polynomial, but here we consider using a fixed–point iteration. The first thing we need to do is algebraically manipulate the equation $f(x) = 0$ to obtain an equation of the form $x = g(x)$. There are many ways we might do this; consider, for example:

- Approach 1.
$$x^3 - x - 1 = 0 \quad \Rightarrow \quad x^3 = x + 1. \quad \Rightarrow \quad x = \sqrt[3]{x+1}$$
In this case we have the fixed–point iteration: $x_{n+1} = \sqrt[3]{x_n + 1}$.

- Approach 2.
$$x^3 - x - 1 = 0 \quad \Rightarrow \quad x = x^3 - 1.$$
In this case we have the fixed–point iteration: $x_{n+1} = x_n^3 - 1$.

Fig. 6.9 displays plots showing the intersection points of the graphs of $y = x$ and $y = g(x)$ for each of the above fixed–point iteration functions.

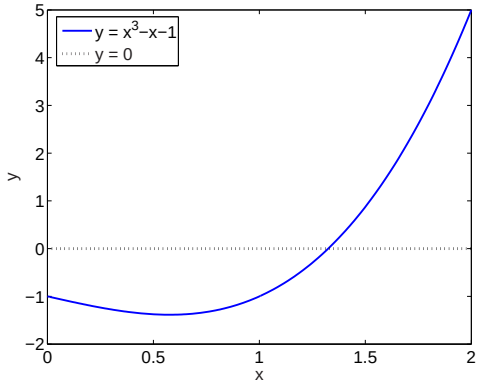

Figure 6.8: Graph of $f(x) = x^3 - x - 1$.

From the discussion in the previous example, we might expect the fixed–point iteration

$$x_{n+1} = \sqrt[3]{x_n + 1}$$

to converge very quickly because near the intersection point the slope of $g(x) = \sqrt[3]{x+1}$ is very flat. On the other hand, we might expect the fixed–point iteration

$$x_{n+1} = x_n^3 - 1$$

to diverge because near the intersection point the slope of $g(x) = x^3 - 1$ is very steep. This is, in fact, what we observe in our computational results. Starting from the point $x_0 = 1.0$ for each of these iteration functions, we compute the results in Table 6.2. The entry HUGE in the table corresponds to a number clearly too large to be of interest; the iteration ultimately overflows on any computer or calculator! The iterates generated using the iteration function $g(x) = \sqrt[3]{x+1}$ converge to the root $x_* = 1.3247\cdots$, while the iterates generated using the iteration function $g(x) = x^3 - 1$ diverge.

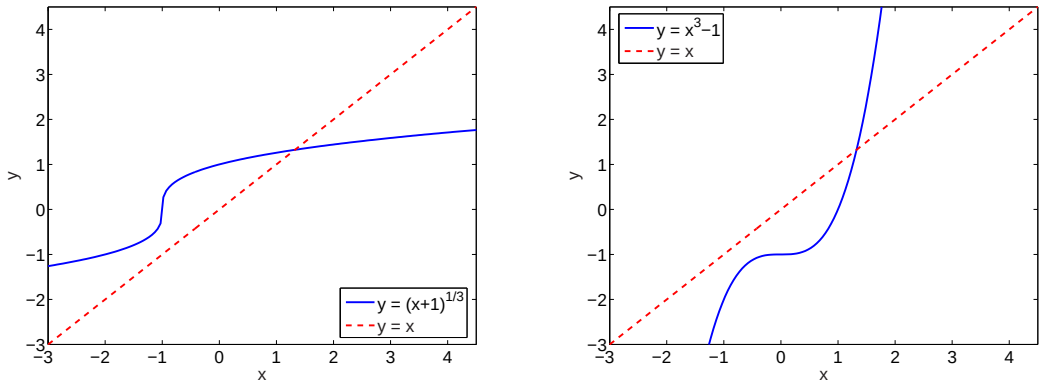

Figure 6.9: Graphs of $y = g(x)$ and $y = x$, where $g(x) = \sqrt[3]{x+1}$ (left) and $g(x) = x^3 - 1$ (right).

Problem 6.2.1. *Assume that the function g is continuous and that the sequence $\{x_n\}_{n=0}^{\infty}$ converges to a fixed–point x_*. Employ a limiting argument to show that x_* satisfies $x_* = g(x_*)$.*

	Iteration Function	
	$x_{n+1} = \sqrt[3]{x_n + 1}$	$x_{n+1} = x_n^3 - 1$
n	x_n	x_n
0	1.000	1.000
1	1.260	0.000
2	1.312	-1.000
3	1.322	-2.000
4	1.324	-9.000
5	1.325	-730.0
6	1.325	HUGE

Table 6.2: Computing the real root of $x^3 - x - 1 = 0$ by fixed–point iteration.

Problem 6.2.2. *Consider the function $f(x) = x - e^{-x}$. The equation $f(x) = 0$ has a root x_* in $(0, 1)$. Use Rolle's Theorem (see Problem 4.2.1) to prove that this root x_* of $f(x)$ is simple.*

Problem 6.2.3. *1. Show algebraically by rearrangement that the equation $x = -\ln(x)$ has the same roots as the equation $x = e^{-x}$*

2. Show algebraically by rearrangement that the equation $x = \sqrt[3]{x + 1}$ has the same roots as the equation $x^3 - x - 1 = 0$

3. Show algebraically by rearrangement that the equation $x = x^3 - 1$ has the same roots as the equation $x^3 - x - 1 = 0$

Problem 6.2.4. *Prove that the equation $x = -\ln(x)$ has a unique simple real root.*

Problem 6.2.5. *Prove that the equation $x^3 - x - 1 = 0$ has a unique simple real root.*

Problem 6.2.6. *For $f(x) = x^3 - x - 1$, show that the rearrangement $x^3 - x - 1 = x(x^2 - 1) - 1 = 0$ leads to a fixed–point iteration function $g(x) = \dfrac{1}{x^2 - 1}$, and show that the rearrangement $x^3 - x - 1 = (x - 1)(x^2 + x + 1) - x = 0$ leads to the iteration function $g(x) = 1 + \dfrac{x}{x^2 + x + 1}$. How do the iterates generated by each of these fixed–point functions behave? [In each case compute a few iterates starting from $x_0 = 1.5$.]*

6.2.2 Convergence Analysis of Fixed–Point Iteration

In the previous subsection we observed that the convergence behavior of the fixed–point iteration appears to be related to the magnitude of the slope of the iteration function, $|g'(x)|$, near the fixed–point. In this subsection we solidify these observations with mathematical arguments.

Given an initial estimate x_0 of the fixed–point x_* of the equation $x = g(x)$, we need two properties of the iteration function g to ensure that the sequence of iterates $\{x_n\}_{n=1}^{\infty}$ defined by

$$x_{n+1} = g(x_n), \quad n = 0, 1, \ldots,$$

converges to x_*:

- $x_* = g(x_*)$; that is, x_* is a fixed–point of the iteration function g.
- For any n, let's define $e_n = x_n - x_*$ to be the **error** in x_n considered as an approximation to x_*. Then, for some constant $N > 0$ and all $n > N$ the sequence of absolute errors $\{|e_n|\}_{n=0}^{\infty}$ decreases monotonically to 0; that is, for each such value of n, the iterate x_{n+1} is closer to the point x_* than was the previous iterate x_n.

Assuming that the derivative g' is continuous wherever we need it to be, from the Mean Value Theorem (Problem 6.1.3), we have

$$\begin{aligned} e_{n+1} &= x_{n+1} - x_* \\ &= g(x_n) - g(x_*) \\ &= g'(\eta_n) \cdot (x_n - x_*) \\ &= g'(\eta_n) \cdot e_n \end{aligned}$$

where η_n is a point between x_* and x_n.

Summarizing, the *key equation*

$$e_{n+1} = g'(\eta_n) \cdot e_n$$

states that the new error e_{n+1} is a factor $g'(\eta_n)$ times the old error e_n.

Now, the new iterate x_{n+1} is closer to the root x_* than is the old iterate x_n if and only if $|e_{n+1}| < |e_n|$. This is guaranteed if and only if $|g'(\eta_n)| \leq G < 1$ for some constant G. So, informally, if $|g'(\eta_n)| \leq G < 1$ for each value of η_n arising in the iteration, the absolute errors tend to zero and the iteration converges.

More specifically:

- If for some constant $0 < G < 1$, $|g'(x)| \leq G$ (the **tangent line condition**) for all $x \in I = [x_* - r, x_* + r]$ then, for every starting point $x_0 \in I$, the sequence $\{x_n\}_{n=0}^{\infty}$ is *well defined* and *converges* to the fixed–point x_*. (Note that, as $x_n \to x_*$, $g'(x_n) \to g'(x_*)$ and so $g'(\eta_n) \to g'(x_*)$ since η_n is "trapped" between x_n and x_*.)

- The smaller the value of $|g'(x_*)|$, the more rapid is the ultimate rate of convergence. Ultimately (that is, when the iterates x_n are so close to x_* that $g'(\eta_n) \approx g'(x_*)$) each iteration reduces the magnitude of the error by a factor of about $|g'(x_*)|$.

- If $g'(x_*) > 0$, convergence is ultimately one–sided. Because, when x_n is close enough to x_* then $g'(\eta_n)$ has the same sign as $g'(x_*) > 0$ since we assume that $g'(x)$ is continuous near x_*. Hence the error e_{r+1} has the same sign as the error e_r for all values of $r > n$.

- By a similar argument, if $g'(x_*) < 0$, ultimately (that is, when the iterates x_n are close enough to x_*) convergence is two–sided because the signs of the errors e_r and e_{r+1} are opposite for each value of $r > n$.

We have chosen the interval I to be *centered* at x_*. This is to simplify the theory. Then, if x_0 lies in the interval I so do all future iterates x_n. The disadvantage of this choice is that we don't know the interval I until we have calculated the answer x_*! However, what this result does tell us is what to expect if we start close enough to the solution. If $|g'(x_*)| \leq G < 1$, $g'(x)$ is continuous in a neighborhood of x_*, and x_0 is close enough to x_* then the fixed–point iteration

$$x_{n+1} = g(x_n), \quad n = 0, 1, \ldots,$$

converges to x_*. The convergence rate is *linear* if $g'(x_*) \neq 0$ and *superlinear* if $g'(x_*) = 0$; that is, ultimately

$$e_{n+1} \approx g'(x_*) e_n$$

Formally, if the errors satisfy

$$\lim_{n \to \infty} \frac{|e_{n+1}|}{|e_n|^p} = K$$

where K is a positive constant, then the convergence rate is p. The case $p = 1$ is labeled linear, $p = 2$ quadratic, $p = 3$ cubic, etc.; any case with $p > 1$ is superlinear. The following examples illustrate this theory.

Example 6.2.4. Consider the fixed–point problem

$$x = e^{-x}.$$

Here we have $g(x) = e^{-x}$, $g'(x) = -e^{-x}$, and from Examples 6.2.1 and 6.2.2, we know this problem has a fixed–point $x_* = 0.5671\cdots$. Although we have already observed that the fixed–point iteration converges for this problem, we can now explain mathematically why this should be the case.

- If $x > 0$ then $-1 < g'(x) < 0$. Thus, for all $x \in (0, \infty)$, $|g'(x)| < 1$. In particular, we can choose an interval, $I \approx [0.001, 1.133]$ which is approximately centered symmetrically about x_*. Since $|g'(x)| < 1$ for all $x \in I$, then for any $x_0 \in I$, the fixed–point iteration will converge.
- Since $g'(x_*) = -0.567\cdots < 0$, convergence is ultimately two–sided. This can be observed geometrically from Fig. 6.7, where we see x_0 is to the left of the fixed–point, x_1 is to the right, x_2 is to the left, and so on.
- Ultimately, the error at each iteration is reduced by a factor of approximately

  $$\frac{|e_{n+1}|}{|e_n|} \approx |g'(x)| \approx |-0.567|,$$

  as can be seen from the computed results shown in Table 6.3.

n	x_n	e_n	e_n/e_{n-1}
0	0.5000	-0.0671	—
1	0.6065	0.0394	-0.587
2	0.5452	-0.0219	-0.556
3	0.5797	0.0126	-0.573
4	0.5601	-0.0071	-0.564
5	0.5712	0.0040	-0.569

Table 6.3: Errors and error ratios for fixed–point iteration with $g(x) = e^{-x}$

Example 6.2.5. Consider the fixed–point problem

$$x = \sqrt[3]{x+1}.$$

Here we have $g(x) = \sqrt[3]{x+1}$, $\quad g'(x) = \dfrac{1}{3(x+1)^{2/3}}$, and from Example 6.2.3, we know this problem has a fixed–point $x_* = 1.3247\cdots$. An analysis of the convergence behavior of the fixed–point iteration for this problem is as follows:

- If $x \geq 0$ then $x + 1 \geq 1$, and hence $0 \leq \dfrac{1}{x+1} \leq 1$. Therefore $0 \leq g'(x) = \dfrac{1}{3(x+1)^{2/3}} \leq \dfrac{1}{3}$. So $|g'(x)| \leq \dfrac{1}{3}$ for $x \in [0, \infty)$. In particular, choose an interval, $I \approx [0, 2.649]$ which is approximately symmetrically centered around x_*. Since $|g'(x)| < 1$ for all $x \in I$, then for any $x_0 \in I$, the fixed–point iteration will converge.
- Since $g'(x_*) \approx 0.189 > 0$, convergence is ultimately one–sided, and, since $|g'(x_*)|$ is quite small, convergence is speedy. This can be observed from the results shown in Table 6.2, where we see that all of $x_0, x_1, x_2, \ldots$ are to the left of (i.e., less than) x_*.

n	x_n	e_n	e_n/e_{n-1}
0	1.000	-0.325	—
1	1.260	-0.065	0.200
2	1.312	-0.012	0.192
3	1.322	-0.002	0.190
4	1.324	-0.000	0.190

Table 6.4: Errors and error ratios for fixed–point iteration with $g(x) = \sqrt[3]{x+1}$

- The error at each iteration is ultimately reduced by a factor of approximately

$$\frac{|e_{n+1}|}{|e_n|} \approx |g'(x)| \approx |0.189|,$$

as can be seen from the computed results shown in Table 6.4.

Problem 6.2.7. *For the divergent sequences shown in Tables 6.1 and 6.2 above, argue why*

$$|x_* - x_{n+1}| > |x_* - x_n|$$

whenever x_n is sufficiently close, but not equal, to x_*.

Problem 6.2.8. *Analyze the convergence, or divergence, of the fixed–point iteration for each of the iteration functions*

1. $g(x) \equiv \dfrac{1}{x^2 - 1}$
2. $g(x) \equiv 1 + \dfrac{x}{x^2 + x + 1}$

See Problem 6.2.6 for the derivation of these iteration functions. Does your analysis reveal the behavior of the iterations that you computed in Problem 6.2.6?

Problem 6.2.9. *Consider the fixed–point equation $x = 2\sin x$ for which there is a root $x_* \approx 1.90$ radians. Why would you expect the fixed–point iteration $x_{n+1} = 2\sin x_n$ to converge from a starting point x_0 close enough to x_*? Would you expect convergence to be ultimately one–sided or two–sided? Why?*

Use the iteration $x_{n+1} = 2\sin x_n$ starting (i) from $x_0 = 2.0$ and (ii) from $x_0 = 1.4$. In each case continue until you have 3 correct digits in the answer. Discuss the convergence behavior of these iterations in the light of your observations above.

6.3 Root Finding Methods

Let us turn now to finding roots – that is, points x_* such that $f(x_*) = 0$. The first two methods we consider, the Newton and secant iterations, are closely related to the fixed–point iteration discussed in Section 6.2. We then describe the bisection method, which has a slower rate of convergence than the Newton and secant iterations, but has the advantage that convergence can be guaranteed for relative accuracies greater than ϵ_{DP}. The final method we discuss is rather different than the first three, and is based on inverse interpolation.

6.3.1 The Newton Iteration

Suppose $f(x)$ is a given function, and we wish to compute a simple root of $f(x) = 0$. A systematic procedure for constructing an equivalent fixed–point problem, $x = g(x)$, is to choose the **Newton Iteration Function**

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

The associated fixed–point iteration $x_{n+1} = g(x_n)$ leads to the well–known **Newton Iteration**

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} .$$

Notice that if x_* satisfies $x_* = g(x_*)$, and if $f'(x_*) \neq 0$, then $f(x_*) = 0$, and hence x_* is a simple root of $f(x) = 0$.

How does this choice of iteration function arise? Here are two developments:

- An algebraic derivation can be obtained by expanding $f(z)$ around x using a Taylor series:

  $$f(z) = f(x) + (z - x)f'(x) + (z - x)^2 \frac{f''(x)}{2} + \cdots$$

  A Taylor polynomial approximation is obtained by terminating the infinite series after a finite number of terms. In particular, a linear polynomial approximation is given by

  $$f(z) \approx f(x) + (z - x)f'(x) .$$

  Setting $z = x_{n+1}$, $f(x_{n+1}) = 0$ and $x = x_n$ we obtain

  $$0 \approx f(x_n) + (x_{n+1} - x_n)f'(x_n) .$$

  Rearranging this relation gives the Newton iteration. Note, this is "valid" because we assume x_{n+1} is much closer to x_* than is x_n, so, usually, $f(x_{n+1})$ is much closer to zero than is $f(x_n)$.

- For a geometric derivation, observe that the curve $y = f(x)$ crosses the x-axis at $x = x_*$. In point-slope form, the tangent to $y = f(x)$ at the point $(x_n, f(x_n))$ is given by $y - f(x_n) = f'(x_n)(x - x_n)$. Let x_{n+1} be defined as the value of x where this tangent crosses the x-axis (see Fig. 6.10). Then the point $(x_{n+1}, 0)$ is on the tangent line, so we obtain

  $$0 - f(x_n) = f'(x_n)(x_{n+1} - x_n) .$$

  Rearranging this equation, we obtain the Newton iteration.

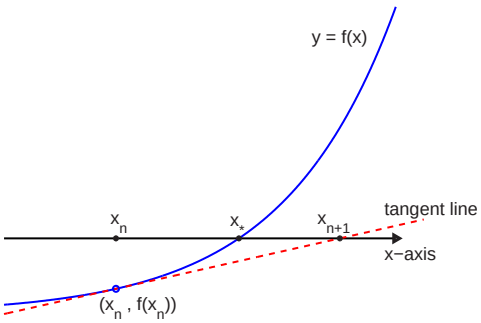

Figure 6.10: Illustration of Newton's method. x_{n+1} is the location where the tangent line to $y = f(x)$ through the point $(x_n, f(x_n))$ crosses the x-axis.

A short computation shows that, for the Newton iteration function g,

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

If we assume that the root x_* is simple, then $f(x_*) = 0$ and $f'(x_*) \neq 0$, so

$$g'(x_*) = \frac{f(x_*)f''(x_*)}{f'(x_*)^2} = 0$$

From the previous section we know that, when x_n is close enough to x_*, then $e_{n+1} \approx g'(x_*)e_n$. So $g'(x_*) = 0$ implies e_{n+1} is very much smaller in magnitude than e_n when $|e_n|$ is itself sufficiently small, indeed convergence will be at least superlinear.

Example 6.3.1. Suppose $f(x) = x - e^{-x}$, and consider using Newton's method to compute a root of $f(x) = 0$. In this case, $f'(x) = 1 + e^{-x}$, and the Newton iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n - e^{-x_n}}{1 + e^{-x_n}} = \frac{e^{x_n}(x_n + 1)}{1 + e^{x_n}}\,.$$

In Table 6.5 we give the iterates, errors and error ratios for the Newton iteration . The correct digits for the iterates, x_n, are underlined. Notice that the number of correct digits approximately doubles for each iteration. Also observe that the magnitude of the ratio

$$\frac{|e_n|}{|e_{n-1}|^2}$$

is essentially constant, indicating quadratic convergence.

n	x_n	e_n	e_n/e_{n-1}	e_n/e_{n-1}^2
0	0.500000000000000	-6.7E-02	—	—
1	0.566311003197218	-8.3E-04	1.2E-02	0.1846
2	0.567143165034862	-1.3E-07	1.5E-04	0.1809
3	0.567143290409781	2.9E-13	-2.3E-06	-0.1716

Table 6.5: Errors and error ratios for the Newton iteration applied to $f(x) = x - e^{-x}$.

Example 6.3.2. Suppose $f(x) = x^3 - x - 1$, and consider using Newton's method to compute a root of $f(x) = 0$. In this case, $f'(x) = 3x^2 - 1$, and the Newton iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^3 - x_n - 1}{3x_n^2 - 1} = \frac{2x_n^3 + 1}{3x_n^2 - 1}\,.$$

In Table 6.6 we give the iterates, errors and error ratios for the Newton iteration . The correct digits for the iterates, x_n, are underlined. For this example, after the first iteration (once we are "close enough" to the root), we see again that the number of correct digits approximately doubles for each iteration. Moreover, the magnitude of the ratio

$$\frac{|e_n|}{|e_{n-1}|^2}$$

is essentially constant, indicating quadratic convergence.

n	x_n	e_n	e_n/e_{n-1}	e_n/e_{n-1}^2
0	1.00000000000000	-3.2E-01	—	—
1	1.50000000000000	1.8E-01	-5.4E-01	1.7E-00
2	1.34782608695652	2.3E-02	1.3E-01	7.5E-01
3	1.32520039895091	4.8E-04	2.1E-02	9.0E-01
4	1.32471817399905	2.2E-07	4.5E-04	9.3E-01
5	1.32471795724479	4.4E-14	2.0E-07	9.3E-01

Table 6.6: Errors and error ratios for the Newton iteration applied to $f(x) = x^3 - x - 1$.

These examples suggest that at each Newton iteration the number of correct digits in x_n approximately doubles: this behavior is typical for quadratically convergent iterations. Can we confirm this theoretically? First observe by definition of fixed–points and fixed–point iteration, we can represent the error as

$$\begin{aligned} e_{n+1} &= x_{n+1} - x_* \\ &= g(x_n) - g(x_*)\,. \end{aligned}$$

If we now expand $g(x_n)$ around x_* using a Taylor series, we obtain

$$\begin{aligned} e_{n+1} &= x_{n+1} - x_* \\ &= g(x_n) - g(x_*) \\ &= \left(g(x_*) + (x_n - x_*)g'(x_*) + (x_n - x_*)^2 \frac{g''(x_*)}{2} + \cdots \right) - g(x_*) \\ &= e_n g'(x_*) + e_n^2 \frac{g''(x_*)}{2} + \cdots \end{aligned}$$

where $e_n = x_n - x_*$. Recall that for Newton's iteration, $g'(x_*) = 0$. Then if we neglect the higher order terms not shown here, and assume $g''(x_*) \neq 0$ as it is in most cases, we obtain the error equation

$$e_{n+1} \approx \frac{g''(x_*)}{2} \cdot e_n^2$$

(For example, for the iteration in Table 6.5 $\dfrac{e_{n+1}}{e_n^2} \approx \dfrac{g''(x_*)}{2} \approx 0.1809$ and we have quadratic convergence.)

Now, let's check that this error equation implies ultimately doubling the number of correct digits. Dividing both sides of the error equation by x_* leads to the relation

$$r_{n+1} \approx \frac{x_*\, g''(x_*)}{2} \cdot r_n^2$$

where the relative error r_n in e_n is defined by $r_n \equiv \dfrac{e_n}{x_*}$. Suppose that x_n is correct to about t decimal digits, so that $r_n \approx 10^{-t}$ and choose d to be the integer such that

$$\frac{x_* g''(x_*)}{2} \approx \pm 10^d$$

where the sign of $\pm$ is chosen according to the sign of the left hand side. Then,

$$r_{n+1} \approx \frac{x_*\, g''(x_*)}{2} \cdot r_n^2 \approx \pm 10^d \cdot 10^{-2t} = \pm 10^{d-2t}$$

As the iteration converges, t grows and, ultimately, $2t$ will become much larger than d so then $r_{n+1} \approx \pm 10^{-2t}$. Hence, for sufficiently large values t, when r_n is accurate to about t decimal digits then r_{n+1} is accurate to about $2t$ decimal digits. This approximate doubling of the number of correct digits at each iteration is evident in Tables 6.5 and 6.6. When the error e_n is sufficiently small that

the error equation is accurate then e_{n+1} has the same sign as $g''(x_*)$. Then, convergence for the Newton iteration is one–sided: from above if $g''(x_*) > 0$ and from below if $g''(x_*) < 0$.

Summarizing — for a simple root x_* of f:

- The Newton iteration converges when started from any point x_0 that is "close enough" to the root x_* assuming $g'(x)$ is continuous in a closed interval containing x_*, because the Newton iteration is a fixed–point iteration with $|g'(x_*)| < 1$
- When the error e_n is sufficiently small that the error equation is accurate, the number of correct digits in x_n approximately doubles at each Newton iteration.
- When the error e_n is sufficiently small that the error equation is accurate, the Newton iteration converges from one side.

Note, the point x_0 that is "close enough" to guarantee convergence in the first point above may not provide "an error e_n that is sufficiently small that the error equation is accurate" in the second and third points above. The following example illustrates this matter.

What happens if f has a multiple root? Let's consider a simple case when $f(x) = (x - x_*)^r F(x)$ where $F(x_*) \neq 0$ and $r > 1$. Then, differentiating $f(x)$ twice and inserting the results into the formula $g'(x) = \dfrac{f(x)f''(x)}{f'(x)^2}$ derived above, we get $g'(x) \approx \dfrac{r-1}{r}$ when $(x - x_*)$ is small. So, at any point x close enough to the root x_*, we have $g'(x) > 0$ and $|g'(x)| < 1$, implying one sided linear convergence. Even though we have chosen to simplify the analysis by making a special choice of $f(x)$, our conclusions are correct more generally.

Example 6.3.3. Consider the equation $f(x) \equiv x^{20} - 1 = 0$ with real roots ± 1. If we use the Newton iteration the formula is

$$x_{n+1} = x_n - \frac{x_n^{20} - 1}{20x_n^{19}}$$

Starting with $x_0 = 0.5$, which we might consider to be "close" to the root $x_* = 1$ we compute $x_1 = 26213$, $x_2 = 24903$, $x_3 = 23658$ to five digits. The first iteration takes an enormous leap across the root then the iterations "creep" back towards the root. So, we didn't start "close enough" to get quadratic convergence; the iteration is converging linearly. If, instead, we start from $x_0 = 1.5$, to five digits we compute $x_1 = 1.4250$, $x_2 = 1.3538$ and $x_3 = 1.2863$, and convergence is again linear. If we start from $x_0 = 1.1$, to five digits we compute $x_1 = 1.0532$, $x_2 = 1.0192$, $x_3 = 1.0031$, and $x_4 = 1.0001$. We observe quadratic convergence in this last set of iterates.

Problem 6.3.1. *Construct the Newton iteration by deriving the tangent equation to the curve $y = f(x)$ at the point $(x_n, f(x_n))$ and choosing the point x_{n+1} as the place where this line crosses the x-axis. (See Fig. 6.10.)*

Problem 6.3.2. *Assuming that the root x_* of f is simple, show that $g''(x_*) = \dfrac{f''(x_*)}{f'(x_*)}$. Hence, show that*

$$\frac{e_{n+1}}{e_n^2} \approx \frac{f''(x_*)}{2f'(x_*)}$$

Problem 6.3.3. *For the function $f(x) = x^3 - x - 1$, write down the Newton iteration function g and calculate $g'(x)$ and $g''(x)$. Table 6.6 gives a sequence of iterates for this Newton iteration function. Do the computed ratios behave in the way that you would predict?*

Problem 6.3.4. *Consider computing the root x_* of the equation $x = \cos(x)$, that is computing the value x_* such that $x_* = \cos(x_*)$. Use a calculator for the computations required below.*

1. *Use the iteration $x_{n+1} = \cos(x_n)$ starting from $x_0 = 1.0$ and continuing until you have computed x_* correctly to three digits. Present your results in a table. Observe that the iteration converges. Is convergence from one side? Viewing the iteration as being of the form $x_{n+1} = g(x_n)$, by analyzing the iteration function g show that you would expect the iteration to converge and that you can predict the convergence behavior.*

2. *Now use a Newton iteration to compute the root x_* of $x = \cos(x)$ starting from the same value of x_0. Continue the iteration until you obtain the maximum number of digits of accuracy available on your calculator. Present your results in a table, marking the number of correct digits in each iterate. How would you expect the number of correct digits in successive iterates to behave? Is that what you observe? Is convergence from one side? Is that what you would expect and why?*

Problem 6.3.5. *Some computers perform the division $\frac{N}{D}$ by first computing the reciprocal $\frac{1}{D}$ and then computing the product $\frac{N}{D} = N \cdot \frac{1}{D}$. Consider the function $f(x) \equiv D - \frac{1}{x}$. What is the root of $f(x)$? Show that, for this function $f(x)$, the Newton iteration function $g(x) = x - \frac{f(x)}{f'(x)}$ can be written so that evaluating it does not require divisions. Simplify this iteration function so that evaluating it uses as few adds, subtracts, and multiplies as possible.*

Use the iteration that you have derived to compute $\frac{N}{D} = \frac{2}{3}$ without using any divisions.

Problem 6.3.6. *Consider the case when $f(x) = (x - x_*)^r F(x)$ where $F(x_*) \neq 0$ and $r > 1$. Show that $g'(x) = \frac{f(x)f''(x)}{f'(x)^2} \approx \frac{r-1}{r}$ when $(x - x_*)$ is small.*

6.3.2 Secant Iteration

The major disadvantage of the Newton iteration is that the function f' must be evaluated at each iteration. This derivative may be difficult to compute or may be far more expensive to evaluate than the function f. One quite common circumstance where it is difficult to derive f' is when f is defined by a computer program. Recently, the technology of Automatic Differentiation has made computing automatically the values of derivatives defined in such complex ways a possibility but methods that do not require values of $f'(x)$ remain in widespread use.

When the cost of evaluating the derivative is the problem, this cost can often be reduced by using instead the iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{m_n},$$

where m_n is an approximation of $f'(x_n)$. For example, we could use a *difference quotient* (recall the discussion in Section 5.1 on approximating derivatives),

$$m_n = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

This difference quotient uses the already–computed values $f(x)$ from the current and the previous iterate. It yields the **secant iteration**:

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{\left(\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}\right)} \\ &= x_n - \left(\frac{f(x_n)}{f(x_n) - f(x_{n-1})}\right)(x_n - x_{n-1}) \quad \text{[preferred computational form]} \\ &= \frac{x_n f(x_{n-1}) - x_{n-1} f(x_n)}{f(x_{n-1}) - f(x_n)} \quad \text{[computational form to be avoided]} \end{aligned}$$

This iteration requires two starting values x_0 and x_1. Of the three expressions for the iterate x_{n+1}, the second expression

$$x_{n+1} = x_n - \left(\frac{f(x_n)}{f(x_n) - f(x_{n-1})} \right) (x_n - x_{n-1}), \quad n = 1, 2, \ldots$$

provides the "best" implementation. It is written as a *correction* of the most recent iterate in terms of a scaling of the difference of the current and the previous iterate. All three expressions suffer from cancelation; in the last, cancelation can be catastrophic.

When the error e_n is small enough the secant iteration converges, and it can be shown that the error ultimately behaves like $|e_{n+1}| \approx K|e_n|^p$ where $p = \dfrac{1+\sqrt{5}}{2} \approx 1.618$ and K is a positive constant. (Derivation of this result is not trivial.) The value $\dfrac{1+\sqrt{5}}{2}$ is the well-known golden ration whcih we will meet again in the next chapter. Since $p > 1$, the error exhibits *superlinear convergence* but at a rate less than that exemplified by quadratic convergence. However, this superlinear convergence property has a similar effect on the behavior of the error as the quadratic convergence property of the Newton iteration. As long as $f''(x)$ is continuous in an interval containing x_* and the root x_* is simple, we have the following properties:

- If the initial iterates x_0 and x_1 are both sufficiently close to the root x_* then the secant iteration is guaranteed to converge to x_*.
- When the iterates x_n and x_{n-1} are sufficiently close to the root x_* then the secant iteration converges superlinearly; that is, the number of correct digits increases faster than linearly at each iteration. Ultimately the number of correct digits increases by a factor of approximately 1.6 per iteration.

As with the Newton iteration we may get convergence on larger intervals containing x_* than we observe superlinearity.

Example 6.3.4. Consider solving the equation $f(x) \equiv x^3 - x - 1 = 0$ using the secant iteration starting with $x_0 = 1$ and $x_1 = 2$. Observe in Table 6.7 that the errors are tending to zero at an increasing rate but not as fast as in quadratic convergence where the number of correct digits doubles at each iteration. The ratio $|e_i|/|e_{i-1}|^p$ where $p = \dfrac{1+\sqrt{5}}{2}$ is tending to a limit until limiting precision is encountered.

n	x_n	e_n	$\lvert e_n\rvert/\lvert e_{n-1}\rvert^p$
0	1.0	-0.32471795724475	–
1	2.0	0.67528204275525	–
2	1.16666666666666	-0.15805129057809	0.298
3	1.25311203319502	-0.07160592404973	1.417
4	1.33720644584166	0.01248848859691	0.890
5	1.32385009638764	-0.00086786085711	1.043
6	1.32470793653209	-0.00001002071266	0.901
7	1.32471796535382	0.00000000810907	0.995
8	1.32471795724467	-0.00000000000008	0.984
9	1.32471795724475	-0.00000000000000	–

Table 6.7: Secant iterates, errors and error ratios

Problem 6.3.7. *Derive the secant iteration geometrically as follows: Write down the formula for the straight line joining the two points* $(x_{n-1}, f(x_{n-1}))$ *and* $(x_n, f(x_n))$*, then for the next iterate* x_{n+1} *choose the point where this line crosses the* x*-axis. (The figure should be very similar to Fig. 6.10.)*

Problem 6.3.8. *In the three expressions for the secant iteration above, assume that the calculation of* $f(x_n)$ *is performed just once for each value of* x_n*. Write down the cost per iteration of each form in terms of arithmetic operations (adds, multiplies, subtracts and divides) and function,* f*, evaluations.*

6.3.3 Bisection

The methods described so far choose an initial estimate for the root of $f(x)$, then use an iteration that converges in appropriate circumstances. However, normally there is no guarantee that these iterations will converge from an arbitrary initial estimate. Here we discuss iterations where the user must supply more information but the resulting iteration is guaranteed to converge.

The interval $[a, b]$ is a **bracket for a root** of $f(x)$ if $f(a) \cdot f(b) \leq 0$. Clearly, when f is continuous a bracket contains a root of $f(x)$. For, if $f(a) \cdot f(b) = 0$, then either a or b or both is a root of $f(x) = 0$. And, if $f(a) \cdot f(b) < 0$, then $f(x)$ starts with one sign at a and ends with the opposite sign at b, so by continuity $f(x) = 0$ must have a root between a and b. (Mathematically, this last argument appeals to the Intermediate Value Theorem, see Problem 6.1.1.)

Let $[a, b]$ be an initial bracket for a root of $f(x)$. The bisection method refines this bracket (that is, makes it smaller) as follows. Let $m = \dfrac{a+b}{2}$; this divides the interval $[a, b]$ into two subintervals $[a, m]$ and $[m, b]$ each of half of the length of $[a, b]$. We know that at least one of these subintervals must be a bracket for a root (see Problem 6.3.10). If $f(a) \cdot f(m) \leq 0$, then there is a root in $[a, m]$, and this is a new (smaller) bracket of the root. If $f(m) \cdot f(b) \leq 0$, then there is a root in $[m, b]$, and this is a new (smaller) bracket of the root. Thus, one iteration of the bisection process refines an initial bracket $[a, b]$ to a new bracket $[a_{\text{new}}, b_{\text{new}}]$ of half the length. This process can be repeated with the newly computed bracket, until it becomes sufficiently small. Fig. 6.11 illustrates one iteration of the bisection method.

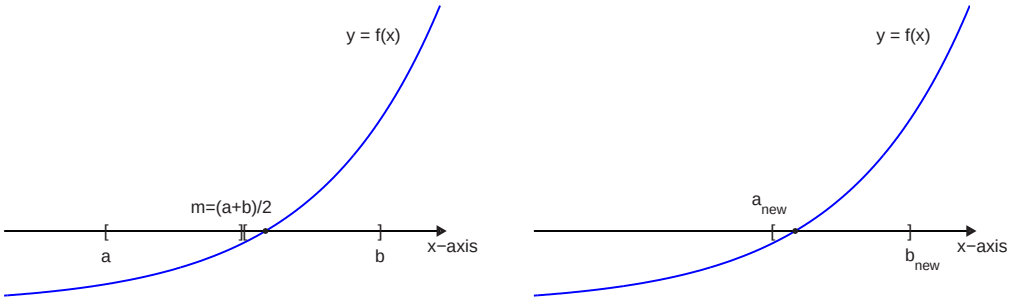

Figure 6.11: Illustration of the bisection method. The left plot shows the initial bracket, $[a, b]$, and the midpoint of this interval, $m = (a + b)/2$. Since the root is in $[m, b]$, the refined bracket is $[a_{\text{new}}, b_{\text{new}}]$, where $a_{\text{new}} := m$ and $b_{\text{new}} := b$.

The simple approach used by the bisection method means that once we determine an initial bracket, the method is guaranteed to converge. Although the Newton and secant iterations require the initial guess to be "close enough" to the root in order to guarantee convergence, there is no such requirement here. The end points of the initial bracket can be very far from the root – as long as the function is continuous, the bisection method will converge.

The disadvantage of bisection is that when the Newton and secant iterations converge, they do so much more quickly than bisection. To explain the convergence behavior of bisection, we let

$[a_n, b_n]$ denote the bracket at the nth iteration, and we define the approximation of the root to be the midpoint of this interval, $x_n = m_n = (a_n + b_n)/2$. It is difficult to get a precise formula for the error, but we can say that because bisection simply cuts the interval in half at each iteration, the magnitude of the error, $|e_n| = |x_n - x_*|$, is *approximately* halved at each iteration. Note that it is possible that x_{n-1} (the midpoint of $[a_{n-1}, b_{n-1}]$) is closer to the root than is x_n (the midpoint of $[a_n, b_n]$), so the error does not necessarily decrease at each iteration. Such a situation can be observed in Fig. 6.11, where the midpoint of $[a, b]$ is closer to the root than is the midpoint of $[a_{\text{new}}, b_{\text{new}}]$. However, since the bracket is being refined, bisection eventually converges, and on average we can expect that $\frac{|e_n|}{|e_{n-1}|} \approx \frac{1}{2}$. Thus, on average, the bisection method converges linearly to a root. Note that this implies that $|e_n| \approx 2^{-n}$, or equivalently

$$-\log_2(|e_n|) \approx n\,.$$

Thus, the (approximate) linear convergence of bisection can be illustrated by showing that the points $(n, -\log_2(|e_n|))$ lie essentially on a straight line of slope one (see Example 6.3.5 and Problem 6.3.12).

Example 6.3.5. Consider solving the equation $f(x) \equiv x^3 - x - 1 = 0$ using the bisection method starting with the initial bracket $[a_0, b_0]$ where $a_0 = 1$ and $b_0 = 2$ and terminating when the length of the bracket $[a_n, b_n]$ is smaller than 10^{-4}. Observe in Table 6.8 that the errors are tending to zero but irregularly. The ratio e_n/e_{n-1} is not tending to a limit. However, Fig. 6.12 illustrates the approximate linear convergence rate, since the points $(n, -\log_2(|e_n|))$ lie essentially on a straight line of slope one.

n	x_n	e_n	e_n/e_{n-1}
0	1.5000000000000000	0.1752820427552499	–
1	1.2500000000000000	-0.0747179572447501	-0.426
2	1.3750000000000000	0.0502820427552499	-0.673
3	1.3125000000000000	-0.0122179572447501	-0.243
4	1.3437500000000000	0.0190320427552499	-1.558
5	1.3281250000000000	0.0034070427552499	0.179
6	1.3203125000000000	-0.0044054572447501	-1.293
7	1.3242187500000000	-0.0004992072447501	0.113
8	1.3261718750000000	0.0014539177552499	-2.912
9	1.3251953125000000	0.0004773552552499	0.328
10	1.3247070312500000	-0.0000109259947501	-0.023
11	1.3249511718750000	0.0002332146302499	-21.345
12	1.3248291015625000	0.0001111443177499	0.477
13	1.3247680664062500	0.0000501091614999	0.451

Table 6.8: Bisection iterates, errors and error ratios for Example 6.3.5. The initial bracket was taken to be $[a_0, b_0] = [1, 2]$, and so the initial estimate of the root is $x_0 = 1.5$.

Problem 6.3.9. *Consider using the bisection method to compute the root of $f(x) = x - e^{-x}$ starting with initial bracket $[0, 1]$ and finishing with a bracket of length 2^{-10}. How many iterations will be needed?*

Problem 6.3.10. *Let the interval $[a, b]$ be a bracket for a root of the function f, and let $m = \dfrac{a+b}{2}$ be the midpoint of $[a, b]$. Show that if neither $[a, m]$ nor $[m, b]$ is a bracket, then neither is $[a, b]$. [Hint: If neither $[a, m]$ nor $[m, b]$ is a bracket, then $f(a)\cdot f(m) > 0$ and $f(m)\cdot f(b) > 0$. Conclude that $f(a) \cdot f(b) > 0$; that is, conclude that $[a, b]$ is not a bracket.] Now, proceed to use this contradiction to show that one of the intervals $[a, m]$ and $[m, b]$ must be a bracket for a root of the function f.*

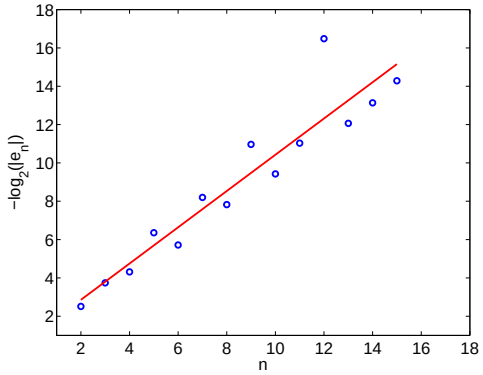

Figure 6.12: Plot of $-\log_2(|e_n|)$ versus n for Example 6.3.5. The points $(n, -\log_2(|e_n|))$ are marked by circles, and the solid line has slope one showing the approximate linear behavior of the points.

Problem 6.3.11. *(Boundary Case) Let $[a, b]$ be a bracket for a root of the function f. In the definition of a bracket, why is it important to allow for the possibility that $f(a) \cdot f(b) = 0$? In what circumstance might we have $f(a) \cdot f(b) = 0$ without having either $f(a) = 0$ or $f(b) = 0$? Is this circumstance likely to occur in practice? How would you modify the algorithm to avoid this problem?*

Problem 6.3.12. *Let $\{[a_n, b_n]\}_{n=0}^{\infty}$ be the sequence of intervals (brackets) produced when bisection is applied to the function $f(x) \equiv x - e^{-x}$ starting with the initial interval (bracket) $[a_0, b_0] = [0, 1]$. Let x_* denote the root of $f(x)$ in the interval $[0, 1]$, and let $e_n = x_n - x_*$, where $x_n = (a_n + b_n)/2$. Fig. 6.13 depicts a plot of $-\log_2(e_n)$ versus n. Show that this plot is consistent with the assumption that, approximately, $e_n = c \cdot 2^{-n}$ for some constant c.*

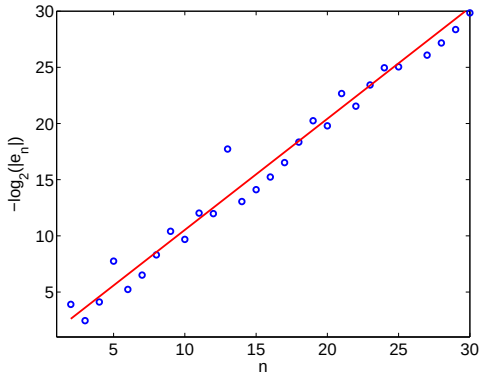

Figure 6.13: Plot of $-\log_2(|e_n|)$ versus n for Problem 6.3.12. The points $(n, -\log_2(|e_n|))$ are marked by circles, and the solid line of slope 1 shows the approximate linear behavior of the points.

6.3.4 Quadratic Inverse Interpolation

A rather different approach to root finding is to use inverse interpolation. The idea is based on the observation that if f has an inverse function, f^{-1}, then

$$f(x) = y \quad \Leftrightarrow \quad x = f^{-1}(y)\,.$$

This relationship implies that if $f^{-1}(y)$ is available, then the root x_* of $f(x) = 0$ is easily computed by evaluating $x_* = f^{-1}(0)$. Unfortunately, it may be very difficult, and in most cases impossible, to

find a closed form expression for $f^{-1}(y)$. Instead of attempting to construct it explicitly, we could try to construct an approximation using an interpolating polynomial, $p(y) \approx f^{-1}(y)$, and then use $p(0)$ as an approximation of x_*. But what degree polynomial should we use, and how do we choose the interpolation points to produce a good approximation? Here, we discuss the most commonly used variant based on *quadratic* inverse interpolation, where the interpolation points are constructed iteratively.

To describe the method, consider a case where we have three points x_{n-2}, x_{n-1} and x_n and we have evaluated the function f at each of these points giving the values $y_{n-i} = f(x_{n-i})$, $i = 0, 1, 2$. Assume that the function f has an inverse function f^{-1}, so that $x_{n-i} = f^{-1}(y_{n-i})$, $i = 0, 1, 2$. Now, make a quadratic (Lagrange form) interpolating function to this inverse data:

$$\begin{aligned} p_2(y) &= \frac{(y-y_{n-1})(y-y_{n-2})}{(y_n-y_{n-1})(y_n-y_{n-2})} f^{-1}(y_n) \\ &\quad + \frac{(y-y_n)(y-y_{n-2})}{(y_{n-1}-y_n)(y_{n-1}-y_{n-2})} f^{-1}(y_{n-1}) \\ &\quad + \frac{(y-y_{n-1})(y-y_n)}{(y_{n-2}-y_{n-1})(y_{n-2}-y_n)} f^{-1}(y_{n-2})\,. \end{aligned}$$

Next, evaluate this expression at $y = 0$ giving

$$\begin{aligned} x_{n+1} = p_2(0) &= \frac{y_{n-1}y_{n-2}}{(y_n-y_{n-1})(y_n-y_{n-2})} x_n \\ &\quad + \frac{y_n y_{n-2}}{(y_{n-1}-y_n)(y_{n-1}-y_{n-2})} x_{n-1} \\ &\quad + \frac{y_{n-1}y_n}{(y_{n-2}-y_{n-1})(y_{n-2}-y_n)} x_{n-2}\,. \end{aligned}$$

Discard the point x_{n-2} and continue with the same process to compute x_{n+2} using the "old" points x_{n-1} and x_n, and the new point x_{n+1}. Proceeding in this way, we obtain an iteration reminiscent of the secant iteration for which the error behaves approximately like $e_{n+1} = Ke_n^{1.44}$. Note that this rate of convergence occurs only when all the x_i currently involved in the formula are close enough to the root x_* of the equation $f(x) = 0$; that is, in the limit quadratic inverse interpolation is a superlinearly convergent method.

Problem 6.3.13. *Derive the formula for the error in interpolation associated with the interpolation formula for $p_2(y)$. Write the error term in terms of derivatives of $f(x)$ rather than of $f^{-1}(y)$, recalling that $\left(f^{-1}\right)'(y) = \dfrac{1}{f'(x)}$.*

6.4 Calculating Square Roots

How does a computer or a calculator compute the square root of a positive real number? In fact, how did engineers calculate square roots before computers were invented? There are many approaches, one dating back to the ancient Babylonians! For example, we could use the identity

$$\sqrt{a} = e^{\frac{1}{2}\ln a}$$

and then use tables of logarithms to find approximate values for $\ln(a)$ and e^b. In fact, some calculators still use this identity along with good routines to calculate logarithms and exponentials.

In this section we describe how to apply root finding techniques to calculate square roots. For any real value $a > 0$, the function $f(x) = x^2 - a$ has two square roots, $x_* = \pm\sqrt{a}$. The Newton

iteration applied to $f(x)$ gives

$$\begin{aligned} x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} &= x_n - \frac{x_n^2 - a}{2x_n} \\ &= \frac{\left(x_n + \dfrac{a}{x_n}\right)}{2} \end{aligned}$$

Note that we only need to perform simple addition, multiplication and division operations to compute x_{n+1}. The (last) algebraically simplified expression is less expensive to compute than the original expression because it involves just one divide, one add, and one division by two, whereas the original expression involves a multiply, two subtractions, a division and a multiplication by two. [When using IEEE Standard floating–point arithmetic, multiplication or division by a power of two of a machine floating–point number involves simply increasing or reducing, respectively, that number's exponent by one, known as a binary shift. This is a low cost operation relative to the standard arithmetic operations.]

From the iteration $x_{n+1} = \dfrac{1}{2}\left(x_n + \dfrac{a}{x_n}\right)$ we observe that since $a > 0$, x_{n+1} has the same sign as x_n for all n. Hence, if the Newton iteration converges, it converges to that root with the same sign as x_0.

Now consider the error in the iterates.

$$e_n \equiv x_n - \sqrt{a}$$

Algebraically, we have

$$\begin{aligned} e_{n+1} = x_{n+1} - \sqrt{a} &= \frac{\left(x_n + \dfrac{a}{x_n}\right)}{2} - \sqrt{a} \\ &= \frac{(x_n - \sqrt{a})^2}{2x_n} = \frac{e_n^2}{2x_n} \end{aligned}$$

From this formula, we see that when $x_0 > 0$ we have $e_1 = x_1 - \sqrt{a} > 0$ so that $x_1 > \sqrt{a} > 0$. It follows similarly that $e_n \geq 0$ for all $n \geq 1$. Hence, if x_n converges to $\sqrt{a}$ then it converges from above. To prove that the iteration converges, note that

$$0 \leq \frac{e_n}{x_n} = \frac{x_n - \sqrt{a}}{x_n - 0} = \frac{\text{distance from } \ x_n \text{ to } \sqrt{a}}{\text{distance from } \ x_n \ \text{ to } \ 0} < 1$$

for $n \geq 1$, so

$$e_{n+1} = \frac{e_n^2}{2x_n} = \left(\frac{e_n}{x_n}\right)\frac{e_n}{2} < \frac{e_n}{2}$$

Hence, for all $n \geq 1$, the error e_n is reduced by a factor of at least $\dfrac{1}{2}$ at each iteration. Therefore the iterates x_n converge to $\sqrt{a}$. Of course, "quadratic convergence rules!" as this is the Newton iteration, so when the error is sufficiently small the number of correct digits in x_n approximately doubles with each iteration. However, if x_0 is far from the root we may not have quadratic convergence initially, and, at worst, the error may only be approximately halved at each iteration.

Not surprisingly, the Newton iteration is widely used in computers' and calculators' mathematical software libraries as a part of the built-in function for determining square roots. Clearly, for this approach to be useful a method is needed to ensure that, from the start of the iteration, the error is sufficiently small that "quadratic convergence rules!". Consider the floating–point number written in the form

$$a = S(a) \cdot 2^{e(a)}$$

where the significand, $S(a)$, satisfies $1 \leq S(a) < 2$ and the exponent $e(a)$ is an integer. (As we saw in sectionFPnumbers, any normalized positive floating point number may be written this way.) If the exponent $e(a)$ is even then

$$\sqrt{a} = \sqrt{S(a)} \cdot 2^{e(a)/2}$$

Hence, we can compute $\sqrt{a}$ from this expression by computing the square root of the significand $S(a)$ and halving the exponent $e(a)$. Similarly, if $e(a)$ is odd then $e(a)-1$ is even, so $a = \{2S(a)\} \cdot 2^{e(a)-1}$ and

$$\sqrt{a} = \sqrt{2S(a)} \cdot 2^{(e(a)-1)/2}$$

Hence, we can compute $\sqrt{a}$ from this expression by computing the square root of twice the significand, $2S(a)$, and halving the adjusted exponent, $e(a)-1$.

Combining these two cases we observe that the significand is in the range

$$1 \le S(a) < 2 \le 2S(a) < 4$$

So, for all values $a > 0$ computing $\sqrt{a}$ is reduced to simply modifying the exponent and computing $\sqrt{b}$ for a value b in the interval [1, 4); here b is either $S(a)$ or it is $2S(a)$ depending on whether the exponent $e(a)$ of a is even or odd, respectively.

This process of reducing the problem of computing square roots of numbers a defined in the semi-infinite interval $(0, \infty)$ to a problem of computing square roots of numbers b defined in the (small) finite interval [1,4) is a form of **range reduction**. Range reduction is used widely in simplifying the computation of mathematical functions because, generally, it is easier (and more accurate and efficient) to find a good approximation to a function at all points in a small interval than at all points in the original, larger interval.

The Newton iteration to compute $\sqrt{b}$ is

$$X_{n+1} = \frac{\left(X_n + \dfrac{b}{X_n}\right)}{2}, \quad n = 0, 1, \ldots$$

where X_0 is chosen carefully in the interval $[1, 2)$ to give convergence in a small (known) number of iterations. On many computers a **seed table** is used to generate the initial iterate X_0. Given the value of a, the value of b =($S(a)$ or $2S(a)$) is computed, then the arithmetic unit supplies the leading bits (binary digits) of b to the seed table which returns an estimate for the square root of the number described by these leading bits. This provides the initial estimate X_0 of $\sqrt{b}$. Starting from X_0, the iteration to compute $\sqrt{b}$ converges to machine accuracy in a *known* maximum number of iterations (usually 3 or 4 iterations). The seed table is constructed so the number of iterations needed to achieve machine accuracy is fixed and known for all numbers for which each seed from the table is used as the initial estimate. Hence, there is no need to check whether the iteration has converged, resulting in some additional savings in computational time.

Problem 6.4.1. *Use the above range reduction strategy (based on powers of 2) combined with the Newton iteration for computing each of $\sqrt{5}$ and $\sqrt{9}$. [Hint: $5 = \frac{5}{4} \cdot 2^2$] (In each case compute b then start the Newton iteration for computing $\sqrt{b}$ from $X_0 = 1.0$). Using a calculator, continue the iteration until you have computed the maximum number of correct digits available. Check for (a) the predicted behavior of the iterates and (b) quadratic convergence.*

Problem 6.4.2. *Consider the function $f(x) = x^3 - a$ which has only one real root, namely $x = \sqrt[3]{a}$. Show that a simplified Newton iteration for this function may be derived as follows*

$$\begin{aligned} x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} &= x_n - \frac{1}{3}\left(\frac{x_n^3 - a}{x_n^2}\right) \\ &= \frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right) \end{aligned}$$

Problem 6.4.3. *In the cube root iteration above show that the second (algebraically simplified) expression for the cube root iteration is less expensive to compute than the first. Assume that the value of $\frac{1}{3}$ is precalculated and in the iteration a multiplication by this value is used where needed.*

Problem 6.4.4. *For $a > 0$ and the cube root iteration*

1. *Show that*
$$e_{n+1} = \frac{(2x_n + \sqrt[3]{a})}{3x_n^2} e_n^2$$
when $x_0 > 0$
2. *Hence show that* $e_1 \geq 0$ *and* $x_1 > \sqrt[3]{a}$ *and, in general, that* $x_n \geq \sqrt[3]{a}$ *for all* $n \geq 1$
3. *When* $x_n \geq \sqrt[3]{a}$ *show that*
$$0 \leq \frac{(2x_n + \sqrt[3]{a})e_n}{3x_n^2} < 1$$
4. *Hence, show that the error is reduced at each iteration and that the Newton iteration converges from above to* $\sqrt[3]{a}$ *from any initial estimate* $x_0 > 0$.

Problem 6.4.5. *We saw that* $\sqrt{x}$ *can be computed for any* $x > 0$ *by range reduction from values of* $\sqrt{X}$ *for* $X \in [1, 4)$. *Develop a similar argument that shows how* $\sqrt[3]{x}$ *can be computed for any* $x > 0$ *by range reduction from values of* $\sqrt[3]{X}$ *for* $X \in [1, 8)$. *What is the corresponding range of values for* X *for computing* $\sqrt[3]{x}$ *when* $x < 0$*?*

Problem 6.4.6. *Use the range reduction derived in the previous problem followed by a Newton iteration to compute each of the values* $\sqrt[3]{10}$ *starting the iteration from* $X_0 = 1.0$, *and* $\sqrt[3]{-20}$ *starting the iteration from* $X_0 = -1.0$.

Problem 6.4.7. *Let* $a > 0$. *Write down the roots of the function*
$$f(x) = a - \frac{1}{x^2}$$
Write down the Newton iteration for computing the positive root of $f(x) = 0$. *Show that the iteration formula can be simplified so that it does not involve divisions. Compute the arithmetic costs per iteration with and without the simplification; recall that multiplications or divisions by two (binary shifts) are less expensive than arithmetic. How would you compute* $\sqrt{a}$ *from the result of the iteration, without using divisions or using additional iterations?*

Problem 6.4.8. *Let* $a > 0$. *Write down the real root of the function*
$$f(x) = a - \frac{1}{x^3}$$
Write down the Newton iteration for computing the real root of $f(x) = 0$. *Show that the iteration formula can be simplified so that it does not involve a division at each iteration. Compute the arithmetic costs per iteration with and without the simplification; identify any parts of the computation that can be computed* a priori, *that is they can be computed just once independently of the number of iterations. How would you compute* $\sqrt[3]{a}$ *from the result of the iteration, without using divisions or using additional iterations?*

6.5 Roots of Polynomials

Specialized versions of the root–finding iterations described in previous sections may be tailored for computing the roots of a polynomial
$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$
of degree n. These methods exploit the fact that we know precisely the number of roots for a polynomial of exact degree n; namely, there are n roots counting multiplicities. In many cases of interest some, or all, of these roots are real.

In this section we concentrate on how the Newton iteration can be used for this purpose. The following issues need to be considered:

- To use a Newton iteration to find a root of a function f requires values of both $f(x)$ and its derivative $f'(x)$. For a polynomial p_n, it is easy to determine both of these values efficiently. In Section 6.5.1, we describe an algorithm called *Horner's rule* that can be used to evaluate polynomials efficiently, and, in general, accurately.
- Once a root x_* of $p_n(x)$ has been found, we may want to divide the factor $(x - x_*)$ out of $p_n(x)$ to obtain a polynomial $p_{n-1}(x)$ of degree $n-1$. We can then find another root of $p_n(x)$ by computing a root of $p_{n-1}(x)$. (If we don't remove the factor $(x - x_*)$ and we use $p_n(x)$ again then we may converge to x_* once more, and we will not know whether this is a valid root.) In Section 6.5.2, we describe how *synthetic division* can be used for this deflation process.
- Finally in Section 6.5.3 we consider what effect errors have on the computed roots, and in particular, discuss the *conditioning* of the roots of a polynomial.

6.5.1 Horner's Rule

The derivation of Horner's rule begins by noting that a polynomial p_n can be expressed as a sequence of nested multiplies. For example, a cubic polynomial

$$p_3(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

may be written as the following sequence of nested multiplies

$$p_3(x) = ((((a_3)x + a_2)x + a_1)x + a_0)\,.$$

Notice that if we use the first equation to evaluate $p_3(x)$, then we must perform two exponentiations (or equivalently additional multiplications), three multiplications, and three additions. On the other hand, if we use the second equation, then we need only three multiplications and three additions. In general, if $p_n(x)$ is a polynomial of degree n, then by using Horner's rule to evaluate $p_n(x)$ we avoid all exponentiation operations, and require only n multiplications and n additions.

To develop an algorithm for Horner's rule, it is helpful to visualize the nested multiplies as a sequence of operations:

$$\begin{array}{rcrcl} & & t_3(x) & = & a_3 \\ & & t_2(x) & = & a_2 + x \cdot t_3(x) \\ & & t_1(x) & = & a_1 + x \cdot t_2(x) \\ p_3(x) & = & t_0(x) & = & a_0 + x \cdot t_1(x) \end{array}$$

where we use $t_i(x)$ to denote "temporary" functions.

Example 6.5.1. Consider evaluating the cubic

$$\begin{array}{rcl} p_3(x) & = & 2 - 5x + 3x^2 - 7x^3 \\ & = & (2) + x \cdot \{(-5) + x \cdot [(3) + x \cdot (-7)]\} \end{array}$$

at $x = 2$. This nested multiplication form of $p_3(x)$ leads to the following sequence of operations for evaluating $p_3(x)$ at any point x:

$$\begin{array}{rcrcl} & & t_3(x) & = & (-7) \\ & & t_2(x) & = & (3) + x \cdot t_3(x) \\ & & t_1(x) & = & (-5) + x \cdot t_2(x) \\ p_3(x) & \equiv & t_0(x) & = & (2) + x \cdot t_1(x) \end{array}$$

Evaluating this sequence of operations for $x = 2$ gives

$$\begin{array}{rcrcl} & & t_3(x) & = & (-7) \\ & & t_2(x) & = & (3) + 2 \cdot (-7) = -11 \\ & & t_1(x) & = & (-5) + 2 \cdot (-11) = -27 \\ p_3(x) & \equiv & t_0(x) & = & (2) + 2 \cdot (-27) = -52 \end{array}$$

which you may check by evaluating the original cubic polynomial directly.

To efficiently evaluate the derivative $p_3'(x)$ we differentiate the sequence of operations above with respect to x to obtain

$$\begin{array}{rcll} & & t_3'(x) & = & 0 \\ & & t_2'(x) & = & t_3(x) + x \cdot t_3'(x) \\ & & t_1'(x) & = & t_2(x) + x \cdot t_2'(x) \\ p_3'(x) & \equiv & t_0'(x) & = & t_1(x) + x \cdot t_1'(x) \end{array}$$

Merging alternate lines of the derivative sequence with that for the polynomial sequence produces a further sequence for evaluating the cubic $p_3(x)$ and its derivative $p_3'(x)$ simultaneously, namely

$$\begin{array}{rcll} & & t_3'(x) & = & 0 \\ & & t_3(x) & = & a_3 \\ & & t_2'(x) & = & t_3(x) + x \cdot t_3'(x) \\ & & t_2(x) & = & a_2 + x \cdot t_3(x) \\ & & t_1'(x) & = & t_2(x) + x \cdot t_2'(x) \\ & & t_1(x) & = & a_1 + x \cdot t_2(x) \\ p_3'(x) & \equiv & t_0'(x) & = & t_1(x) + x \cdot t_1'(x) \\ p_3(x) & \equiv & t_0(x) & = & a_0 + x \cdot t_1(x) \end{array}$$

Example 6.5.2. Continuing with Example 6.5.1

$$\begin{array}{rcll} & & t_3'(x) & = & 0 \\ & & t_3(x) & = & (-7) \\ & & t_2'(x) & = & t_3(x) + x \cdot t_3'(x) \\ & & t_2(x) & = & (3) + x \cdot t_3(x) \\ & & t_1'(x) & = & t_2(x) + x \cdot t_2'(x) \\ & & t_1(x) & = & (-5) + x \cdot t_2(x) \\ p_3'(x) & \equiv & t_0'(x) & = & t_1(x) + x \cdot t_1'(x) \\ p_3(x) & \equiv & t_0(x) & = & (2) + x \cdot t_1(x) \end{array}$$

and evaluating at $x = 2$ we have

$$\begin{array}{rcll} & & t_3'(x) & = & 0 \\ & & t_3(x) & = & (-7) \\ & & t_2'(x) & = & (-7) + 2 \cdot 0 = -7 \\ & & t_2(x) & = & (3) + 2 \cdot (-7) = -11 \\ & & t_1'(x) & = & (-11) + 2 \cdot (-7) = -25 \\ & & t_1(x) & = & (-5) + 2 \cdot (-11) = -27 \\ p_3'(x) & \equiv & t_0'(x) & = & (-27) + 2 \cdot (-25) = -77 \\ p_3(x) & \equiv & t_0(x) & = & (2) + 2 \cdot (-27) = -52 \end{array}$$

Again you may check these values by evaluating the original polynomial and its derivative directly at $x = 2$.

Since the current temporary values $t_i'(x)$ and $t_i(x)$ are used to determine only the next temporary values $t_{i-1}'(x)$ and $t_{i-1}(x)$, we can write the new values $t_{i-1}'(x)$ and $t_{i-1}(x)$ over the old values $t_i'(x)$ and $t_i(x)$ as follows

$$\begin{array}{rcrcl} & & tp & = & 0 \\ & & t & = & a_3 \\ & & tp & = & t + x \cdot tp \\ & & t & = & a_2 + x \cdot t \\ & & tp & = & t + x \cdot tp \\ & & t & = & a_1 + x \cdot t \\ p_3'(x) & \equiv & tp & = & t + x \cdot tp \\ p_3(x) & \equiv & t & = & a_0 + x \cdot t \end{array}$$

For a polynomial of degree n the regular pattern of this code segment is captured in the pseudocode in Fig. 6.14. (We could program round the multiplication of the third line above since we know that the result is zero.)

Horner's Rule to evaluate $p_n(x)$ and $p'_n(x)$

Input: coefficients, a_i of $p(x)$
scalar, x

Output: $t = p_n(x)$ and $tp = p'_n(x)$

```
tp := 0
t := a_n
for i = n − 1 downto 0 do
    tp := t + x ∗ tp
    t := a_i + x ∗ t
next i
```

Figure 6.14: Pseudocode to implement Horner's rule for simultaneously evaluating $p_n(x)$ and $p'_n(x)$

Problem 6.5.1. *Use Horner's rule to compute the first two iterates x_1 and x_2 in Table 6.6.*

Problem 6.5.2. *Write out Horner's rule for evaluating a quartic polynomial*

$$p_4(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

and its derivative $p'_4(x)$. For the polynomial $p_4(x) = 3x^4 - 2x^2 + 1$, compute $p_4(-1)$ and $p'_4(-1)$ using Horner's scheme.

Problem 6.5.3. *What is the cost in arithmetic operations of a single Newton iteration when computing a root of the polynomial $p_n(x)$ using the code segment in Fig. 6.14 to compute both of the values $p_n(x)$ and $p'_n(x)$.*

Problem 6.5.4. *Modify the code segment computing the values $p_n(x)$ and $p'_n(x)$ in Fig. 6.14 so that it also computes the value of the second derivative $p''_n(x)$. (Fill in the assignments to* tpp *below.)*

```
tpp :=?
tp := 0
t := a_n
for i = n − 1 downto 0 do
    tpp :=?
    tp := t + x ∗ tp
    t := a_i + x ∗ t
next i
p''_n(x):= tpp
p'_n(x):= tp
p_n(x):= t
```

6.5.2 Synthetic Division

Consider a cubic polynomial p_3 and assume that we have an approximation α to a root x_* of $p_3(x)$. In synthetic division, we divide the polynomial

$$p_3(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

by the linear factor $x - \alpha$ to produce a quadratic polynomial

$$q_2(x) = b_3x^2 + b_2x + b_1$$

as the quotient and a constant b_0 as the remainder. The defining relation is

$$\frac{p_3(x)}{x-\alpha} = q_2(x) + \frac{b_0}{x-\alpha}$$

To determine the constant b_0, and the coefficients b_1, b_2 and b_3 in $q_2(x)$, we write this relation in the form

$$p_3(x) = (x-\alpha)q_2(x) + b_0$$

Now we substitute in this expression for q_2 and expand the product $(x-\alpha)q_2(x) + b_0$ as a sum of powers of x. As this must equal $p_3(x)$ we write the two power series together:

$$\begin{array}{rcrcrcrcr} (x-\alpha)q_2(x)+b_0 & = & (b_3)x^3 & + & (b_2-\alpha b_3)x^2 & + & (b_1-\alpha b_2)x & + & (b_0-\alpha b_1) \\ p_3(x) & = & a_3x^3 & + & a_2x^2 & + & a_1x & + & a_0 \end{array}$$

The left hand sides of these two equations must be equal for any value x, so we can equate the coefficients of the powers x^i on the right hand sides of each equation (here α is considered to be fixed). This gives

$$\begin{array}{rcl} a_3 & = & b_3 \\ a_2 & = & b_2 - \alpha b_3 \\ a_1 & = & b_1 - \alpha b_2 \\ a_0 & = & b_0 - \alpha b_1 \end{array}$$

Solving for the unknown values b_i, we have

$$\begin{array}{rcl} b_3 & = & a_3 \\ b_2 & = & a_2 + \alpha b_3 \\ b_1 & = & a_1 + \alpha b_2 \\ b_0 & = & a_0 + \alpha b_1 \end{array}$$

We observe that $t_i(\alpha) = b_i$; that is, Horner's rule and synthetic division are different interpretations of the same relation. The intermediate values $t_i(\alpha)$ determined by Horner's rule are the coefficients b_i of the polynomial quotient $q_2(x)$ that is obtained when $p_3(x)$ is divided by $(x-\alpha)$. The result b_0 is the value $t_0(\alpha) = p_3(\alpha)$ of the polynomial $p_3(x)$ when it is evaluated at the point $x = \alpha$. So, when α is a root of $p_3(x) = 0$, the value of $b_0 = 0$.

Synthetic division for a general polynomial $p_n(x)$, of degree n in x, proceeds analogously. Dividing $p_n(x)$ by $(x-\alpha)$ produces a quotient polynomial $q_{n-1}(x)$ of degree $n-1$ and a constant b_0 as remainder. So,

$$\frac{p_n(x)}{x-\alpha} = q_{n-1}(x) + \frac{b_0}{x-\alpha}$$

where

$$q_{n-1}(x) = b_nx^{n-1} + b_{n-1}x^{n-2} + \cdots + b_1$$

Since

$$p_n(x) = (x-\alpha)q_{n-1}(x) + b_0,$$

we find that

$$\begin{array}{rcrcrcccr} (x-\alpha)q_{n-1}(x)+b_0 & = & b_nx^n & + & (b_{n-1}-\alpha b_n)x^{n-1} & + & \cdots & + & (b_0-\alpha b_1) \\ p_n(x) & = & a_nx^n & + & a_{n-1}x^{n-1} & + & \cdots & + & a_0 \end{array}$$

Equating coefficients of the powers x^i,

$$\begin{array}{rcl} a_n & = & b_n \\ a_{n-1} & = & b_{n-1} - \alpha b_n \\ & \vdots & \\ a_0 & = & b_0 - \alpha b_1 \end{array}$$

Solving these equations for the constant b_0 and the unknown coefficients b_i of the polynomial $q_{n-1}(x)$ leads to

$$\begin{aligned} b_n &= a_n \\ b_{n-1} &= a_{n-1} + \alpha b_n \\ &\vdots \\ b_0 &= a_0 + \alpha b_1 \end{aligned}$$

How does the synthetic division algorithm apply to the problem of finding the roots of a polynomial? First, from its relationship to Horner's rule, for a given value of α synthetic division efficiently determines the value of the polynomial $p_n(\alpha)$ (and also of its derivative $p'_n(\alpha)$ using the technique described in the pseudocode given in Fig. 6.14). Furthermore, suppose that we have found a good approximation to the root, $\alpha \approx x_*$. When the polynomial $p_n(x)$ is evaluated at $x = \alpha$ via synthetic division the remainder $b_0 \approx p_n(x_*) = 0$. We treat the remainder b_0 as though it is precisely zero. Therefore, approximately

$$p_n(x) = (x - x_*)q_{n-1}(x)$$

where the quotient polynomial q_{n-1} has degree one less than the polynomial p_n. The polynomial q_{n-1} has as its $(n-1)$ roots values that are close to each of the remaining $(n-1)$ roots of $p_n(x) = 0$. We call q_{n-1} the *deflated polynomial* obtained by removing a linear factor $(x - \alpha)$ approximating $(x - x_*)$ from the polynomial p_n. This process therefore reduces the problem of finding the roots of $p_n(x) = 0$ to

- Find one root α of the polynomial $p_n(x)$
- Deflate the polynomial p_n to derive a polynomial of one degree lower, q_{n-1}
- Repeat this procedure with q_{n-1} replacing p_n

Of course, this process can be "recursed" to find all the real roots of a polynomial equation. If there are only real roots, eventually we reach a polynomial of degree two. Then, we may stop and use the quadratic formula to "finish off".

To use this approach to find all the real *and* all the complex roots of a polynomial equation, we must use complex arithmetic in both the Newton iteration and in the synthetic division algorithm, and we must choose starting estimates for the Newton iteration that lie in the complex plane; that is, the initial estimates x_0 must not be on the real line. Better, related approaches exist which exploit the fact that, for a polynomial with real coefficients, the complex roots arise in conjugate pairs. The two factors corresponding to each complex conjugate pair taken together correspond to an irreducible quadratic factor of the original polynomial. These alternative approaches aim to compute these irreducible quadratic factors directly, in real arithmetic, and each irreducible quadratic factor may then be factored using the quadratic formula.

Example 6.5.3. Consider the quartic polynomial

$$p_4(x) = x^4 - 5x^2 + 4$$

which has roots ± 1 and ± 2. Assume that we have calculated the root $\alpha = -2$ and that we use the synthetic division algorithm to deflate the polynomial $p_4(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ to compute the cubic polynomial $q_3(x) = b_4x^3 + b_3x^2 + b_2x + b_1$ and the constant b_0. The algorithm is

$$\begin{aligned} b_4 &= a_4 \\ b_3 &= a_3 + \alpha b_4 \\ b_2 &= a_2 + \alpha b_3 \\ b_1 &= a_1 + \alpha b_2 \\ b_0 &= a_0 + \alpha b_1 \end{aligned}$$

Substituting the values of the coefficients of the quartic polynomial p_4 and evaluating at $\alpha = -2$ we compute

$$\begin{array}{rcl} b_4 & = & (1) \\ b_3 & = & (0) + (-2)(1) = -2 \\ b_2 & = & (-5) + (-2)(-2) = -1 \\ b_1 & = & (0) + (-2)(-1) = 2 \\ b_0 & = & (4) + (-2)(2) = 0 \end{array}$$

So, $b_0 = 0$ as it should be since $\alpha = -2$ is exactly a root and we have

$$q_3(x) = (1)x^3 + (-2)x^2 + (-1)x + (2)$$

You may check that $q_3(x)$ has the remaining roots ± 1 and 2 of $p_4(x)$.

Example 6.5.4. We'll repeat the previous example but using the approximate root $\alpha = -2.001$ and working to four significant digits throughout. Substituting this value of α we compute

$$\begin{array}{rcl} b_4 & = & (1) \\ b_3 & = & (0) + (-2.001)(1) = -2.001 \\ b_2 & = & (-5) + (-2.001)(-2.001) = -0.9960 \\ b_1 & = & (0) + (-2.001)(-0.9960) = 1.993 \\ b_0 & = & (4) + (-2.001)(1.993) = 0.0120 \end{array}$$

Given that the error in α is only 1 in the fourth significant digit, the value of b_0 is somewhat larger than we might expect. If we assume that $\alpha = -2.001$ is close enough to a root, which is equivalent to assuming that $b_0 = 0$, the reduced cubic polynomial is $q_3(x) = x^3 - 2.001x^2 - 0.996x + 1.993$. Using MATLAB, to four significant digits the cubic polynomial $q_3(x)$ has roots 2.001, 0.998 and -0.998, approximating the roots 2, 1 and -1 of $p_4(x)$, respectively. These are somewhat better approximations to the true roots of the original polynomial than one might anticipate from the size of the discarded remainder b_0.

Problem 6.5.5. *The cubic polynomial $p_3(x) = x^3 - 7x + 6$ has roots 1, 2 and -3. Deflate p_3 by synthetic division to remove the root $\alpha = -3$. This will give a quadratic polynomial q_2. Check that $q_2(x)$ has the appropriate roots. Repeat this process but with an approximate root $\alpha = -3.0001$ working to five significant digits throughout. What are the roots of the resulting polynomial $q_2(x)$.*

Problem 6.5.6. *Divide the polynomial $q_{n-1}(x) = b_n x^{n-1} + b_{n-1}x^{n-2} + \cdots + b_1$ by the linear factor $(x - \alpha)$ to produce a polynomial quotient $r_{n-2}(x)$ and remainder c_1*

$$\frac{q_{n-1}(x)}{x - \alpha} = r_{n-2}(x) + \frac{c_1}{x - \alpha}$$

where $r_{n-2}(x) = c_n x^{n-2} + c_{n-1}x^{n-3} + \cdots + c_2$.

(a) Solve for the unknown coefficients c_i in terms of the values b_i.

(b) Derive a code segment, a loop, that interleaves the computations of the b_i's and c_i's.

Problem 6.5.7. *By differentiating the synthetic division relation $p_n(x) = (x - \alpha)q_{n-1}(x) + b_0$ with respect to the variable x, keeping the value of α fixed, and equating the coefficients of powers of x on both sides of the differentiated relation derive Horner's algorithm for computing the value of the derivative $p_n'(x)$ at any point $x = \alpha$.*

6.5.3 Conditioning of Polynomial Roots

So far, we have considered calculating the roots of a polynomial by repetitive iteration and deflation, and we have left the impression that both processes are simple and safe although the computation of Example 6.5.4 should give us pause for thought. Obviously there are errors when the iteration is not completed, that is the root is not computed exactly, and these errors are clearly propagated into the deflation process which itself introduces further errors into the remaining roots. The overall process is difficult to analyze but a simpler question which we can in part answer is whether the computed polynomial roots are sensitive to these errors. Particularly we might ask the closely related question "Are the polynomial roots sensitive to small changes in the coefficients of the polynomial?" That is, is polynomial root finding an inherently ill–conditioned process? In many applications one algorithmic choice is to reduce a "more complicated" computational problem to one of computing the roots of a high degree polynomial. If the resulting problem is likely to be ill–conditioned, this approach should be considered cautiously.

Consider a polynomial

$$p_n(x) = a_0 + a_1 x + \cdots + a_n x^n \,.$$

Assume that the coefficients of this polynomial are perturbed to give the polynomial

$$P_n(x) = A_0 + A_1 x + \cdots + A_n x^n \,.$$

Let z_0 be a root of $p_n(x)$ and Z_0 be the "corresponding" root of $P_n(x)$; usually, for sufficiently small perturbations of the coefficients a_i there is such a correspondence. By Taylor series

$$P_n(z_0) \approx P_n(Z_0) + (z_0 - Z_0)P_n'(Z_0)$$

and so, since $p_n(z_0) = 0$,

$$P_n(z_0) - p_n(z_0) \approx P_n(Z_0) + (z_0 - Z_0)P_n'(Z_0)\,.$$

Observing that $P_n(Z_0) = 0$, we have

$$z_0 - Z_0 \approx \frac{\sum_{i=0}^{n}(A_i - a_i)z_0^i}{P_n'(Z_0)}\,.$$

So, the error in the root can be large when $P_n'(Z_0) \approx 0$ such as occurs when Z_0 is close to a cluster of roots of $P_n(x)$. We observe that P_n is likely to have a cluster of at least m roots whenever p_n has a multiple root of multiplicity m. [Recall that at a multiple root α of $P_n(x)$ we have $P_n'(\alpha) = 0$.]

One way that the polynomial P_n might arise is essentially directly. That is, if the coefficients of the polynomial p_n are not representable as computer numbers at the working precision then they are rounded; that is, the difference $|a_i - A_i| \approx \max\{|a_i|, |A_i|\}\epsilon_{\text{WP}}$. In this case, a reasonable estimate is

$$z_0 - Z_0 \approx \frac{P_n(z_0)}{P_n'(Z_0)}\epsilon_{\text{WP}}$$

so that even small perturbations that arise in the storage of computer numbers can lead to a significantly sized error. The other way $P_n(x)$ might arise is implicitly as a consequence of *backward error analysis*. For some robust algorithms for computing roots of polynomials, we can show that the roots computed are the roots of a polynomial $P_n(x)$ whose coefficients are usually close to the coefficients of $p_n(x)$

Example 6.5.5. Consider the following polynomial of degree 12:

$$\begin{aligned} p_{12}(x) &= (x-1)(x-2)^2(x-3)(x-4)(x-5)(x-6)^2(x-7)^2(x-8)(x-9) \\ &= x^{12} - 60x^{11} + 1613x^{10} - 25644x^9 + \cdots \end{aligned}$$

If we use, for example, MATLAB (see Section 6.6) to compute the roots of the power series form of $p_{12}(x)$, we obtain the results shown in Table 6.9. Observe that each of the double roots 2, 6 and

exact root	computed root
9	9.00000000561095
8	7.99999993457306
7	7.00045189374144
7	6.99954788307491
6	6.00040397500841
6	5.99959627548739
5	5.00000003491181
4	3.99999999749464
3	3.00000000009097
2	2.00000151119609
2	1.99999848881036
1	0.999999999999976

Table 6.9: Roots of the polynomial $p_{12}(x) = (x-1)(x-2)^2(x-3)(x-4)(x-5)(x-6)^2(x-7)^2(x-8)(x-9)$. The left column shows the exact roots, and the right column shows the computed roots of the power series form, $p_{12}(x) = x^{12} - 60x^{11} + 1613x^{10} - 25644x^9 + \cdots$ as computed with MATLAB's `roots` function (see Section 6.6)
.

7 of the polynomial $p_{12}(x)$ is approximated by a pair of close real roots near 2, 6 and 7. Also note that the double roots are approximated much less accurately than are the simple roots of similar size, and the larger roots are calculated somewhat less accurately than the smaller ones.

Example 6.5.5 and the preceding analysis indicates that multiple and clustered roots are likely to be ill–conditioned, though this does not imply that isolated roots of the same polynomial are well–conditioned. The numerator in the expression for $z_0 - Z_0$ contains "small" coefficients $a_i - A_i$ by definition so, at first glance, it seems that a near-zero denominator gives rise to the only ill–conditioned case. However, this is not so; as a famous example due to Wilkinson shows, the roots of a polynomial of high degree may be ill–conditioned even when all its roots are well–spaced.

Example 6.5.6. Wilkinson devised an example of an ill–conditioned polynomial of degree 20 with the integer roots $1, 2, \ldots, 20$, that is

$$\begin{aligned} p_{20}(x) &= (x-1)(x-2)\cdots(x-20) \\ &= x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + \cdots \end{aligned}$$

To demonstrate ill–conditioning in this case, suppose we use MATLAB (see Section 6.6) to construct the power series form of $p_{20}(x)$, and then compute its roots. Some of the coefficients are sufficiently large integers that they require more digits than are available in MATLAB DP numbers; so, they are rounded. The computed roots of $p_{20}(x)$ are shown in Table 6.10.

Observe that the computed roots are all real but the least accurate differ from the true (integer) roots in the fourth decimal place (recall that MATLAB DP uses about 15 decimal digits in its arithmetic). A calculation of the residuals (that is, evaluations of the computed polynomial) using the MATLAB function `polyval` for the true (integer) roots and for the computed roots shows that both are large. (The true roots have large residuals, when they should mathematically be zero, because of the effects of rounding.) The residuals for the computed roots are about 2 orders of magnitude larger than the residuals for the true roots. This indicates that MATLAB performed satisfactorily – it was responsible for losing only about 2 of the available 15 digits of accuracy. So, the errors in the computed roots are the result mainly of the ill–conditioning of the original polynomial. [Note: The results that we give are machine dependent though you should get approximately the same values using DP arithmetic with any compiler on any computer implementing the IEEE Standard.]

exact root	computed root
20	20.0003383927958
19	18.9970274109358
18	18.0117856688823
17	16.9695256538411
16	16.0508983668794
15	14.9319189435064
14	14.0683793756331
13	12.9471623191454
12	12.0345066828504
11	10.9836103930028
10	10.0062502975018
9	8.99831804962360
8	8.00031035715436
7	6.99996705935380
6	6.00000082304303
5	5.00000023076008
4	3.99999997423112
3	3.00000000086040
2	1.99999999999934
1	0.999999999999828

Table 6.10: Roots of the polynomial $p_{20}(x)$ in Example 6.5.6. The left column shows the exact roots, and the right column shows the computed roots of the power series form of $p_{20}(x)$ as computed with MATLABś `roots` function (see Section 6.6).

Example 6.5.7. Next, we increase the degree of the polynomial to 22 giving $p_{22}(x)$ by adding the (next two) integer roots 21 and 22. Then, we follow the same computation. This time our computation, shown in Table 6.11, gives a mixture of real and complex conjugate pairs of roots with much larger errors than the previous example.

Observe that some of the large roots 21, 20, 15, 12, 11 and 10 of $p_{22}(x)$ are each computed to no more than three correct digits. Worse, each of the pairs of real roots 18 and 19, 16 and 17, and 13 and 14 of $p_{22}(x)$ become complex conjugate pairs of roots of $P_{22}(x)$. They are not even close to being real!

Problem 6.5.8. *Consider the quadratic polynomial $p_2(x) = x^2 - 2x + 1$ with double root $x = 1$. Perturb the quadratic to the form $P_2(x) = x^2 - 2(1+\epsilon)x + 1$ where ϵ is small in magnitude compared to* 1 *but may be either positive or negative. Approximately, by how much is the root of the quadratic perturbed? Does the root remain a double root? If not, do the roots remain real?*

6.6 Matlab Notes

By now our familiarity with MATLAB should lead us to suspect that there are built-in functions that can be used to compute roots, and it is not necessary to write our own implementations of the methods discussed in this chapter. It is, however, instructive to implement some of the methods; at the very least, such exercises can help to improve our programming skills. We therefore begin this section by developing implementations of fixed–point, Newton and bisection methods. We then conclude this section with a thorough discussion of the built-in MATLAB functions `roots` (to find roots of polynomials) and `fzero` (to find a root of a more general function). We emphasize that it is

exact root	computed root
22	22.0044934641580
21	20.9514187857428
20	20.1656378752978
19	18.5836448660249 + 0.441345126232861i
18	18.5836448660249 - 0.441345126232861i
17	16.3917246244130 + 0.462094344544644i
16	16.3917246244130 - 0.462094344544644i
15	15.0905963295356
14	13.4984691077096 + 0.371867167940926i
13	13.4984691077096 - 0.371867167940926i
12	11.6990708240321
11	11.1741488784148
10	9.95843441261174
9	9.00982455499210
8	7.99858075332924
7	7.00011857014703
6	5.99999925365778
5	4.99999900194002
4	4.00000010389144
3	2.99999999591073
2	2.00000000004415
1	1.00000000000012

Table 6.11: Roots of the polynomial $p_{22}(x)$ in Example 6.5.7. The left column shows the exact roots, and the right column shows the computed roots of the power series form of $p_{22}(x)$ as computed with MATLAB's `roots` function (see Section 6.6).

typically best to use one of these built-in functions, which have been thoroughly tested, rather than one of our own implementations.

6.6.1 Fixed–Point Iteration

Recall that the basic fixed–point iteration is given by

$$x_{n+1} = g(x_n), \quad n = 0, 1, 2, ...$$

The iterates, x_{n+1}, can be computed using a `for` loop. To begin the iteration, though, we need an initial guess, x_0, and we need a function, $g(x)$. If $g(x)$ is simple, then it is usually best to define it as an anonymous function, and if $g(x)$ is complicated, then it is usually best to define it as a function m-file.

Example 6.6.1. Suppose we want to compute 12 iterations of $g(x) = e^{-x}$, using the initial guess $x_0 = 0.5$ (see Table 6.1). Since $g(x)$ is a very simple function, we define it as an anonymous function, and use a `for` loop to generate the iterates:

```
g = @(x) exp(-x);
x = 0.5;
for n = 1:12
  x = g(x);
end
```

If we want to test the fixed–point iteration for several different $g(x)$, then it would be helpful to have a MATLAB program (that is, a function m-file) that works for arbitrary $g(x)$. In order to write this function, we need to consider a few issues:

- In the previous example, we chose, somewhat arbitrarily, to compute 12 iterations. For other problems this may be too many or two few iterations. To increase flexibility, we should:
  - Stop the iteration if the computed solution is sufficiently accurate. This can be done, for example, by stopping the iteration if the relative change between successive iterates is less than some specified tolerance. That is, if

    $$|x_{n+1} - x_n| \leq \text{tol} * |x_n|.$$

    This can be implemented by inserting the following conditional statement inside the `for` loop:

    ```
    if ( abs(x(n+1)-x(n)) <= tol*max(abs(x(n)), abs(x(n+1))))
      break
    end
    ```

    A value of `tol` must be either defined in the code, or specified by the user. The command `break` terminates execution of the `for` loop.
  - Allow the user to specify a maximum number of iterations, and execute the loop until the above convergence criterion is satisfied, or until the maximum number of iterations is reached.
- It might be useful to return all of the computed iterations. This can be done easily by creating a vector containing $x_0, x_1, \cdots$.

Combining these items with the basic code given in Example 6.6.1 we obtain the following function:

```
function [x, flag] = FixedPoint(g, x0, tol, nmax)
%
%     [x, flag] = FixedPoint(g, x0, tol, nmax);
%
% Find a fixed--point of g(x) using the iteration:  x(n+1) = g(x(n))
%
% Input:  g - an anonymous function or function handle,
%        x0 - an initial guess of the fixed--point,
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%         flag - set to 0 for normal exit and to 1
%                for exceeding maximum number of iterations
%
x(1) = x0; flag = 1;
for n = 1:nmax
  x(n+1) = g(x(n));
  if ( abs(x(n+1)-x(n)) <= tol*max(abs(x(n)), abs(x(n+1))))
    flag = 0;
    break
  end
end
x = x(:);
```

The final statement in the function, `x = x(:)`, is used to ensure `x` is returned as a column vector. This, of course, is not necessary, and is done only for cosmetic purposes.

Example 6.6.2. To illustrate how to use the function `FixedPoint`, consider $g(x) = e^{-x}$ with the initial guess $x_0 = 0.5$.

(a) If we want to compute exactly 12 iterations, we can set `nmax = 12` and `tol = 0`. That is, we use the statements:

```
g = @(x) exp(-x);
[x, flag] = FixedPoint(g, 0.5, 0, 12);
```

The same result could be computed using just one line of code:

```
[x, flag] = FixedPoint(@(x) exp(-x), 0.5, 0, 12);
```

Note that on exit, `flag = 1`, indicating that the maximum number of iterations has been exceeded, and that the iteration did not converge within the specified tolerance. Of course this is not surprising since we set `tol = 0`!

(b) If we want to see how many iterations it takes to reach a certain stopping tolerance, say `tol` = 10^{-8}, then we should choose a relatively large value for `nmax`. For example, we could compute

```
[x, flag] = FixedPoint(@(x) exp(-x), 0.5, 1e-8, 1000);
```

On exit, we see that `flag = 0`, and so the iteration has converged within the specified tolerance. The number of iterations is then `length(x) - 1` (recall the first entry in the vector is the initial guess). For this example, we find that it takes 31 iterations.

Problem 6.6.1. *Implement* `FixedPoint` *and use it to:*

(a) Produce a table corresponding to Table 6.1 starting from the value $x_0 = 0.6$

(b) Produce a table corresponding to Table 6.2 starting from the value $x_0 = 0.0$

(c) Produce a table corresponding to Table 6.2 starting from the value $x_0 = 2.0$

Recall that it is possible to produced formatted tables in MATLAB *using the* `sprintf` *and* `disp` *commands; see Section 1.3.5.*

Problem 6.6.2. *Implement* `FixedPoint` *and use it to compute 10 iterations using each* $g(x)$ *given in Problem 6.2.6, with* $x_0 = 1.5$.

Problem 6.6.3. *Rewrite the function* `FixedPoint` *using* `while` *instead of the combination of* `for` *and* `if`*. Note that it is necessary to implement a counter to determine when the number of iterations has reached* `nmax`.

6.6.2 Newton and Secant Methods

The Newton and secant methods are used to find roots of $f(x) = 0$. Implementation of these methods is very similar to the fixed–point iteration. For example, Newton's method can be implemented as follows:

```
function x = Newton(f, df, x0, tol, nmax)
%
%     x = Newton(f, df, x0, tol, nmax);
%
% Use Newton's method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%        df - an anonymous function of function handle for f'(x)
%        x0 - an initial guess of the root,
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%
x(1) = x0;
for n = 1:nmax
  x(n+1) = x(n) - f(x(n)) / df(x(n));
  if ( abs(x(n+1)-x(n)) <= tol*abs(x(n)) )
    break
  end
end
x = x(:);
```

Note that in Newton's method we must input two functions, $f(x)$ and $f'(x)$. It might be prudent to include a check, and to print an error message, if `df(x(n))` is zero, or smaller in magnitude than a specified value.

Example 6.6.3. Suppose we want to find a root of $f(x) = 0$ where $f(x) = x - e^{-x}$, with initial guess $x_0 = 0.5$. Then, using `Newton`, we can compute an approximation of this root using the following MATLAB statements:

```
f = @(x) x - exp(-x);
df = @(x) 1 + exp(-x);
x = Newton(f, df, 0.5, 1e-8, 10);
```

or, in one line of code, using

```
x = Newton(@(x) x - exp(-x), @(x) 1 + exp(-x), 0.5, 1e-8, 10);
```

For this problem, only 4 iterations of Newton's method are needed to converge to the specified tolerance.

Problem 6.6.4. *Using the* `Fixed Point` *code above as a template, modify* `Newton` *to check more carefully for convergence and to use a flag to indicate whether convergence has occurred before the maximum number of iterations has been exceeded. Also include in your code a check for small derivative values.*

Problem 6.6.5. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Test both codes at a simple root of a quadratic polynomial $f(x)$ that is concave up, choosing the initial iterates x_0 (and, in the case of secant, x_1) close to this root. Construct tables containing the values of the iterates, x_n and corresponding function values, $f(x_n)$. Is there any pattern to the signs of the consecutive $f(x)$ values?*

Problem 6.6.6. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Use both codes to compute the root of $f(x) = x - e^{-x}$. For Newton, use the initial guess $x_0 = 0.5$, and for secant use $x_0 = 0$ and $x_1 = 1$. Check the ratios of the errors in successive iterates as in Table 6.7 to determine if they are consistent with the convergence rates discussed in this chapter. Repeat the secant method with initial iterates $x_0 = 1$ and $x_1 = 0$.*

Problem 6.6.7. *Using the* `Newton` *code above, write a similar implementation* `Secant` *for the secant iteration. Use both methods to compute the root of $f(x) = \ln(x-3) + \sin(x) + 1$, $x > 3$. For Newton, use the initial guess $x_0 = 4$, and for secant use $x_0 = 3.25$ and $x_1 = 4$. Check the ratios of the errors in successive iterates as in Table 6.7 to determine if they are consistent with the convergence rates discussed in this chapter.*

6.6.3 Bisection Method

Recall that bisection begins with an initial bracket, $[a, b]$ containing the root, and proceeds as follows:

- First compute the midpoint, $m = (a + b)/2$.
- Determine if a root is in $[a, m]$ or $[m, b]$. If the root is in $[a, m]$, rename m as b, and continue. Similarly, if the root is in $[m, b]$, rename m as a and continue.
- Stop if the length of the interval becomes sufficiently small.

Note that we can determine if the root is in $[a, m]$ by checking to see if $f(a) \cdot f(m) \leq 0$. With these observations, an implementation of the bisection method could have the form:

```
function x = Bisection0(f, a, b, tol, nmax)
%
%     x = Bisection0(f, a, b, tol, nmax);
%
% Use bisection method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%     [a,b] - an initial bracket containing the root
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...
%
x(1) = a;   x(2) = b;
for n = 2:nmax
  m = (a+b)/2;   x(n+1) = m;
  if f(a)*f(m) < 0
    b = m;
  else
    a = m;
  end
  if ( abs(b-a) <= tol*abs(a) )
    break
  end
end
x = x(:);
```

Note that this implementation requires that we evaluate $f(x)$ twice for every iteration (to compute $f(a)$ and $f(b)$). Since this is typically the most expensive part of the method, it is a good idea to reduce the number of function evaluations as much as possible. The bisection method can be implemented so that only one function evaluation is needed at each iteration. In the algorithm below we have taken care to keep track of whether the maximum number of iterations has been exceeded and to check for zeros and avoid underflows in our treatment of the function values generated during the iteration.

Problem 6.6.8. *Use the* `Bisection` *code to compute a root of* $f(x) = x - e^{-x}$. *Use the initial bracket* $[0, 1]$. *Produce a table of results similar to Table 6.8.*

Problem 6.6.9. *Using the* `Bisection`, `Secant` *and* `Newton` *codes, experimentally compare the convergence behavior of each method for the function* $f(x) = x - e^{-x}$.

```
function [x, flag] = Bisection(f, a, b, tol, nmax)
%
%     [x, flag]  = Bisection(f, a, b, tol, nmax);
%
% Use bisection method to find a root of f(x) = 0.
%
% Input:  f - an anonymous function or function handle for f(x)
%     [a,b] - an initial bracket containing the root
%       tol - a (relative) stopping tolerance, and
%      nmax - specifies maximum number of iterations.
%
% Output: x - a vector containing the iterates, x0, x1, ...,
%             with last iterate the final approximation to the root
%         flag - set to 0 for normal exit and to 1
%                for exceeding maximum number of iterations
%
x(1) = a;  x(2) = b;  fa = f(a);  fb = f(b);  flag = 1;
if (sign(fa) ~= 0 & sign(fb) ~= 0)
  for n = 2:nmax
    c = (a+b)/2;  x(n+1) = c;  fc = f(c);
    if sign(fc) == 0
      flag = 0;
      break
    end
    if sign(fa) ~= sign(fc)
      b = c;  fb = fc;
    else
      a = c;  fa = fc;
    end
    if (abs(b-a) <= tol*max(abs(a), abs(b)))
      flag = 0;
      break
    end
  end
else
  if sign(fa) == 0
    x(2) = a; x(1) = b;
  end
  flag = 0;
end
x = x(:);
```

6.6.4 The `roots` and `fzero` Functions

As previously mentioned, MATLAB provides two built-in root finding methods:

`fzero`	finds a root of $f(x) = 0$, where $f(x)$ is a general function of one variable
`roots`	finds all roots of $p(x) = 0$, where $p(x)$ is a polynomial

In this section we discuss in more detail how to use these built-in MATLAB functions.

Computing a solution of $f(x) = 0$.

The MATLAB function `fzero` is used to find zeros of a general function of one variable. The implementation uses a combination of bisection, secant and inverse quadratic interpolation. The basic calling syntax for `fzero` is:

```
x = fzero(fun, x0)
```

where `fun` can be an inline or anonymous function, or a handle to a function M–file. The initial guess, `x0` can be a scalar (that is, a single initial guess), or it can be a bracket (that is, a vector containing two values) on the root. Note that if `x0` is a single initial guess, then `fzero` first attempts to find a bracket by sampling on successively wider intervals containing `x0` until a bracket is determined. If a bracket for the root is known, then it is usually best to supply it, rather than a single initial guess.

Example 6.6.4. Consider the function $f(x) = x - e^{-x}$. Because $f(0) < 0$ and $f(1) > 0$, a bracket for a root of $f(x) = 0$ is $[0, 1]$. This root can be found as follows:

```
x = fzero(@(x) x-exp(-x), [0, 1])
```

We could have used a single initial guess, such as:

```
x = fzero(@(x) x-exp(-x), 0.5)
```

but as mentioned above, it is usually best to provide a bracket if one is known.

Problem 6.6.10. *Use* `fzero` *to find a root of $f(x) = 0$, with $f(x) = \sin x$. For initial guess, use $x_0 = 1$, $x_0 = 5$, and the bracket $[1, 5]$. Explain why a different root is computed in each case.*

Problem 6.6.11. *Consider the function $f(x) = \frac{1}{x} - 1$, and note that $f(1) = 0$. Thus, we might expect that $x_0 = 0.5$ is a reasonable initial guess for* `fzero` *to compute a solution of $f(x) = 0$. Try this, and see what* `fzero` *computes. Can you explain what happens here? Now use the initial guess to be the bracket $[0.5, 1.5]$. What does* `fzero` *compute in this case?*

If a root, x_*, has multiplicity greater than one, then $f'(x_*) = 0$, and hence the root is not simple. `fzero` can find roots which have odd multiplicity, but it cannot find roots of even multiplicity, as illustrated in the following example.

Example 6.6.5. Consider the function $f(x) = x^2 e^x$, which has a root $x_* = 0$ of multiplicity 2.

(a) If we use the initial guess $x_0 = 1$, then

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, 1)
```

computes $x \approx -925.8190$. At first this result may seem completely ridiculous, but if we compute

```
f(xstar)
```

the result is 0. Notice that $\lim_{x \to -\infty} x^2 e^x = 0$, thus it appears that `fzero` "finds" $x \approx -\infty$.

(b) If we try the initial guess $x_0 = -1$, then

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, -1)
```

then `fzero` fails because it cannot find an interval containing a sign change.

(c) We know the root is $x_* = 0$, so we might try the bracket $[-1, 1]$. However, if we compute

```
f = @(x) x.*x.*exp(x);
xstar = fzero(f, [-1, 1])
```

an error occurs because the initial bracket does not satisfy the sign change condition, $f(-1)f(1) < 0$.

Problem 6.6.12. *The functions $f(x) = x^2 - 4x + 4$ and $f(x) = x^3 - 6x^2 + 12x - 8$ satisfy $f(2) = 0$. Attempt to find this root using* `fzero`*. Experiment with a variety of initial guesses. Why does* `fzero` *have trouble finding the root for one of the functions, but not for the other?*

It is possible to reset certain default parameters used by `fzero` (such as stopping tolerance and maximum number of iterations), and it is possible to obtain more information on exactly what is done during the computation of the root (such as number of iterations and number of function evaluations) by using the calling syntax

```
[x, fval, exitflag, output] = fzero(fun, x0, options)
```

where

- `x` is the computed approximation of the root.
- `fval` is the function value of the computed root, $f(x_*)$. Note that if a root is found, then this value should be close to zero.
- `exitflag` provides information about what condition caused `fzero` to terminate; see `doc fzero` for more information.
- `output` is a structure array that contains information such as which algorithms were used (e.g., bisection, interpolation, secant), number of function evaluations and number of iterations.
- `options` is an input parameter that can be used, for example, to reset the stopping tolerance, and to request that results of each iteration be displayed in the command window. The options are set using the built-in MATLAB function `optimset`.

The following examples illustrate how to use `options` and `output`.

Example 6.6.6. Consider $f(x) = x - e^{-x}$. We know there is a root in the interval $[0, 1]$. Suppose we want to investigate how the cost of computing the root is affected by the choice of the initial guess. Since the cost is reflected in the number of function evaluations, we use `output` for this purpose.

(a) Using the initial guess $x_0 = 0$, we can compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), 0);
output
```

we see that `output` displays the information:

```
 intervaliterations: 10
         iterations: 6
          funcCount: 27
          algorithm: 'bisection, interpolation'
            message: 'Zero found in the interval [-0.64, 0.64]'
```

The important quantity here is `funcCount`, which indicates that a total of 27 function evaluations was needed to compute the root.

(b) On the other hand, if we use $x_0 = 0.5$, and compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), 0.5);
output
```

we see that 18 function evaluations are needed.

(c) Finally, if we use the bracket $[0, 1]$, and compute:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1]);
output
```

we see that only 8 function evaluations are needed.

Notice from this example that when we provide a bracket, the term `interval iterations`, which is returned by the structure `output`, is 0. This is because `fzero` does not need to do an initial search for a bracket, and thus the total number of function evaluations is substantially reduced.

Example 6.6.7. Again, consider $f(x) = x - e^{-x}$, which has a root in $[0, 1]$. Suppose we want to investigate how the cost of computing the root is affected by the choice of stopping tolerance on x. To investigate this, we must first use the MATLAB function `optimset` to define the structure `options`.

(a) Suppose we want to set the tolerance to 10^{-4}. Then we can use the command:

```
options = optimset('TolX', 1e-4);
```

If we then execute the commands:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1], options);
output
```

we see that only 5 function evaluations are needed.

(b) To increase the tolerance to 10^{-8}, we can use the command:

```
options = optimset('TolX', 1e-8);
```

Executing the commands:

```
[x, fval, exitflag, output] = fzero(@(x) x-exp(-x), [0, 1], options);
output
```

shows that 7 function evaluations are needed to compute the approximation of the root.

Example 6.6.8. Suppose we want to display results of each iteration when we compute an approximation of the root of $f(x) = x - e^{-x}$. For this, we use `optimset` as follows:

```
options = optimiset('Display', 'iter');
```

If we then use `fzero` as:

```
x = fzero(@(x) x-exp(-x), [0, 1], options);
```

then the following information is displayed in the MATLAB command window:

```
 Func-count    x          f(x)             Procedure
    2              1      0.632121         initial
    3         0.6127      0.0708139        interpolation
    4        0.56707   -0.000115417        interpolation
    5       0.567144    9.44811e-07        interpolation
    6       0.567143    1.25912e-11        interpolation
    7       0.567143   -1.11022e-16        interpolation
    8       0.567143   -1.11022e-16        interpolation

Zero found in the interval [0, 1]
```

Notice that a total of 8 function evaluations were needed, which matches (as it should) the number found in part (c) of Example 6.6.6.

Problem 6.6.13. *Note that we can use* `optimset` *to reset* `TolX` *and to request display of the iterations in one command, such as:*

```
options = optimset('TolX', 1e-7, 'Display', 'iter');
```

Use this set of options with $f(x) = x - e^{-x}$*, and various initial guesses (e.g., 0, 1 and [0,1]) in* `fzero` *and comment on the computed results.*

Roots of polynomials

The MATLAB function `fzero` cannot, in general, be effectively used to compute roots of polynomials. For example, it cannot compute the roots of $f(x) = x^2$ because $x = 0$ is a root of even multiplicity. However, there is a special purpose method, called `roots`, that can be used to compute all roots of a polynomial.

Recall from our discussion of polynomial interpolation (section 4.5) MATLAB assumes polynomials are written in the canonical form

$$p(x) = a_1 x^n + a_2 x_{n-1} + \cdots + a_n x + a_{n+1},$$

and that they are represented with a vector (either row or column) containing the coefficients:

$$a = \begin{bmatrix} a_1 & a_2 & \cdots & a_n & a_{n+1} \end{bmatrix}.$$

If such a vector is defined, then the roots or $p(x)$ can be computed using the built-in function `roots`. Because it is beyond the scope of this book, we have not discussed the basic algorithm used by `roots` (which computes the eigenvalues of a *companion matrix* defined by the coefficients of $p(x)$), but it is a very simple function to use. In particular, execution of the statement

```
r = roots(a);
```

produces a (column) vector `r` containing the roots of the polynomial defined by the coefficients in the vector `a`.

Example 6.6.9. Consider $x^4 - 5x^2 + 4 = 0$. The following MATLAB commands:

```
a = [1 0 -5 0 4];
r = roots(a)
```

compute a vector `r` containing the roots 2, 1, -2 and -1.

The MATLAB function `poly` can be used to create the coefficients of a polynomial with given roots. That is, if `r` is a vector containing the roots of a polynomial, then

```
a = poly(r);
```

constructs a vector containing the coefficients of a polynomial whose roots are given by the entries in the vector `r`. Basically, `poly` multiplies out the factored form

$$p(x) = (x - r_1)(x - r_2) \cdots (x - r_n)$$

to obtain the canonical power series form

$$p(x) = x^n + a_2 x^{n-1} + \cdots + a_n x + a_{n+1}.$$

Example 6.6.10. The roots of $(2x + 1)(x - 3)(x + 7) = 0$ are obviously $-\frac{1}{2}, 3$ and -7. Executing the MATLAB statements

```
r = [-1/2, 3, -7];
a = poly(r)
```

constructs the vector `a = [1, 4.5, -19, -10.5]`.

Problem 6.6.14. *Consider the polynomial given in Example 6.5.5. Use the* MATLAB *function* `poly` *to construct the coefficients of the power series form of* $p_{12}(x)$*. Then compute the roots using the* MATLAB *function* `roots`*. Use the* `format` *command so that computed results display 15 digits.*

Problem 6.6.15. *Consider the polynomial* $p_{20}(x)$ *given in Example 6.5.6. Compute the results shown in that example using the* MATLAB *functions* `poly` *and* `roots`*. Use the* `format` *command so that computed results display 15 digits.*

Problem 6.6.16. *Consider the polynomial* $p_{22}(x)$ *given in Example 6.5.7. Compute the results shown in that example using the* MATLAB *functions* `poly` *and* `roots`*. Use the* `format` *command so that computed results display 15 digits.*

Chapter 7

Univariate Minimization

A major problem in scientific computing is optimization. In many real life problems we need to maximize or minimize a quantity over choices of a number of variables, possibly with some additional constraints on the variables. For example, we may want to maximize a profit function that depends on cost and production variables, with the constraint that cost must be positive. Here, we consider the simplest such problem, the minimization of a function of a single variable without side constraints. The methods we describe can simply be adapted for maximization problems by changing the sign of the function.

7.1 Introduction

Recall from Calculus that when you are set the problem of determining a minimum of a continuously differentiable function f on a closed interval $[a, b]$, first you determine the derivative function f'. Then you find all the critical values x in $[a, b]$ where $f'(x) = 0$, say $x_1, x_2, \ldots, x_p$. Finally, you determine the global minimum $\min_{x\in[a,b]} f(x) = \min\{f(a), f(x_1), f(x_2), \ldots, f(x_p), f(b)\}$. There is no need to check whether the critical values $x_1, x_2, \ldots, x_p$ correspond to local minima as we determine the global minimum by enumeration.

Example 7.1.1. To find the minimum of the continuously differentiable function $f(x) = -e^{-x}\sin(x)$ on the closed interval $[0, 1.5]$, first we compute the derivative $f'(x) = -e^{-x}(-\sin(x) + \cos(x))$. Next, we find the critical values x where $f'(x) = 0$. Since e^{-x} is never zero, we must have $-\sin(x)+\cos(x) = 0$; that is, we need the values x such that $\tan(x) = 1$ since $\cos(x) \neq 0$ on $[0, 1.5]$. On $[0, 1.5]$ this equation is satisfied at just one point $x = \frac{\pi}{4}$. Now,

$$\min\left\{f(0), f\left(\frac{\pi}{4}\right), f(1.5)\right\} = \min\{0, -0.322396941945, -0.222571216108\} = -0.322396941945$$

is the global minimum located at $x = \frac{\pi}{4}$.

This approach suggests that we may find the extrema of $f(x)$ by computing the zeros of $f'(x)$. We could find individual extrema by solving the equation $f'(x) = 0$ by using, say Newton's method which would also involve computing $f''(x)$ for use in the iteration

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

or if the second derivative is difficult to obtain by using the secant method for solving $f'(x) = 0$.

In computational practice it is usually infeasible to find a global minimum, unless we know in advance how many local extrema $f(x)$ has. So, here we concentrate on finding just one local

extremum, a local minimum. Also, we begin with methods that use only values of $f(x)$ in finding the local minimum then move to methods which combine using values of $f'(x)$ with values of $f(x)$.

We consider how to determine the location x_m of a minimum of the function f on the interval $[a, b]$. We say that $[a, b]$ is a **bracket for the minimum** of f if x_m is in $[a, b]$. Throughout, we assume that the function f is **U–shaped** on the interval $[a, b]$; i.e., it is continuous, *strictly decreasing* on $[a, x_m)$, and *strictly increasing* on $(x_m, b]$. When the function f has a continuous derivative, then for f to be U–shaped it is sufficient that the derivative f' be negative on $[a, x_m)$ and positive on $(x_m, b]$. When the function f has a continuous second derivative, then for f to be U–shaped it is sufficient that f be concave up on the interval $[a, b]$ and $f'(x_m) = 0$. Of course, we may assume that a function is U–shaped when in reality it is not – usually we have no way to check. So, occasionally we will discuss what happens to our algorithms if the underlying assumption of U–shapedness turns out not to be correct.

Let the interval $[a, b]$ be a bracket for the minimum of a U–shaped function f. The basic process used to refine such a bracket, i.e., to make it shorter, uses **search points** c and d where $a < c < d < b$. Given such search points, it follows that (see Figure 7.1)

1. if $f(c) \leq f(d)$, then $[a, d]$ is a bracket for the minimum, and
2. if $f(c) \geq f(d)$, then $[c, b]$ is a bracket for the minimum.

To prove that 1 is true, suppose that both $f(c) \leq f(d)$ and $x_m > d$. Now $x_m > d$ implies that both of the points c and d lie to the left of the point x_m where the function f is strictly decreasing, so $f(c) > f(d)$. But this contradicts the supposition that $f(c) \leq f(d)$, so both of $f(c) \leq f(d)$ and $x_m > d$ cannot be true. Therefore, if $f(c) \leq f(d)$ is true, then $x_m > d$ must be false, i.e., the point x_m lies in the interval $[a, d]$. So, the statement in Part 1 is true. The proof of 2 is similar; see Problem 7.1.2.

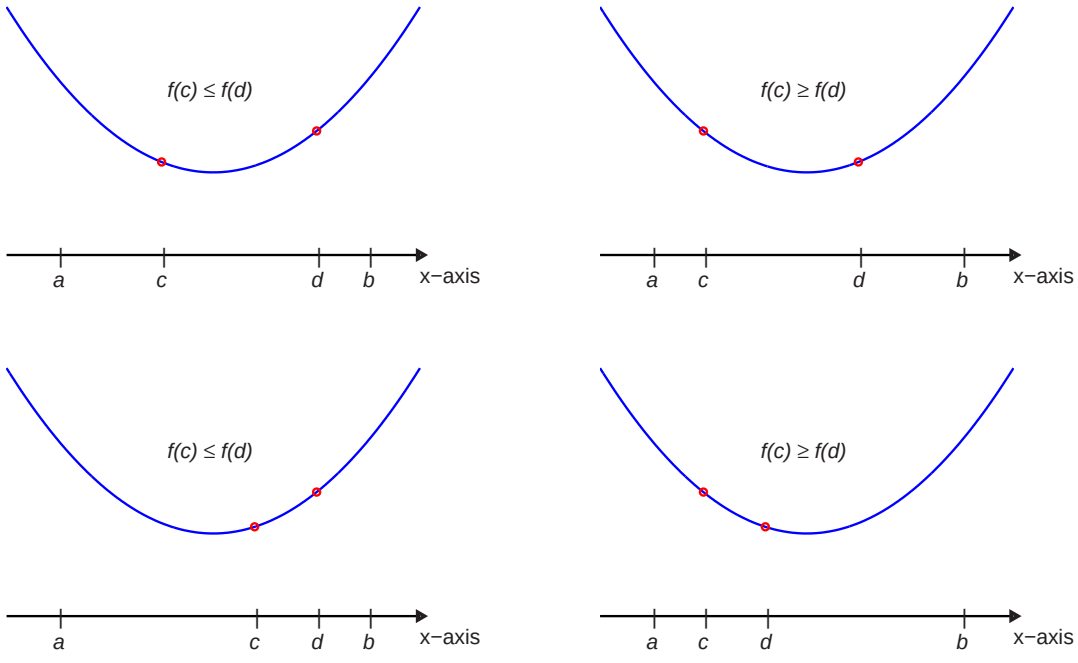

Figure 7.1: Illustration of bracket refining to find a minimum of a function. The plots on the left illustrate that if $f(c) \leq f(d)$, then $[a, d]$ is a bracket for the minimum. The plots on the right illustrate that if $f(c) \geq f(d)$, then $[c, b]$ is a bracket for the minimum.

Unlike when refining a bracket for a root, refining a bracket for the minimum generally requires the function to be evaluated at two distinct points inside the bracket. To understand why, let

$[a, b]$ be a bracket for the minimum of the U–shaped function f, and let c be some point in the open interval (a, b). There is no decision rule that compares the values $f(a)$, $f(b)$ and $f(c)$ and determines correctly which of the intervals, either $[a, c]$ or $[c, b]$, is a bracket for the minimum. To see why, pick three values v_a, v_b and v_c for which $v_c < \min\{v_a, v_b\}$ and plot the points (a, v_a), (b, v_b) and (c, v_c). Consider Figure 7.2, which shows the graphs of the values of two U–shaped functions $f_1(x)$ and $f_2(x)$ that pass through these three points but the minimum of $f_1(x)$ is located to the right of the point c and the minimum of $f_2(x)$ is located to the left of c. Observe that v_a, v_b and v_c are the common values assumed by these two functions at the points a, b and c, respectively. If the decision rule compares the values v_a, v_b and v_c and declares the interval $[c, b]$ to be the refined bracket, then this would be correct for the U–shaped function f_1 but incorrect for the U–shaped function f_2. On the other hand, if the decision rule compares the values v_a, v_b and v_c and declares the interval $[a, c]$ to be the refined bracket, then it would be correct for the U–shaped function f_2 but incorrect for the U–shaped function f_1. The conclusion is that, generally, a decision rule that refines a bracket for the minimum of a U–shaped function f must evaluate the function at more than one distinct point inside the open interval (a, b).

Say we evaluate f at an interior point c and we find that $f(a) < f(c) < f(b)$. This does not violate the assumption that f is U–shaped; it simply tells us that if there is a unique local minimum it must lie in the interval $[a, c]$. However, if $c \in (a, b)$ and $f(c) \geq \max\{f(a), f(b)\}$ then the assumption that f is U–shaped is violated. Our methods below will involve using two interior points at each iteration. In this case there are other ways that U-shapedness can be violated (and which should be checked for in the algorithm). For example, let $a < c < d < b$ and $f(a) < f(c) < f(d) < f(b)$. This is possible when f is U–shaped but if $f(a) < f(c) > f(d) < f(b)$ then f cannot be U–shaped.

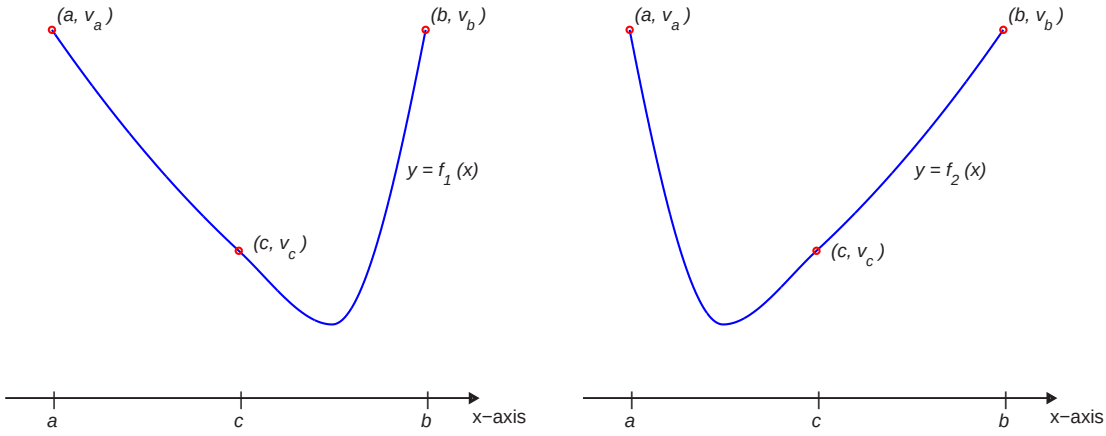

Figure 7.2: The plots in this figure illustrate that if the function is evaluated at only one point, c, in the interval $[a, b]$, then it may not be possible to determine whether the minimum of the function is to the left or to the right of c.

Problem 7.1.1. *Show that $x = \frac{\pi}{4}$ is a local minimum of $f(x) = -e^{-x}\sin(x)$. Find a local maximum of $f(x)$.*

Problem 7.1.2. *Prove the statement presented in Part 2. Hint: Suppose that both $f(c) \geq f(d)$ and $x_m < c$ are true. Describe the contradiction obtained from this supposition. Conclude that if $f(c) \geq f(d)$ is true, then $x_m < c$ must be false; i.e., the point x_m lies in the interval $[c, b]$.*

Problem 7.1.3. *Let $f(x)$ be a U–shaped function defined on the interval $[a, b]$. For every choice of the point $c \in (a, b)$, show that $f(c) < \max\{f(a), f(b)\}$. Hint: Show that the function f cannot be U–shaped on the interval $[a, b]$ if there is a value of $c \in (a, b)$ for which $f(c) \geq \max\{f(a), f(b)\}$.*

Problem 7.1.4. *Let the function f be U–shaped on the interval $[a, b]$, and let c be some point in the open interval (a, b) for which $f(c) \geq \min\{f(a), f(b)\}$. If $f(c) \geq f(a)$, then show that the interval $[a, c]$ is a bracket for the minimum. Similarly, if $f(c) \geq f(b)$, then show that the interval $[c, b]$ is a bracket for the minimum.*

Problem 7.1.5. *Pick three values v_a, v_b and v_c such that $v_c < \min\{v_a, v_b\}$ and plot the points (a, v_a), (c, v_c) and (b, v_b). Sketch the graph of two piecewise linear (V-shaped) functions that pass through these three points and yet assume their minimum value at points on opposite sides of the point c. Argue why these piecewise linear functions are U–shaped functions.*

7.2 Search Methods not Involving Derivatives

Here, we describe methods which only use values of $f(x)$ in the search for the location of its local minimum. In each case the methods proceed by reducing the length of a bracket on the location. The first method makes no use of the absolute size of the function in determining the iterates. The second uses interpolation formulas and hence implicitly incorporates the function's size.

7.2.1 Golden Section Search

The golden section search for the minimum of a U–shaped function f is analogous to the bisection method for finding a root of $f(x)$. Golden section is characterized by how the search points c and d are chosen. If $[a, b]$ is a bracket for the minimum of the function f, then its associated **golden section search points** c and d are

$$\begin{aligned} c &\equiv a + r(b - a) \\ d &\equiv b - r(b - a) \end{aligned}$$

where

$$r \equiv \frac{3 - \sqrt{5}}{2} \approx 0.382\ldots$$

Because $r < 1$, c is the golden section search point nearest the endpoint a and we say that a and c are *associated*. Similarly, we say that the points d and b are associated. Observe that the endpoints of each of the possible refined brackets, that is $[a, d]$ or $[c, b]$, consist of *one* golden section search point and the endpoint associated with the *other* golden section search point.

The name golden section search is derived from the fact that the length of the original bracket divided by the length of either possible refined bracket is

$$\frac{1}{1 - r} = \frac{1 + \sqrt{5}}{2} \approx 1.618,$$

a value known in antiquity as the **golden section** constant.

Choosing c and d to be the golden section search points has two advantages. First, each refinement step of golden section search reduces the length of the bracket by a factor of $1 - r \approx 0.618$. Second, the two golden section search points for the original bracket occur in the refined bracket one as an endpoint and the other as an associated golden section search point. This is illustrated in Table 7.1 and Figure 7.3 for a bracket and its two possible refinements. Note how the golden section search points of the original bracket appear in the refined bracket. One is an endpoint and the other is that endpoint's associated golden section search point. As a consequence, the value $f(x)$ at the golden section search point common to both the original and refined brackets can and should be reused in the next refinement step!

Bracket, $[a_i, b_i]$		Left Endpoint a_i	Left GSSP c_i	Right GSSP d_i	Right Endpoint b_i
Original Bracket	$[0, 1]$	0	0.382	0.618	1
Left Refinement	$[0, 0.618]$	0	0.236	0.382	0.618
Right Refinement	$[0.382, 1]$	0.382	0.618	0.764	1

Table 7.1: Endpoints and golden section search points (GSSP). The left refinement case is illustrated in the left part of Figure 7.3, and the right refinement is illustrated in the right part of Figure 7.3

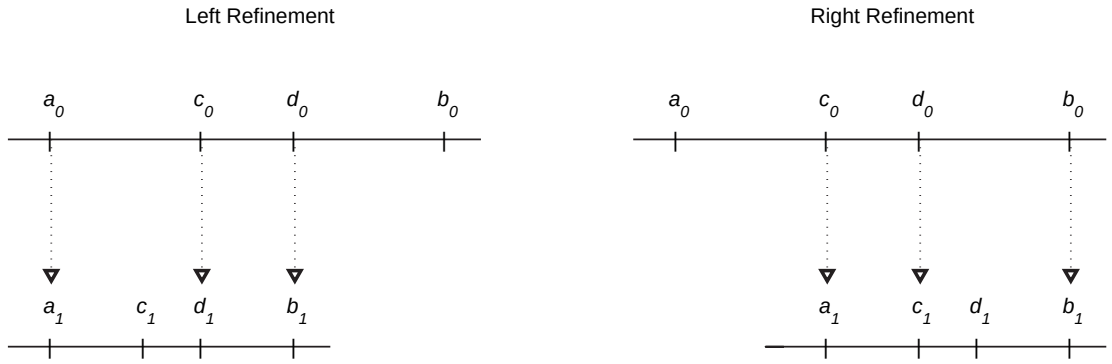

Figure 7.3: The figure illustrates the two possibilities for refining a bracket in the golden section search method. Notice that only one new point needs to be computed for each refinement.

Figure 7.4 displays a pseudocode for golden section search; it assumes that the initial points a and b as well the function $f(x)$ are provided. At the top of the while-loop the only points known are the endpoints a, b and the golden section search point c associated with a. For refinement, the code computes the golden section search point d associated with b, via the technique in Problem 7.2.3, then the if-statement identifies the refined bracket. This refined bracket consists of the interval between the golden section search point with the largest value of $f(x)$ and the endpoint associated with the other golden section search point. The body of the if-statement applies the label c to the golden section search point that has the largest value $f(x)$, fc to its associated function value, a to the endpoint associated with c, and b to the other endpoint. Note that the computation of the point d and the refinement of the bracket work equally well whether $a < b$ or $a > b$. This pseudocode for golden section search incurs a startup cost of one function evaluation, and then a cost of one function evaluation for each refinement step. Generally, in costing the algorithm the startup cost is either ignored or is considered to be amortized over the cost of all the function evaluations used in the later refinement steps. So, each step of golden section search for the minimum reduces the length of the bracket for the minimum by a factor of about 0.618 at a cost of about one function evaluation. Error detection should be included in an algorithm for golden section search. For example, if it turns out that $f(d) \geq \max\{f(a), f(b)\}$, then it should be reported that the function f is not U–shaped so there may be more than one local extremum in the interval $[a, b]$. However, the algorithm could be permitted to continue searching for a local minimum with the starting bracket $[d, b]$ and interior point c; note, the pseudocode terminates in these circumstances.

What termination criteria should we use? Suppose the point x lies in a bracket $[a, b]$ for the minimum of $f(x)$. This implies that $|x - x_m| \leq |b - a|$. So, a common practice is to stop the search when the bracket is sufficiently short, that is, when

$$|b - a| \leq \sqrt{\frac{\epsilon}{L}}$$

Golden Section Search

Input: function to minimize, $f(x)$
initial bracket, $[a, b]$

Output: scalars c, fc, where $fc = f(c) \approx \min_{a \le x \le b} f(x)$

$r = (3 - \sqrt{5})/2$
$c = a + r(b - a)$; $fc = f(c)$
if $fc > fa$ or $fc > fb$ then Stop
while termination criteria not satisfied do
  $d = c + r(b - c)$; $fd = f(d)$
  if $fc \le fd$ then
    $b = a$; $fb = fa$; $a = d$; $fa = fd$
  else
    $a = c$; $fc = fa$; $c = d$; $fc = fd$
  endif
  if $fc > fa$ or $fc > fb$ then Stop
end while

Figure 7.4: Pseudocode *Golden Section Search*

where ϵ is the working precision machine epsilon and L is a positive constant which we determine below. This termination criterion can be justified as follows. Rearrange this inequality so that it reads

$$L(b - a)^2 \le \epsilon$$

The left hand side of this inequality provides an upper bound on the absolute relative error in accepting $f(x)$ as the value of $f(x_m)$ while the right hand side provides an upper bound on the absolute relative error in the computed value $F(x)$ of $f(x)$. So, this termination criterion states that the search be stopped when the difference between the values $f(x)$ and $f(x_m)$ could be attributed to rounding error in the computed value of $f(x_m)$. There are two reasons why the value $F(x)$ differs from the value $f(x_m)$. The first reason is that $f(x)$ is computed using floating point arithmetic. So, the best we can expect is that $F(x)$ is the rounded value of $f(x)$ from which it follows that

$$\left| \frac{F(x) - f(x)}{f(x)} \right| \le \epsilon$$

The second reason is that the point x differs from x_m. If f has two continuous derivatives, then since $f'(x_m) = 0$, Taylor's series states that as $x \to x_m$,

$$f(x) \approx f(x_m) + \frac{f''(x_m)}{2}(x - x_m)^2 = f(x_m)\,(1 + \psi)$$

where

$$\psi = \frac{f''(x_m)}{2f(x_m)}(x - x_m)^2$$

So, the relative error in accepting the value $f(x)$ as an approximation of $f(x_m)$ satisfies the bound

$$\left| \frac{f(x) - f(x_m)}{f(x_m)} \right| \lessapprox |\psi|$$

Recall that $|x - x_m| \le |b - a|$, so

$$|\psi| \le \left| \frac{f''(x_m)}{2f(x_m)} \right| (b - a)^2 = L(b - a)^2$$

where we have identified

$$L = \left| \frac{f''(x_m)}{2f(x_m)} \right|$$

In terms of the values ψ and ϵ, the proposed termination criterion becomes $|\psi| \leq \epsilon$. In words, the error in accepting the value $f(x)$ as the value of $f(x_m)$ is no larger than an amount that could be attributed to the rounding error in the computed value of $f(x_m)$. Of course, in practice we don't know the value of L, so we choose a value such as $L = 2$, hoping it won't turn out to be so much too small that it prevents termination of the algorithm. It is possible to estimate L using finite differences or by differentiating polynomials interpolating to the most recently calculated values of $f(x)$ to approximate $f''(x_m)$ but the necessary information is not readily available until we have a good approximation to x_m.

This termination criterion also has an important consequence. If we switch from single to double precision arithmetic, we should expect to determine x_m only to just over 4 additional decimal digits of accuracy. This follows because $\sqrt{\epsilon_{\text{SP}}} \approx 10^{-3.5}$ and $\sqrt{\epsilon_{\text{DP}}} \approx 10^{-8}$, so the switch in precision causes the square root of the working precision machine epsilon to be reduced by only a factor of $10^{-4.5}$.

Example 7.2.1. We find a local minimum of $f(x) = -e^{-x}\sin(x)$ on the interval $[0, 1.5]$ using golden section search. We terminate when the bracket is smaller than 10^{-5}. We implemented the pseudocode in Figure 7.4 in DP and terminated when the bracket was smaller than 10^{-5}; results are in Table 7.2. Observe the irregular behavior of the error, somewhat reminiscent of the behavior of the error for the bisection method in root finding.

iterate	$f(iterate)$	*error*
0.000000	0.000000	-0.785398
1.500000	-0.222571	0.714602
0.572949	-0.305676	-0.212449
0.927051	-0.316517	0.141653
1.145898	-0.289667	0.360500
0.791796	-0.322384	0.006398
0.708204	-0.320375	-0.077194
0.843459	-0.321352	0.058061
0.759867	-0.322183	-0.025531
0.811529	-0.322181	0.026131
0.779600	-0.322386	-0.005798
0.772063	-0.322339	-0.013336
0.784259	-0.322397	-0.001140
0.787138	-0.322396	0.001739
0.782479	-0.322394	-0.002919
0.785358	-0.322397	-0.000040
0.786038	-0.322397	0.000640
0.784938	-0.322397	-0.000460
0.785618	-0.322397	0.000220
0.785198	-0.322397	-0.000200
0.785457	-0.322397	0.000059
0.785297	-0.322397	-0.000101
0.785396	-0.322397	-0.000002

Table 7.2: Golden section – iterates, function values and errors in iterates

Problem 7.2.1. *Show that the length of each of the two possible refined brackets,* $[a, d]$ *and* $[c, b]$*, is* $(1 - r)(b - a)$*. What is the numerical value of* $1 - r$*?*

Problem 7.2.2. *Consider the formulas for the golden section search points c and d. Show that if we swap the values a and b, then the values c and d are swapped too, i.e., the formula for c after the swap is the same as the formula for d before the swap, and vice versa. Remark: The conclusion is that, regardless of whether $a < b$ or $b < a$, the formula for c determines the golden section search point associated with a, and the formula for d determines the golden section search point associated with b.*

Problem 7.2.3. *Let the points a and b be given, and let c and d be their associated golden section search points. Show that*

$$d = c + r(b - c)$$

Computing the point d this way has the advantage that d surely lies between c and b.

Problem 7.2.4. *Show that r is a root of the quadratic equation $r^2 - 3r + 1 = 0$. Algebraically rearrange this equation to show that (a) $(1-r)^2 = r$, (b) $r(2-r) = 1-r$, and (c) $1-2r = r(1-r)$.*

Problem 7.2.5. *Given the initial bracket for the minimum is $[0, 1]$, show explicitly the computation of the remaining numbers in Table 7.1.*

Problem 7.2.6. *Construct a table similar to Table 7.1 for a general initial bracket $[a, b]$.*

7.2.2 Quadratic Interpolation Search

For brevity, we use the phrase **quadratic search** to describe the search for the minimum of a U–shaped function f using quadratic interpolation. What distinguishes quadratic search for the minimum from golden section search is how the search points c and d are chosen. We assume that an initial bracket $[a, b]$ for the minimum of the function $f(x)$ is given. We start by choosing a point c in the open interval (a, b). To start the quadratic search iteration we need $f(c) < \min\{f(a), f(b)\}$. Next, d is determined as the point in the open interval (a, b) where the quadratic polynomial that interpolates the data $(c, f(c))$, $(a, f(a))$ and $(b, f(b))$ assumes its unique minimum value; if $d = c$ an error is reported. In fact, it is quite possible that if $d \approx c$ then the location of the minimum is in their vicinity but if $d = c$ we have no simple way of shortening the bracket of the location. Finally, the bracket is refined as usual; of the search points c and d, the one corresponding to the larger value of f becomes an endpoint of the refined bracket and the other is (re-)labeled c. (In the pseudocode in Figure 7.5 we order the points c and d such that $c < d$ to simplify the coding.)

Let us demonstrate that the point d lies between the midpoints of the intervals $[a, c]$ and $[c, b]$, and so lies in the open interval (a, b). (Of course, that d lies in (a, b) is intuitively obvious but the result we prove tells us a little more, that d cannot be closer than half the distance to a given endpoint than c is from the same endpoint.) To start, let $P_2(x)$ be the quadratic polynomial that interpolates the data $(c, f(c))$, $(a, f(a))$ and $(b, f(b))$. With the data so ordered, the Newton form is

$$P_2(x) = g_0 + (x - c)g_1 + (x - c)(x - a)g_2$$

where the coefficients g_0, g_1 and g_2 are chosen so that $P_2(c) = f(c)$, $P_2(a) = f(a)$ and $P_2(b) = f(b)$. The solution of these equations is

$$\begin{aligned} g_0 &= f(c) \\ g_1 &= \frac{f(c) - f(a)}{c - a} \\ g_2 &= \frac{\left(\dfrac{f(b) - f(c)}{b - c}\right) - g_1}{b - a} \end{aligned}$$

Because $f(c) < \min\{f(a), f(b)\}$, it is easy to see that $g_1 < 0$ and $g_2 > 0$. Consequently, the quadratic polynomial P_2 is concave up and achieves its least value at the point

$$d \equiv \frac{a + c}{2} - \frac{g_1}{2g_2}$$

where $P'(d) = 0$. Because $g_1 < 0$ and $g_2 > 0$, it follows that $d > \dfrac{a+c}{2}$. Similarly, if the Newton form of the same interpolating polynomial $P_2(x)$ is constructed but with the data ordered as $(c, f(c))$, $(b, f(b))$ and $(a, f(a))$, then it is easy to show that $d < \dfrac{c+b}{2}$. Therefore, the point d lies between the midpoints of the intervals $[a, c]$ and $[c, b]$.

Generally, quadratic search for the minimum works well. However, for some functions it converges very slowly. In these cases, a pattern develops where successive iterations reset the same endpoint, either a or b, to a nearby point thereby causing the length of the bracket to be only minimally refined. This pattern may be broken by keeping track of the assignments and, when necessary, inserting a special step. For example, a special step might replace the point d with the midpoint of one of the two intervals $[a, c]$ and $[c, b]$, whichever is the longer interval.

In the pseudocode in Figure 7.5, we could use any representation of the quadratic polynomial $P_2(x)$, including one of those used in the analysis above. However, in the interests of transparency and because it should lead to a slightly more accurate computation when rounding error and cancelation impact the calculation, we choose a form centered on the current internal point, c :

$$P_2(x) = A(x-c)^2 + B(x-c) + C$$

The minimum of this quadratic is taken at the location

$$d = c - \frac{B}{2A}$$

so at each iteration we calculate the new point, d, as a correction to c. The point c will usually be a good approximation to the location of the minimum as the iteration proceeds so the correction should be small.

Example 7.2.2. We find a local minimum of $f(x) = -e^{-x}\sin(x)$ on the interval $[a, b] = [0, 1.5]$ using quadratic interpolation search. We terminate the iteration when the bracket is smaller than 10^{-5}. We implemented the pseudocode given in Figure 7.5; results are given in Table 7.3. Observe that the error, that is the distance between the point c and the answer, becomes small quickly but that the bracket takes longer to reduce in length to the required size, hence the need for the last few iterates in the table. Indeed, the left hand end of the bracket remains at $a = 0.75$ from the second iteration until the penultimate iteration.

iterate	$f(iterate)$	*error*
0.000000	0.000000	-0.785398
1.500000	-0.222571	0.714602
0.750000	-0.321983	-0.035398
0.948066	-0.314754	0.162668
0.811887	-0.322175	0.026489
0.786261	-0.322397	0.000862
0.785712	-0.322397	0.000314
0.785412	-0.322397	0.000014
0.785402	-0.322397	0.000004
0.785398	-0.322397	0.000000

Table 7.3: Quadratic interpolation search – iterates, function values and errors in iterates

Problem 7.2.7. *Let f be a U–shaped function on the interval $[a, b]$ and let c be a point in (a, b) for which $f(c) < \min\{f(a), f(b)\}$. Let the quadratic polynomial $P_2(x)$ interpolate $f(x)$ at the points $x = a$, $x = c$ and $x = b$, so that $P_2(c) < \min\{P_2(a), P_2(b)\}$. Without calculating the quadratic $P_2(x)$, argue why $P_2(x)$ is concave up and why the location d of its minimum lies in (a, b).*

Quadratic Search

Input: function to minimize, $f(x)$
initial bracket, $[a, b]$

Output: scalars c, fc, where $fc = f(c) \approx \min_{a \le x \le b} f(x)$

$fa = f(a)$; $fb = f(b)$
$c = 0.5(a + b)$; $fc = f(c)$
if $fc \ge fa$ or $fc \ge fb$ then Stop
while termination criteria not satisfied do
  $A = [(fa - fc) * (b - c) - (fb - fc) * (a - c)]/[(a - c) * (b - c) * (a - b)]$
  $B = [(fa - fc) * (b - c)^2 - (fb - fc) * (a - c)^2]/[(a - c) * (b - c) * (b - a)]$
  $d = c - B/(2 * A)$; $fd = f(d)$
  if $c > d$ then, swap c and d, and fc and fd
    $temp = c$; $c = d$; $d = temp$
    $temp = fc$; $fc = fd$; $fd = temp$
  endif
  if $fc \le fd$ then
    $b = d$; $fb = fd$
  else
    $a = c$; $fa = fc$
    $c = d$; $fc = fd$
  endif
  if $fc \ge fa$ or $fc \ge fb$ then Stop
end while

Figure 7.5: Pseudocode *Quadratic Search*. The initial choice of $c = 0.5 * (a + b)$ is arbitrary, but sensible.

Problem 7.2.8. *Suppose that f is a U–shaped function on the interval $[a, b]$ and c is a point in (a, b) for which $f(c) < \min\{f(a), f(b)\}$. Let P_2 be the quadratic polynomial that interpolates to the data $(a, f(a))$, $(c, f(c))$ and $(b, f(b))$. Use the Mean Value Theorem (see Problem 6.1.3) to demonstrate that there is a point $u \in (a, c)$ for which $P_2'(u) < 0$, and a point $v \in (c, b)$ for which $P_2'(v) > 0$. Use the Intermediate Value Theorem (see Problem 6.1.1) to conclude there is a point d between the points u and v for which $P_2'(d) = 0$.*

Problem 7.2.9. *Show that the interpolation conditions in Problem 7.2.8 lead to the solution listed for the coefficients g_0, g_1 and g_2. Argue why the condition $f(c) < \min\{f(a), f(b)\}$ implies that both $g_1 < 0$ and $g_2 > 0$.*

Problem 7.2.10. *Compute the derivative P_2' and determine the value d for which $P_2'(d) = 0$. Hint: The derivative $P_2'(x)$ is a linear function of x.*

Problem 7.2.11. *Compute the interpolating quadratic polynomial P_2, but this time with the data ordered $(c, f(c))$, $(b, f(b))$ and $(a, f(a))$. For this data, the Newton form of this polynomial is*

$$P_2(x) = h_0 + (x - c)h_1 + (x - c)(x - b)h_2$$

From the interpolating conditions, derive formulas for the coefficients h_1, h_2 and h_3. Argue why $f(c) < \min\{f(a), f(b)\}$ implies that both $h_1 > 0$ and $h_2 > 0$. Argue why $h_2 = g_2$. Determine the value d for which $P_2'(d) = 0$, and argue why $d < \dfrac{c+b}{2}$. Hint: The quadratic interpolating polynomial for this data is unique, regardless of how the data is ordered. So, the power series coefficients of the quadratic $P_2(x)$ are unique. How are the coefficients g_2 and h_2 in this form related to the coefficient of x^2 of the Newton form of $P_2(x)$?

Problem 7.2.12. *Consider the U–shaped function*

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

on $[a, b] = [0.01, 0.99]$. Using $c = 0.02$ as the initial value, carry out three iterations of quadratic search for the minimum. What is the value of x_m for this function, and how is it related to the values c and d computed by each iteration? Remark: The bracket endpoint b remains frozen in this example for many iterations.

7.3 Using the Derivative

Suppose that we can compute values of both the function f and its derivative f' at any point x. There follows a description of an algorithm that incorporates information from both f and f' in the search for location of the local minimum of f. This method extends that of the previous section.

7.3.1 Cubic Interpolation Search

For brevity, we use the phrase **cubic search** to describe the search for the minimum of a U–shaped function f using cubic interpolation. Let f be U–shaped on $[a, b]$. We suppose that $f'(a)f'(b) < 0$, for otherwise f has an endpoint extremum. Because f is U–shaped, it follows that $f'(a) < 0$ and $f'(b) > 0$.

The key computation performed by cubic search is the construction of the cubic Hermite polynomial P_3 that interpolates values $f(x)$ and $f'(x)$ at the endpoints a and b; that is,

$$P_3(a) = f(a), \;\; P_3'(a) = f'(a), \;\; P_3(b) = f(b), \;\; P_3'(b) = f'(b)$$

This interpolating polynomial was introduced in Problem 4.1.21. If $t \equiv (x - a)/h$ with $h \equiv b - a$, then

$$P_3(x) = f(a)\phi(t) + f(b)\phi(1 - t) + hf'(a)\psi(t) - hf'(b)\psi(1 - t)$$

where the cubic polynomials $\phi(s) = (1+2s)(1-s)^2$ and $\psi(s) = s(1-s)^2$. However, here we choose a more convenient form

$$P_3(x) = A(x-c)^3 + B(x-c)^2 + C(x-c) + D$$

where $c = \dfrac{a+b}{2}$. This choice gives a symmetry to the interpolating conditions and they are solved simply to give

$$\begin{aligned} A &= [d(f'(b)+f'(a)) - (f(b)-f(a))]/(4d^3) \\ B &= [f'(b)-f'(a)]/(4d) \\ C &= [3(f(b)-f(a)) - d(f'(b)+f'(a))]/(4d) \\ D &= [f(a)+f(b)]/2 - d[f'(b)-f'(a)]/4 \end{aligned}$$

where $d = \dfrac{b-a}{2}$.

Note that $P_3'(x) = 0$ is a quadratic equation that has at most two roots. Since $P_3'(a) = f'(a) < 0$ and $P_3'(b) = f'(b) > 0$, by the Intermediate Value Theorem one root c lies in the open interval (a,b) and the other root, if it exists, must lie outside $[a,b]$. (Remember that the coefficient of x^2 *may be zero* so there may only be one root of $P_3'(x) = 0$.) The root $c \in (a,b)$ corresponds to a local minimum value of the function $P_3(x)$. The roots of $P_3'(x) = 0$ are $\dfrac{a+b}{2} + \dfrac{-B \pm \sqrt{B^2 - 3AC}}{3A}$. We choose the smaller of the two values in the second term as we know just one root is in (a,b) and we are correcting from the midpoint of the interval.

The cubic search assumes that an initial bracket $[a,b]$ for the minimum of the function f is given. In the pseudocode in Figure 7.6, at startup we determine the values $f(a)$, $f'(a)$, $f(b)$ and $f'(b)$ and check that $f'(a) < 0$ and $f'(b) > 0$ reporting an error if these inequalities are not satisfied. The cubic search iteration begins by computing the location c of the unique minimum of the cubic Hermite interpolating function $P_3(x)$ that interpolates both $f(x)$ and $f'(x)$ at the endpoints a and b. If $f'(c) \approx 0$, then the cubic search stops because the point c is also the location of the minimum of the function f. Also, if $f(c) \geq \max\{f(a), f(b)\}$, we report an error because the function f cannot be U–shaped. If $f'(c) < 0$, then the cubic search moves the endpoint a to the point c by re-labeling c as a, and re-labeling $f(c)$ as $f(a)$. Similarly, if $f'(c) > 0$, then the cubic search moves the endpoint b to the point c. The process continues until the bracket on the location of the minimum is reduced to a length less than a user error tolerance.

Example 7.3.1. We find a local minimum of $f(x) = -e^{-x}\sin(x)$ on the interval $[a,b] = [0, 1.5]$ using cubic interpolation search. We terminate the iteration when the bracket is smaller than 10^{-5}. We implemented the pseudocode given in Figure 7.6 and the results are given in Table 7.3. Observe that the iteration terminates because at the fourth iteration (and thereafter) $f'(c)$ is zero to machine accuracy. If we don't include the check on $f'(c)$, the iteration never terminates since the calculated values of $f'(c)$ are very small and have the wrong sign. This results in the algorithm "shrinking" the interval very slowly from the incorrect end.

iterate	$f(iterate)$	$f'(iterate)$	*error*
7.809528e-001	-3.223906e-001	-2.879105e-003	-4.445365e-003
7.857741e-001	-3.223969e-001	2.423153e-004	3.759439e-004
7.853982e-001	-3.223969e-001	-7.308578e-010	-1.252987e-009
7.853982e-001	-3.223969e-001	-5.551115e-017	-1.195117e-010

Table 7.4: Cubic interpolation search – iterates, function and derivative values and errors in iterates

Cubic Search

Input: function to minimize, $f(x)$
initial bracket, $[a, b]$

Output: scalars c, fc, where $fc = f(c) \approx \min_{a \le x \le b} f(x)$

$fa = f(a); fb = f(b); fap = f'(a); fbp = f'(b)$
$c = 0.5(a + b); fc = f(c); fcp = f'(c)$
if $fc \ge fa$ and $fc \ge fb$ then Stop
if $fap \ge 0$ or $fbp \le 0$ then Stop
$M = \max\{|fap|, |fbp|\}$
while termination criteria not satisfied do
  $d = 0.5(b - a)$
  $A = [d(fbp + fap) - (fb - fa)]/(4d^3)$
  $B = (fbp - fap)/(4d)$
  $C = [3(fb - fa) - d(fbp + fap)]/(4d)$
  $D = (fb + fa)/2 - d(fbp - fap)/4$
  if $B > 0$ then
    $c = 0.5(a + b) + (-B + \sqrt{B^2 - 3AC})/(3A)$
  else
    $c = 0.5(a + b) + (-B - \sqrt{B^2 - 3AC})/(3A)$
  endif
  $fc = f(c); fcp = f'(c)$
  if $|fcp| < roundtol \cdot M$ then
    Terminate with c, fc as output
  endif
  if $fc \ge fa$ and $fc \ge fb$ then Stop
  if $fcp > 0$ then
    $b = c; fb = fc; fbp = fcp$
  else
    $a = c; fa = fc; fap = fcp$
  endif
end while

Figure 7.6: Pseudocode *Cubic Search*. The initial choice of $c = 0.5 * (a + b)$ is arbitrary, but sensible.

Problem 7.3.1. *Suppose f is a U–shaped function on $[a, b]$ with $f'(a)f'(b) \geq 0$. Show that f must have an endpoint extremum.*

Problem 7.3.2. *Suppose f is a U–shaped function on $[a, b]$ with $f'(a)f'(b) < 0$. Show that we must have $f'(a) < 0$ and $f'(b) > 0$.*

Problem 7.3.3. *Let f be a U–shaped function on $[a, b]$ with $f'(a) < 0$ and $f'(b) > 0$. Let $P_3(x)$ be the cubic Hermite polynomial that interpolates values $f(x)$ and $f'(x)$ at the endpoints a and b. In the Newton form, we can write its derivative*

$$P_3'(x) = g_0 + (x - a)g_1 + (x - a)(x - b)g_2$$

Show that the derivative interpolation conditions imply that $g_0 < 0$ and $g_1 > 0$. Assume that $g_2 \neq 0$. Observe that the roots of $P_3'(x) = 0$ are located at the points where the straight line $y = g_0 + (x-a)g_1$ intersects the parabola $y = -(x - a)(x - b)g_2$. Sketch this line and this parabola and show that, regardless of the value of the coefficient g_2, the line and the parabola always intersect at two distinct points. Show that only one of these intersection points lies in (a, b).

Problem 7.3.4. *Let f be a U–shaped function on $[a, b]$ with $f'(a) < 0$ and $f'(b) > 0$. Let $P_3(x)$ be the cubic Hermite polynomial that interpolates values $f(x)$ and $f'(x)$ at the endpoints a and b. By counting turning points, show that $P_3(x)$ has a unique minimum located at a point c in (a, b).*

Problem 7.3.5. *Let f be a U–shaped function on $[a, b]$ with $f'(a) < 0$ and $f'(b) > 0$. Let the function $Q(x) = P_3'(x)$ where $P_3(x)$ is the cubic Hermite polynomial that interpolates values $f(x)$ and $f'(x)$ at the endpoints a and b. Note that $Q(x)$ is a quadratic with $Q'(a) < 0$ and $Q'(b) > 0$.*

- *Use the Intermediate Value Theorem (see Problem 6.1.1) to show that there is a point $c \in (a, b)$ where $Q(c) = 0$.*
- *Use the Mean Value Theorem (see Problem 6.1.3) to show that there is a point $u \in (a, c)$ where $Q'(u) > 0$. Use the Mean Value Theorem again to show that there is a point $v \in (c, b)$ where $Q'(v) > 0$.*
- *Using the fact that $Q'(x)$ is a linear polynomial, and the facts established above show that $Q'(c) > 0$.*

Conclude that the cubic polynomial $P_3(x)$ has a unique minimum located at a point $c \in (a, b)$.

Problem 7.3.6. *Consider the U–shaped function*

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

on $[a, b] = [0.01, 0.99]$. Using $c = 0.5$ as the initial value, carry out three iterations of cubic search for the minimum.

7.4 Matlab Notes

Using the basic techniques developed for iterative methods in Section 6.6 and the pseudocodes presented in the previous section, it is not difficult to write MATLAB implementations of the optimization methods discussed in this chapter. We therefore leave these as exercises.

Problem 7.4.1. *Write a* MATLAB *function M–file that implements the golden section search method, and use it to find a local minimum of $f(x) = e^{-x} - sin(x)$ on the interval $[0, 3]$. Start with the bracket $[1, 2]$ and iterate until the length of the bracket is less than 10^{-2}.*

Problem 7.4.2. *Write a* MATLAB *function M–file that implements the quadratic interpolation search for the minimum, with the midpoint of* $[a, b]$ *as the initial point c. Report all applicable errors. Now consider using this to find a minimum of the function*

$$f(x) = \frac{x^4 + 1}{x^2 - 2x + 1}$$

on $[a, b] = [-10, 10]$. *Is this function U–shaped on the interval* $[-10, 10]$*?*

Problem 7.4.3. *Consider the U–shaped function*

$$f(x) = \begin{cases} 1 - x & \text{if } x \leq 0.9 \\ x - 0.8 & \text{otherwise} \end{cases}$$

on $[a, b] = [0, 1]$. *Use quadratic interpolation search for the minimum starting with the midpoint of* $[a, b]$ *as the initial point c. Report all applicable errors. Note that it may be easier to use a function M–file to implement* $f(x)$*, rather than an anonymous function, for this particular problem.*

Problem 7.4.4. *Use the quadratic interpolation search to find a local minimum of* $f(x) = e^{-x} - sin(x)$ *on the interval* $[0, 1]$. *Start with the bracket* $[1, 2]$ *and iterate until the length of the bracket is less than* 10^{-5}.

Problem 7.4.5. *Consider the U–shaped function*

$$f(x) = \begin{cases} 1 - x & \text{if } x \leq 0.9 \\ x - 0.8 & \text{otherwise} \end{cases}$$

on $[a, b] = [0, 1]$. *Use cubic interpolation to search for the minimum. Report all applicable errors. Note that it may be easier to use a function M–file to implement* $f(x)$*, rather than an anonymous function, for this particular problem.*

Problem 7.4.6. *Use the cubic interpolation search to find a local minimum of* $f(x) = e^{-x} - sin(x)$ *on the interval* $[0, 1]$. *Start with the bracket* $[1, 2]$ *and iterate until the length of the bracket is less than* 10^{-5}.

MATLAB's main built-in function for finding a local minimum of a function of one variable is `fminbnd`, which implements a combination of golden section and quadratic (parabolic) interpolation search. The basic usage is

```
x = fminbnd(fun, x1, x2)
```

where, as usual, `fun` can be an inline or anonymous function, or a handle to a function M–file, and `[x1, x2]` is an initial bracket of the minimum.

Example 7.4.1. To find the minimum of $f(x) = -e^{-x}\sin(x)$ on the interval $[0, 1.5]$, we can use the MATLAB statement:

```
x = fminbnd(@(x) -exp(-x).*sin(x), 0, 1.5)
```

The computed result is $x \approx 0.7854$, which is an approximation of $\frac{\pi}{4}$.

Example 7.4.2. Recall that $\max\{f(x)\} = \min\{-f(x)\}$. Thus, if we want to find the maximum of $f(x) = \dfrac{1}{x^2 - 6x + 8}$ on the interval $[0, 6]$, we can use the MATLAB statement:

```
x = fminbnd(@(x) -1./(x.*x-6*x+10), 0, 6)
```

which computes $x = 3$. It is not difficult to confirm that $f(3)$ is the maximum of $f(x)$ on the interval $[0, 6]$.

It is important to keep in mind that if the $f(x)$ is not U shaped, then `fminbnd` does not guarantee to find a global minimum on an interval, as illustrated in the following example.

Example 7.4.3. Consider the function $g(x) = x - \cos(7x)$ on the interval $[-4, 4]$. A plot of $g(x)$ is shown in Figure 7.7. Notice that the minimum is located approximately at the point $(-3.6109, -4.6006)$. If we compute

```
x = fminbnd(@(x) x-cos(7*x), -4, 4)
```

the result is $x \approx -0.9181$. Although this is a local minimum, it is not the global minimum of $f(x)$ on the interval $[-4, 4]$. If instead, we compute

```
x = fminbnd(@(x) x-cos(7*x), -4, 3)
```

the result is $x \approx -3.6109$, which is the global minimum. Interestingly, though, if we further refine the initial interval and compute

```
x = fminbnd(@(x) x-cos(7*x), -4, 2)
```

the result is $x \approx -2.7133$, yet another local minimum.

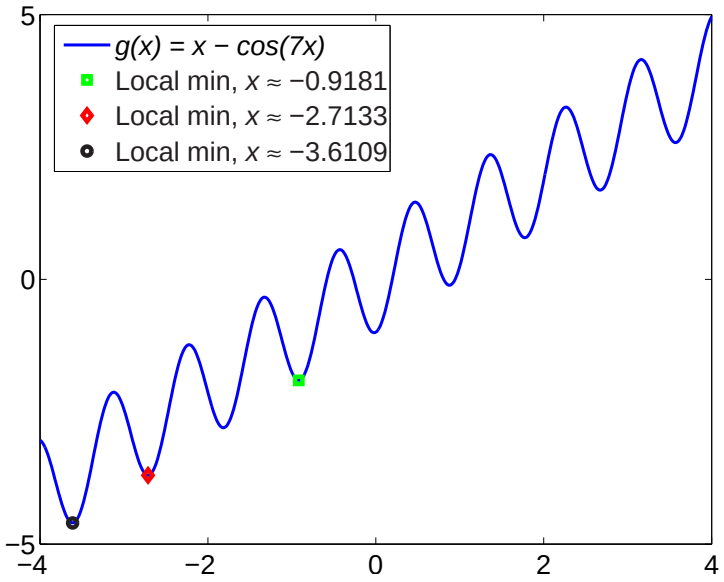

Figure 7.7: Plot of the function $g(x) = x - \cos(7x)$ on the interval $[-4, 4]$.

Problem 7.4.7. *Use* `fminbnd` *to compute a minimum of* $f(x) = x - \cos(7x)$ *with initial intervals* $[-4, z]$*, with* $z = 4, 3, 2, 1, 0, -1, -2, -3$*. What does* `fminbnd` *compute in each case?*

Problem 7.4.8. *Use* `fminbnd` *to find all local minima of* $f(x) = x^2 - \cos(4x)$*. What starting interval did you use in each case? Hint: You might want to first plot the function, and use it to determine appropriate starting intervals.*

As with `fzero` (see Section 6.6.4), it is possible to reset certain default parameters used by `fminbnd` (such as stopping tolerance and maximum number of iterations), and it is possible to obtain more information on exactly what is done during the computation of the minimum (such as number of iterations and number of function evaluations) by using the calling syntax

```
[x, fval, exitflag, output] = fminbnd(fun, x1, x2, options)
```

where

- `x` is the computed approximation of the local minimizer in the interval `[x1, x2]`.
- `fval` is the function value $f(x_{\min})$.
- `exitflag` provides information about what condition caused `fminbnd` to terminate; see the Help page on `fminbnd` for additional information.
- `output` is a structure array that contains information such as which algorithms were used (e.g., golden section search or parabolic interpolation), number of function evaluations and number of iterations.
- `options` is an input parameter that can be used, for example, to reset the stopping tolerance, and to request that results of each iteration be displayed in the command window. The options are set using the built-in MATLAB function `optimset`.

The following example illustrates how to use `options` and `output`.

Example 7.4.4. Consider the function $f(x) = \dfrac{1}{x(1-x)^2}$. It is not difficult to show that $f(x)$ has a local minimum at $x = \frac{1}{3}$. Suppose we execute the MATLAB commands:

```
options = optimset('TolX', 1e-15, 'Display', 'iter');
[x, fval, exitflag, output] = fminbnd(@(x) 1./(x.*(1-x).^2), 0.001, 0.999, options);
```

Because we used `optimset` to set the `options` field to display the iterations, the following information is displayed in the command window:

```
 Func-count     x          f(x)         Procedure
    1       0.382202       6.8551        initial
    2       0.617798      11.0807        golden
    3       0.236596      7.25244        golden
    4       0.334568      6.75007        parabolic
    5       0.336492      6.75045        parabolic
    6       0.333257         6.75        parabolic
    7       0.333332         6.75        parabolic
    8       0.333333         6.75        parabolic
    9       0.333333         6.75        parabolic
   10       0.333333         6.75        parabolic
   11       0.333333         6.75        parabolic
   12       0.333333         6.75        golden
   13       0.333333         6.75        parabolic

Optimization terminated:
the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-15
```

which shows that some iterations perform a golden section search, while others perform a quadratic interpolation search. Quadratic search is preferred in cases where it will not lead to difficulties such as placing a new point too close to the endpoints. (The table uses `golden` to refer to golden section search, and `parabolic` to refer to quadratic interpolation search. The term `initial` simply

indicates the initial choice which, in the case shown, is a golden section point.) We observe that a total of 13 function evaluations are needed to compute the minimum. The final six iterations all seem to be the same from the output but they are in fact spent computing the final value of `x`, not shown here, to the fifteen digits specified using `optimset`. If we display `output` it provides the following information:

```
iterations: 12
 funcCount: 13
 algorithm: 'golden section search, parabolic interpolation'
   message: [1x111 char]
```

The "message" can be viewed by simply entering the following command:

```
output.message
```

Problem 7.4.9. *Use* `fminbnd` *to find a minimum of* $f(x) = e^{-x} - \sin(x)$ *starting with the bracket* $[1, 2]$. *How many iterations used a golden section search, and how many used a parabolic interpolation search? Use the default* `TolX`. *Do the results change if* `TolX` *is changed to* 10^{-15}*?*

Problem 7.4.10. *Use* `fminbnd` *to find a minimum of* $f(x) = |x - 7|$ *starting with the bracket* $[6, 8]$. *How many iterations used a golden section search, and how many used a parabolic interpolation search? Use the default* `TolX`. *Do the results change if* `TolX` *is changed to* 10^{-15}*?*

Problem 7.4.11. *Repeat Problem 7.4.7, and report how many iterations used a golden section search, and how many used a parabolic interpolation search. Do the results change if* `TolX` *is changed to* 10^{-15}*?*

Problem 7.4.12. *Repeat Problem 7.4.8, and report how many iterations used a golden section search, and how many used a parabolic interpolation search. Do the results change if* `TolX` *is changed to* 10^{-15}*?*

Problem 7.4.13. *Consider finding a minimum of* $f(x) = e^{-x} - \sin(x)$ *starting with the bracket* $[1, 2]$ *so that the final "bracket" is of length* 10^{-4}. *Implement*

1. *Golden Section search as given in the pseudocode in figure 7.4.*
2. *Quadratic Interpolation search as given in the pseudocode in figure 7.5.*
3. *Cubic Interpolation search as given in the pseudocode in figure 7.6.*

each in a MATLAB *M–file then solve this problem using your code. Compare the number of iterations needed in each case with those required by the* MATLAB *function* `fminbnd` *to solve the problem to the required accuracy.*

Index