Contents

Preface

1 Number Representations and Errors **1**
1.1 Introduction 1
1.2 Floating-point number 2
1.3 Sources of errors 8
1.4 Measures of error and precision 12
1.5 Floating-point arithmetic 15

2 Iterative Solution of Equations **20**
2.1 Introduction 20
2.2 The bisection method 22
2.3 Function iteration 27
2.4 Newton's method 35
2.5 The secant method 41
2.6 2 equations in 2 unknowns: Newton's method 45
2.7 MATLAB functions for equation solving 49

3 Approximate Evaluation of Functions **52**
3.1 Introduction 52
3.2 Series expansions 53
3.3 CORDIC algorithms 60
3.4 MATLAB functions 73

4 Interpolation **74**
4.1 Introduction 74
4.2 Lagrange interpolation 77
4.3 Difference representations 84
4.4 Splines 100
4.5 MATLAB interpolation functions 115

5 Numerical Calculus **119**
5.1 Introduction 119
5.2 Numerical integration: interpolatory quadrature rules 122
5.3 Composite formulas 132
5.4 Practical numerical integration 140
5.5 Improper integrals 149
5.6 Numerical differentiation 153
5.7 Maxima and minima 159
5.8 MATLAB functions for numerical calculus 168

6 Differential Equations **171**
6.1 Introduction and Euler's method 171
6.2 Runge–Kutta methods 178
6.3 Multistep methods 186
6.4 Systems of differential equations 193
6.5 Boundary-value problems: (1) shooting methods 198
6.6 Boundary-value problems: (2) finite difference methods 204
6.7 MATLAB functions for ordinary differential equations 208

7 Linear Equations **210**
7.1 Introduction 210
7.2 Gauss elimination 212
7.3 *LU* factorization: iterative refinement 225
7.4 Iterative methods 232
7.5 Linear least squares approximation 239
7.6 Eigenvalues 248
7.7 MATLAB's linear algebra functions 257

Appendices
A MATLAB Basics 261
B Answers to Selected Exercises 282

References and Further Reading 298

Index 299

Preface

This book is a revision of the first edition *Guide to Numerical Analysis,* which was developed from a variety of introductory courses in numerical methods and analysis that I taught at the University of Lancaster and the University of Maryland. Over the intervening ten years or so, I have taught similar courses at the Naval Academy. The level and content of the courses has varied considerably but they have included all the material presented here. The intention of this book is to provide a gentle and sympathetic introduction to many of the problems of scientific computing, and the wide variety of methods used for their solution. The book is therefore suitable for a first course in numerical mathematics or scientific computing – whether it be for mathematics majors, or for students of science, engineering or economics, for example.

This, of course, precludes the possibility of providing a fully rigorous treatment of all the most up-to-date algorithms and their analyses. The intention is rather to give an appreciation of the need for numerical methods for the solution of different types of problem, and to discuss the basic approaches. For each of the problems this is followed by at least some mathematical justification – and, most importantly, examples – to provide both practical evidence and motivation for the reader. The level of mathematical justification is determined largely by the desire to keep the mathematical prerequisites to a minimum. Thus, for example, no knowledge of linear algebra is assumed beyond the most basic matrix algebra, and analytic results are based on a sound knowledge of the calculus.

Inevitably this means that some methods, such as those for numerical solution of ordinary differential equations, are not derived in a detailed and rigorous manner. Such methods are nonetheless justified and/or derived by appealing to other techniques discussed earlier in the book such as those for numerical integration and differentiation. In contrast, in situations where rigorous explanation can be included without substituting bafflement for enlightenment, this is given. In Chapter 2, on iterative solution of equations, for instance, a detailed analysis of the convergence of some iterative schemes is presented. (A brief summary of basic results on convergence of sequences is included as a reminder to the reader.)

As in the first edition, practical justification of the methods is presented through computer examples and exercises. However, a major change from that first edition is that these are now achieved through the use of MATLAB[1] rather than the BASIC programming language. MATLAB is an extremely powerful package for numerical

[1] MATLAB is a registered trademark of The MathWorks, Inc.

computing and programming which has many 'add-on' toolboxes. However we use nothing outside the basic MATLAB package in this text. Each chapter concludes with a section describing some of the relevant MATLAB functions – which will often include implementations of methods beyond the scope of this book. It is a mark of the power and success of packages such as MATLAB that programming languages such as FORTRAN which were formerly regarded as 'high-level' languages have become 'low-level' ones in the modern era of scientific computing. The book has an Appendix devoted to the basics of the MATLAB package, its language and programming.

The MATLAB example code used in this book is not intended to exemplify sophisticated or robust pieces of software; it is simply illustrative of the methods under discussion. This is important for a beginning student since writing robust code necessitates taking into account all sorts of difficulties well beyond our present scope. Our objective here is to get the basic ideas across. The interested student can ponder some of these difficulties in subsequent courses.

As to the content of the book: Chapter 1 is devoted to floating-point arithmetic and errors and, throughout the book we take time to consider the precision of our numerical solutions, and how this can be safeguarded or improved. Subsequent chapters deal with the *solution of* (nonlinear) *equations*, now including pairs of equations in two unknowns (Chapter 2); the *approximate evaluation of functions*, both by series approximations and the CORDIC algorithms that are used in almost all calculators and PCs (Chapter 3); *interpolation* with both polynomial and cubic spline interpolation discussed (Chapter 4); *numerical calculus* including integration, differentiation and location of extrema (Chapter 5); *ordinary differential equations* including Runge-Kutta and multistep methods as well as systems of differential equations and shooting and finite difference methods for boundary value problems (Chapter 6). Chapter 7 is devoted to *linear equations*. This fundamental topic is deliberately left to the end to allow the earlier chapters to motivate its treatment. This chapter has now been augmented with a section on eigenvalues. However, Chapter 7 is independent of any of the specific material and so can easily be covered earlier if desired.

The intention of this book remains to provide the student with an introduction to this subject which is not, in its combined demands of computing, motivation, manipulation and analysis, paced such that only the most able can 'see the wood for the trees'. The major effort of programming is therefore removed from the reader, as are the harder parts of the analysis. The algebraic manipulation is largely unavoidable but is carefully explained – especially in the early stages. The motivation for the numerical methods is provided by the examples – which are not all of types that can be solved directly – and the numerical methods themselves provide the motivation for the necessary analysis.

In writing the Preface for the first edition, it was important to thank two good friends David Towers, the series editor, and Charles Clenshaw who was a great source of help in the writing process. To those I should add many colleagues who

have influenced me over the last ten years. Where the influence is positive it is to their credit, if it is ever negative it is probably because I did not take their good advice! Notable among these are Frank Olver, Dan Lozier, Alan Feldstein, Jim Buchanan, George Nakos and Bob Williams. The struggles of hundreds of students have also helped me see the difficulties, and, I hope, have led to better explanations of many topics.

Annapolis PETER R. TURNER

CHAPTER 3

Approximate Evaluation of Functions

Aims and objectives

In this chapter, we address the question of how to obtain good *approximations* to some of the standard elementary functions. We introduce both the traditional approach using series expansions and its modern equivalent, the CORDIC algorithms used by almost all calculators and personal computer chips. With easy access to so many functions at the touch of a button, one of the primary objectives is to convince the reader that there is a problem to be solved – as well as giving an introduction to two powerful techniques.

3.1 Introduction

The next several chapters are concerned with computing functional quantities beginning with the evaluation of some of the basic functions of mathematics. The elementary functions such as sine, cosine, exponential and logarithm functions are not computable *exactly* except in very special cases. Some are even defined in terms of integrals which cannot be computed exactly by standard algebra or calculus techniques. For example, the natural logarithm function is often defined as

$$\ln x = \int_1^x \frac{1}{t}\,dt$$

Techniques for numerical evaluation of integrals such as this will be discussed in Chapter 5.

Other elementary functions, such as arctan, are defined as inverses of functions which themselves cannot be readily evaluated exactly. Implicitly, this defines them as solutions of nonlinear equations which cannot be solved algebraically. Techniques such as those used in Chapter 2 are then potential methods of evaluating such functions.

In many cases there are better techniques available. Many of these are based on either the use of series expansions (Section 3.2) or the CORDIC algorithms (Section 3.3) which are widely used in calculators and personal computer (PC) hardware. If you consult the MATLAB manual to see what methods are used for some of the elementary and special functions of mathematics, you will see other

techniques mentioned, such as rational approximation and asymptotic series. The specifics of these methods are beyond our scope here but many of the principles used here are also employed in those techniques to ensure the desired accuracy in the resulting approximations.

In the case of series expansions, the important theoretical issues are finding the radius of convergence of the series expansion, and then determining the number of terms that are needed in order to reduce the truncation error below some prescribed tolerance. Once that is achieved, the summation of the appropriate terms of the series is usually a straightforward task.

For both series expansions and the CORDIC algorithms there is further difficulty in handling arguments outside their intervals of convergence. These problems can often be overcome by some *range reduction* method. As an (apparently) simple example of this, we know that the basic trigonometric functions sin and cos are periodic with period 2π. It is sufficient therefore to have good algorithms for computing these functions for arguments in the interval $[0, 2\pi]$; larger arguments can be reduced to this interval by subtracting an appropriate multiple of 2π. As we shall see there is much more to it than this implies.

3.2 Series expansions

There are two absolutely fundamental power series from which many of the other important examples are derived: the *geometric series*

$$\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k = 1 + x + x^2 + \cdots \qquad (|x| < 1) \tag{3.1}$$

and the *exponential series*

$$\exp(x) = e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \qquad (\text{all } x) \tag{3.2}$$

Other important series expansions are easily derived from these, or by Taylor or MacLaurin expansions. Using the identity

$$e^{ix} = \cos x + i \sin x$$

where i is the imaginary $\sqrt{-1}$, we get the series for the two basic trigonometric functions:

$$\cos x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} \cdots \qquad (\text{all } x) \tag{3.3}$$

$$\sin x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} \cdots \qquad (\text{all } x) \tag{3.4}$$

(Alternatively, if you are unfamiliar with complex numbers, these are the MacLaurin series for these functions.) Series expansions for the hyperbolic functions can be obtained in a similar manner

$$\cosh x = \frac{1}{2}(e^x + e^{-x}) = \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \qquad (\text{all } x) \qquad (3.5)$$

$$\sinh x = \frac{1}{2}(e^x - e^{-x}) = \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots \qquad (\text{all } x) \qquad (3.6)$$

By integrating the power series (3.2) we get

$$\ln(1-x) = -\sum_{k=0}^{\infty} \frac{x^{k+1}}{k+1} = -x - \frac{x^2}{2} - \frac{x^3}{3} - \cdots \qquad (|x| < 1) \qquad (3.7)$$

and, replacing x by $-x$,

$$\ln(1+x) = -\sum_{k=0}^{\infty} \frac{(-1)^{k+1}x^{k+1}}{k+1} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4}\cdots \qquad (|x| < 1) \qquad (3.8)$$

For more details on the derivations of these formulas, consult your calculus text.

The rest of this section is devoted to some important and illustrative examples.

Example 1 The series in (3.8) is convergent for $x = 1$. It follows that

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4}\cdots$$

Use the first 8 terms of this series to estimate $\ln 2$. How many terms would be needed to compute $\ln 2$ with an error less than 10^{-6} using this series? (Note: The true value of $\ln 2 \approx 0.693\,147\,18$)

The first 8 terms yield

$$\ln 2 \approx 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} = 0.634\,523\,81$$

which has an error close to 0.06.

Since the series (3.8) is an alternating series of decreasing terms (for $0 < x \leq 1$), the truncation error is smaller than the first term omitted. To force this truncation error to be smaller than 10^{-6} would therefore require that the first term omitted is smaller than $1/1\,000\,000$. That is, the first one million terms would suffice.

This is obviously not a practical approach. We shall return to the natural logarithm function later.

Example 2 Derive a series expansion for the function $\arctan x$. Use this series and the identity $\arctan 1 = \pi/4$ to compute π.

First, we know that

$$\frac{d}{dt}\arctan t = \frac{1}{1+t^2}$$

and so, for $|t| < 1$, using (3.1) with $x = -t^2$ we get

$$\frac{d}{dt}\arctan t = 1 - t^2 + t^4 - t^6 \cdots$$

Power series may be integrated term-by-term within their radius of convergence. Hence, for $|x| < 1$,

$$\begin{aligned} \arctan x &= \int_0^x \left(1 - t^2 + t^4 - t^6 \cdots\right) dt \\ &= x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \cdots \end{aligned} \tag{3.9}$$

This series is also convergent for $x = 1$ and so we may deduce that

$$\frac{\pi}{4} = \arctan 1 = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \cdots$$

The first eight terms yield the approximation

$$\pi \approx 4 \sum_{k=0}^{7} \frac{(-1)^k}{2k+1} = 3.017\,071\,8$$

Adding the next term we get $\pi \approx 3.017\,071\,8 + 4/17 = 3.252\,365\,9$. It is apparent that many more terms will be needed to obtain a good approximation to π using this series.

For *single*-precision floating-point, we would require an error in the series approximation of $\pi/4$ smaller than 2^{-25}. Since the series is alternating with decreasing terms, we could stop once the first term omitted is smaller than this bound: $2^{24} = 16\,777\,216$ would suffice! For IEEE double precision, as is used by MATLAB, the required number of terms rises to $2^{53} \approx 10^{16}$. Even on a very fast *gigaflop* (10^9 floating-point operations per second) computer, we would be waiting about 10 million seconds, or nearly 4 months to obtain a value for π. Typing `pi` at the MATLAB prompt should convince you that better techniques must be available! We shall consider some improvements shortly.

These first two examples using series approximations make it plain that finding an elegant mathematical expression for a quantity is not the same as finding a good algorithm for its evaluation!

Example 3 Find the number of terms of the exponential series that are needed for $\exp x$ to have error $< 10^{-5}$ for $|x| \leq 2$

First we observe that the tail of the exponential series truncated after N terms increases with $|x|$. Also the truncation error for $x > 0$ will be greater than that for $-x$ since the series for $\exp(-x)$ will be alternating in sign. It is sufficient therefore to consider $x = 2$.

We shall denote by $E_N(x)$ the truncation error in the approximation using N terms:

$$\exp x \approx \sum_{k=0}^{N-1} \frac{x^k}{k!}$$

Then, we obtain, for $x > 0$

$$\begin{aligned} E_N(x) &= \sum_{k=N}^{\infty} \frac{x^k}{k!} = \frac{x^N}{N!} + \frac{x^{N+1}}{(N+1)!} + \frac{x^{N+2}}{(N+2)!} + \cdots \\ &= \frac{x^N}{N!}\left[1 + \frac{x}{N+1} + \frac{x^2}{(N+2)(N+1)} + \cdots\right] \\ &\leq \frac{x^N}{N!}\left[1 + \frac{x}{N+1} + \frac{x^2}{(N+1)^2} + \cdots\right] \\ &= \frac{x^N}{N!} \cdot \frac{1}{1 - x/(N+1)} \end{aligned}$$

provided $x < N + 1$. For $x = 2$, this simplifies to

$$E_N(2) \leq \frac{2^N}{N!} \cdot \frac{N+1}{N-1}$$

and we require this quantity to be smaller than 10^{-5}. Now $2^{11}/11! = 5.130\,671\,8 \times 10^{-5}$, while $2^{12}/12! = 8.551\,119\,7 \times 10^{-6}$. We must check the effect of the factor $\dfrac{N+1}{N-1} = \dfrac{13}{11} = 1.181\,818\,2$. Since $(1.181\,818\,2)(8.551\,119\,7) = 10.105\,869$, 12 terms are not quite sufficient. $N = 13$ terms are needed: $\dfrac{2^{13}}{13!} \cdot \dfrac{14}{12} = 1.534\,816\,4 \times 10^{-6}$.

We note that for $|x| \leq 1$ in Example 3 we obtain $E_N(1) \leq \dfrac{N+1}{N \cdot N!} < 10^{-5}$ for $N \geq 9$. For $|x| \leq 1/2$, just 7 terms are needed. The number of terms required increases rapidly with x. These ideas can be used as a basis for *range reduction*, so that the series would be used only for very small values of x. For example, we could use the 7 terms to obtain $e^{1/2}$ and then take

$$e^2 = \left(e^1\right)^2 = \left(e^{1/2}\right)^4$$

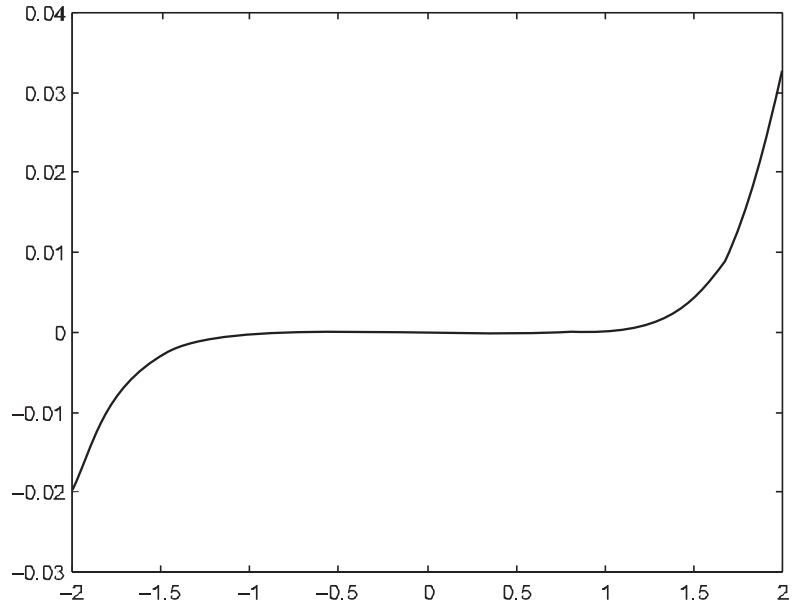

Figure 3.1 Series approximation error for e^x

to obtain exp 2. Greater care would be needed over the precision obtained from the series to allow for any loss of accuracy in squaring the result twice. The details are unimportant here, the point is that the series expansion can provide the basis for a good algorithm.

The magnitude of the error in any of these approximations to e^x increases rapidly with $|x|$, as can be seen from Figure 3.1.

The curve plotted is the error $e^x - \sum_{k=0}^{6} x^k/k!$. We see that the error remains very small throughout $[-1, 1]$ but rises sharply outside this interval indicating that more terms are needed there. The truncated exponential series is computed using the function m-file

```
function y=expn(x,n)
% Evaluates the first n terms of the exponential series
s=ones(size(x));
t=s;
for k=1:n-1
   t=t.*x/k;
   s=s+t;
end
y=s;
```

In the remaining examples of this section, we discover that substantial improvements are possible for the first two examples, too.

Example 4 Develop a series for $\ln((1+x)/(1-x))$, use it for the evaluation of $\ln 2$ with error less than 10^{-6}

We can use the series (3.7) and (3.8) to obtain

$$\begin{aligned} \ln\frac{1+x}{1-x} &= \ln(1+x) - \ln(1-x) \\ &= 2\left[x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots\right] \end{aligned} \tag{3.10}$$

Also $\dfrac{1+x}{1-x} = 2$ for $x = 1/3$ and so

$$\ln 2 = \frac{2}{3}\left[1 + \frac{(1/3)^2}{3} + \frac{(1/3)^4}{5} + \cdots\right] = \frac{2}{3}\sum_{k=0}^{\infty}\frac{1}{(2k+1)3^{2k}}$$

The truncation error incurred by using just the first N terms is then bounded by

$$\frac{2}{3}\cdot\frac{1}{2N+1}\cdot\frac{1}{9^N} < 10^{-6}$$

for $N \geq 6$ so that just 6 terms suffice. This compares very favourably with the 1 million that were required in Example 3. Using these 6 terms, we obtain the approximation

$$\ln 2 \approx \frac{2}{3}\sum_{k=0}^{5}\frac{1}{(2k+1)3^{2k}} = 0.693\,147\,07$$

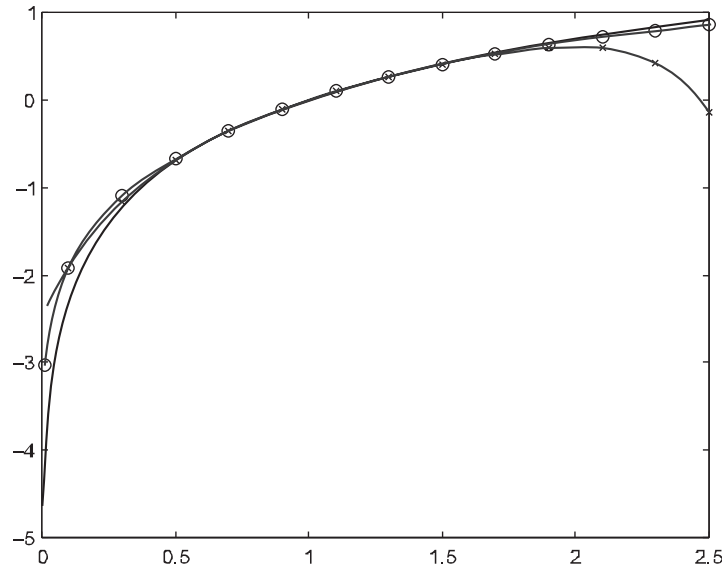

Figure 3.2 Series approximations to ln x

Recall the true value of $\ln 2 \approx 0.693\,147\,18$, so that the actual error is close to 10^{-7}, which is well within our tolerance.

The relative effectiveness of this approximation and the original power series (3.8) is illustrated in Figure 3.2. The true natural logarithm is plotted with a solid line, the sum of the first 6 terms of (3.8) with $\times$s and the first 6 terms of (3.10) with the os.

It is apparent that the new approximation reproduces the curve better over a wider range.

Example 5 Compute π using the identity $\arctan 1/\sqrt{3} = \pi/6$

In Example 2, we obtained the series expansion $\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \cdots$ from which we have

$$\begin{aligned}\pi &= 6\left[3^{-1/2} - \frac{3^{-3/2}}{3} + \frac{3^{-5/2}}{5} \cdots\right] \\ &= \frac{6}{\sqrt{3}}\left[1 - \frac{1}{3(3)} + \frac{1}{5(3)^2} \cdots\right] = 2\sqrt{3}\sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)3^k}\end{aligned}$$

The truncation error for this alternating series is bounded by the first term omitted. For single precision floating-point evaluation of π we would require this truncation error to be smaller than 2^{-23}, and so we seek N such that

$$\frac{2\sqrt{3}}{(2N+1)3^N} < 2^{-23}$$

This is satisfied for $N \geq 13$. This should be compared with the nearly 17 million terms needed to achieve the same accuracy in Example 2. Using these terms we obtain

$$\pi \approx 2\sqrt{3}\sum_{k=0}^{13} \frac{(-1)^k}{(2k+1)3^k} = 3.141\,592\,6$$

which is indeed a good approximation.

For IEEE double-precision (MATLAB) accuracy, the approximation in Example 5 requires just 30 terms. The 4 months on a fast computer for Example 2 would be reduced to about one-millionth of a second! Clearly the choice and design of computer algorithms can have a marked effect. The actual methods implemented on modern machines are much more efficient than these but the analysis of this section should give some idea of what can be achieved.

Exercises: Section 3.2

1 Write a MATLAB m-file to approximate the natural logarithm using the first 6 terms of (3.8). Use it to estimate $\ln 1.25$.

2 How many terms of the series (3.8) are needed to approximate $\ln 1.25$ with error smaller than 10^{-6}? Evaluate this approximation and verify that the error is indeed within the tolerance.

3 Use the fact that $\ln 10 = \ln 8 + \ln 1.25$, the result of Exercise 2, and an accurate value for $\ln 2$ to estimate $\ln 10$ with error smaller than 10^{-6}.

4 Determine the number of terms of the exponential series that are needed to obtain $e^{0.1}$ with error smaller than 10^{-10}. Evaluate the sum of these terms and verify that the desired accuracy is achieved.

5 Write a MATLAB m-file to compute the natural logarithm function using n terms of the approximation in Example 4. Graph the error in this approximation to $\ln x$ for $x \in (0, 2.5)$ using 8 terms.

6 The erf function, or 'error function', defined by

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp\left(-t^2\right) dt$$

is an important function in statistics. Derive the series expansion

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)k!}$$

Use the first 10 terms of this series to obtain a graph of this function over the interval $[0, 4]$. Compare this with the built-in erf function.

3.3 CORDIC algorithms

In this section, we concentrate on a topic which is in many ways a modern equivalent of interpolation in tables of logarithmic or trigonometric functions. Interpolation methods are still needed for special functions, and as a basis for numerical integration and the solution of differential equations. These aspects will be discussed in Chapters 4–6. The elementary functions such as $\ln, \exp, \sin$ and $\cos$ *are* available at the touch of a button on your calculator or computer.

Perhaps the most surprising aspect of this is that all these functions are computed on most calculators and PCs by minor variations of the same algorithm. The so-called *CORDIC* (*CO*ordinate *R*otation *DI*gital *C*omputer) algorithms were first developed by Volder for solving trigonometric problems in in-flight navigational computers. (In mathematics, it is usual that methods or theorems bear the names of their discoverers, and so perhaps these algorithms should be known as Volder methods. The use of the acronym CORDIC is testament to their having been discovered by an engineer.)

In this section, we shall concern ourselves only with binary versions of the CORDIC algorithms. Hand calculators typically use decimal-based versions but the extra detail of the decimal form of CORDIC algorithms does not enhance the understanding of the underlying ideas. To get a basic idea, we begin by observing that the long multiplication algorithm for binary integers is particularly simple, consisting solely of shifts and additions.

For example consider the multiplication 73×47. The decimal calculation requires 4 single-digit multiplications (each yielding a 2-digit result) with 2 carries and then a further addition with another carry. As a binary operation

$$
\begin{array}{rr}
73 \times 47 = & 1001001 \\
 & \times\ 101111 \\
\hline
 & 1001001 \\
 & 10010010 \\
 & 100100100 \\
 & 1001001000 \\
 & 100100100000 \\
\hline
 & 110101100111
\end{array}
$$

Each of these terms is just 1001001 shifted the appropriate number of places.

The principal objective of the CORDIC algorithm is to achieve a similar level of simplicity for the elementary functions. All the CORDIC algorithms are based on an ingenious decomposition of the argument and/or the required answer in terms of simple constants, so that only additions and exponent shifts are needed. The following theorem lies at the heart of the matter.

Theorem 1 *Suppose that the numbers* σ_k $(k = 0, 1, \ldots, n)$ *are positive, decreasing and that*

$$\sigma_k \leq \sum_{j=k+1}^{n} \sigma_j + \sigma_n \tag{3.11}$$

Suppose too that

$$|r| \leq \sum_{j=0}^{n} \sigma_j + \sigma_n \tag{3.12}$$

Then the sequence defined by $s_0 = 0$ *and*

$$s_{k+1} = s_k + \delta_k \sigma_k \qquad (k = 0, 1, \ldots, n)$$

where $\delta_k = \operatorname{sgn}(r - s_k)$ *satisfies, for each k,*

$$|r - s_k| \leq \sum_{j=k}^{n} \sigma_j + \sigma_n \tag{3.13}$$

In particular, s_{n+1} *approximates r with error bounded by* σ_n*, that is,*

$$|r - s_{n+1}| \leq \sigma_n$$

Remark 1 Note that sgn is the usual signum function defined by

$$\operatorname{sgn}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Proof The proof is by induction. First, we have

$$|r - s_0| = |r| \leq \sum_{j=0}^{n} \sigma_j + \sigma_n$$

so that (3.13) holds for $k = 0$.

Now, assuming the result holds for some value of k, and noting that δ_k is chosen to have the same sign as $|r - s_k|$, we obtain

$$|r - s_{k+1}| = |r - s_k - \delta_k \sigma_k| = ||r - s_k| - \sigma_k|$$

Then, by (3.11) and the induction hypothesis, we deduce that

$$-\left(\sum_{j=k+1}^{n} \sigma_j + \sigma_n\right) \leq -\sigma_k \leq |r - s_k| - \sigma_k$$

$$\leq \left(\sum_{j=k}^{n} \sigma_j + \sigma_n\right) - \sigma_k = \sum_{j=k+1}^{n} \sigma_j + \sigma_n$$

as required. ■

Theorem 1 shows that, if we choose a positive decreasing sequence $\sigma_0, \sigma_1, \ldots, \sigma_n$ to satisfy (3.11), then any number in the interval $[-E, E]$ where $E = \sum_{j=0}^{n} \sigma_j + \sigma_n$ can be written in the form $\pm\sigma_0 \pm \sigma_1 \pm \ldots \pm \sigma_n$ with an error no worse than σ_n.

Example 6 **The values $\sigma_k = 2^{-k}$ satisfy the conditions of Theorem 1, and so for any $r \in [-2, 2]$, we can write**

$$r \approx \pm 1 \pm \frac{1}{2} \pm \frac{1}{4} \pm \cdots \pm \frac{1}{2^n}$$

with error less than 2^{-n}. Find this decomposition of 1.2345 using $n = 5$

We set $s_0 = 0$ and choose $\delta_k = \operatorname{sgn}(r - s_k)$. We get

$\delta_0 = +1$	$s_1 = 0 + 1(2^0) = 1$	$\lvert r - s_1\rvert = 0.2345$	$< \sigma_0 = 1$
$\delta_1 = +1$	$s_2 = 1 + 1(2^{-1}) = 1.5$	$\lvert r - s_2\rvert = 0.265\,5$	$< \sigma_1 = 1/2$
$\delta_2 = -1$	$s_3 = 1.5 - 1(2^{-2}) = 1.25$	$\lvert r - s_3\rvert = 0.0155$	$< \sigma_2 = 1/4$
$\delta_3 = -1$	$s_4 = 1.25 - 1(2^{-3}) = 1.125$	$\lvert r - s_4\rvert = 0.109\,5$	$< \sigma_3 = 1/8$
$\delta_4 = +1$	$s_5 = 1.125 + 1(2^{-4}) = 1.187\,5$	$\lvert r - s_5\rvert = 0.047$	$< \sigma_4 = 1/16$
$\delta_5 = +1$	$s_6 = 1.187\,5 + 1(2^{-5}) = 1.218\,75$	$\lvert r - s_6\rvert = 0.0157\,5$	$< \sigma_5 = 1/32$

Clearly, this process could be continued to any desired accuracy. It is important to observe that the convergence is not monotone – even the absolute values of the errors can increase in individual steps as we see in steps 2 and 4 of Example 6.

The following general algorithm, with appropriate choices for the parameters m and σ_k, yields approximations to a wide class of elementary functions.

Algorithm 1 General CORDIC Algorithm

Inputs Starting values x_0, y_0, z_0
Parameter $m = -1$, 0, or $+1$, and corresponding sequence (σ_k)
Mode: either *rotation mode* or *vectoring mode*

Compute three sequences:

$$x_{k+1} = x_k - m\delta_k y_k 2^{-k} \tag{3.14}$$

$$y_{k+1} = y_k + \delta_k x_k 2^{-k} \tag{3.15}$$

$$z_{k+1} = z_k - \delta_k \sigma_k \tag{3.16}$$

where

$$\delta_k = \begin{cases} \operatorname{sgn}(z_k) \text{ for rotation mode} \\ -\operatorname{sgn}(y_k) \text{ for vectoring mode} \end{cases} \tag{3.17}$$

The value of m depends on the operation to be performed: $m = 0$ for arithmetic operations, $m = 1$ for trigonometric functions and $m = -1$ for hyperbolic functions. Details of the choice of the various parameters and modes for particular functions are summarized in Tables 3.1–3.3 below.

The names of the 2 modes are a historical accident owing to the development of the algorithms for navigational purposes. They are not of great value in understanding the methods.

Example 7 CORDIC Division

For multiplication and division we use $m = 0$ and $\sigma_k = 2^{-k}$ which we have already seen satisfy (3.11). Division is performed using the vectoring mode: with $z_0 = 0$, we find that

$$|z_{n+1} - y_0/x_0| < 2^{-n}$$

provided that the quotient $y_0/x_0 \in [-2, 2]$. To see this, we note first that, since $m = 0$, $x_k = x_0$ for every k, the algorithm reduces to just the 2 equations

$$y_{k+1} = y_k + \delta_k x_0 2^{-k}$$
$$z_{k+1} = z_k - \delta_k 2^{-k}$$

with $\delta_k = -\operatorname{sgn}(y_k)$.

In this case therefore (3.15) represents the decomposition

$$y_0 \approx -\sum_{k=0}^{n} \delta_k x_0 2^{-k}$$

or, equivalently,

$$\frac{y_0}{z_0} \approx -\sum_{k=0}^{n} \delta_k 2^{-k}$$

with error less than 2^{-n}. However, from (3.16) we see that $z_{n+1} = -\sum_{k=0}^{n} \delta_k 2^{-k}$.

As a specific example, consider the division $1.2/2.3$. Then $x_0 = 2.3$ and

$y_0 = 1.2$	$z_0 = 0$	$\delta_0 = -\text{sgn}(1.2) = -1$
$y_1 = 1.2 - (2.3)2^0 = -1.1$	$z_1 = 0 - (-1)2^0 = 1$	$\delta_1 = -\text{sgn}(-1.1) = +1$
$y_2 = -1.1 + (2.3)2^{-1} = 0.05$	$z_2 = 1 - (1)2^{-1} = 0.5$	$\delta_2 = -1$
$y_3 = 0.05 - (2.3)2^{-2} = -0.525$	$z_3 = 0.5 + 2^{-2} = 0.75$	$\delta_3 = +1$
$y_4 = -0.525 + (2.3)2^{-3} = -0.237\,5$	$z_4 = 0.75 - 2^{-3} = 0.625$	$\delta_4 = +1$
$y_5 = -0.09375$	$z_5 = 0.5625$	$\delta_5 = +1$
$y_6 = -0.021875$	$z_6 = 0.53125$	$\vdots$

The values of y_k are being driven towards 0 by the choice of the signs, and z_k is approaching $1.2/2.3 = 0.52174$ to 5 decimals. The error in z_6 is about 0.01 which is indeed smaller than $2^{-5} = 0.03125$.

We see that the most complicated operation involved here is the calculation of y_k, which involves no more than a binary shift of x_0 and the addition or subtraction of this quantity from the previous value. The algorithm entails no more than shifts and adds, which was the objective behind the development of CORDIC algorithms.

Multiplication can be performed using the same CORDIC algorithm in rotation mode. Setting $y_0 = 0$, we obtain $y_{n+1} \approx x_0 z_0$ with error bounded by $x_0 2^{-n}$. The details are left to the exercises.

Thus far, we have paid little attention to the condition (3.11) of Theorem 1, which for multiplication and division states that the decomposition works for

$$|r| \leq \sum_{k=0}^{n} 2^{-k} + 2^{-n} = 2$$

For division, this imposes the requirement that $|y_0/x_0| \leq 2$ which is automatically satisfied for (the mantissas of) normalized binary floating-point numbers which lie in $[1, 2)$. Similarly for floating-point multiplication, both operands are within the appropriate range. With n chosen appropriately, and with $x_0 \in [1, 2)$ the error bound for multiplication also guarantees the correct *relative* precision in the product.

The use of CORDIC algorithms for multiplication and division is summarized in Table 3.1. Here, as in Tables 3.2 and 3.3 for trigonometric and hyperbolic functions

Table 3.1 CORDIC algorithms for arithmetic operations

$m = 0$, $\sigma_k = 2^{-k}$ for $k = 0, 1, \ldots, n$

Function	*Mode*	*Initial values*	*Output*	*Error bound*	*Useful domain*
$*$	R	$y_0 = 0$	$y_{n+1} \approx x_0 z_0$	$\lvert x_0 \rvert 2^{-n}$	$\lvert x_0 \rvert, \lvert z_0 \rvert \leq 2$
$/$	V	$z_0 = 0$	$z_{n+1} \approx y_0 / x_0$	2^{-n}	$\lvert y_0 / x_0 \rvert \leq 2$

(pp. 68 and 72), R and V represent the Rotation and Vectoring modes, respectively. In each of the tables, a 'useful domain' is quoted for each operation. This is not necessarily the complete domain of convergence for the algorithm but it indicates a useful practical domain for which convergence can be established.

Example 8 CORDIC trigonometric functions

For the trigonometric functions, we use $m = 1$ and take

$$\sigma_k = \arctan 2^{-k} \qquad (k = 0, 1, \ldots, n)$$

The fact that this sequence satisfies (3.11) can be established by an application of the Mean Value Theorem.

The rotation mode provides a technique for computing $\sin\theta$ and $\cos\theta$ by setting $z_0 = \theta$ and decomposing this as $\sum \delta_k \sigma_k$. Writing $s_k = \theta - z_k$, it follows from (3.16) that

$$s_{k+1} = s_k + \delta_k \sigma_k$$

Hence, using the facts that sine and cosine are odd and even functions, respectively, we have

$$\begin{aligned} \cos(s_{k+1}) &= \cos(s_k + \delta_k \sigma_k) \\ &= \cos(s_k)\cos(\delta_k \sigma_k) - \sin(s_k)\sin(\delta_k \sigma_k) \\ &= \cos(s_k)\cos(\sigma_k) - \delta_k \sin(s_k)\sin(\sigma_k) \end{aligned}$$

and, similarly,

$$\sin(s_{k+1}) = \sin(s_k)\cos(\sigma_k) + \delta_k \cos(s_k)\sin(\sigma_k)$$

Dividing both these equations by $\cos(\sigma_k)$ and observing that $\tan(\sigma_k) = 2^{-k}$, we now obtain

$$\frac{\cos(s_{k+1})}{\cos(\sigma_k)} = \cos(s_k) - \delta_k 2^{-k} \sin(s_k) \tag{3.18}$$

$$\frac{\sin(s_{k+1})}{\cos(\sigma_k)} = \sin(s_k) + \delta_k 2^{-k} \cos(s_k) \tag{3.19}$$

From Theorem 1 it follows that $|z_{n+1}| = |s_{n+1} - \theta| \leq \sigma_n$. Apart from the divisor $\cos(\sigma_k)$, (3.18) and (3.19) resemble (3.14) and (3.15) with $m = 1$, $x_k = \cos(s_k)$, and $y_k = \sin(s_k)$. The factors $\cos(\sigma_k)$ and their product

$$K_T = \prod_{k=0}^{n} \cos(\sigma_k)$$

can be precomputed. The initial values of x_k and y_k can be premultiplied by this constant K_T. That is, we set

$$x_0 = K_T \cos(s_0) = K_T \cos(0) = K_T$$
$$y_0 = K_T \sin(s_0) = K_T \sin(0) = 0$$

The effect is that after the $n + 1$ steps are completed, we have

$$x_{n+1} \approx \cos\theta, \quad y_{n+1} \approx \sin\theta$$

each with error less than 2^{-n}.

To illustrate this algorithm we shall compute $\sin(1)$ and $\cos(1)$ using just 4 steps ($n = 3$). In this case we use

$$\sigma_0 = \arctan 1 = 0.7854$$
$$\sigma_1 = \arctan(1/2) = 0.4636$$
$$\sigma_2 = \arctan(1/4) = 0.2450$$
$$\sigma_3 = \arctan(1/8) = 0.1244$$

and

$$K_T = \cos(\sigma_0)\cos(\sigma_1)\cos(\sigma_2)\cos(\sigma_3) = 0.6088$$

Then, with $\delta_k = \text{sgn}(z_k), m = 1$ we get, using (3.14)–(3.16)

$$\begin{array}{llll} x_0 = 0.6088 & y_0 = 0 & z_0 = 1 & \delta_0 = +1 \\ x_1 = 0.6088 & y_1 = 0.6088 & z_1 = 0.2146 & \delta_1 = +1 \\ x_2 = 0.3044 & y_2 = 0.9132 & z_2 = -0.2490 & \delta_2 = -1 \\ x_3 = 0.5327 & y_3 = 0.8371 & z_3 = -0.0040 & \delta_3 = -1 \\ x_4 = 0.6373 & y_4 = 0.7705 & z_4 = 0.1204 & \delta_4 = +1 \end{array}$$

from which we deduce that $\cos 1 \approx 0.6373$ and $\sin 1 \approx 0.7705$, each with error less than $2^{-3} = 0.125$. (The true values are 0.5403 and 0.8415.)

Note that, as with the division algorithm, the errors are not necessarily reduced at each iteration. The compensating advantage is that we know *in advance* the exact number of steps that are needed to achieve a specified accuracy. In the case of the trigonometric functions, we should also note that it is not until the completion of the predetermined number of steps that we really have approximations to $\cos\theta$ and $\sin\theta$ because of the initial scaling by K_T which depends on the number of steps to be used.

Program **MATLAB m-file for sin and cos using CORDIC algorithm with 40 steps**

```
function y=Cordictrig(z)
% Computes cos and sin of z using 40 steps of CORDIC algorithm
% y(1), y(2) are approximations of cos(z), sin(z) respectively
s=2.^-(0:39); sig=atan(s);
KT=prod(cos(sig));
x1=KT; y1=0;
for k=1:40
   x0=x1; y0=y1;
   if z>=0, del=1; else del=-1; end
   x1=x0-del*y0*s(k);
   y1=y0+del*x0*s(k);
   z=z-del*sig(k);
end
y(1)=x1; y(2)=y1;
```

The results from this m-file should be accurate to within 2^{-39}. As an example, the command

```
» cs=cordictrig(1)
```

yields the result

```
cs =
0.540302305868555   0.841470984807631
```

which each have errors smaller than 2^{-41} as can be verified using

```
» log2(abs([cos(1)-cs(1),sin(1)-cs(2)]))
ans =
   −41.1327212602903   −41.7753953184615
```

The vectoring mode of the trigonometric CORDIC provides algorithms for both the inverse tangent function and for the magnitude of a 2-dimensional vector. Specifically, with $z_0 = 0$, we can use it to compute

$$z_{n+1} \approx \arctan(y_0/x_0)$$

and

$$x_{n+1} \approx \sqrt{x_0^2 + y_0^2}/K_T$$

If the initial values x_0, y_0 are scaled by K_T, the arctangent value is unchanged and the square root becomes just $\sqrt{x_0^2 + y_0^2}$, which is to say we have precisely the appropriate output for conversion between Cartesian and polar coordinates in the plane.

Remark 2 It may appear from the program above that this algorithm requires knowledge of both arctan and cos in order to compute these functions. However, we should note that in a hardware implementation only those special values σ_k and $\cos\sigma_k$ are needed and these would be precomputed and stored on the chip.

As with the arithmetic functions, we must consider the intervals of convergence for these CORDIC trigonometric functions. Since the angle θ is decomposed as $\sum \delta_k\sigma_k$, this algorithm will be applicable for any $|\theta| \leq \sum \sigma_k + \sigma_n \approx 1.74$ which is greater than $\pi/2$ so that range reduction could be implemented to allow the CORDIC algorithm to compute any value of these functions. Essentially this involves subtracting the appropriate integer multiple of 2π and then adjusting the answers for the correct quadrant. For arguments outside a moderate interval, this range reduction is not trivial. We do not concern ourselves with the details here.

The complete domain of the arctangent can be covered since, using $x_0, y_0 \in [-2, 2]$, the range of values of y_0/x_0 spans the whole real line. For the geometric coordinate transformation, this range remains sufficient since we can scale both coordinates by an appropriate binary exponent. As before we summarize the trigonometric CORDIC algorithms in a table (Table 3.2).

Table 3.2 CORDIC algorithms for trigonometric functions

$m = 1$, $\sigma_k = \arctan 2^{-k}$ for $k = 0, 1, \ldots, n$, $K_T = \prod_{k=0}^{n} \cos(\sigma_k)$

Function	*Mode*	*Initial values*	*Output*	*Useful domain*
cos, sin	R	$x_0 = K_T$ $y_0 = 0$	$x_{n+1} \approx \cos z_0$ $y_{n+1} \approx \sin z_0$	$\lvert x_0\rvert, \lvert z_0\rvert \leq \pi/2$
arctan	V	$z_0 = 0$	$z_{n+1} \approx \arctan(y_0/x_0)$	$\lvert x_0\rvert, \lvert y_0\rvert \leq 2$
$\lvert\lvert\cdot\rvert\rvert$	V	$x_0 = xK_T$ $y_0 = yK_T$	$x_{n+1} \approx \sqrt{x^2 + y^2}$	$\lvert x\rvert, \lvert y\rvert \leq 2$

Example 9 Hyperbolic, exponential and logarithmic functions

The CORDIC algorithms for these functions are very similar to those for the trigonometric functions. This time we take $m = -1$ and $\sigma_k = \tanh^{-1} 2^{-k}$ for $k \geq 1$. Equations similar to (3.18) and (3.19) can be derived from the corresponding identities for the hyperbolic functions. There is one very important difference, however. The quantities σ_k just defined *do not* satisfy

condition (3.11). It can be shown, however, that, if the steps for $k = 4, 13, 40, \ldots$, or, in general, $k = (3^j - 1)/2$, are repeated then all the conditions of Theorem 1 are satisfied. Corresponding to the quantity K_T used for the trigonometric functions, this time we take

$$K_H = \prod \cosh \sigma_k$$

where the product includes repetitions of the appropriate factors.

In the rotation mode if $x_1 = K_H$ and $y_1 = 0$ then

$$x_{n+1} \approx \cosh z_1$$
$$y_{n+1} \approx \sinh z_1$$

each with error smaller than 2^{-n}. From these we can obtain the exponential function since

$$e^{z_1} = \cosh z_1 + \sinh z_1$$

With $z_1 = 0$, the vectoring mode can be used to obtain

$$\begin{aligned} z_{n+1} &\approx \tanh^{-1}(y_1/x_1) \\ &= \frac{1}{2} \ln w \end{aligned} \tag{3.20}$$

if $x_1 = w + 1,\ y_1 = w - 1$. Also

$$x_{n+1} \approx \frac{\sqrt{x_1^2 - y_1^2}}{K_H} = \frac{\sqrt{w}}{K_H} \tag{3.21}$$

if $x_1 = w + 1/4,\ y_1 = w - 1/4$.

We illustrate these algorithms with the estimation of $e^{0.2}$ using $n = 5$. The basic algorithm can be simplified somewhat for the exponential function. Equations (3.14) and (3.15) with $m = -1$ become

$$x_{k+1} = x_k + \delta_k y_k 2^{-k}$$
$$y_{k+1} = y_k + \delta_k x_k 2^{-k}$$

and writing $u_k = x_k + y_k$ we obtain the single equation

$$u_{k+1} = u_k + \delta_k u_k 2^{-k} \tag{3.22}$$

Now using $n = 5$, which corresponds to 6 steps, and allowing for the repetition for $k = 4$, we use

$$\sigma_1 = \tanh^{-1} 2^{-1} = 0.5493,\ \sigma_2 = 0.2554,$$
$$\sigma_3 = 0.1257,\ \sigma_4 = 0.0626,\ \sigma_5 = 0.0626$$

from which we have

$$\begin{aligned} K_H &= \cosh(0.5493) \cosh(0.2554) \cosh(0.1257) \cosh^2(0.0626) \\ &= 1.2067 \end{aligned}$$

Then we obtain

$$\begin{array}{lll} u_1 = 1.2067 & z_1 = 0.2000 & \delta_1 = +1 \\ u_2 = 1.8101 & z_2 = -0.3493 & \delta_2 = -1 \\ u_3 = 1.3576 & z_3 = -0.0939 & \delta_3 = -1 \\ u_4 = 1.1879 & z_4 = 0.0318 & \delta_4 = +1 \\ u_5 = 1.2621 & z_5 = -0.0308 & \delta_5 = -1 \\ u_6 = 1.1832 & z_6 = 0.0318 & \delta_6 = +1 \end{array}$$

Hence, we obtain $e^{0.2} \approx u_7 = 1.1832 + (1.1832)2^{-5} = 1.2202$, which should be compared with the true value 1.2214.

In the following program, the set of steps that are repeated includes $k = 1$. This has the benefit of increasing the range of applicability of the algorithm. This program implements the vectoring mode. With careful choice of the inputs, the outputs yield not only the inverse hyperbolic tangent but also the natural logarithm and square-root functions.

Program

MATLAB m-file for $\tanh^{-1}$ using CORDIC algorithm with 40 steps plus repetitions

```
function out=cordichypv(x,y)
% Cordic algorithm for hyperbolic functions
% Vectoring mode, 40 steps plus repetitions for 1,4,13,40
v=[1,1:4,4:13,13:40,40];
% Note this takes account of the repetitions
s=2.^-v;
sig=atanh(s);
KH=prod(cosh(sig));
z=0; x1=x; y1=y;
for k=1:44
  x0=x1; y0=y1;
  if y0>=0, del=-1; else del=1; end
  x1=x0+del*y0*s(k);
  y1=y0+del*x0*s(k);
  z=z-del*sig(k);
end
out=[x1,y1,z];
```

For example, the command

```
» v=cordichypv(2,1)
```

yields the output vector

```
v = 1.24223904144032, 9.7878783548196e-014, 0.549306144333976
```

the third element of which is the approximation to $\tanh^{-1}(1/2)$ which has true value 0.549306144334055. The error is about 10^{-13}.

The second element of the output merely confirms that the CORDIC iterations have forced y_k to approach 0. The first element is the final x_{n+1} value which should be close to $\sqrt{2^2-1^2}/K_H = \sqrt{3}/1.39429751423739 = 1.242\,239$ to 6 decimals.

Equations (3.20) and (3.21) show how to modify the initial values to obtain other desired outputs. For example, to get $\ln 2$, we must choose $x_1 = 2+1 = 3$, $y_1 = 2-1 = 1$. This will result in $z_{n+1} \approx (1/2)\ln 3$ and so the third element of the output vector must be doubled:

```
» v=cordichypv(3,1);
» ln2=2*v(3)
ln2 =
   0.69314718056046
```

which should be compared with the true value 0.693147180559945.

The convergence domain for the hyperbolic functions with the repetitions used is $\sum \sigma_k \approx 1.74$ so that $\cosh z$, $\sinh z$ and $\exp z$ may be computed for $|z| \leq 1.74$. This range can be extended in a variety of ways. One convenient method is to write larger arguments in the form

$$z = z_1 + p\ln 2$$

where p is an integer chosen so that $z_1 \in [0, \ln 2)$. Then z_1 is in the range of applicability and we can then use

$$e^z = e^{p\ln 2}e^{z_1} = 2^p e^{z_1}$$

which will be the normalized binary floating-point representation. The error in the value of e^{z_1} will be 2^{-n+1}, where n is the number of steps used. It is therefore easy to determine in advance the number of steps needed for any particular floating-point format. This number of steps remains fixed for all arguments.

The CORDIC algorithms for the hyperbolic functions are summarized in Table 3.3.

It is necessary here to give a word of warning about the possibility of meaningless computation. As an illustration, suppose that a hypothetical calculator works to 7 decimal digits so that a number, A, is represented as $a \times 10^{\alpha}$ with $1 \leq a < 10$. The difference between A and the next representable number is then $10^{\alpha-6}$ which is certainly greater than 2π whenever $\alpha \geq 7$. To try to give a specific value to, say, $\cos A$ is then plainly meaningless since more than one complete period of the cosine function would share the same representation. Nonetheless most computers and calculators will attribute a specific value to $\cos A$, which emphasizes the point that *any* output from a computer or calculator should be treated with suspicion until the situation has been analyzed carefully.

Before leaving the subject of approximate evaluation of functions, it should be stressed that, although many practical algorithms are based on the ideas presented here, these are by no means the only ones available. The different routines used in various computers provide ample testimony to the variety and blend of approaches

Table 3.3 CORDIC algorithms for hyperbolic functions

$m = -1$, $\sigma_k = \tanh^{-1} 2^{-k}$ for $k = 1, 2, \ldots, n$, $K_H = \prod \cosh(\sigma_k)$ with repetitions for $k = 1, 4, 13, 40, \ldots$

Function	*Mode*	*Initial values*	*Output*	*Useful domain*
cosh sinh exp	R	$x_0 = K_H$ $y_0 = 0$	$x_{n+1} \approx \cosh z_0$ $y_{n+1} \approx \sinh z_0$ $x_{n+1} + y_{n+1} \approx e^{z_0}$	$\lvert x_0\rvert, \lvert y_0\rvert \leq 1.7$
$\tanh^{-1}$	V	$z_1 = 0$	$z_{n+1} \approx \tanh^{-1}(y_1/x_1)$	$\lvert y_1\rvert < \lvert x_1\rvert$ $\lvert y_1\rvert \leq 2$
$\ln w$	V	$x_1 = w + 1$ $y_1 = w - 1$	$z_{n+1} \approx \frac{1}{2} \ln w$	$1/2 \leq w \leq 2$
$\sqrt{w}$	V	$x_1 = K_H(w + 1/4)$ $y_1 = K_H(w - 1/4)$	$x_{n+1} \approx \sqrt{w}$	$1 \leq w \leq 4$

which may be useful in different circumstances. The interpolation-based methods of Chapter 4 also play an important role, as do others that are beyond our present objectives.

Exercises: Section 3.3

1 Approximate 0.12345 in the form $\pm 1 \pm 1/2 \pm \cdots \pm 1/32$. Verify that the error satisfies the bound given in Theorem 1.

2 Show that $\sigma_k = 2^{-k}$ for $k = 0, 1, \ldots, n$ satisfies the condition (3.11) of Theorem 1.

3 Write a MATLAB m-file to obtain the CORDIC decomposition of a number $r \in [-2, 2]$ using $\sigma_k = 2^{-k}$ for $k = 0, 1, \ldots, 40$. Test it for $r = \pm 0.12345$ and ± 1.2345 and verify that the error bounds satisfy (3.13).

4 Use the CORDIC multiplication algorithm to compute 1.23×1.12 with error less than 2^{-7}.

5 Show that $\sigma_k = \arctan 2^{-k}$ for $k = 0, 1, \ldots, n$ satisfies condition (3.11) of Theorem 1. Use the CORDIC algorithm to compute $\sin(0.5)$ and $\cos(0.5)$ with $n = 5$.

6 Write a MATLAB m-file to compute the vectoring mode of the trigonometric CORDIC algorithm using 40 steps. Use it to convert the Cartesian coordinates $(2, 1)$ to plane polar coordinates.

7 Derive a simplified CORDIC scheme for evaluating the function e^{-z}. Use this algorithm with $n = 6$ to approximate $e^{-0.25}$.

8 Use the CORDIC algorithm to approximate $\ln 1.5$ using $n = 6$.

9 Write an m-file for the rotation mode of the CORDIC algorithm for hyperbolic functions using $n = 40$. (Don't forget the repeated steps!) Use this to evaluate $\sinh x$ for $x = -1.5 : 0.1 : 1.5$. Graph the error function for these values and verify that the errors are appropriately bounded.

3.4 MATLAB functions

MATLAB has algorithms built in for all the standard 'elementary' mathematical functions, such as the trigonometric functions and their inverses, the exponential function, the natural logarithm and the hyperbolic functions and their inverses.

Their names are usually the expected ones with the convention that the inverse functions are prefixed with the letter a so that atan is MATLAB's arctan function. The syntax is also as you would expect. Some of the basic ones are included in Table 3.4. All the standard MATLAB functions can be applied elementwise to vectors and matrices – a fact which is particularly helpful for graphics.

Table 3.4 MATLAB elementary functions

Mathematical notation	*MATLAB*	*Mathematical notation*	*MATLAB*
$\sin x$	sin(x)	$e^x = \exp(x)$	exp(x)
$\cos x$	cos(x)	$\ln x$	log(x)
$\tan x$	tan(x)	$\log_{10} x$	log10(x)
$\arctan x$	atan(x)	$\log_2 x$	log2(x)
$\arcsin x$	asin(x)	$\cosh x$	cosh(x)
$\sqrt{x}$	sqrt(x)	$\sinh x$	sinh(x)
$\lvert x \rvert$	abs(x)	$\tanh^{-1} x$	atanh(x)

Modern PCs can evaluate most of these in hardware using algorithms similar to those described in this chapter. However, MATLAB *does not* use these hardware functions because its results should be the same independent of the hardware platform being used. (Most UNIX workstations do not have the ability to compute these in hardware on their RISC (Reduced Instruction Set Computer) chips.) MATLAB employs very efficient software code for these functions.

In addition to these 'elementary' functions, MATLAB has built-in m-files for many other 'special' functions. These include the erf function (which was introduced in the Exercises to Section 3.2), the beta and gamma functions, the Bessel functions and many others. Again the syntax is much as would be expected for these different functions. You can check the details in your MATLAB documentation when you need to use any of these functions.