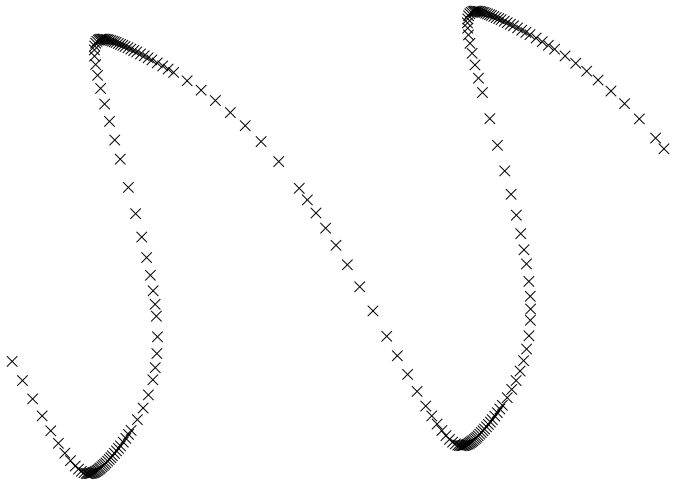
Lars Eldén Linde Wittmeyer-Koch Hans Bruun Nielsen

Introduction to

Numerical Computation

– analysis and MATLAB® illustrations



June 2004

Contents

Preface

This book is intended as a textbook for an introductory course in scientific computation at an advanced undergraduate level. The mathematical prerequisites should be covered by a first-year calculus and algebra course. The book is a translation and revision of a corresponding book by the first two authors, published in Swedish, *Numeriska beräkningar – analys och illustrationer med MATLAB®*.

The aim of the book is to give an introduction to some of the basic ideas in numerical analysis. We cover a comparatively wide range of material, including classical algorithms for the solution of nonlinear equations and linear systems, and methods for interpolation, integration and approximation. We also give an introduction to some areas, which we think are important in the applications that a science or engineering student will meet. These areas include floating point computer arithmetic and standard functions, splines, finite elements and discrete cosine transform. In a few areas (approximation and linear systems of equations) we have chosen to give a somewhat more comprehensive treatment.

The presentation is heavily influenced by the development of computers and mathematical software. In an environment like MATLAB[1)] it is possible by simple commands to perform advanced calculations on a personal computer. This has reduced the general need for students to learn about algorithmic details, but it maybe has increased the need to learn about properties of the algorithms. With the ready availability of advanced software one can easily solve large problems, and in order to be able to choose the right algorithm, one must know about the complexity of the algorithms (the execution time and storage demand) and the possibility of exploiting structure, eg a band matrix.

[1)] MATLAB® is a registered trade mark of The MathWorks, Inc. We use MATLAB 6.5 in the examples.

In our opinion an introductory course should emphasize the understanding of methods and algorithms. In order to really grasp what is involved in numerical computations, the student must first perform computations using a simple calculator. We supply a number of exercises intended for this. As the title indicates, we illustrate the algorithms in MATLAB. In the implementations we focus on simplicity and readability rather than robustness and efficiency. Therefore, these MATLAB functions should not be considered as genuine mathematical software. To see how much more complicated real mathematical software is, one could compare our `adaptrk45` with MATLABs `ode45` (49 and 616 lines of MATLAB code, respectively).

Some of our MATLAB functions are collected in a toolbox incbox, which is available from the authors' homepages together with other supplementary material like, eg, a table of formulas.

We wish to thank our colleagues at the Department of Mathematics, Linköping University and the department of Informatics and Mathematical Modelling, Technical University of Denmark, for their help and support during the preparation of the manuscript.

Linköping, June 2004

Lars Eldén
www.math.liu.se/∼laeld/

Linde Wittmeyer-Koch
www.math.liu.se/∼liwit/

Lyngby, June 2004

Hans Bruun Nielsen
www.imm.dtu.dk/∼hbn/

Chapter 1

Introduction

1.1. Mathematical Models and Numerical Approximations

Mathematical models are basic tools in scientific problem solving. Typically, some fundamental laws of nature are used to derive one or more equations that model the phenomenon under study. Questions that are posed in connection with the problem area may be answered through mathematical treatment of the equations. This is illustrated schematically in Figure 1.1.

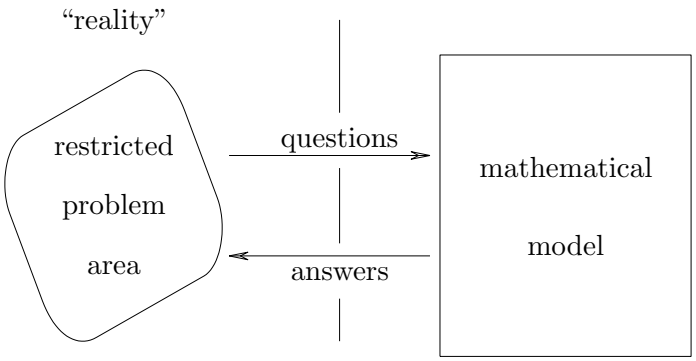


Figure 1.1. *Mathematical model.*

It is important to remember that the problem area described by the mathematical model may be very narrow. Further, there are often simplifying assumptions. Therefore, the mathematical model is not an exact description of reality, and the answers that the model produces must be checked and compared with experimental results.

Example. Assume that we throw a ball vertically up with velocity v_0. How high does the ball get?

We choose a simple mathematical model, where gravity is the only force acting on the ball. Let y denote the height of the ball above the ground, and let v denote its velocity. Both are functions of time t since the ball was thrown.

According to Newton's second law of motion we have $\dfrac{dv}{dt} = -g$, where g is the acceleration of gravity. This gives

$$v(t) = v_0 - gt \ .$$

The ball reaches its summit at time t_1 given by $v(t_1) = 0$,

$$t_1 = \frac{v_0}{g} \ ,$$

and the height y_1 is found by integrating $v(t) = \dfrac{dy}{dt}$:

$$y_1 = \int_0^{t_1} v(t)\, dt = \int_0^{t_1} (v_0 - gt)dt = v_0 t_1 - \tfrac{1}{2} g t_1^2 \ .$$

If the initial velocity is $v_0 = 25\, m/s$ (and $g = 9.81\, m/s^2$), then $t_1 = 2.55\, s$ and $s_1 = 31.9\, m$. ∎

In this example it is realistic to ignore air resistance and also that the acceleration of gravity is a decreasing function of the height. If we want to study the launching of a rocket, the model gets more complicated.

Example. Consider a small rocket (eg for atmospheric research), which is launched vertically[1)] . The rocket weighs 300 kg, including 180 kg fuel. After start the engine consumes 10 kg fuel every second, and gives a vertical thrust of 5000 N. The rocket is further affected by gravitation (with acceleration of gravity $g = 9.81$ m/s^2) and by air resistance $0.1v^2$, where v is the velocity, measured in m/s.

Thus, the mass M, height h and velocity v are functions of time. They satisfy

$$M(t) = 300 - 10t \ ,$$

$$v = \frac{dh}{dt} \ ,$$

and Newton's equation of force, $F = (Mv)' = Mv' + M'v$, or

$$Mv' = 5000 - Mg - 0.1v^2 - M'v \ .$$

[1)] This example is taken from T.J. Akai, *Applied Numerical Methods for Engineers*, Wiley, 1994.

We insert the expression for M and see that we have to solve a system of differential equations

$$\begin{aligned} \frac{dh}{dt} &= v \ , \\ \frac{dv}{dt} &= \frac{5000 - 0.1v^2 + 10v}{300 - 10t} - g \ , \end{aligned} \tag{1.1.1}$$

with the initial condition $h(0) = v(0) = 0$.

Note that the model is valid only for $t \leq 18$. At that time all the fuel has been used, so there is no upward driving force and the mass is constant. ∎

The mathematical model in this example is an *initial value problem* for a system of differential equations: Given the knowledge that $h(0) = v(0) = 0$ we want to determine $h(t)$ and $v(t)$ for $t > 0$. The differential equation is nonlinear in v, and there is no simple way to solve it analytically, ie to find an explicit expression for the solution.

The mathematical model is a good description of the phenomenon under study, but it does not directly help us to answer questions about the behaviour of the rocket.

Essentially we have the choice between two alternatives. Either we can simplify the model so that an analytic solution can be found. For the problem in the example, this is the case if we ignore the air resistance, so that both the unknowns h and v occur linearly in the differential equations. The alternative is to introduce *numerical approximations*.

Example. By means of the MATLAB function `ode45` it is easy to compute approximations to the solution at discrete values of t. First, we define a function that implements the right hand side in the system of differential equations:

```
function  dy = rocket(t,y)
dy = [y(2)
      (5000 - 0.1*y(2)^2 + 10*y(2))/(300 - 10*t) - 9.81];
```

The two components of the vector `y` represent h and v, respectively. Now we call `ode45`

```
>> [t,Y] = ode45(@rocket,[0 18],[0 0]);
```

The second input variable tells that the problem should be solved for $0 \leq t \leq 18$, and the third variable tells that both components of `y` are zero for $t = 0$.

The output is a vector `t` with times and an array `Y` with corresponding values of the two components of `y`. The commands

```
>> plot(t,Y(:,1),'x')
>> xlabel('time'),  ylabel('height')
```

display height as function of time and put text on the axes, see Figure 1.2.

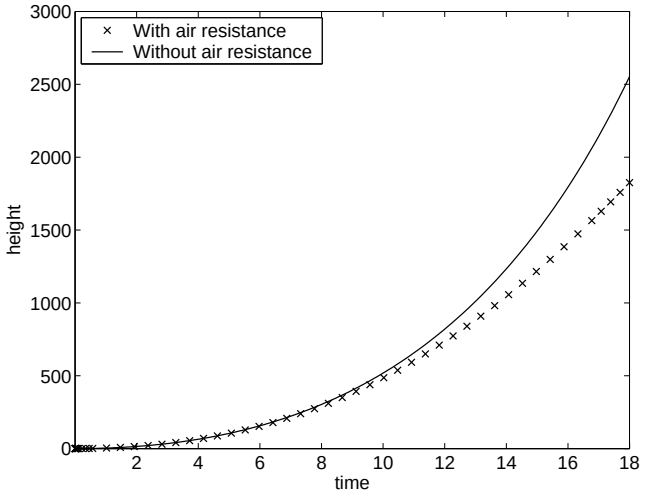


Figure 1.2. *Approximations of the solution to the differential equation* (1.1.1).

For comparison we also show the solution without the air resistance term,[2)]

$$h(t) = (500 - 15g)\left(30 \log \frac{30}{30 - t} - t\right) - \tfrac{1}{4}\, gt^2 \ .$$

■

The example illustrates that if we simplify the model, so that we can solve the mathematical problem explicitly, then the results may change so much that they cannot be used to give a reliable answer to the questions that we want to ask. Errors are also introduced in the numerical solution, but here the answers are more reliable because it is possible to estimate how the approximations affect the accuracy of the solution. The two alternatives are illustrated schematically in Figure 1.3.

In the rocket example the mathematical problem is to determine a function that satisfies the initial value problem. We replaced this by the numerical problem of computing approximations of this function at

[2)] Throughout the book we use $\log x$ to denote the natural logarithm of x.

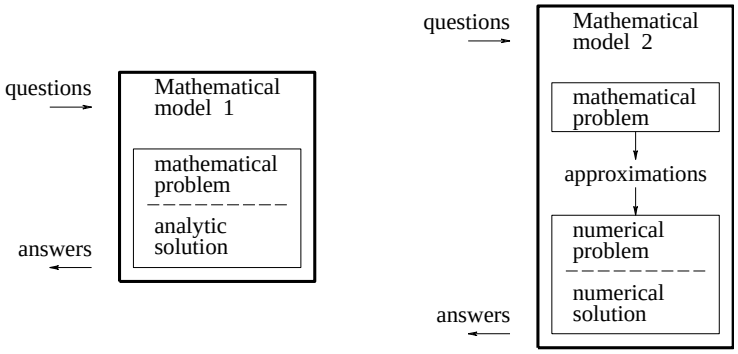


Figure 1.3. *Mathematical model and numerical problem.*

a number of discrete points. We will now make the notion of a numerical problem somewhat more precise.

A *numerical problem* is a clear and unambiguous description of the functional relation between *input data*, ie the "independent variables" of the problem, and *output data*, ie the desired results. Both input data and output data consist of a *finite number* of quantities.

In the rocket example the input data are the initial values ($h(0) = v(0) = 0$), and the output data are the approximate values of the solution at discrete points. In Chapter 10 we describe how the output data are computed. The description is in the form of an algorithm.

An *algorithm* for a numerical problem is a complete description of a finite number of well-defined operations, through which each permissible input data vector is transformed into an output data vector.

In this connection an operation is an arithmetic or logical operation, or it may be a previously defined algorithm. An algorithm may be described loosely or in greater detail. A comprehensive description is obtained when an algorithm is formulated in a programming language.

The objective of *numerical analysis* is to construct and analyze numerical methods and algorithms for the solution of practical computational problems. In connection with the above discussion we can give the following examples of interesting questions in numerical analysis:

- How large is the discretization error when a derivative is replaced by a difference quotient?
- How long does it take to solve a certain numerical problem using a certain algorithm on a certain computer?
- How do the rounding errors of the computer arithmetic influence the accuracy of the solution?

The answer to the first question will sometimes be given in terms of a "*big O*" expression. The notation

$$g(h) = O(h^p) \quad \text{as } h \to 0$$

means that there are numbers p, K and d such that $|g(h)| \leq K \cdot h^p$ for all $|h| \leq d$. Often, we just write $g(h) = O(h^p)$, and presuppose "as $h \to 0$".

The "big O" concept may also be used in answers to the second question: Let n denote the "size" of a numerical problem. We say that the computational work involved in solving the problem is $w(n) = O(n^q)$ if the number of operations used can be expressed as

$$w(n) = c_1 n^q + c_2 n^{q-1} + \cdots + c_{q+1} \ ,$$

where q is a positive integer and $c_1 > 0$. As $n \to \infty$ the first term dominates and $w(n) \simeq c_1 n^q$.

1.2. Numerical Computation

Numerical analysis is a branch of applied mathematics and dates back thousands of years, when man first felt the need to construct a calendar, compute areas and volumes, etc. Short biographies of some of the people contributing to the development can be found in a special chapter at the end of the book.

As a separate discipline, numerical analysis can be considered to start in the 1600s. At that time computation was done by hand, using paper, pen and logarithmic tables. During the 1900s manual and later electrical calculators were developed, but as late as the early 1960s the slide rule was the dominating calculation tool for technical computations. It is interesting to note the large change that has happened during the last 55 years, since the advent of computers. As an example we will mention that in the late 1940s the computer solution of a linear system of equations

with 15 unknowns might take two days. Now, any personal computer performs this task in a fraction of a second. Today a large system of linear equations may have more than 10^5 unknowns, and if the system is structured (eg if most of the elements in the matrix of the system are zero), then a powerful computer can handle problems of dimension 10^7 – 10^8.

Both with simple and advanced computations the programming environment is at least as important as the speed and the memory size of the computer. The first computers were programmed in machine language, ie the computer's own instructions, which were at a hardware level. The 1950s saw the advent of algorithmic programming languages, that allow a more "mathematical" formulation of the algorithms. Such languages are under constant development, taking into account changes in computer architecture[3)], and are still the basis of numerical computations.

Starting in the early 1980s new algorithmic programming languages have been developed, with data types at a higher level and with powerful standard computations built into the language. In this book we shall use MATLAB to illustrate algorithms and show how relatively complicated computations can be made by simple instructions. The presentation is not dependent on the use of MATLAB to illustrate the principles; similar instructions can be found eg in Maple or Matematica. We use MATLAB because it has very high quality and because it is widely used in natural and technical sciences. We want to emphasize that this book is not meant as a textbook in MATLAB. Also the reader who knows another programming language like eg Fortran 90 or C++ should be able to understand the MATLAB programs without great difficulty.

The basic data type in MATLAB (MATrix LABoratory) is a matrix (scalar quantities are matrices of dimension one), and matrix operations like addition and multiplication are part of the language. There are also algorithms for more advanced manipulation of matrices; eg a linear system of equations $Ax = b$ is solved (by means of Gaussian elimination) by the command `x = A\b`. There are many further numerical algorithms available, and it is easy to supply with your own algorithms. In the previous section we used a program for the solution of differential equations

[3)] The first version of Fortran came in the late 1950s. Fortran 90, which was presented around 1990, was developed for object oriented programming. A new and further modernized Fortran standard is planned for release in 2005.

and implemented the code that described our equations.

MATLAB also contains an abundance of algorithms for graphical output[4], for symbolic calculation, a debugger, a report generating system, and a windows system for file handling. As mentioned, it is easy to extend the set of available operations, and there a number of "toolboxes" for different applications. As an example, most of the MATLAB functions discussed in this book are collected in a toolbox named incbox, which can be downloaded from the authors' homepages. In short, it can be said that MATLAB is a programming environment rather than a programming language.

References

The definitions of a numerical problem and an algorithm were taken from

G. Dahlquist, Å. Björck, *Numerical Methods*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

There are many textbooks in MATLAB, eg

D.J. Higham, N.J. Higham, *Matlab Guide*, SIAM, Philadelphia, PA, 2000.

R. Pratap, *Getting Started with MATLAB, Version 6*, Oxford University Press, Oxford, 2002.

[4] Except for Figures 2.3, 3.5, 5.7, 5.15 and 10.16 all the figures in this book were made in MATLAB.

Chapter 2

Error Analysis and Computer Arithmetic

In Chapter 1 we illustrated how approximations are introduced in the solution of mathematical problems that cannot be solved exactly. One of the tasks in *numerical analysis* is to estimate the accuracy of the result of a numerical computation. In this chapter we discuss different sources of error that affect the computed result and we derive methods for error estimation. In particular we discuss some properties of computer arithmetic. Finally, we describe the main features of the standard for floating point arithmetic, which was adopted by IEEE[1)] in 1985.

2.1. Sources of Error

Basically there are three types of error that affect the result of a numerical computation

1. **Errors in the input data** are often unavoidable. The input data may be results of measurements with limited accuracy, or real numbers which must be represented with a fixed number of digits.
2. **Rounding errors** arise when computations are performed using a fixed number of digits.
3. **Truncation errors** arise when "an infinite process is replaced by a finite one", eg when an infinite series is approximated by a partial sum, or when a function is approximated by a straight line.

[1)] Institute for Electrical and Electronics Engineers. Pronounced "I triple E".

Truncation errors will be discussed in connection with different numerical methods. In this chapter we shall examine the other two sources of error.

The different types of error are illustrated in Figure 2.1, which refers to the discussion in Chapter 1.

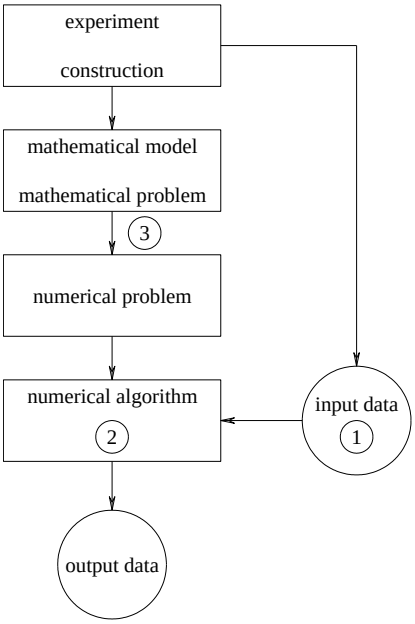


Figure 2.1. *Sources of error.*

We shall use the following notation

R_{X} error in the result, coming from errors in the input data,

R_{XF} error in the result, coming from errors in the function values used,

R_{C} rounding error,

R_{T} truncation error.

The error type R_{XF} is a special case of R_{X}.

2.2. Basic Concepts

Let a denote an *exact value*, and $\overline{a}$ an *approximation* of a, eg

$$a = \sqrt{2}, \qquad \overline{a} = 1.414\ .$$

We introduce the definitions

Absolute error in $\overline{a}$: $\quad \Delta a = \overline{a} - a$.
Relative error in $\overline{a}$: $\quad \dfrac{\Delta a}{a}$, $\quad (a \neq 0)$.

Example. In the above example we have

$$\Delta a = 1.414 - \sqrt{2} = -0.0002135\ldots \quad ,$$

$$\frac{\Delta a}{a} = \frac{-0.0002135\ldots}{\sqrt{2}} = -0.0001510\ldots \quad .$$

■

In many cases we only know a bound on the magnitude of the absolute error of an approximation. Also, it is often practical to give an estimate of the magnitude of the absolute and relative error, even if more information is available.

Example. Continuing with our example we can write

$$|\Delta a| \leq 0.00022 \ , \qquad \left|\frac{\Delta a}{a}\right| \leq 0.00016 \ ,$$

or

$$|\Delta a| \leq 0.0003 \ , \qquad \left|\frac{\Delta a}{a}\right| \leq 0.0002 \ .$$

■

Note that we must always round upwards in order that the inequalities shall hold. Relative errors are often given in percentages; in the last example the error is at most 0.02%.

The following three statements are equivalent

$1^\circ \qquad \overline{a} = 1.414, \quad |\Delta a| \leq 0.22 \cdot 10^{-3}$,

$2^\circ \qquad a = 1.414 \pm 0.22 \cdot 10^{-3}$,

$3^\circ \qquad 1.41378 \leq a \leq 1.41422$.

There are two ways to reduce the number of digits in a numerical value: *rounding* and *chopping*. We first consider rounding of decimal numbers to t digits: Let η denote the part of the number that corresponds to positions to the right of the tth decimal. If $\eta < 0.5 \cdot 10^{-t}$, then the tth decimal is left unchanged and it is raised by 1 if $\eta > 0.5 \cdot 10^{-t}$. In the limit case where $\eta = 0.5 \cdot 10^{-t}$, the tth decimal is raised by one if it is odd and left unchanged if it is even. This is called *rounding to even*. With *chopping* all the decimals after the tth are ignored.

Example. Rounding to 3 decimals:

$$\begin{aligned} 1.23767 &\quad \text{is rounded to} \quad 1.238\ , \\ 0.774501 &\quad \text{is rounded to} \quad 0.775\ , \\ 6.3225 &\quad \text{is rounded to} \quad 6.322\ , \\ 6.3235 &\quad \text{is rounded to} \quad 6.324\ . \end{aligned}$$

Chopping to 3 decimals:

$$0.69999 \quad \text{is chopped to} \quad 0.699\ .$$

■

It is important to remember that errors are introduced when numbers are rounded or chopped. From the above rules we see that when a number is rounded to t decimals, then the error is at most $0.5 \cdot 10^{-t}$, while the chopping error can be 10^{-t}. Note that chopping errors are systematic: the chopped result is always closer to zero than the original number. When an approximate value is rounded or chopped, then the associated error must be added to the error bound.

Example. Let $b = 11.2376 \pm 0.1$. Since the absolute error can be one unit in the first decimal, it is not meaningful to give four decimals, and we round to one decimal, $b_{\text{rd}} = 11.2$. The rounding error is

$$|R_{\text{C}}| = |b_{\text{rd}} - b| = |11.2 - 11.2376| = 0.0376 < 0.04\ .$$

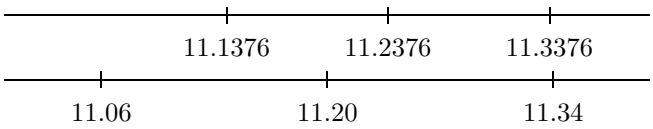
We must add the rounding error to the original error bound,

$$b = 11.2 \pm 0.14\ .$$

This is easily seen if we write

$$\begin{aligned} |b_{\text{rd}} - b| &= |b_{\text{rd}} - \overline{b} + \overline{b} - b| \\ &\leq |b_{\text{rd}} - \overline{b}| + |\overline{b} - b| < 0.04 + 0.1 = 0.14\ . \end{aligned}$$

The corresponding intervals are illustrated below



Notice that the the rounded interval $[11.1, 11.3]$ does not necessarily contain the exact value. ■

The following definitions relate to the concepts of absolute and relative error.

If $|\Delta a| = |\overline{a} - a| \leq 0.5 \cdot 10^{-t}$, then the approximate value $\overline{a}$ is said to have t *correct decimals*.

In an approximate value with $t > 0$ correct decimals, the digits in positions with unit $\geq 10^{-t}$ are called *significant digits*. Leading zeros are not counted; they only indicate the position of the decimal point.

The definitions are easily modified to cover the case when the magnitude of the absolute error is larger than 0.5.

Example. From the definitions we have

Approximation with error bound	Correct decimals	Significant digits
$0.001234 \pm 0.5 \cdot 10^{-5}$	5	3
$56.789 \pm 0.5 \cdot 10^{-3}$	3	5
210000 ± 5000		2

■

Note that the approximation

$$a = 1.789 \pm 0.005$$

has two correct decimals even though the exact value may be 1.794.

The principles for rounding and chopping and the concept of significant digits are completely analogous in other number systems than the decimal system; see Sections 2.4 – 2.5.

2.3. Error Propagation

When approximate values are used in computations, their errors will, of course, give rise to errors in the results. We shall derive some simple methods for estimating how errors in the data are propagated in computations.

In practical applications error analysis is often closely related to the technology for construction of devices and measurement of physical quantities.

Example. The efficiency of a certain type of solar energy collectors can be computed by the formula

$$\eta = K\,\frac{QT_{\mathrm{d}}}{I}\ ,$$

where K is a constant, known to high accuracy; Q denotes volume flow; T_{d} denotes temperature difference between ingoing and outgoing fluid; and I denotes irradiance. The accuracies with which we can measure Q, T_{d} and I depend on the construction of the solar collector.

Assume that we have computed the efficiencies 0.76 and 0.70 for two solar collectors S1 and S2, and that the errors in the data can be estimated as follows,

Collector	S1	S2
Q	1.5%	0.5%
T_{d}	1%	1%
I	3.6%	2%

Based on these data, can we be sure that S1 has a larger efficiency than S2? We return to this example when we have derived mathematical tools that can help us answer the question. ∎

First, assume that we shall compute $f(x)$, where f is a differentiable function. Further, assume that we know an approximation of x with an error bound: $x = \overline{x} \pm \epsilon$. If f is monotone (increasing or decreasing), then we can estimate the propagated error simply by computing $f(\overline{x} - \epsilon)$ and $f(\overline{x} + \epsilon)$:

$$\begin{aligned} |R_{\mathrm{X}}| = |\Delta f| &= |f(\overline{x}) - f(x)| \\ &\leq \max\{|f(\overline{x} - \epsilon) - f(\overline{x})|, |f(\overline{x} + \epsilon) - f(\overline{x})|\}\ . \end{aligned} \tag{2.3.1}$$

A more generally applicable method is provided by the following theorem, which will be used repeatedly in the book.

Theorem 2.3.1. Mean value theorem of differential calculus. If the function f is differentiable, then there is a point ξ between $\overline{x}$ and x such that

$$\Delta f = f(\overline{x}) - f(x) = f'(\xi)(\overline{x} - x)\ .$$

The theorem is illustrated in Figure 2.2.

When the mean value theorem is used for practical error estimation, the derivative is computed at $\overline{x}$, and the error bound is adjusted by adding an appropriate "safety correction".

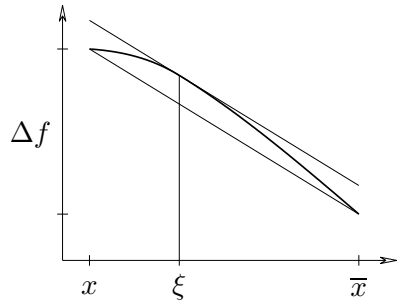


Figure 2.2. *Mean value theorem.*

Example. We shall compute $f(a) = \sqrt{a}$ for $a = 2.05 \pm 0.01$. The mean value theorem gives

$$\Delta f = f'(\xi)\Delta a = \frac{1}{2\sqrt{\xi}}\,\Delta a\ .$$

We can estimate[2)]

$$|\Delta f| \lesssim \frac{1}{2\sqrt{2.05}}\,|\Delta a| \le \frac{0.01}{2\sqrt{2.05}} = 0.00349\ldots \le 0.0036\ .$$

The function $f(x) = \sqrt{x}$ is monotone, and by means of (2.3.1) we get

$$\begin{aligned} |\Delta f| &\le \max\{|\sqrt{2.04} - \sqrt{2.05}|,\ |\sqrt{2.06} - \sqrt{2.05}|\} \\ &\le \max\{0.00350,\ 0.00349\} = 0.0035\ . \end{aligned}$$

■

In general there are more than one datum in a computation, and all of them may be approximations. We first examine error propagation for the four simple arithmetic operations.

Let $y = x_1 + x_2$ and assume that we know approximations $\overline{x}_1$ and $\overline{x}_2$. From the definition of absolute error we get

$$\Delta y = \overline{y} - y = \overline{x}_1 + \overline{x}_2 - (x_1 + x_2) = \Delta x_1 + \Delta x_2\ .$$

If we only know bounds for the absolute errors in $\overline{x}_1$ and $\overline{x}_2$, we must take absolute values and use the triangle inequality,

$$|\Delta y| = |\Delta x_1 + \Delta x_2| \le |\Delta x_1| + |\Delta x_2|\ .$$

A similar analysis of the subtraction $y = x_1 - x_2$ shows that

$$\Delta y = \Delta x_1 - \Delta x_2, \qquad |\Delta y| \le |\Delta x_1| + |\Delta x_2|\ .$$

We summarize the results for addition and subtraction.

[2)] The symbol "$\lesssim$" means "less than or approximately equal to".

	$y = x_1 + x_2$,	$y = x_1 - x_2$,
Absolute error :	$\Delta y = \Delta x_1 + \Delta x_2$,	$\Delta y = \Delta x_1 - \Delta x_2$,
Error bound :	$\lvert\Delta y\rvert \le \lvert\Delta x_1\rvert + \lvert\Delta x_2\rvert$,	$\lvert\Delta y\rvert \le \lvert\Delta x_1\rvert + \lvert\Delta x_2\rvert$.

This can easily be generalized to an arbitrary number of data. Eg for $y = \sum_{i=1}^{n} x_i$ we get $|\Delta y| \le \sum_{i=1}^{n} |\Delta x_i|$.

Next, consider the multiplication $y = x_1 x_2$. We get

$$\begin{aligned}\Delta y &= \overline{x}_1\overline{x}_2 - x_1x_2 = (x_1 + \Delta x_1)(x_2 + \Delta x_2) - x_1x_2 \\ &= x_1\Delta x_2 + x_2\Delta x_1 + \Delta x_1\Delta x_2 \ .\end{aligned}$$

It is convenient to consider the relative errors,

$$\frac{\Delta y}{y} = \frac{\Delta x_2}{x_2} + \frac{\Delta x_1}{x_1} + \frac{\Delta x_1}{x_1}\,\frac{\Delta x_2}{x_2} \ .$$

If the relative errors in $\overline{x}_1$ and $\overline{x}_2$ are small, we can ignore the last term, so that

$$\frac{\Delta y}{y} \simeq \frac{\Delta x_1}{x_1} + \frac{\Delta x_2}{x_2} \ ,$$

and if we take absolute values and use the triangle inequality, we get the bound

$$\left|\frac{\Delta y}{y}\right| \lesssim \left|\frac{\Delta x_1}{x_1}\right| + \left|\frac{\Delta x_2}{x_2}\right| \quad .$$

By a similar argument for the division $y = x_1/x_2$ we get

$$\frac{\Delta y}{y} \simeq \frac{\Delta x_1}{x_1} - \frac{\Delta x_2}{x_2} \ , \qquad \left|\frac{\Delta y}{y}\right| \lesssim \left|\frac{\Delta x_1}{x_1}\right| + \left|\frac{\Delta x_2}{x_2}\right| \quad .$$

We summarize,

	$y = x_1 \cdot x_2$,	$y = x_1/x_2$,
Relative error :	$\dfrac{\Delta y}{y} \simeq \dfrac{\Delta x_1}{x_1} + \dfrac{\Delta x_2}{x_2}$,	$\dfrac{\Delta y}{y} \simeq \dfrac{\Delta x_1}{x_1} - \dfrac{\Delta x_2}{x_2}$,
Error bound :	$\left\lvert\dfrac{\Delta y}{y}\right\rvert \lesssim \left\lvert\dfrac{\Delta x_1}{x_1}\right\rvert + \left\lvert\dfrac{\Delta x_2}{x_2}\right\rvert$,	$\left\lvert\dfrac{\Delta y}{y}\right\rvert \lesssim \left\lvert\dfrac{\Delta x_1}{x_1}\right\rvert + \left\lvert\dfrac{\Delta x_2}{x_2}\right\rvert$.

Example. Now we can solve the solar collector problem. The error propagation formulas for multiplication and division give

$$\left|\frac{\Delta\eta}{\eta}\right| \leq \left|\frac{\Delta Q}{Q}\right| + \left|\frac{\Delta T_{\mathrm{d}}}{T_{\mathrm{d}}}\right| + \left|\frac{\Delta I}{I}\right| \ .$$

For collector S1 we get

$$\left|\frac{\Delta\eta}{\eta}\right| \leq (1.5 + 1 + 3.6) \cdot 10^{-2} = 0.061 \ ,$$

so that

$$|\Delta\eta| \lesssim 0.76 \cdot 0.061 < 0.046 \ .$$

Thus, the efficiency for S1 is in the interval $0.714 \leq \eta_1 \leq 0.806$.

The similar computation for S2 gives $0.675 \leq \eta_2 \leq 0.725$.

The two intervals overlap, and therefore we cannot be sure that the solar collector S1 is better than S2. ∎

The following generalization of the mean value theorem is useful for examination of error propagation in the evaluation of a function f of n variables, $x_1, x_2, \ldots, x_n$.

Theorem 2.3.2. If the real valued function f is differentiable in a neighbourhood of $x = (x_1, x_2, \ldots, x_n)$ and $\overline{x} = x + \Delta x$, is a point in that neighbourhood, then there is a number θ, $0 < \theta < 1$, such that

$$\Delta f = f(\overline{x}) - f(x) = \sum_{k=1}^{n} \frac{\partial f}{\partial x_k}(x + \theta\Delta x)\Delta x_k \ .$$

Proof. Define the function $F(t) = f(x + t\Delta x)$. The mean value theorem for a function of one variable and the chain rule for differentiation give

$$\Delta f = F(1) - F(0) = F'(\theta) = \sum_{k=1}^{n} \frac{\partial f}{\partial x_k}(x + \theta\Delta x_k)\ \Delta x_k \ .$$

□

When this theorem is used for practical error estimation, the partial derivatives are evaluated at $x = \overline{x}$ (the given approximation). When there are only bound for the errors in the argument $\overline{x}$, one can get a bound for Δf by using the triangle inequality.

General error propagation formula:

$$\Delta f \simeq \sum_{k=1}^{n} \frac{\partial f}{\partial x_k}(\overline{x})\, \Delta x_k \ .$$

Maximal error bound:

$$|\Delta f| \lesssim \sum_{k=1}^{n} \left| \frac{\partial f}{\partial x_k}(\overline{x})\, \Delta x_k \right| \ .$$

Example. Let $y = \sin(x_1^2 x_2)$, where $x_1 = 0.75 \pm 10^{-2}$ and $x_2 = 0.413 \pm 3 \cdot 10^{-3}$.

The maximal error bound gives

$$\begin{aligned} |\Delta y| &\lesssim |\cos(x_1^2 x_2) \cdot 2 x_1 x_2 \Delta x_1| + |\cos(x_1^2 x_2) \cdot x_1^2 \Delta x_2| \\ &\lesssim 0.974 \cdot 2 \cdot 0.75 \cdot 0.413 \cdot 10^{-2} + 0.974 \cdot 0.75^2 \cdot 3 \cdot 10^{-3} \leq 0.0077 \ . \end{aligned}$$

The approximate value is

$$\overline{y} = \sin(0.75^2 \cdot 0.413) = 0.230229 \pm 0.5 \cdot 10^{-6} \ .$$

If we round this to 0.23, we get a rounding error less than $0.03 \cdot 10^{-2}$, so that

$$y = 0.23 \pm (0.77 \cdot 10^{-2} + 0.03 \cdot 10^{-2}) = 0.23 \pm 0.8 \cdot 10^{-2} \ .$$

■

The maximal error bound is very pessimistic if the number of variables is large. In such cases it is sometimes better to use statistical methods and to compute an average value for the error of the result, a so-called *mean error*. A common practice is to use *experimental perturbation analysis*, ie the input data are varied more or less systematically, and it is checked how sensitive the output data are to such perturbations.

In the above presentation the problem was to compute a real valued function of several variables. If we have a vector valued function f, ie

$$f = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{pmatrix} \ ,$$

then the methods can be used for error estimation of each component separately, cf Exercise E4.

The relative error of an approximate value is a measure of the information content of the approximation. If we want to measure a physical quantity, eg the distance between two points, it generally calls for more advanced equipment if we want six significant digits instead of two. When

the approximation is used in numerical computation, it is important that there is no unnecessary loss of information.

Example. The approximations

$$x_1 = 10.123456 \pm 0.5 \cdot 10^{-6}, \qquad x_2 = 10.123788 \pm 0.5 \cdot 10^{-6}$$

both have 8 significant digits and relative error less than $0.5 \cdot 10^{-7}$. When we subtract,

$$y = x_1 - x_2 = -0.000332 \pm 10^{-6} \ ,$$

the approximation of y also has a small absolute error, but the relative error can be estimated as

$$\left| \frac{\Delta y}{y} \right| \leq \frac{10^{-6}}{0.000331} < 0.4 \cdot 10^{-2} \ .$$

Thus, the approximation has only two significant digits. ∎

This is an example of *cancellation*: When we subtract two almost equal numbers with errors, the result has fewer significant digits, and the relative error is larger. Such loss of accuracy can often be avoided by a mathematically equivalent reformulation of the expression that has to be computed.

Example. The second degree equation

$$x^2 - 18x + 1 = 0$$

has the solutions $x_1 = 9 + \sqrt{80}$, $x_2 = 9 - \sqrt{80}$. If $\sqrt{80}$ is given with four correct decimals, we get

$$x_1 = 9 + 8.9443 \pm 0.5 \cdot 10^{-4} = 17.9443 \pm 0.5 \cdot 10^{-4} \ ,$$
$$x_2 = 9 - 8.9443 \pm 0.5 \cdot 10^{-4} = \ \ 0.0557 \pm 0.5 \cdot 10^{-4} \ .$$

Thus, the approximation of x_1 has six significant digits, and x_2 has only three. Cancellation is avoided by rewriting

$$x_2 = \frac{(9 - \sqrt{80})(9 + \sqrt{80})}{9 + \sqrt{80}} = \frac{1}{9 + \sqrt{80}} = \frac{1}{17.9443 \pm 0.5 \cdot 10^{-4}} \ .$$

Then we get

$$\overline{x}_2 = \frac{1}{17.9443} = 0.055728002\ldots \ .$$

The error propagation rule for division says that the relative error in $\overline{x}_2$ (coming from the error in the approximation of $\sqrt{80}$) is at most

$$\frac{0.5 \cdot 10^{-4}}{17.9443} < 0.3 \cdot 10^{-5} \ ,$$

so the absolute error is bounded by $0.3 \cdot 10^{-5} \cdot 0.05573 < 0.17 \cdot 10^{-6}$.

If we round $\overline{x}_2$ to seven decimals, we get $x_2 = 0.0557280 \pm 0.2 \cdot 10^{-6}$.

Like the approximation of $\sqrt{80}$ this approximation of x_2 has five significant digits. ∎

Two more examples of reformulations for avoiding cancellation are

$$\sqrt{1+x} - \sqrt{1-x} = \frac{2x}{\sqrt{1+x} + \sqrt{1-x}} \ ,$$
$$\log a - \log b = \log \frac{a}{b} \ .$$

If it is difficult to find a suitable reformulation of an expression of the form

$$f(a+x) - f(a) \ ,$$

then cancellation can be avoided by using an appropriate number of terms in the Taylor expansion around a,

$$f(a+x) - f(a) \simeq f'(a)x + \frac{1}{2!} f''(a)x^2 + \cdots + \frac{1}{r!} f^{(r)}(a)x^r \ .$$

Example. For small values of x we have

$$1 - \cos x \simeq \frac{x^2}{2!} + \frac{x^4}{4!} \ .$$ ∎

2.4. Representation of Numbers in Computers

The decimal number system is a *position system* with *base* (or *radix*) 10. Most computers use a position system with another base (eg 2 or 16), and therefore we start this section by recalling how numbers are represented in a position system with arbitrary base β. Let β be a natural number, $\beta \geq 2$; any positive real number can be written

$$(d_n \ldots d_2 d_1 d_0 . d_{-1} d_{-2} \ldots)_\beta \ ,$$

where all the d_j are integers between 0 and $\beta - 1$. The value of such a number is

$$d_n \beta^n + \cdots + d_2 \beta^2 + d_1 \beta^1 + d_0 \beta^0 + d_{-1} \beta^{-1} + d_{-2} \beta^{-2} + \cdots \ .$$

Example.
$$(760)_8 = 7 \cdot 8^2 + 6 \cdot 8^1 + 0 \cdot 8^0 = (496)_{10} \ ,$$
$$(101.101)_2 = 2^2 + 2^0 + 2^{-1} + 2^{-3} = (5.625)_{10} \ ,$$
$$(0.333\ldots)_{10} = 3 \cdot 10^{-1} + 3 \cdot 10^{-2} + 3 \cdot 10^{-3} + \cdots = \frac{1}{3} \ .$$
■

The architecture of most computers is based on the principle that data are stored in main memory with a fixed amount of information as a unit. This unit is called a *word*, and the *word length* is the number of bits per word. Most computers have word length 32 bits, but some use 64 bits.

Integers can, of course, be represented exactly in a computer, provided that the word length is large enough for storing all the digits.

In contrast, a real number cannot in general be represented exactly in a computer. There are two reasons for this: Errors may arise when a number is converted from one number system to another, eg

$$(0.1)_{10} = (0.0001100110011\ldots)_2 \ .$$

Thus, the number $(0.1)_{10}$ cannot be represented exactly in a computer with a binary number system. The other reason is that errors may arise because of the finite word length of the computer.

How should real numbers be represented in a computer? The first computers (in the 1940s and early 1950s) used *fixed point representation*: For each computation the user decided how many digits in a computer word were to be used for representing respectively the integer and the fractional parts of a real number. Obviously, with this method it is difficult to represent simultaneously both large and small numbers. Assume eg that we have a decimal representation with word length six digits, and that we use three digits for decimal parts. The largest and smallest positive numbers that can be represented are 999.999 and 0.001, respectively. Note that small numbers are represented with larger relative errors than large numbers.

This difficulty is overcome if real numbers are represented in the *exponential form* that we generally use for very small and very large numbers. Eg we would not write

$$0.00000000789 \ , \qquad 6540000000000 \ ,$$

but rather

$$7.89 \cdot 10^{-9} \ , \qquad 6.54 \cdot 10^{12} \ .$$

This way of representing real numbers is called *floating point representation* (as opposed to fixed point).

In the number system with base β any real number $X \neq 0$ can be written in the form

$$X = M \cdot \beta^E \ ,$$

where E is an integer, and

$$\begin{aligned} M &= \pm D_0.D_1D_2D_3\ldots \ , \\ 1 &\leq D_0 \leq \beta{-}1 \ , \\ 0 &\leq D_i \leq \beta{-}1, \quad i = 1, 2, \ldots \ . \end{aligned}$$

M may be a number with infinitely many digits. When a number written in this form is to be stored in a computer, it must be reduced — by rounding or chopping. Assume that $t+1$ digits are used for representing M. Then we store the number

$$x = m \cdot \beta^e \ ,$$

where m is equal to M, reduced to $t{+}1$ digits,

$$\begin{aligned} m &= \pm d_0.d_1d_2d_3\ldots d_t \ , \\ 1 &\leq d_0 \leq \beta{-}1 \ , \\ 0 &\leq d_i \leq \beta{-}1, \quad i = 1, 2, \ldots, t \ , \end{aligned}$$

and[3)] $e = E$. The number m is called the *significand* or *mantissa*, and e is called the *exponent*. The digits to the right of the point in m are called the *fraction*. From the expression for m it follows that

$$1 \leq |m| < \beta \ .$$

We say that x is a *normalized floating point number*.

The range of the numbers that can be represented in the computer depends on the amount of storage that is reserved for the exponent. The limits of e can be written

$$L \leq e \leq U \ ,$$

where L and U are negative and positive integers, respectively. If the result of a computation is a floating point number with $e > U$, then the computer issues an error signal.This type of error is called *overflow*. The

[3)] In the extreme case where we use rounding; all $D_i = \beta - 1$, $i = 0, 1, \ldots, t$; and we have to augment the last digit by 1, we get $m = \pm 1.00\ldots0$ and $e = E + 1$. We shall ignore this case in the following presentation, but the results regarding relative error are also valid for this extreme case.

corresponding error with $e < L$ is called *underflow*. It is often handled by assigning the value 0 to the result.

A set of normalized floating point numbers is uniquely characterized by β (the base), t (the precision), and $[L, U]$ (the range of the exponent). We shall refer to *the floating point system* (β, t, L, U).

The floating point system (β, t, L, U) is the set of *normalized floating point numbers in the number system with base β and t digits for the fraction* (equivalent to $t+1$ digits in the significand), ie all numbers of the form

$$x = m \cdot \beta^e \ ,$$

where

$$\begin{aligned} m &= \pm d_0.d_1d_2d_3 \ldots d_t \ , \\ 0 &\le d_i \le \beta - 1, \quad i = 1, 2, \ldots, t \ , \\ 1 &\le |m| < \beta \ , \end{aligned}$$

and where the exponent satisfies

$$L \le e \le U \ .$$

It is important to realize that the floating point numbers are not evenly distributed over the real axis. As an example, the positive floating point numbers in the system $(\beta, t, L, U) = (2, 2, -2, 1)$ are shown in Figure 2.3.

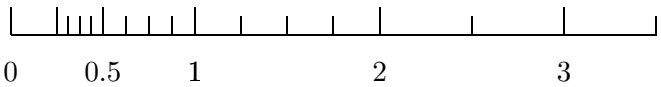


Figure 2.3. *The positive numbers in the floating point system* $(2, 2, -2, 1)$.

Some typical values of (β, t, L, U) are

	β	t	L	U
IEEE standard, single precision	2	23	-126	127
double precision	2	52	-1022	1023
TI-85 pocket calculator	10	13	-999	999

Double precision floating point numbers are available in several programming languages, eg Fortran and C, and it is the standard format in

MATLAB. Usually such numbers are implemented so that two computer words are used for storing one number; this gives higher precision and a larger range. We return to this in Section 2.8.

Again, we want to emphasize that our notation "The floating point system (β, t, L, U)" means that the *fraction* occupies t digits. This notation is consistent with the IEEE standard for floating point arithmetic, see Section 2.8. In older literature floating point numbers are often normalized so that $\beta^{-1} \leq |m| < 1$, and there (β, t, L, U) means that the *significand* (equal to the fraction) occupies t digits.

2.5. Rounding Errors in Floating Point Representation

When numbers are represented in the floating point system (β, t, L, U), we get rounding errors because of the limited precision. We shall derive a bound for the relative error.

Assume that a real number $X \neq 0$ can be written (exactly)

$$X = M \cdot \beta^e, \qquad 1 \leq |M| < \beta \ ,$$

and let $x = m \cdot \beta^e$, where m is equal to M, rounded to $t{+}1$ digits. Then

$$|m - M| \leq \tfrac{1}{2}\beta^{-t} \ ,$$

and we get a bound for the absolute error,

$$|x - X| \leq \tfrac{1}{2}\beta^{-t} \cdot \beta^e \ .$$

This leads to the following bound for the relative error:

$$\frac{|x - X|}{|X|} \ \leq \ \frac{\frac{1}{2}\beta^{-t} \cdot \beta^e}{|M| \cdot \beta^e} \ = \ \frac{\frac{1}{2}\beta^{-t}}{|M|} \ \leq \ \tfrac{1}{2}\beta^{-t} \ .$$

The last inequality follows from the condition $|M| \geq 1$. Thus, we have shown

Theorem 2.5.1. The *relative rounding error* in floating point representation can be estimated as

$$\frac{|x - X|}{|X|} \leq \mu, \qquad \mu = \tfrac{1}{2}\beta^{-t} \ .$$

μ is called the *unit roundoff*.

Note that the bound for the relative error is independent of the magnitude of X. This means that both large and small numbers are represented with the same relative accuracy.

Sometimes it is convenient to use an equivalent formulation:

There is an ϵ such that

$$x = X(1+\epsilon), \qquad |\epsilon| \le \mu \ .$$

In Section 2.7 we shall see that this formulation is useful in the analysis of accumulated rounding errors in connection with sequences of arithmetic operations.

If we have a computer with binary arithmetic, using $t+1$ digits in the significand, how accurate is this computer, expressed in terms of decimal numbers? We must answer this question in order to know how many decimal digits it is relevant to print.

Example. The floating point system $(2, 23, -126, 127)$ has unit roundoff

$$\mu = \tfrac{1}{2} \cdot 2^{-23} = 2^{-24} \simeq 5.96 \cdot 10^{-8} \simeq 0.5 \cdot 10^{-7} \ .$$

Thus, this system is roughly equivalent to a floating point system with $(\beta, t) = (10, 7)$. ∎

Alternative formulations of the above question are: "How many decimal digits correspond to $t+1$ binary?" and "If the unit roundoff in a binary floating point system is $\mu = 0.5 \cdot 2^{-t}$, how many digits must we have in a decimal system with approximately the same unit roundoff?" This was the formulation used in the example. In general we have to solve the equation

$$0.5 \cdot 10^{-s} = 0.5 \cdot 2^{-t} \ ,$$

with respect to s. Taking logarithms, we get $s = t \log_{10} 2 \simeq 0.3t$.

Rule of thumb:

t binary digits correspond to $0.3t$ decimal digits.

s decimal digits correspond to $3.3s$ binary digits.

Example. The IEEE standard for single precision arithmetic prescribes $t = 23$ binary digits in the fraction. This corresponds approximately to a decimal floating point system with $s = 7$, since $23 \log_{10} 2 \simeq 6.9$.

The standard for MATLAB has $(\beta, t) = (2, 52)$. This corresponds to a decimal system with $s = 16$ digits in the fraction.

```
>> format long e
>> y = sin(pi/4)
y = 7.071067811865475e-01
```

The result is displayed with 15 digits in the fraction. ∎

The reasoning in this section can easily be modified to floating point systems with *chopping*. The only difference is that then the unit roundoff is $\mu_c = 2\mu = \beta^{-t}$. Floating point arithmetic with chopping is easier to implement than arithmetic with rounding, but is rarely used today because the IEEE standard for floating point arithmetic prescribes that rounding should always be used, see Section 2.8.

2.6. Arithmetic Operations with Floating Point Numbers

The aim of this section is not to describe in full detail how floating point arithmetic can be implemented. Rather, we want to show that under certain assumptions it is possible to perform floating point operations with good accuracy. This accuracy should then be requested from all implementations.

Since rounding errors arise already when real numbers are stored in the computer, one can hardly expect that arithmetic operations can be performed without errors. As an example, let a, b and c be variables with $t+1$ digits, and consider the statement

$$a := b * c \ .$$

In general the product of two $t+1$ digit numbers has $2t+1$ or $2t+2$ digits, so the significand must be reduced (by rounding or chopping) before the result can be stored in a.

Before 1985 there did not exist a standard for the implementation of floating point arithmetic, and different computer manufacturers chose different solutions, depending on economic and other reasons. In this section we shall describe a somewhat simplified arithmetic in order to be able to explain the principles of floating point arithmetic without giving

too many details. A survey of the IEEE floating point standard is given in Section 2.8.

We shall describe an arithmetic for the floating point system (β, t, L, U) and assume rounding. In the numerical examples we use the system $(10, 3, -9, 9)$.

Computers have special registers for performing arithmetic operations. The length of these registers (the number of digits they hold) determine how accurately floating point operations can be performed. We shall assume that the arithmetic registers hold $2t + 4$ digits in the significand. It is possible to perform the arithmetic operations with the same accuracy (and faster) using shorter registers, but then more complicated algorithms are needed.

Apart from arithmetic and logical operations one must be able to perform *shift operations*, which are used in connection with normalization and to make two floating point numbers have the same exponent. As an example, a left shift implies that the exponent is decreased:

$$0.031 \cdot 10^1 = 3.100 \cdot 10^{-1} \ .$$

We first discuss floating point addition (and at the same time subtraction, since $x - y = x + (-y)$). Let

$$x = m_x \beta^{e_x} \ , \qquad y = m_y \beta^{e_y} \ ,$$

and let $z = fl[x + y]$ denote the result of the floating point addition. We assume that $e_x \geq e_y$. If $e_x > e_y$, then y is shifted $e_x - e_y$ positions to the right before the addition:

$$\begin{aligned} 1.234 \cdot 10^0 + 4.567 \cdot 10^{-2} &= (1.234 + 0.04567) \cdot 10^0 \\ &= 1.27967 \cdot 10^0 \doteq 1.280 \cdot 10^0 \ . \end{aligned}$$

(The symbol "$\doteq$" means "is rounded to"). If $e_x - e_y \geq t + 3$, then $fl[x + y] = x$. Eg

$$1.234 \cdot 10^0 + 4.567 \cdot 10^{-6} = 1.234004567 \cdot 10^0 \doteq 1.234 \cdot 10^0 \ .$$

We get the following addition algorithm[4)]:

[4)] This and the following algorithms are at a low level and depend on the computer's hardware. We cannot give them in MATLAB, but present them in pseudo code.

Floating point addition $z := x + y$;

$e_z := e_x$; ($e_x \geq e_y$ is assumed)
if $e_x - e_y \geq t+3$ **then**
  $m_z := m_x$;
else
  $m_y := m_y/\beta^{e_x - e_y}$; (right shift $e_x - e_y$ positions)
  $m_z := m_x + m_y$;
  Normalize; (see below)
endif

If $e_x - e_y < t+3$, then m_y can be stored exactly after the shift, since the arithmetic register is assumed to hold $2t+4$ digits. Also the addition $m_x + m_y$ is performed without error. In general, the result of these operations may be an *unnormalized* floating point number $z = m_z \cdot \beta^{m_z}$, with $|m_z| \geq \beta$ or $|m_z| < 1$, eg $5.678 \cdot 10^0 + 4.567 \cdot 10^0 = 10.245 \cdot 10^0$ or $5.678 \cdot 10^0 + (-5.612 \cdot 10^0) = 0.066 \cdot 10^0$. In such cases the floating point number is normalized by appropriate shifts. Further, the significand must be rounded to t+1 digits. These two tasks are performed by the following algorithm that takes an unnormalized, nonzero floating point number $m \cdot \beta^e$ as input and gives a normalized floating point number x as output.

Normalize
if $|m| \geq \beta$ **then**
  $m := m/\beta$; $e := e+1$; (right shift one position)
else
  while $|m| < 1$ **do**
    $m := m * \beta$; $e := e-1$; (left shift one position)
endif
Round m to t+1 digits;
if $|m| = \beta$ **then**
  $m := m/\beta$; $e := e+1$; (right shift one position)
endif
if $e > U$ **then**
  $x :=$ `Inf`; (exponent overflow)
elseif $e < L$ **then**
  $x := 0$; (exponent underflow)
else
  $x = m \cdot \beta^e$; (the normal case)
endif

The if–statement after the rounding is needed because the rounding to $t+1$ digits can give an unnormalized result:

$$9.9995 \cdot 10^3 \doteq 10.000 \cdot 10^3 = 1.000 \cdot 10^4 \ .$$

The multiplication and division algorithms are simple:

FLOATING POINT MULTIPLICATION $z := x * y$;

$e_z := e_x + e_y$;
$m_z = m_x * m_y$
Normalize;

FLOATING POINT DIVISION $z := x/y$;

if $y = 0$ **then**
division by zero; (error signal[5])
else
$e_z := e_x - e_y$;
$m_z = m_x/m_y$
Normalize;
endif

We have assumed that the arithmetic registers hold $2t + 4$ digits. This implies that the results of addition and multiplication are exact before normalization and rounding. Therefore, the only error in these operations is the rounding error. A careful analysis of the division algorithm shows that the division of the significands can be performed so that the $2t + 4$ digits are correct. Therefore, the fundamental error estimate for floating point representation (Theorem 2.5.1) is valid for floating point arithmetic:

Theorem 2.6.1. Let $\odot$ denote any of the arithmetic operators $+, -, *$ and $/$, and assume that $x \odot y \neq 0$ and that the arithmetic registers are as described above. Then

$$\left| \frac{fl[x \odot y] - x \odot y}{x \odot y} \right| \leq \mu \ ,$$

or, equivalently,

$$fl[x \odot y] = (x \odot y)(1 + \epsilon) \ ,$$

for some ϵ that satisfies $|\epsilon| \leq \mu$. $\mu = \frac{1}{2}\beta^{-t}$ is the unit roundoff.

[5] In the IEEE standard the error signal is $z :=$ `Inf` if $x \neq 0$ and $z :=$ `NaN` (Not–a–Number) if $x = 0$, see Section 2.8.

It can be shown that the theorem is valid even if the arithmetic registers hold $t+4$ digits, only, provided that the algorithms are modified accordingly.

A consequence of the errors in floating point arithmetic is that some of the usual mathematical laws are no longer valid. Eg the associative law for addition does not necessarily hold. It may happen that

$$fl[fl[a+b]+c] \neq fl[a+fl[b+c]] \ .$$

Example. Let $a = 9.876 \cdot 10^4$, $b = -9.880 \cdot 10^4$, $c = 3.456 \cdot 10^1$, and use the floating point system $(10, 3, -9, 9)$ with rounding. Then

$$fl[fl[a+b]+c] = fl[-4.000 \cdot 10^1 + 3.456 \cdot 10^1] = -5.440 \cdot 10^1 \ ,$$

and

$$fl[a+fl[b+c]] = fl[9.876 \cdot 10^4 - 9.877 \cdot 10^4] = -1.000 \cdot 10^1 \ .$$ ∎

Another consequence of the errors is that it is seldom meaningful to test for equality between floating point numbers. Let `x` and `y` be floating point numbers that are results of earlier computation. Then there is very small probability that the boolean expression `x == y` will be true. Therefore, instead of

```
if  x == y
```

one should write

```
if  abs(x-y) < delta*max(abs(x), abs(y))
```

where `delta` is some small number, slightly larger than the unit roundoff μ.

2.7. Accumulated Errors

As an example of error accumulation in repeated floating point operations we shall consider the computation of a sum,

$$S_n = \sum_{k=1}^{n} x_k \ .$$

We assume that the sum is computed in the natural order and let $\hat{S}_i$ denote the computed partial sum,

$$\hat{S}_1 := x_1$$
$$\hat{S}_i := fl[\hat{S}_{i-1} + x_i], \quad i = 2, 3, \ldots, n \ .$$

If we use the error estimate for addition in the form

$$fl[a+b] = (a+b)(1+\epsilon), \quad |\epsilon| \le \mu \ ,$$

we see that

$$\hat{S}_i = (\hat{S}_{i-1} + x_i)(1+\epsilon_i), \quad |\epsilon_i| \le \mu; \quad i = 2, 3, \ldots, n \ .$$

A simple induction argument shows that the final result can be written in the form

$$\hat{S}_n = \hat{x}_1 + \hat{x}_2 + \cdots + \hat{x}_n \ , \tag{2.7.1a}$$

where

$$\begin{aligned} \hat{x}_1 &= x_1(1+\epsilon_2)(1+\epsilon_3)\cdots(1+\epsilon_n) \ , \\ \hat{x}_i &= x_i(1+\epsilon_i)(1+\epsilon_{i+1})\cdots(1+\epsilon_n), \quad i = 2, 3, \ldots, n \ . \end{aligned} \tag{2.7.1b}$$

To be able to obtain practical error estimates, we need the following lemma, the proof of which is left as an exercise.

Lemma 2.7.1. Let the numbers $\epsilon_1, \epsilon_2, \ldots, \epsilon_r$ satisfy $|\epsilon_i| \le \mu$, $i = 1, 2, \ldots, r$, and assume that $r\mu \le 0.1$. Then there is a number δ_r such that

$$(1+\epsilon_1)(1+\epsilon_2)\cdots(1+\epsilon_r) = 1 + \delta_r \ ,$$

and

$$|\delta_r| \le 1.06\, r\, \mu \ .$$

Now we can derive two types of results, which give error estimates for summation in floating point arithmetic.

Theorem 2.7.2. Forward analysis. If $n\mu \le 0.1$, then the error in the computed sum can be estimated as

$$|\hat{S}_n - S_n| \le |x_1|\,|\delta_{n-1}| + |x_2|\,|\delta_{n-1}| + |x_3|\,|\delta_{n-2}| + \cdots + |x_n|\,|\delta_1| \ ,$$

where

$$|\delta_i| \le i \cdot 1.06\mu, \quad i = 1, 2, \ldots, n-1 \ .$$

Proof. According to (2.7.1) and Lemma 2.7.1 we can write

$$\hat{S} = x_1(1+\delta_{n-1}) + x_2(1+\delta_{n-1}) + x_3(1+\delta_{n-2}) + \cdots + x_n(1+\delta_1) \ ,$$

where the δ_i satisfy the inequality in the theorem. Subtract S_n and use the triangle inequality. □

Forward analysis is the type of error analysis that we used at the beginning of this chapter. However, it is difficult to use this method to analyze such a fundamental algorithm as Gaussian elimination for the solution of a linear system of equations. The first correct error analysis of this algorithm was made in the mid 1950s by means of *backward error analysis*.

In backward analysis one shows that the approximate solution $\hat{S}$ which has been computed for the problem $\mathcal{P}$ is the *exact solution* of a *perturbed problem* $\hat{\mathcal{P}}$, and estimate the "distance" between $\hat{\mathcal{P}}$ and $\mathcal{P}$. By means of perturbation analysis of the problem it is then possible to estimate the difference between $\hat{S}$ and the true solution S. Examples of this are given in Chapters 4 and 8.

We cite the following description of the aim of backward error analysis from J.R. Rice, *Matrix computations and mathematical software*, McGraw-Hill, New York, 1981.

> "The objective of backward error analysis is to stop worrying about whether one has the "exact" answer, because this is not a well-defined thing in most real-world situations. What one wants is to find an answer which is the true mathematical solution to a problem which is within the domain of uncertainty of the original problem. Any result that does this must be acceptable as an answer to the problem, at least with the philosophy of backward error analysis."

In the summation case we can formulate

Theorem 2.7.3. Backward analysis. The computed sum can be expressed as

$$\hat{S}_n = \hat{x}_1 + \hat{x}_2 + \cdots + \hat{x}_n \ ,$$

where

$$\hat{x}_1 = x_1(1 + \delta_{n-1}) \ ,$$
$$\hat{x}_i = x_i(1 + \delta_{n+1-i}), \quad i = 2, 3, \ldots, n \ .$$

If $n\mu \leq 0.1$, then

$$|\delta_k| \leq k \cdot 1.06\mu, \quad k = 1, 2, \ldots, n-1 \ .$$

The error estimates in these two theorems lead to an important conclusion: We can rewrite the estimates in the form

$$|\hat{S}_n - S_n| \leq \big((n-1)|x_1| + (n-1)|x_2| + (n-2)|x_3| + \cdots + |x_n|\big)1.06\mu \ ,$$

This shows that in order to minimize the error bound, we should add the terms in increasing order of magnitude, since the first terms have the largest factors.

Example. Let

$$\begin{aligned} x_1 &= 1.234 \cdot 10^1 \ , \\ x_2 &= 3.453 \cdot 10^0 \ , \\ x_3 &= 3.442 \cdot 10^{-2} \ , \\ x_4 &= 4.667 \cdot 10^{-3} \ , \\ x_5 &= 9.876 \cdot 10^{-4} \ , \end{aligned}$$

and use the floating point system $(10, 3, -90.9)$ with rounding. Summation in decreasing and increasing order gives

$$\begin{aligned} &\text{decreasing order:} \quad \hat{S}_5 = 1.592 \cdot 10^1 \ , \\ &\text{increasing order:} \quad \hat{S}_5 = 1.583 \cdot 10^1 \ . \end{aligned}$$

The exact result rounded to 6 decimals is $S_5 = 1.583306 \cdot 10^1$. ∎

Similarly, a relatively large error may arise when a slowly converging series is summed in decreasing order.

Example. We have computed the sum

$$\sum_{n=1}^{30000} \frac{1}{n^2}$$

in the floating point system $(2, 23, -126, 127)$ with rounding. If we take the terms in increasing order of n, we get the result 1.644725, while we get 1.644901 if we sum in decreasing order. The last result is equal to the true value of the sum rounded to 24 binary digits.

It should be pointed out that the major part of the difference between the two results is due to the fact that when we sum in decreasing order, the last 25904 terms do not contribute to the sum because

$$fl[S + \tfrac{1}{n^2}] = S \quad \text{for } n > 4096 \ ,$$

where S is the summation variable. We leave it as an exercise to show this.

∎

2.8. IEEE Standard for Floating Point Arithmetic

Above all, it is the development of microcomputers that has made it necessary to standardize floating point arithmetic. The aim is to facilitate *portability*, ie a program should run on different computers without changes, and if two computers conform to the standard, then the program should give identical results (or almost identical; see the end of this section).

A proposal for a standard for binary floating point arithmetic was presented in 1979. Some changes were made, and the standard was adopted in 1985. It has been implemented in most computers[6)]. We shall present the most important parts of the standard for binary floating point arithmetic without going into too much detail.

The standard defines four floating point formats divided into two groups, *basic* and *extended*, each in a *single precision* and a *double precision* version. The single precision basic format requires a word of length 32 bits, organized as shown in Figure 2.4.

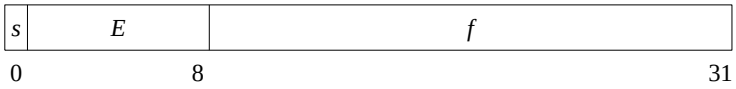


Figure 2.4. *Basic format, single precision.*

The components of a floating point number x are the sign s (one bit), the biased exponent E (8 bits) and the fraction f (23 bits). The value v of x is

a. $v = (-1)^s(1.f)2^{E-127}$ if $0 < E < 255$.

b. $v = (-1)^s(0.f)2^{-126}$ if $E = 0$ and $f \neq 0$.

c. $v = (-1)^s 0$ if $E = 0$ and $f = 0$.

d. $v =$ `NaN` (Not-a-Number, see below) if $E = 255$ and $f \neq 0$.

e. $v = (-1)^s$`Inf` (Infinity) if $E = 255$ and $f = 0$.

[6)] In 1987 a base-independent standard was adopted. This was motivated by the pocket calculators, which normally use the base $\beta = 10$. There have also been computers using base 8 (octal system, using 3 bits per digit) and base 16 (hexadecimal system, using 4 bits per digit). A major argument for these systems is that they need fewer shifts in connection with the arithmetic operations.

The normal case is a. Due to the normalization the significand satisfies $1 \leq |m| < 2$. Thus, the integer part is always one, and by not storing this digit we save an extra bit for the fraction. Also note that the exponent is stored in biased (or shifted) form. The range of values that can be obtained with 8 bits is

$$[(00000000)_2, (11111111)_2] = [(0)_{10}, (255)_{10}] \ .$$

The two extreme values are reserved (cf the above table), so the range of values for the true exponent $e = E - 127$ is

$$[L, U] = [1 - 127, 254 - 127] = [-126, 127] \ .$$

This is in accordance with the table on page 23.

Example. The numbers 1, $4.125 = (1 + \frac{1}{32})2^2$ and $-0.09375 = -1.5 \cdot 2^{-4}$ are stored as

0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0	1	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1	0	1	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

■

One reason for introducing `NaN` (Not-a-Number) and `Inf` (Infinity), items d. and e. above, is to make debugging easier. Both of these are returned as the result in an exceptional case. Eg the result of $1/0$ and $0/0$ is `Inf` and `NaN`, respectively. Also, `NaN` is the result of a computation involving an uninitialized variable, and `Inf` is returned in case of overflow. Rather than stopping the execution of the program it may be preferable to continue, and analyze the output for this debugging information.

In the floating point arithmetic of most computers the result is put to zero in case of an underflow. The IEEE standard allows the option *"gradual underflow"*, item b.

Example. If $e < -126$, then the floating point number is unnormalized. Eg the number stored as

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

is $2^{-4} \cdot 2^{-126} = 2^{-130}$. Note that the leading zeros in f are not significant, and the bound of the relative representation error grows with increasing number of leading zeros.

The smallest nonzero, positive number that can be represented in this way is

$$2^{-23} \cdot 2^{-126} = 2^{-149} \simeq 1.40 \cdot 10^{-45} \ .$$

The smallest normalized, positive number is

$$2^{-126} \simeq 1.18 \cdot 10^{-38} \ ,$$

and the largest number is

$$(2 - 2^{-23}) \cdot 2^{127} \simeq 3.40 \cdot 10^{38} \ .$$

■

The basic double precision format is analogous to the single precision format. Here, 64 bits are used as illustrated in Figure 2.5. The fraction

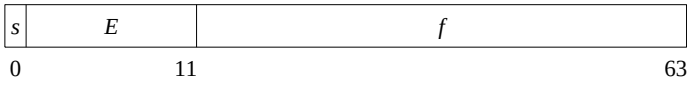


Figure 2.5. *Basic format, double precision.*

f is given with $t = 52$ binary digits, the biased exponent is $E = e + 1023$ and the range of positive, normalized floating point numbers is

$$[2^{-1022}, (2 - 2^{-52})2^{1023}] \simeq [2.22 \cdot 10^{-308}, 1.80 \cdot 10^{308}] \ .$$

Details of the extended single and double formats are left to the implementer, but there must be at least one sign bit and

$$\text{Extended single:} \quad t \geq 31, \quad L \leq -1022, \quad U \geq 1023 \ .$$

$$\text{Extended double:} \quad t \geq 63, \quad L \leq -16382, \quad U \geq 16383 \ .$$

Example. Intel microprocessors live up to these requirements. They use 80 bits both for extended single and extended double. One bit is used for the sign, 15 bits for the biased exponent, and 64 bits for the significand, corresponding to 63 bits for the fraction. ■

Implementations of the standard must provide the four simple arithmetic operations, the square root function and binary–decimal conversion. When every operand is normalized, then an operation (also the square root function) must be performed such that the result is equal to the rounded result of the same operation performed with infinite precision. This implies that Theorem 2.6.1 is valid.

The standard specifies that rounding is done according to the rules in Section 2.2. In particular, rounding to even must be used in the limit case.

The extended formats can be used (by the compiler in some high level languages) both for avoiding overflow and underflow and to give better accuracy.

Example. The computation of $s := \sqrt{x_1^2 + x_2^2}$ may give overflow or underflow, even if the result can be represented as a normalized floating point number, see Exercise E9. If the compiler uses extended precision for the computed squares and their sum, then overflow or underflow cannot occur. ∎

Example. The "length" of a vector with n elements,

$$l = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2} ,$$

can be computed by the following program.

Extended real s;
$s := 0$;
for $i := 1$ **to** n **do** $s := s + x_i * x_i$;
$l := \sqrt{s}$;

If l can be represented (eg in single precision), overflow or underflow cannot occur. Further, since the significand of the extended format has more digits, l will be computed more accurately than in the case where s is a basic format variable. If n is not too large, then l can even be equal to the exact result rounded to the basic format. ∎

In the beginning of this section we mentioned that even if two computers both apply to the IEEE standard for floating point arithmetic, the same program does not necessarily give identical results when run on the two computers. A difference is caused by different implementations of extended formats. An example is given in Computer Exercise C4.

Exercises

E1. How accurately do we need to know π in order to be able to compute $\sqrt{\pi}$ with four correct decimals?

E2. Derive the error propagation formula for division.

E3. (a) Derive the error propagation formula for the function $y = \log x$.

(b) Use the result from (a) to derive the error propagation formula for the function $y = f(x_1, x_2, x_3) = x_1^{\alpha_1} x_2^{\alpha_2} x_3^{\alpha_3}$. (This technique is called logarithmic differentiation).

E4. Let f be a function from $\mathbb{R}^n$ to $\mathbb{R}^m$, and assume that we want to compute $f(\overline{a})$, where $\overline{a}$ is an approximation of a. Show that the general error propagation formula applied to each component of f leads to

$$\Delta f \simeq J\, \Delta a \ ,$$

where J is the $m \times n$ matrix (the Jacobi matrix) with elements

$$(J)_{ij} = \frac{\partial f_i}{\partial x_j} \ .$$

E5. (a) Use Taylor expansion to avoid cancellation in $e^x - e^{-x}$, x close to 0.

Use reformulation to avoid cancellation in the following expressions

(b) $\sin x - \cos x$, x close to $\pi/4$,

(c) $1 - \cos x$, x close to 0,

(d) $\left(\sqrt{1+x^2} - \sqrt{1-x^2}\right)^{-1}$, x close to 0.

E6. Let x be a normalized floating point number in the system (β, t, L, U). Show that $r \le |x| \le R$, where

$$r = \beta^L, \qquad R = (\beta - \beta^{-t})\beta^U \ .$$

E7. Assume that x and y are binary floating point numbers that satisfy $xy > 0$ and $|y| \le |x| \le 2|y|$. Show that $fl[x - y] = x - y$.

E8. (a) Show that $fl[1 + x] = 1$ for all $x \in [0, \mu]$, where μ is the unit roundoff.

(b) Show that $fl[1 + x] > 1$ for all $x > \mu$.

(c) Let $1+\epsilon$ be the smallest floating point number greater than 1. Determine ϵ in the floating point system $(2, t, L, U)$ and compare it to μ. (This number is sometimes called the "*machine epsilon*"; it is given by `eps` in MATLAB).

E9. Show that the computation of $s = \sqrt{x_1^2 + x_2^2}$ can give overflow or underflow even if s is in the range of the floating point system. (As examples take $x_1 = x_2 = 8 \cdot 10^5$ and $x_1 = x_2 = 2 \cdot 10^{-5}$ in the system $(10, 4, -9, 9)$). Rewrite the computation so that over- and underflow is avoided for all data x_1, x_2 such that the result s can be represented.

E10. Assume that $r\mu \leq 0.1$ and that $|\epsilon_i| \leq \mu$, $i = 1, 2, \ldots, r$. Show that

$$\delta_r = (1+\epsilon_1)(1+\epsilon_2)\cdots(1+\epsilon_r) - 1$$

satisfies $|\delta_r| \leq 1.06\, r\, \mu$.

Hint: First show that $|\delta_r| \leq (1+\mu)^r - 1$. Next use that $(1+x)^n \leq e^{nx}$ for $x \geq 0$ and and make a series expansion.

E11. Let $S_n = \sum_{i=1}^n x_i y_i$ and let $\hat{S}_n$ denote the computed result. Show that

$$|\hat{S}_n - S_n| \leq 1.06\mu \cdot \sum_{i=1}^{n} (n+2-i)|x_i y_i| \ ,$$

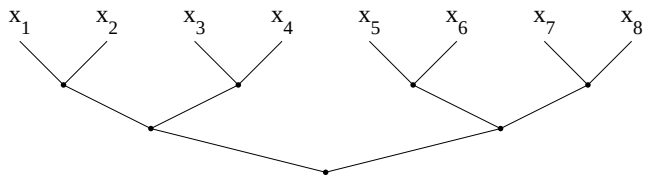
cf Theorem 2.7.2.

E12. Consider the problem in the second example on page 33. Show that when the summation is performed in decreasing order, then

$$fl[S + \tfrac{1}{n^2}] = S \quad \text{for } n > 4096 \ ,$$

where S is the summation variable.

E13. Assume that $n = 2^k$ for some positive integer k, and that the sum $S_n = \sum_{i=1}^n x_i$ is computed in the order illustrated by the figure:



Do a backward error analysis of this summation algorithm and compare it with Theorem 2.7.3.

E14. (a) Show that the rounding error in floating point representation can be estimated as

$$\frac{|x - X|}{|x|} \leq \mu \ ,$$

cf Theorem 2.5.1.

(b) Similarly, it can be shown that if x and y are floating point numbers and $\odot$ is any of the four simple arithmetic operators, then

$$fl[x \odot y] - x \odot y = \theta\, fl[x \odot y], \quad |\theta| \leq \mu \ .$$

Use this to derive the following estimate of the accumulated error in floating point summation:

$$|\hat{S}_n - S_n| \leq \mu \sum_{i=1}^{n} |\hat{S}_i| \ .$$

This estimate shows that in order to get a small error, the summation should be made such that the partial sums are small.

(The exercise is taken from the paper of Espelid given below.)

E15. If extended precision is not available, it may be simulated in connection with arithmetic operations.

(a) Let $|a|$ and $|b|$ be two floating point numbers and $\overline{c} = fl[a+b]$. The error $e = \overline{c} - (a+b)$ can be computed by the algorithm

$$\begin{array}{lll} \textbf{if} & |a| < |b| & \textbf{then} \quad e := (b - \overline{c}) + a; \\ & & \textbf{else} \quad e := (a - \overline{c}) + b; \end{array}$$

(a1) Use the algorithm in the floating point system $(10, 4, -9, 9)$ with $a = 1.2345$, $b = 0.045678$.

The idea of computing both $\overline{c}$ and e in $\overline{c} + e = a + b$ can be used to improve the accuracy of a computed sum of n elements, see Computer Exercise C5. It is sometimes called "Møller's device" and is discussed pp 83–88 in Higham's book given in the references below.

(b) Also in connection with multiplication we are able to get a good estimate of the error: Let x be a $t{+}1$ digit number, and write it as

$$x = x_1 + x_2 \ ,$$

where the first $r{+}1$ digits in the significand of x_1 are obtained by rounding x, and there are $t{-}r$ trailing zero digits.
Let $y_1 + y_2$ denote a similar splitting of the floating point number y. Then

$$x\,y = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2 \ .$$

Now, assume that $r + 1 \le \frac{1}{2}(t{+}1)$. Then the first three products are computed without error, and if r is not too small, then the error in $fl[x_2 * y_2]$ will be small compared to xy. By means of Møller's device we can write the above expression in the form

$$x\,y = p_1 + p_2 + \epsilon \ ,$$

where $|p_2| \le \mu|xy|$ and ϵ is even smaller.

(b1) Use this idea in the floating point system $(10, 4, -9, 9)$ with $r = 1$, $a = 4.3216$, $b = 7.6543$.

(b2) Analyze the method in IEEE double precision when $r = 23$ (corresponding to single precision).

In Computer Exercise C5 we shall use these two ideas to compute a product sum, cf Exercise E11, and in the first exercise of the next chapter we treat another example of the splitting idea.

Computer Exercises

C1. The standard format in MATLAB is IEEE basic double, so the unit roundoff is $\mu = 2^{-53} \simeq 1.11 \cdot 10^{-16}$.

(a) MATLAB has a number of predeclared variables that reflect the floating point system. Print `realmin`, `realmax` and `eps`, and compare them to the results of Exercises E6 and E8.

(b) Find the smallest, strictly positive floating point number in MATLAB.

C2. Assume that we want to implement $\sinh x$ in IEEE single precision.

(a) Test the accuracy of the expressions

$$S_1(x) = \frac{e^x - e^{-x}}{2} , \qquad S_2(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!}$$

for $x = 10^{-1}, 10^{-2}, \ldots, 10^{-8}$.

Hint: The MATLAB command `y = sinh(x)` returns the value $\sinh$ `x` with a relative error bounded by 2^{-53}, and `double(single(a))` rounds the MATLAB variable `a` to single precision (with 29 trailing zeros in the fraction).

(b) Prove that S_2 gives full accuracy (ie a relative error bounded by $\mu = 2^{-24}$) for all x such that $0 < |x| \leq 0.1$.

C3. Write a program for computing the exponential function e^x by means of the Maclaurin series

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots .$$

Compute and add new terms as long as they affect the sum. Investigate the accuracy for large positive and negative arguments. Explain why the accuracy is so poor for large negative x.

C4. The theoretical determination of the smallest floating point number x such that $fl[1 + x] > 1$ on a certain computer is quite difficult. It demands profound knowledge of details in the implementation of floating point arithmetic on the computer. The following MATLAB program was executed on a computer that conforms with the floating point standard.

```
y = 1;  i = 100;
while  y == 1
  i = i - 1;
  x = 2^(-53) * (1 + 2^(-i));
  y = 1 + x;
end
disp(i)
```

The computer uses extended double precision format in the arithmetic operations and uses double roundings: First $1+x$ is rounded to 63 bits in the fraction (extended precision) and next to 52 bits (double precision). What is the result of the program?

What result do you get on your computer?

On a computer that does not use double rounding one gets the result "51". Explain why.

(The example was constructed by Nick Higham.)

C5. Make a MATLAB function that takes two n–vectors a and b as input and returns the product sum $S = \sum_{i=1}^{n} a_i b_i$. The function should use the ideas given in Exercise E15. Products should be computed with a splitting corresponding to single precision, $r = 23$, and Møller's device should be used in the summation.

Compare the result with the output from `dot(a,b)` on the problem given by $a_1 = b_1 = 1$, $a_2 = 2^{-30} + 1$, $b_2 = 2^{-30} - 1$.

Hint: The splitting of a number `x` can be obtained by the commands

```
x1 = double(single(x));   x2 = x - x1;
```

and the summation can be done by `pesum` from `incbox`.

References

The historical development of the representation of numbers is a fascinating chapter in the cultural history of mankind. A nice survey is given in

D.E. Knuth, *The Art of Computer Programming, Volume 2. Seminumerical Algorithms*, 3rd edition, Addison–Wesley, Reading, Mass., 1997.

One is tempted to believe that the binary number system is a fruit of the development of computers. As a matter of fact, several mathematicians in the 17th and 18th centuries used binary representation for number theoretical research. Knuth's book gives a good presentation of floating point arithmetic, both single, double and extended precision. He also shows how to perform single precision arithmetic in a register with $t+4$ bits such that Theorem 2.6.1 is valid.

The first summary of error analysis for floating point arithmetic was given in

J.H. Wilkinson, *Rounding Errors in Algebraic Processes*, Prentice–Hall, Englewood Cliffs, New Jersey, 1963.

The standard for floating point system has been published by the IEEE

ANSI/IEEE 754, *Binary Floating Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1985.

The standard has its own home page

http:grouper.ieee.org/groups/754

Interesting discussions of the standard are given in

D. Goldberg, *What every Computer Scientist Should Know about Floating-Point Arithmetic*, ACM Computing Surveys 23(1991), 5–48.

M.L. Overton, *Numerical Computing with IEEE floating point arithmetic (including one Theorem, one Rule of Thumb and One Hundred and One Exercises)*, SIAM, Philadelphia, PA, 2001.

Floating point summation is discussed in

N.J. Higham, *The Accuracy of Floating-Point Summation*, SIAM J. Sci. Stat. Comput. **14** (1993), 783–799.

T.O. Espelid, *On Floating-Point Summation*, SIAM Review **37** (1995), 603–607.

Error analysis for a number of numerical algorithms is described in

N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd edition, SIAM, Philadelphia, PA, 2002.

Chapter 3

Function Evaluation

3.1. Introduction

Numerical computation frequently involves the elementary functions of mathematics (in connection with mathematical software they are often called *standard functions*). This is the case both when the calculation is done by hand and when a computer is used. It is important to be able to compute these functions easily and efficiently. In hand calculations it is sometimes practical to use Taylor expansions of the elementary functions. Series expansions of non-elementary functions are sometimes used, eg, for the solution of differential equations.

When series expansions are used for numerical computations, the sum of the series is approximated by a partial sum. Then one must estimate the truncation error, the remainder term. In this chapter we describe the simplest remainder term estimates.

Computer implementations of standard functions are often based on the approximation of the function by a rational function. Here it is important both that the approximation satisfies the accuracy requirements, and that it can be evaluated fast. We briefly describe how a standard function can be implemented. Then we discuss some numerical aspects of range reduction, ie how mathematical identities can be used to reduce the argument for the function to an interval close to zero. Finally, we give a simplified description of a technique for implementation of trigonometric functions.

3.2. Remainder Term Estimates

Let $S = \sum_{n=1}^{\infty} a_n$ be a convergent series (we shall also use S to denote the sum of the series). We assume that we cannot compute the sum analytically, but have to approximate it numerically.

The *partial sum* S_N is defined as

$$S_N = \sum_{n=1}^{N} a_n \ .$$

We shall use the partial sum as an approximation of S. The corresponding truncation error, the *remainder term*, is

$$R_N = S - S_N = \sum_{n=N+1}^{\infty} a_n \ .$$

We shall estimate the truncation error, ie find an upper bound on $|R_N|$.

First, consider the case when the series is alternating and the absolute value of the terms tends monotonically to zero:

$$a_n a_{n+1} < 0, \quad |a_n| > |a_{n+1}|, \ n = N, N+1, \ldots, \quad \lim_{x\to\infty} a_n = 0 \ .$$

Figure 3.1 illustrates the partial sums as a function of N. We have chosen

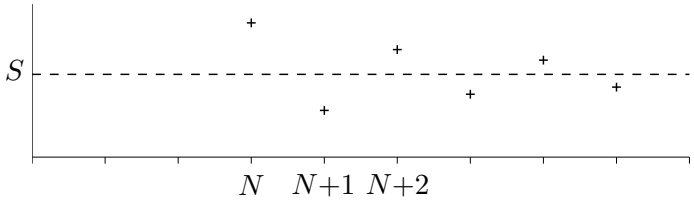


Figure 3.1. *Partial sums of an alternating series.*

N so that $S_N \geq S$. Then it is seen that $S_{N+1} \leq S$, $S_{N+2} \geq S$, etc. The partial sums S_N, S_{N+2}, S_{N+4}, ... form a monotone, bounded sequence. Thus, the series is convergent. We also see that the remainder term can be approximated by the first neglected term.

The remainder term of a convergent, alternating series can be estimated by the first neglected term,

$$|R_N| \leq |a_{N+1}| \ .$$

Example. Let $S = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n^2}$. We shall compute S with three correct decimals. How many terms are needed in the partial sum?

The condition is $|R_N| \le \frac{1}{(N+1)^2} \le 0.5 \cdot 10^{-3}$, which is equivalent to $N \ge \sqrt{2000} - 1 \simeq 43.7$. Hence, we must use 44 terms in the partial sum.

We can easily get a good approximation of S with somewhat less work: From Figure 3.1 we see that if we approximate S by $S_N + \frac{1}{2}a_{N+1}$, then the error estimate is only half as large,

$$S = \left(S_N + \tfrac{1}{2}a_{N+1}\right) \pm \tfrac{1}{2}|a_{N+1}| \ .$$

With this improvement we get the inequality $\frac{1}{2(N+1)^2} \le 0.5 \cdot 10^{-3}$, which leads to $N \ge \sqrt{1000} - 1 \simeq 30.6$. Thus, we only need to include 31 terms. ∎

Now, consider the remainder term of a positive series, ie a series with positive terms, $S = \sum_{n=1}^{\infty} a_n$, $a_n \ge 0$. In some cases such a series can be written

$$S = \sum_{n=1}^{\infty} f(n) \ ,$$

where $f(x)$ is a simple function, which is positive and monotonically decreasing for x sufficiently large. If a primitive function[1)] F of f is known, then we can use the following relation between a sum and an integral,

$$R_N = \sum_{n=N+1}^{\infty} f(n) \le \int_N^{\infty} f(x)\,dx = -F(N) \ .$$

(We have of course assumed that $F(x) \to 0$ as $x \to \infty$). Figure 3.2 shows that the inequality holds.

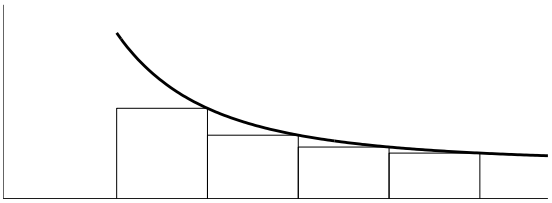


Figure 3.2. *Estimation of a remainder term by an integral.*

[1)] F is a *primitive function* for the function f, if $F'(x) = f(x)$.

Estimation by an integral. Let $S = \sum_{n=N+1}^{\infty} f(n)$ with $f(x)$ positive and monotonically decreasing for $x > N$. The remainder term can be estimated by

$$R_N = \sum_{n=N+1}^{\infty} f(n) \le \int_N^{\infty} f(x)\,dx \ .$$

Example. Let $S = \sum_{n=1}^{\infty} \frac{1}{n^4}$. We shall compute S with three correct decimals.

The integral estimate gives

$$R_N \le \int_N^{\infty} x^{-4}\,dx = \frac{1}{3N^3} \le 0.5 \cdot 10^{-3} \ ,$$

so that $N \ge (2000/3)^{1/3} \simeq 8.7$. To get three correct decimals we must therefore include 9 terms in the partial sum. We compute

$$S \simeq 1.081937 \ .$$

Here, we have summed backwards, ie started with the smallest terms, cf Section 2.7. Our computer is assumed to have unit roundoff $\mu < 10^{-7}$, and therefore rounding errors in the summation are negligible compared to the other errors. The remainder term estimate becomes

$$R_9 \le \frac{1}{3 \cdot 9^3} \le 0.458 \cdot 10^{-3} \ ,$$

and we get $S = 1.081937 \pm 0.458 \cdot 10^{-3} = 1.0819 \pm 0.0005$. ∎

Another method for estimating the remainder term of a positive series is sometimes useful: If each term in a positive series is less than the corresponding term in another series, whose sum is known (there is an analytic expression for it), then one can estimate the remainder term, using the known series.

Comparison with a known series. Assume that

$$0 \le a_n \le b_n \ , \quad n \ge N+1 \ ,$$

and that $T_N = \sum_{n=N+1}^{\infty} b_n$ is known (and convergent). Then

$$R_N = \sum_{n=N+1}^{\infty} a_n \le \sum_{n=N+1}^{\infty} b_n = T_N \ .$$

In many cases the known series is a geometric series

$$T_N = \sum_{n=N+1}^{\infty} c \cdot r^{n-(N+1)}$$
$$= c\left(1 + r + r^2 + \cdots\right) = \frac{c}{1-r} \ , \quad |r| < 1 \ .$$

Example. The first two terms in the Maclaurin series[2)] for the exponential function are

$$e^x \simeq 1 + x \ .$$

In how large an interval does this approximation give us four correct decimals?

The remainder term is

$$R_1 = \sum_{n=2}^{\infty} \frac{x^n}{n!} \ .$$

There is no simple way of estimating this by an integral. Therefore we rewrite the series so that we can estimate by a geometric series:

$$R_1 = \frac{x^2}{2}\left(1 + \frac{x}{3} + \frac{x^2}{3\cdot 4} + \frac{x^3}{3\cdot 4\cdot 5} + \cdots\right)$$
$$\leq \frac{x^2}{2}\left(1 + \frac{x}{3} + \frac{x^2}{3\cdot 3} + \frac{x^3}{3\cdot 3\cdot 3} + \cdots\right) = \frac{x^2/2}{1 - x/3} \ .$$

By solving the inequality

$$\frac{x^2/2}{1-x/3} \leq 0.5 \cdot 10^{-4} \ ,$$

(do that!) we find that $e^x \simeq 1 + x$ gives four correct decimals for

$$0 \leq x \leq 0.009983 \ .$$

■

3.3. Standard Functions

We already noted the importance of being able to evaluate standard functions efficiently and accurately on a computer. Most programming languages include standard functions that can be used without having to worry about how they are implemented. In some applications, however,

[2)] A *Maclaurin series* is a Taylor series expansion of a function about 0.

where one works with a very simple processor, it may be necessary for the programmer to implement the standard functions that are needed.

In this and the next sections we give a brief introduction to some methods for implementing standard functions. First, we give examples of approximations of elementary functions, that can be used for the implementations. It is outside the scope of this book to show how these approximations can be derived. Some basic principles are given in Chapter 9, and we refer to the literature given at the end of this chapter and Chapter 9. Also see Section 4.7, where we discuss an implementation of the square root function.

As a rule, the functions are only approximated for small arguments. In the next section we describe how they can be computed for arbitrary argument by means of so-called range reduction.

The basic requirements when implementing standard functions are

1) the relative error of the approximation must be smaller than a given tolerance — in general the unit roundoff of the floating point system;
2) the algorithm for computing the approximate function values must be fast.

Since we only approximate the function for small arguments, it is natural to use a series expansion around $x = 0$. As an example, consider the approximation by the first four terms in the Maclaurin expansion

$$\sin x \simeq p(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \ .$$

Using the above estimate of the remainder term for an alternating series, we see that for $0 \le x \le \pi/2$ this approximation has an *absolute error* less than

$$\frac{(\pi/2)^9}{9!} < 1.61 \cdot 10^{-4} \ .$$

Maclaurin expansions are accurate for small arguments, but they become increasingly bad as we move away from the origin. One can determine the coefficients of a polynomial so that it becomes a good approximation of the function (in our case the sine function) in *the whole interval*, cf Chapter 9. Figure 3.3 shows the relative error when $\sin x$ is approximated by $p(x)$ and another degree 7 polynomial

$$\sin x \simeq q(x) = b_0 x + b_1 x^3 + b_2 x^5 + b_3 x^7 \ ,$$

where

$$b_0 = 0.9999990604, \qquad b_1 = -0.1666555396 \ ,$$
$$b_2 = 0.0083118989, \qquad b_3 = -0.0001848812 \ .$$

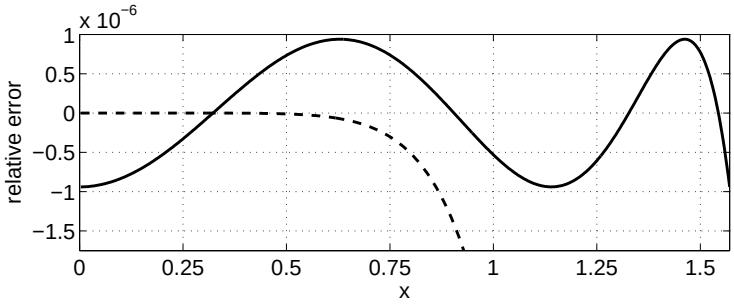


Figure 3.3. *Relative error in the approximation of* $\sin x$ *by the polynomials* $p(x)$ *(dashed line) and* $q(x)$ *(full line).*

It is seen that the error $|(p(x) - \sin x)/\sin x|$ is very small for $x \lesssim 0.75$, but it grows fast when x is larger. The relative error with the approximation $q(x)$ is bounded by 10^{-6} for $0 < x \leq \pi/2$. Both $p(x)$ and $q(x)$ can be evaluated at the cost of five multiplications and three additions: First compute x^2 (one multiplication) and then the polynomial according to the formula[3)]

$$q(x) = (((b_3 \cdot x^2 + b_2) \cdot x^2 + b_1) \cdot x^2 + b_0) \cdot x \ .$$

Some functions can be computed even more efficiently if they are approximated by a *rational function*,

$$f(x) \simeq \frac{p(x)}{q(x)} \ ,$$

where p and q are polynomials. As an example we consider the function $f(x) = 2^x$. In the next section we shall see that if we can get a good approximation of 2^x for small values of x, then it is convenient to approximate the exponential function by using this and the relation

$$e^x = 2^{x \log_2 e} \ .$$

If we approximate 2^x on the interval $[-\frac{1}{2}, \frac{1}{2}]$ by a polynomial,

$$2^x \simeq p(x) \ ,$$

and require that the relative error be less than 10^{-10}, then we must use a

[3)] This method is called *Horner's rule*. It is discussed further in Section 4.6.

polynomial of degree 7. To evaluate this polynomial (with Horner's rule) we need 7 multiplications and 7 additions. If, instead we use the rational approximation

$$2^x \simeq r(x) = \frac{q(x^2) + xs(x^2)}{q(x^2) - xs(x^2)} \ ,$$

where

$$q(y) = 20.8189237703844 + y \ ,$$

$$s(y) = 7.2152891433094 + 0.0576900726822y \ ,$$

then we also get the required accuracy, but the computation is considerably faster (provided that division is not much slower than multiplication). The cost is 3 multiplications, 4 additions and 1 division.

Example. The following MATLAB function implements this rational approximation of 2^x. If `x` is a vector, then `r` is a vector of the same type, with `r(i)` holding the approximation to $2^{\texttt{x(i)}}$.

```
function  r = exp2(x);
% Rational approximation of  2^x
x2 = x.*x;
q = 20.8189237703844 + x2;
xs = x.*(7.2152891433094 + 0.0576900726822*x2);
r = (q + xs) ./ (q - xs);
```

■

The relative error in the approximation is shown in Figure 3.4.

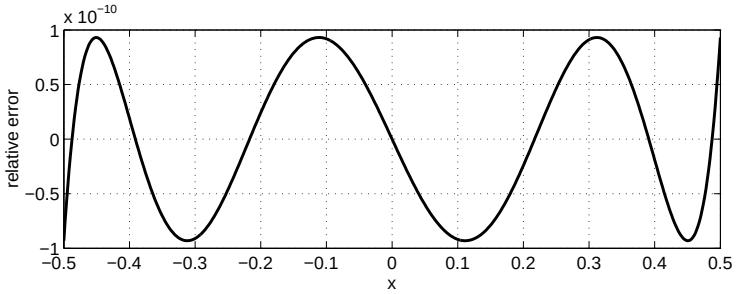


Figure 3.4. *Relative error in the approximation of* 2^x *by the rational function* $r(x)$.

Note that the function satisfies

$$2^{-x} = \frac{1}{2^x} \ ,$$

and our choice of the form of the rational approximation satisfies the same

relation,

$$r(-x) = \frac{q(x^2) - xs(x^2)}{q(x^2) + xs(x^2)} = \frac{1}{r(x)} \ .$$

This similarity is one of the reasons why this simple $r(x)$ is such a good approximation of 2^x.

In conclusion we state that as a rule there are better approximations of the standard functions than the Maclaurin expansions. For some functions it is more efficient to approximate by rational functions. In both cases the coefficients are stored, possibly in hardware. There are books with tables of approximations of the standard functions; see the references at the end of this chapter.

3.4. Range Reduction

In the previous section we gave examples of approximations of standard functions for *small* arguments. In this section we shall discuss how to do *range reduction*, ie use mathematical identities to reduce the function evaluation for an arbitrary argument to the evaluation for a small argument.

For trigonometric functions one uses the periodicity, of course. We take $\sin x$ as an example. Assume that we have a function $A(x)$, eg a polynomial, that approximates $\sin x$ well for $0 \leq x \leq \pi/2$. Using the identity

$$\sin x = \sin(x + k \cdot 2\pi) \ ,$$

for any integer k, we see that a given argument x can be reduced to the interval $[-\pi, \pi]$:

$$v = x - p \cdot 2\pi \ ,$$

for some integer p. The ensuing computation can be described as follows

$$\begin{array}{rrll} \textbf{if} & v > \pi/2 & \textbf{then} & \sin x \simeq A(\pi - v) \\ \textbf{elseif} & v \geq 0 & \textbf{then} & \sin x \simeq A(v) \\ \textbf{elseif} & v \geq -\pi/2 & \textbf{then} & \sin x \simeq -A(-v) \\ \textbf{else} & & & \sin x \simeq -A(v + \pi) \end{array}$$

In all cases the argument of A is a number $u \in [0, \pi/2]$, which satisfies

$$u = |x - n\pi| \ ,$$

for some integer n. If the given x is large, there will be cancellation, cf Section 2.3. We shall analyze rounding errors in a floating point system with unit roundoff μ. We assume that $x - n\pi \geq 0$, that x and n are exact, and that π is represented to full accuracy in the floating point system, ie

$$\overline{\pi} = \pi(1+\epsilon_1), \qquad |\epsilon_1| \leq \mu \ .$$

(In the sequel any ϵ_k satisfies $|\epsilon_k| \leq \mu$). The computed approximation of u, $\overline{u} = fl[x - n\overline{\pi}]$ can then be written

$$\begin{aligned} \overline{u} &= (x - n\overline{\pi}(1+\epsilon_2))(1+\epsilon_3) \\ &= x(1+\epsilon_3) - n\pi(1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3) = u + \Delta u \ . \end{aligned}$$

If we neglect terms that are $O(\mu^2)$, we find that the error Δu can be estimated as

$$|\Delta u| \simeq |(x - n\pi)\epsilon_3 - n\pi(\epsilon_1 + \epsilon_2)| \leq (|u| + 2|n|\pi)\mu \ .$$

Example. Assume that $x = 1000$ and that we shall compute $\sin x$ in IEEE single precision, ie in the floating point system $(2, 23, -126, 127)$. We get

$$n = 318, \qquad u = 1000 - 318\pi = 0.973536\ldots \ .$$

The unit roundoff is $\mu = 2^{-24}$, and we get a bound for the absolute error,

$$|\Delta u| \leq 1999.1\mu < 1.2 \cdot 10^{-4} \ .$$

Now, we use the maximal error bound estimate (see page 18):

$$|\Delta(\sin x)| \leq |(\cos u)\Delta u| \leq (\cos 0.973) \cdot 1.2 \cdot 10^{-4} < 0.68 \cdot 10^{-4} \ ,$$

so that

$$\left| \frac{\Delta(\sin x)}{\sin x} \right| \leq \frac{0.68 \cdot 10^{-4}}{\sin 0.973} < 0.83 \cdot 10^{-4} \ .$$

This is, of course, an unacceptably large error in a floating point system with unit roundoff $\mu \simeq 6 \cdot 10^{-8}$. ∎

The cancellation is reduced if the range reduction is performed in double precision (or simulated extended precision; see the exercises at the end of this chapter).

Example. The IEEE double precision format has unit roundoff $\mu = 2^{-53} \simeq 1.1 \cdot 10^{-16}$. If the range reduction of the previous example is performed in double precision, we get the estimate

$$\left| \frac{\Delta(\sin x)}{\sin x} \right| \leq 1.6 \cdot 10^{-13} \ .$$

Go through the estimations! ∎

In the previous section we indicated that it may be suitable to implement the *exponential function* using the formula

$$e^x = 2^{x \log_2 e} \ .$$

The computation involves four steps

$$\begin{aligned} u &:= x \log_2 e \ , && (\log_2 e \text{ is assumed to be stored accurately}) \\ v &:= u - [u] \ , && ([u] \text{ denotes the closest integer}) \\ w &:= 2^v \ , && (\text{rational approximation}) \\ y &:= w \cdot 2^{[u]} \ . \end{aligned}$$

The algorithm is based on using the identity

$$2^{v+[u]} = 2^v \cdot 2^{[u]}$$

for the range reduction, and it is seen that $-0.5 \leq v \leq 0.5$. If we work in a floating point system with base 2, then the multiplication by $2^{[u]}$ is performed simply by adding the integer $[u]$ to the exponent of w.

Example. Assume that we shall implement the exponential function in IEEE single precision, ie in the floating point system $(2, 23, -126, 127)$. Then e^x is to be computed for $-87.33 \leq x \leq 88.72$ (smaller values of x give underflow and larger values give overflow), and the result shall have a relative error less than the unit roundoff $\mu = 2^{-24} \simeq 6 \cdot 10^{-8}$. We shall examine how the error in the first step of the above algorithm is propagated in the computation.

To meet the accuracy requirement, the statement

$$u := x \cdot \log_2 e$$

must be executed in an extended precision format with unit roundoff μ_1, say. Then we get the computed value

$$\overline{u} = u(1+\epsilon_1)(1+\epsilon_2), \qquad |\epsilon_1|, |\epsilon_2| \leq \mu_1 \ ,$$

where the two error factors come from roundoff errors in the representation of $\log_2 e$ and the multiplication, respectively. We shall assume that $[\overline{u}] = [u]$, and get

$$\overline{v} = \overline{u} - [\overline{u}] = u - [u] + u(\epsilon_1 + \epsilon_2) = v + u(\epsilon_1 + \epsilon_2) \ .$$

Thus, the error in $\overline{v}$ is bounded by

$$|\Delta v| \lesssim 2|u|\mu_1 \ .$$

The error in v is propagated in the value $w = 2^v$. Using the maximal error estimate we get

$$\Delta w \simeq \frac{dw}{dv} \Delta w = w \log 2 \cdot \Delta v \ ,$$

and the relative bound

$$\left|\frac{\Delta w}{w}\right| \lesssim \log 2 \cdot 2|u|\mu_1 = 2|x|\mu_1 \ .$$

In the reformulation we used that $u = x \log_2 e = x/\log 2$.

How many extra bits are needed to ensure that the relative error in w is less than μ for all x such that $|x| \leq 88.72$? Let the significand in the extended format have $24+s$ bits, then the condition is $2 \cdot 88.72 \cdot 2^{-24-s} \leq 2^{-24}$, which gives

$$s \geq \frac{\log(2 \cdot 88.72)}{\log 2} \simeq 7.47 \ .$$

Thus, the significand of the extended format must have at least 8 extra bits. This is in accordance with the IEEE standard, see Section 2.8.

IEEE double precision is the floating point system $(2, 52, -1023, 1024)$. In this system the largest x such that e^x does not overflow is $x \simeq 709.98$, and a similar error estimate leads to

$$s \geq \frac{\log(2 \cdot 709.98)}{\log 2} \simeq 10.47 \ .$$

This is satisfied when the double precision extended format conforms with the IEEE standard, which requires at least 11 extra bits in the significand. ∎

3.5. An Algorithm for Evaluation of Trigonometric Functions

Earlier in this chapter we discussed how elementary functions can be evaluated using polynomial or rational approximations. These are the classical techniques, that are relatively easy to implement, eg in assembler code. When a processor is designed using VLSI technology, however, it may be advantageous to implement the elementary functions at a lower level, ie with simpler operations than multiplications. This is possible because, with VLSI one can construct more complicated and larger (meaning with more simple components) systems than with earlier technology. What is gained is speed; function values can be computed in approximately the same time as it takes to compute one or a couple of divisions.

In this section we shall describe a method for computing the trigonometric functions, which is called *Cordic* (Coordinate rotation digital computer). It was first presented in 1959, and it has, eg, been used for

INTEL's 8087 processor. It can be generalized to compute square roots, exponential and logarithmic functions, and also multiplication and division. Our presentation is simplified; we aim at discussing the basic principles rather than describing an actual implementation.

Let β be an angle, given in radians, and assume that $0 < \beta < \pi/2$ and that we shall compute $\sin\beta$. For reasons of speed the computations in the Cordic algorithm are performed in fixed point, which means that it cannot be used if β is very small. We start by identifying a range $[0, a]$ with the "very small" numbers, and show what to do there. Next, for $\beta \in [a, \pi/2]$ the basic idea is to write β in the form

$$\beta = v_0 \pm \gamma_0 \pm \gamma_1 \pm \gamma_2 \pm \cdots ,$$

where the $\{\gamma_i\}$ is a given decreasing sequence of angles, chosen so that the sequence is finite for a finite precision number β, and so that $\sin\beta$ is computed by a recursive formula that only involves shifts of the significand and additions.

The computer is assumed to have a binary floating point system $(2, t, L, U)$, and the Cordic algorithm is executed in fixed point format with $2t$ bits. In the floating point system $\beta \in [0, \pi/2]$ is given by

$$\beta = m2^{-e} ,$$

where m is the normalized significand and e is a nonnegative integer. To transform β to the fixed point format, we put the significand $m = 1.d_1d_2\ldots d_t$ in a register with t extra bits, and shift it e steps to the right, see Figure 3.5. If $e > t$, then m will be shifted partly or completely outside the register, and loss of accuracy occurs.

$$1.d_1d_2\ldots d_t\underbrace{00\ldots0}_{t \text{ zeros}} \quad \longrightarrow \quad \underbrace{0.0\ldots0}_{e \text{ zeros}}1d_1d_2\ldots d_t0\ldots0$$

Figure 3.5. *Conversion to fixed point format.*

For very small β the approximation

$$\sin\beta \simeq \beta$$

is good. According to Section 3.2 the relative truncation error is approximately

$$\frac{\frac{1}{6}\beta^3}{\beta} = \frac{1}{6}\,\beta^2 .$$

How large can $\beta = m2^{-e}$ be without this estimate exceeding the unit roundoff? The answer is the solution to the inequality

$$\tfrac{1}{6}\left(m2^{-e}\right)^2 \leq \mu = \tfrac{1}{2}2^{-t} \ .$$

This must hold for all $m \in [1, 2[$, and we get

$$\tfrac{4}{6}2^{-2e} \leq \tfrac{1}{2}2^{-t} \ ,$$

which is equivalent to

$$e \geq \tfrac{1}{2}(-\log_2 0.75 + t) \ .$$

In other words: If $e \geq \frac{1}{2}(t+1)$, then the relative error in the approximation $\sin\beta \simeq \beta$ is less than μ. This means that we need only use the Cordic algorithm for angles

$$\beta = m2^{-e}, \qquad 0 \leq e \leq \lceil (t+1)/2 \rceil \ ,$$

where $\lceil p \rceil$ denotes the integer part of p. This implies, cf Figure 3.5, that all the shifted digits stay inside the register, ie no errors are introduced in the conversion from floating point to fixed point format.

Next, we describe the basic idea in the Cordic algorithm without any assumptions about the representation of β. This is an angle, given in radians, and we shall compute $\sin\beta$, which is the y-coordinate of the vector v in the figure.

The computation can be done as follows: Start with the vector

$$v_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \ .$$

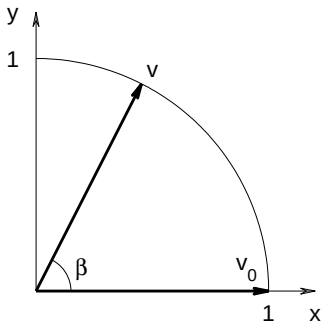


Figure 3.6. *Compute* $\sin\beta$.

Rotate this vector by a given angle γ_0 in the positive direction, and let v_1 denote the corresponding vector, see Figure 3.7.

Define $\beta_0 = \beta$, and put

$$\beta_1 = \beta_0 - \gamma_0 \ .$$

If $\beta_1 > 0$, then rotate v_1 a given angle γ_1 in the positive direction, denote the corresponding vector v_2 (see Figure 3.8) and put

$$\beta_2 = \beta_1 - \gamma_1 \ .$$

(If $\beta_1 < 0$, ie if v_1 has passed v, then rotate v_1 the angle $-\gamma_1$, and put

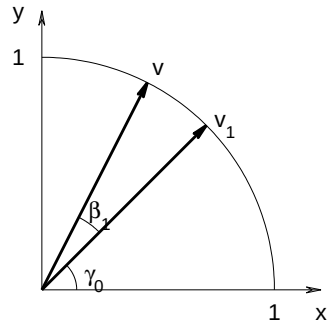


Figure 3.7. *Rotate v_0 to the position v_1.*

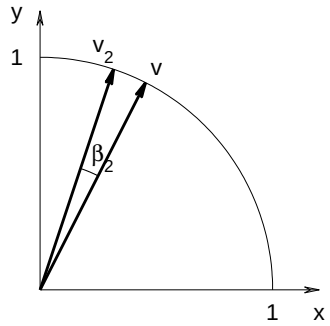


Figure 3.8. *The angle β_2 is negative.*

$\beta_2 = \beta_1 + \gamma_1$ instead). In Figure 3.8 the vector v_2 has passed v and β_2 is negative.

We continue in this way, successively rotating the vector v_i to v_{i+1} by an angle $-\gamma_i$ or γ_i depending on, whether v_i has passed v or not. The process stops when v_i approximates v to the required accuracy. Before we discuss the choice of the γ_i we need a precise formulation of the rotations.

Let (x_i, y_i) be the coordinates of the vector v_i. This vector has length one, and we let θ_i denote its angle with the x-axis. Then

$$x_i = \cos\theta_i, \qquad y_i = \sin\theta_i \ ,$$

and the relation $\theta_{i+1} = \theta_i + \gamma_i$ and use of well-known trigonometric formulas give

$$\begin{aligned} x_{i+1} &= \cos(\theta_i + \gamma_i) = \cos\theta_i \cos\gamma_i - \sin\theta_i \sin\gamma_i = x_i \cos\gamma_i - y_i \sin\gamma_i \ , \\ y_{i+1} &= \sin(\theta_i + \gamma_i) = \sin\gamma_i \cos\theta_i + \cos\gamma_i \sin\theta_i = x_i \sin\gamma_i + y_i \cos\gamma_i \ . \end{aligned}$$

This shows that the rotation from v_i to v_{i+1} can be expressed by

$$v_{i+1} = P_i v_i, \qquad P_i = \begin{pmatrix} \cos\gamma_i & -\sin\gamma_i \\ \sin\gamma_i & \cos\gamma_i \end{pmatrix} \ .$$

For obvious reasons the matrix P_i is called a *rotation matrix*. We shall discuss such matrices in more detail in Section 8.15.

Let us summarize:

Cordic algorithm; preliminary version. Let $\gamma_0, \gamma_1, \ldots$ be a given sequence of rotation angles, and define initial values

$$v_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad \beta_0 = \beta \ .$$

The vectors v_i and angles β_i are computed recursively from

$$v_{i+1} = P_i v_i, \quad \beta_{i+1} = \beta_i - \sigma_i \gamma_i, \quad i = 0, 1, 2, \ldots \ ,$$

where

$$\sigma_i = \text{sign}(\beta_i), \qquad P_i = \begin{pmatrix} \cos\gamma_i & -\sigma_i \sin\gamma_i \\ \sigma_i \sin\gamma_i & \cos\gamma_i \end{pmatrix} \ .$$

This is an iterative method for computing the coordinates of the vector v, ie $\cos\beta$ and $\sin\beta$. It can be shown to converge for suitable choices of the angles $\{\gamma_0, \gamma_1, \ldots\}$. (If, eg, we choose $\gamma_i = 2^{-i}\pi/4$, then the algorithm has similarities with the bisection method).

This preliminary version of the algorithm is neither fast nor simple. Each step of the recursion involves four multiplications and three additions. Now we shall see that the algorithm is a finite recursion when it is applied to an angle β given in fixed point format, and that it can be implemented efficiently in a computer, if the angles γ_i are chosen appropriately. We remind the reader that our presentation is greatly simplified.

The matrix-vector multiplications are the most costly part of the algorithm. Introduce

$$c_i = \cos\gamma_i, \quad s_i = \sin\gamma_i, \quad t_i = \frac{s_i}{c_i} = \tan\gamma_i \ .$$

Then we can write the ith rotation in the form

$$P_i v_i = \begin{pmatrix} c_i & -\sigma_i s_i \\ \sigma_i s_i & c_i \end{pmatrix} v_i = c_i Q_i v_i, \quad Q_i = \begin{pmatrix} 1 & -\sigma_i t_i \\ \sigma_i t_i & 1 \end{pmatrix} \ .$$

If we choose the angles γ_i so that

$$t_i = 2^{-i} \ ,$$

then the multiplication by the matrix Q_i becomes very simple: two shift operations and two additions,

$$Q_i v_i = \begin{pmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} x_i - \sigma_i 2^{-i} y_i \\ \sigma_i 2^{-i} x_i + y_i \end{pmatrix} \ .$$

Assume that also x_i and y_i are stored in fixed point format. Then it is obvious that the algorithm is finite: after $2t$ steps the shift is so large

that everything is shifted outside the register. The matrix Q_i becomes an identity matrix in the fixed point arithmetic. The recursion for β_i is

$$\beta_{i+1} = \beta_i - \sigma_i \gamma_i, \quad \gamma_i = \arctan 2^{-i}, \quad i = 0, 1, 2, \ldots \ .$$

Since $\arctan x \leq x$, this also shows that the recursion stops after $2t$ steps: γ_{2t} cannot be represented in the fixed point format.

The approximation of the vector v is

$$\begin{aligned} v_{2t} &= c_{2t-1} \cdots c_1 c_0 Q_{2t-1} \cdots Q_1 Q_0 v_0 \\ &= \tau Q_{2t-1} \cdots Q_1 Q_0 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= Q_{2t-1} \cdots Q_1 Q_0 \begin{pmatrix} \tau \\ 0 \end{pmatrix} \ . \end{aligned}$$

Note that $\tau = c_{2t-1} \cdots c_1 c_0$ depends on the choice of $\{\gamma_i\}$, not on the value of β. Therefore, τ can be computed once and for all.

We summarize:

The Cordic algorithm.

$v := \begin{pmatrix} \tau \\ 0 \end{pmatrix}$;
for $i := 0, 1, 2, \ldots, 2t-1$ **do**
begin
  $\sigma := \text{sign}(\beta)$;
  $v := \begin{pmatrix} 1 & -\sigma 2^{-i} \\ \sigma 2^{-i} & 1 \end{pmatrix} v$;
  $\beta := \beta - \sigma \gamma_i$;
end

The result is $v = \begin{pmatrix} \cos\beta \\ \sin\beta \end{pmatrix}$ in the fixed point format.

We wish to emphasize the following:

1. The algorithm uses fixed point arithmetic only. This is faster than floating point arithmetic. The operations are simple: shifts and additions. The only logical operation is a test of the sign of β.

2. The angles $\gamma_i = \arctan 2^{-i}$, $i = 0, 1, \ldots, 2t-1$, must be stored. For small angles, however, we have $\arctan 2^{-i} = 2^{-i}$ in the finite precision, and we do not need to store these angles. This way we can reduce the size of the table.

Example. As a simple example, let $t = 4$ and $\beta = 1.1875 = (1.0011)_2$. We find $\tau \simeq 0.6073$, which is stored in the fixed point format as $(0.10011011)_2$. In the table below we give the performance of the algorithm (in the same format).

i	γ_i	σ_i	$\lvert\beta_i\rvert$	x_{i+1}	y_{i+1}
0	$(0.11001001)_2$	1	$(1.00110000)_2$	$(0.10011011)_2$	$(0.10011011)_2$
1	$(0.01110111)_2$	1	$(0.01100111)_2$	$(0.01001110)_2$	$(0.11101000)_2$
2	$(0.00111111)_2$	-1	$(0.00010000)_2$	$(0.10001000)_2$	$(0.11010101)_2$
3	$(0.00100000)_2$	1	$(0.00101111)_2$	$(0.01101110)_2$	$(0.11100110)_2$
4	$(0.00010000)_2$	1	$(0.00001111)_2$	$(0.01100000)_2$	$(0.11101100)_2$
5	$(0.00001000)_2$	-1	$(0.00000001)_2$	$(0.01100111)_2$	$(0.11101001)_2$
6	$(0.00000100)_2$	1	$(0.00000111)_2$	$(0.01100100)_2$	$(0.11101010)_2$
7	$(0.00000010)_2$	1	$(0.00000011)_2$	$(0.01100011)_2$	$(0.11101010)_2$

The resulting approximation to $\sin\beta$ is $(1.1101)_2 \cdot 2^{-1} = 0.90625$. The true value is $\sin\beta = 0.92743\ldots$, and the relative error is

$$\frac{|0.90625 - \sin\beta|}{\sin\beta} \simeq 0.02284 < \mu = \tfrac{1}{2}2^{-4} = 0.03125\ .$$

■

Exercises

E1. Assume that $\sin x$ shall be computed in IEEE single precision and that we do not have access to extended precision. In the range reduction

$$u = x - n\pi$$

we can reduce cancellation by using the idea from Exercise E15(b) in the previous chapter: Write π in the form

$$\pi = \pi_0 + R\ ,$$

where π_0 is exactly representable in the floating point system, with a number of trailing zeros, eg

$$\pi_0 = (3.140625)_{10} = 2^1 \cdot (1.1001001)_2\ .$$

In the floating point system $(2, 23, -126, 127)$ this number has 16 trailing zeros. The error is

$$R = \pi - \pi_0 = 0.00096765\ldots\ ,$$

ie a small number. Let r denote the representation of R in the floating point system.

Now, the reduced argument is computed as

$$u = (x - n\pi_0) - nr \ .$$

x is a floating point number, and the difference in the parenthesis can be computed without error if n is not too large.

Let $x = 1000$ and $n = 318$ (cf the examples on page 54).

(a) Show that $(x - n\pi_0)$ is computed without error.

(b) Estimate the error in the computed $\overline{u}$.

E2. Cancellation in connection with range reduction for a trigonometric function can be disastrous when the argument x is close to an integer multiple of π. Let $u = x - n\pi$ and assume that u is small. Show that

$$\left|\frac{\Delta(\sin x)}{\sin x}\right| \lesssim \left|\frac{\Delta u}{u}\right| \ .$$

Assume that the range reduction is done in IEEE double precision. In how large an interval around $n\pi$ can we **not** compute $\sin x$ to full accuracy in single precision, ie with an error smaller than the unit roundoff? Only the errors from the range reduction are to be taken into account.

E3. Show that the recursions in the Cordic algorithm for the two components of the vector v are independent for $i > t$ and that

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} x_i - \sigma_i 2^{-i} y_t \\ y_i + \sigma_i 2^{-i} x_t \end{pmatrix}, \quad i = t+1, t+2, \ldots, 2t-1 \ .$$

E4. Compute τ in the Cordic algorithm for $t = 23$.

E5. For which values of i do we have $\arctan 2^{-i} \doteq 2^{-i}$ in a fixed point arithmetic with 46 binary digits and one integer digit?

References

Much of the theory for remainder term estimates is given in textbooks in analysis. More about transformation of series can be found in

C.-E. Fröberg, *Numerical Mathematics, Theory and Computer Applications*, The Benjamin/Cummings Publishing Company, Menlo Park, 1985.

The following books have tables of polynomial and rational approximations of standard functions

> J.F. Hart et al., *Computer Approximations*, John Wiley and Sons, New York, 1968.
>
> W.J. Cody, Jr. and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Englewood Cliffs, N.J., 1980.

The Cordic algorithm is described in more detail in

> M.D. Ercegovac and T. Lang, *Digital Arithmetic*, Elsevier Science, 2003.

Chapter 4

Nonlinear Equations

4.1. Introduction

Example. A real root of the equation

$$f(x) = x - \cos x = 0$$

is an intersection between the graph of f and the x-axis. From the figure we see that the equation has a root x^* close to 0.75.

How can we determine a good approximation to the root, and how can we estimate the accuracy of the approximation?

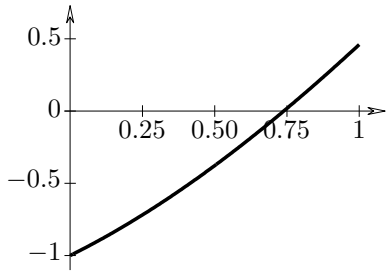


Figure 4.1. *The function* $f(x) = x - \cos x$.

■

In this chapter we study methods for solving an equation

$$f(x) = 0 \ ,$$

where f is a continuous, real-valued function of a real variable. When needed, we require further regularity properties, eg that f is differentiable. In Section 4.8 we shall briefly discuss the solution of systems of nonlinear equations.

Normally, a nonlinear equation $f(x) = 0$ cannot be solved analytically, ie the solution cannot be expressed in a form involving only elementary arithmetic operations and square roots. Algebraic equations (equations where $f(x)$ is a polynomial) can be solved analytically if the degree is at most four, but it can be shown that this is in general not possible for higher degrees. Also, it is generally not possible to give an explicit expression for the solution to a transcendent equation (that includes tran-

scendental function like the exponential function and trigonometric functions). Sometimes we do not even know an analytic expression for $f(x)$; it may eg be defined as the solution to a differential equation, see Chapter 10. In all these cases one has to use a numerical method to solve the equation $f(x) = 0$.

Let x^* denote a root of the nonlinear function f. Then we can write

$$f(x) = (x - x^*)^q g(x) \ ,$$

with $g(x^*) \neq 0$. The exponent q is the *multiplicity* of the root. It follows that

$$f'(x) = q(x - x^*)^{q-1} g(x) + (x - x^*)^q g'(x) \ ,$$

and if $q > 1$ then also $f'(x^*) = 0$. Unless otherwise indicated, we shall assume that x^* is a *simple root*, ie $q = 1$, in which case $f'(x^*) \neq 0$.

The basic idea behind most numerical methods for solving nonlinear equations is, first to find a crude approximation x_0 to the root. Next, from x_0 construct a sequence $\{x_k\}_{k=1}^{\infty}$ that converges to the root,

$$\lim_{k \to \infty} x_k \ = \ x^* \ .$$

A method that produces such a sequence is called an *iteration method*.

We start by describing some ways of constructing iteration methods. Next, we discuss conditions for convergence. In most applications it is desirable that the sequence converges fast. We define order of convergence and investigate it for some methods. In practice calculations are made with finite precision, and we examine the accuracy that can be obtained. In case of convergence this accuracy is achieved after a finite number of steps, so that the infinite sequence $\{x_k\}_{k=1}^{\infty}$ is stopped after a finite number of steps. We also describe a method for implementing the square root function on a computer.

4.2. Crude Localization

When solving a nonlinear equation $f(x) = 0$ you first have to find a crude approximation x_0 to the desired root. There are three ways to do this,

1. from a graphical presentation of the function,
2. by tabulating the function,
3. by use of the bisection method.

From a graph one can often get more information than just the approximate location of the root.

Example. Figure 4.1 shows that the function $f(x) = x - \cos x$ has only one root in the interval $[0, 1]$ and it is close to $x_0 = 0.75$. If we draw the function in a larger interval we see that the function has precisely one root.

We can make a small table of $f(x) = x - \cos x$:

x	$\cos x$	$f(x)$
0.7	0.7648	−0.0648
0.8	0.6967	0.1033

Since f is continuous and there is a sign change in the function values, there must be a root in the interval $[0.7, 0.8]$. For x_0 we can eg take one of the endpoints or the midpoint of this interval. ∎

When we know an interval that contains a root – a so-called *bracket* – we can successively refine it by the *bisection method*: Let $[a, b]$ denote the interval with $f(a) \cdot f(b) < 0$, and let m be the midpoint, $m = (a+b)/2$. If $f(m) \cdot f(b) > 0$, then the root is in $[a, m]$ ($b := m$), otherwise it is in $[m, b]$ ($a := m$), and this subdivision can be repeated.

Example. For the function $f(x) = x - \cos x$ we saw in the previous example that $[a, b] = [0.7, 0.8]$ is a bracket, and $f(b) > 0$. Since $f(0.75) = 0.0183 > 0$, we get the new bracket $[0.7, 0.75]$. Next, $f(0.725) < 0$, so that the next bracket is $[0.725, 0.75]$, etc.

The following MATLAB function implements the bisection method.

```
function  [a, b] = bisection(f, a0,b0, tol)
% Bisection to find root of  f.  Start interval [a0,b0]
% with  f(a0)*f(b0) < 0  (is not checked)
% Repeat until the interval is smaller than tol
a = a0;  b = b0;  sfb = sign(feval(f, b))  % initialize
while  b-a > tol
  x = (a + b)/2;   sfx = sign(feval(f, x));
  disp([a b sfx])
  if  sfx == 0                                % f(x) = 0
    a = x;  b = x;  break       % return with a = b = x
  elseif  sfx == sfb,  b = x;
  else,                a = x; end
end
```

We further define

```
function  y = fx(x)
y = x - cos(x);
```

Then the command

```
>> [a b] = bisection(@fx, 0.7, 0.8, 1e-3);
```

gives

```
sfb = 1
  a           b          sfx
0.70000000  0.80000000     1
0.70000000  0.75000000    -1
0.72500000  0.75000000    -1
0.73750000  0.75000000     1
0.73750000  0.74375000     1
0.73750000  0.74062500    -1
0.73906250  0.74062500     1
```

The resulting interval is $[a, b] = [0.73906250,\ 0.73984375]$. ∎

It is clear that this method always converges, but the speed of convergence is low. After k steps with the method the length of the interval is 2^{-k} times the length of the initial interval (except in the unlikely case where an interval midpoint is a root of f).

Example. In the previous example we started with an interval of length 0.1. In order to get $b-a \leq \texttt{tol} = 10^{-3}$ we have to reduce the interval length by a factor 100, so we need 7 steps ($2^7 = 128 > 100$, while $2^6 = 64 < 100$). To get $b-a \leq 10^{-6}$ we need 17 steps; the midpoint of the resulting interval is the root with 6 correct decimals. ∎

4.3. Iteration Methods

Because of the slow convergence the bisection method should only be used for crude localization of the root. In this section we describe methods with faster convergence. The idea is to use information about the function f and the approximate location of the root to compute a new and more accurate approximation. As a rule of thumb we can say that the more information about f one uses, the faster convergence one gets.

A natural idea is to approximate the curve $y = f(x)$ by its tangent at the current point, see Figure 4.2. Suppose that we have an approximation x_0 to the root. We take the tangent to the curve at the point $(x_0, f(x_0))$ and let the intersection between the tangent and the x-axis be

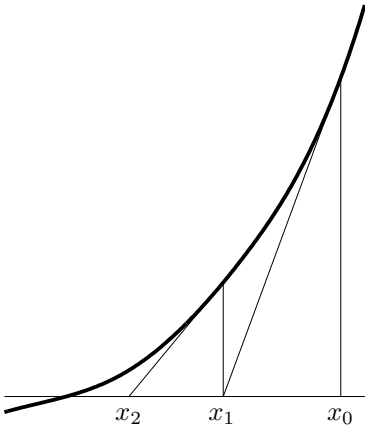


Figure 4.2. *Approximate the curve by its tangent.*

the next approximation x_1. The tangent has the equation

$$y - f(x_0) = f'(x_0)(x - x_0) \ ,$$

and by setting $y = 0$ we get the intersection with the x-axis,

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \ .$$

We can proceed the same way from this approximation to the root, and the general formula is

Definition 4.3.1. Newton-Raphson's method

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \ .$$

Instead of the geometric derivation we could have used an analytic approach: For a given approximation x_k to the root we seek h such that $f(x_k{+}h) = 0$. We expect h to be small, so that a good approximation is provided by the first two terms of a Taylor expansion around x_k,

$$f(x_k{+}h) \simeq f(x_k) + f'(x_k)h \ .$$

If $f'(x_k) \neq 0$, we can equate the right-hand side with zero and solve for h. This, of course, is only an approximation to the true value $h = x^* - x_k$, so we index the solution with k,

$$h_k = -\frac{f(x_k)}{f'(x_k)} \ ,$$

and take $x_{k+1} = x_k + h_k$ as the next approximation to the root. The result agrees with Definition 4.3.1.

In Newton-Raphson's method we use values of both the function and its derivative, and according to the rule of thumb in the first paragraph of this section we can expect fast convergence.

Example. The problem of the previous examples has

$$f(x) = x - \cos x, \qquad f'(x) = 1 + \sin x \ .$$

This can be implemented in MATLAB by

```
function  [f, df] = fdf(x)
f = x - cos(x);
df = 1 + sin(x);
```

The following MATLAB script performs two iterations with Newton-Raphson's method from the starting point $x_0 = 0.7$.

```
x = 0.7;
for  k = 0:1
  [f, df] = fdf(x);   dx = f/df;
  x = x - dx;
end
```

We printed out the results:

```
k     x_k               f(x_k)     f'(x_k)      dx
0   0.700000000000    -6.48e-02    1.64422   -0.039436497848
1   0.739436497848     5.88e-04    1.67387    0.000351337383
2   0.739085160465     4.56e-08    1.67361    0.000000027250
```

There seems to be fast convergence, but since we do not get the function value zero, we must ask the question: *How accurate is the approximation* $\overline{x} = x_2 = 0.739085160465$? The answer is given in Section 4.5. ∎

In the next section we shall study convergence criteria and rate of convergence in some detail. For that discussion it is practical to write Newton-Raphson's method in the form

$$x_{k+1} = \varphi(x_k) \ ,$$

where the *iteration function* $\varphi(x)$ is

$$\varphi(x) = x - \frac{f(x)}{f'(x)} \ .$$

Note that the equation $x = \varphi(x)$ is an equivalent way of writing $f(x) = 0$: the equation $f(x) = 0$ can be put into the form $x = \varphi(x)$ by elementary mathematical operations, and vice versa. Since x^* is a solution to the

equation $f(x) = 0$, we have

$$x^* = \varphi(x^*) \ .$$

x^* is said to be a *fixed point* of the map $x \mapsto \varphi(x)$, and an iteration method $x_{k+1} = \varphi(x_k)$ is called a *fixed point iteration*.

Fixed point iterations can be obtained in other ways than the one leading to Newton-Raphson's method.

Example. The equation $x - \cos x = 0$ can be reformulated to $x = \cos x$, and the corresponding iteration method is

$$x_{k+1} = \cos x_k \ .$$

Starting with $x_0 = 0.7$ we get the results shown in the following table. It seems that the sequence $x_1, x_2, \ldots$ converges and that the error is reduced by a factor about 0.7 in each iteration.

k	x_k	$\epsilon_k = x_k - x^*$	$\epsilon_k/\epsilon_{k-1}$
0	0.70000000000000	-3.9085e-02	
1	0.76484218728449	2.5757e-02	0.6590
2	0.72149163959753	-1.7593e-02	0.6831
3	0.75082132883945	1.1736e-02	0.6671
4	0.73112877257336	-7.9564e-03	0.6779
⋮			
48	0.73908513299150	-2.2366e-10	0.6736
49	0.73908513336582	1.5066e-10	0.6736
50	0.73908513311367	-1.0149e-10	0.6736

The iteration process is illustrated in Figure 4.3.

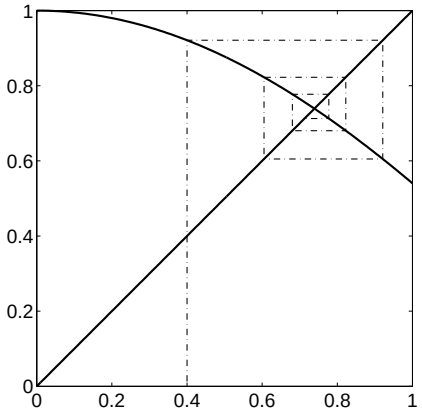


Figure 4.3. *Convergent fixed point iteration.* $x_0 = 0.4$.

Another equivalent reformulation of $f(x) = x - \cos x = 0$ is $x = \arccos x$. The iteration method $x_{k+1} = \arccos x_k$ is divergent, however:

k	x_k	$\epsilon_k = x_k - x^*$	$\epsilon_k/\epsilon_{k-1}$
0	0.70000000000000	-3.9085e-02	
1	0.79539883018414	5.6314e-02	1.4408
2	0.65113098931890	-8.7954e-02	1.5619
3	0.86172266836514	1.2264e-01	1.3943
4	0.53214115489645	-2.0694e-01	1.6874
5	1.00966880945946	2.7058e-01	1.3075

■

Obviously, we need to analyze under which conditions the function φ gives a convergent fixed point iteration $x_{k+1} = \varphi(x_k)$. This is the subject of the next section. Before that, however, we present an iteration method, which is not of the form $x_{k+1} = \varphi(x_k)$.

Consider the equation $f(x) = 0$, where f is differentiable, but the derivative is not readily available (an example is given in Section 10.11). Suppose that we have two approximations to the root, x_0 and x_1, then we can approximate the curve by the secant trough the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$ and take intersection between the secant and the x-axis as the next approximation x_2, see Figure 4.4.

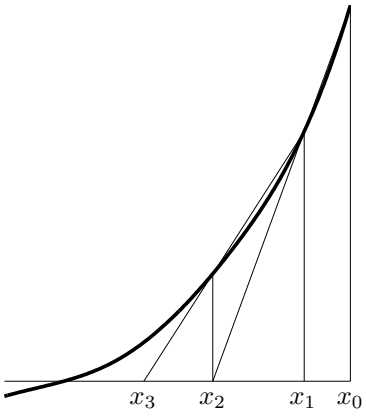


Figure 4.4. *Approximate the curve by a secant.*

The secant has the equation

$$y - f(x_1) = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0) \ ,$$

and by setting $y = 0$ we get the intersection with the x-axis,

$$x_2 = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)} \ .$$

This generalizes:

Definition 4.3.2. The secant method

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \ , \quad k = 1, 2, \ldots$$

The secant method can also be derived from Newton-Raphson's method by using the approximation[1)]

$$f'(x_k) \simeq \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \ .$$

Example. In the first example on page 67 we found that the equation $x - \cos x = 0$ has a root in the interval $[0.7, 0.8]$, and we use $x_0 = 0.7$, $x_1 = 0.8$. The following MATLAB script performs three steps with the secant method; the function `fx` was defined on page 67.

```
xx = [0.7 0.8];  ff = fx(xx);
for  k = 2 : 4
  nx = xx(2) - ff(2)*diff(xx)/diff(ff);
  xx = [xx(2) nx];   ff = [ff(2) fx(nx)];
end
```

We printed out the results:

```
k    x_k                 f(x_k)          x_k - x*
0  0.70000000000000   -6.48422e-02   -3.90851e-02
1  0.80000000000000    1.03293e-01    6.09149e-02
2  0.73856544025090   -8.69665e-04   -5.19693e-04
3  0.73907836214467   -1.13321e-05   -6.77107e-06
4  0.73908513399236    1.30073e-09    7.77202e-10
```

By looking at the function values and comparing with the table on page 70 we see that in this case the secant method converges almost as fast as Newton-Raphson's method. We return to this at the end of the next section. ∎

[1)] The right-hand side is a difference approximation to the derivative $f'(x_k)$; see Section 6.2

4.4. Convergence Criteria and Rate of Convergence

Consider the fixed point iteration

$$x_{k+1} = \varphi(x_k) \ .$$

Convergence of the sequence $\{x_k\}_{k=0}^{\infty}$ to x^* is equivalent with convergence of the sequence $\{x_k - x^*\}_{k=0}^{\infty}$ to 0. Since $x^* = \varphi(x^*)$, we can write

$$x_k - x^* = \varphi(x_{k-1}) - \varphi(x^*) \ ,$$

and by means of the mean value theorem we get

$$x_k - x^* = \varphi'(\xi_k)(x_{k-1} - x^*) \ ,$$

where ξ_k is between x_{k-1} and x^*. If

$$|\varphi'(\xi_k)| \le m < 1$$

for some constant m, then

$$|x_k - x^*| \le m|x_{k-1} - x^*| < |x_{k-1} - x^*| \ .$$

The condition $|\varphi'(x)| \le m < 1$ close to x^* is a sufficient condition for convergence, since then

$$\begin{aligned} |x_k - x^*| &\le m|x_{k-1} - x^*| \le m^2|x_{k-2} - x^*| \\ &\le \cdots \le m^k|x_0 - x^*| \ , \end{aligned} \tag{4.4.1}$$

and $m^k \to 0$ for $k \to \infty$. We formulate this more strictly as a theorem.

Theorem 4.4.1. Assume that the iteration function φ has a real fixed point x^*, and that

$$|\varphi'(x)| \le m < 1$$

for all $x \in \mathcal{I}$, where $\mathcal{I}$ is an interval around x^*,

$$\mathcal{I} = \{x \big| \ |x - x^*| \le \delta\} \ ,$$

for some δ. If $x_0 \in \mathcal{I}$, then

a) $x_k \in \mathcal{I}, \quad k = 1, 2, \ldots \ ,$

b) $\lim\limits_{k\to\infty} x_k = x^* \ ,$

c) x^* is the only root in $\mathcal{I}$ of the equation $x = \varphi(x)$.

Proof. a) is proved by induction. Assume that $x_{k-1} \in \mathcal{I}$. The mean value theorem gives

$$x_k - x^* = \varphi(x_{k-1}) - \varphi(x^*) = \varphi'(\xi_k)(x_{k-1} - x^*) \ ,$$

and since x_k lies between x_{k-1} and x^*, ξ_k must lie in $\mathcal{I}$. Therefore,

$$|x_k - x^*| \leq m|x_{k-1} - x^*| \leq m\delta < \delta \ ,$$

which means that $x_k \in \mathcal{I}$, and a) is proved.

From the above argument it follows that $|x_k - x^*| \leq m|x_{k-1} - x^*|$ is true for $k = 1, 2, \ldots$. Therefore (4.4.1) holds, and $m < 1$ implies that

$$\lim_{k\to\infty} |x_k - x^*| = 0 \ ,$$

and we have proved b).

Uniqueness is proved by contradiction: Assume that there is another point $\hat{x}^* \in \mathcal{I}$, $\hat{x}^* = \varphi(\hat{x}^*)$, $\hat{x}^* \neq x^*$. The mean value theorem and the assumptions give

$$|x^* - \hat{x}^*| = |\varphi'(\xi)| \cdot |x^* - \hat{x}^*| \leq m|x^* - \hat{x}^*| < |x^* - \hat{x}^*| \ ,$$

which is a contradiction. □

Example. Consider the two fixed point iterations from the example starting on page 71,

$$\varphi(x) = \cos x \ , \qquad \varphi(x) = \arccos x \ .$$

Both of these have the fixed point $x^* \simeq 0.739085$.

In the first case we have $\varphi'(x) = -\sin x$ and $|\varphi'(x)| \simeq 0.6736 < 0.7$ close to x^*. According to the theory the iteration is convergent, and this agrees with the experimental result.

In the second case we have $\varphi'(x) = -1/\sqrt{1-x^2}$ and $|\varphi'(x)| \simeq 1.485 > 1$ close to the root. Hence, the iteration is divergent. ■

From the proof of Theorem 4.4.1 its is seen that the smaller m is, the faster is the convergence of the iteration. For Newton-Raphson's method we get

$$\varphi(x) = x - \frac{f(x)}{f'(x)} \ , \qquad \varphi'(x) = \frac{f(x)f''(x)}{(f'(x))^2} \ .$$

We see that $\varphi'(x^*) = 0$, and if φ' is continuous, then $\varphi'(x)$ is small for x close to x^*. Therefore, Newton-Raphson's method should converge rapidly once one is close to the root. To see how fast the convergence is, we make a Taylor expansion around x^*:

$$\varphi(x_k) = \varphi(x^*) + \varphi'(x^*)(x_k - x^*) + \tfrac{1}{2}\varphi''(\xi_k)(x_k - x^*)^2 \ ,$$

where ξ_k is between x_k and x^*. Since $\varphi(x^*) = x^*$; $\varphi'(x^*) = 0$ and $\varphi(x_k) = x_{k+1}$, we get

$$x_{k+1} - x^* = \tfrac{1}{2}\varphi''(\xi_k)(x_k - x^*)^2 \ .$$

This shows that Newton-Raphson's method converges faster than fixed point methods in general.

In order to compare different iteration methods we make the following definition.

Definition 4.4.2. Let $x_0, x_1, x_2, \ldots$ be a sequence that converges to x^*. The *order of convergence* of the sequence is p, defined as the largest positive number such that

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^p} = C < \infty \ .$$

C is called the *asymptotic error constant*.

For $p = 1$ and $p = 2$ the convergence is said to be *linear* and *quadratic*, respectively.

Often we say that an iteration method has order of convergence p if it generates sequences with this order of convergence. In general, convergent fixed point iterations have linear convergence with asymptotic error constant $C = \varphi'(x^*)$. Newton-Raphson's method has quadratic convergence with

$$C = \tfrac{1}{2}\varphi''(x^*) = \frac{f''(x^*)}{2f'(x^*)} \ .$$

Example. For $a > 0$ we can compute $\sqrt{a}$ by solving the equation $f(x) = x^2 - a = 0$. Newton-Raphson's method applied to this equation gives

$$x_{k+1} = \tfrac{1}{2}\left(x_k + \frac{a}{x_k}\right) \ .$$

(Show this!) For $a = 3$, $x_0 = 2$ we get

k	x_k	$x_k^2 - 3$	$\epsilon_k = x_k - \sqrt{3}$	$\lvert\epsilon_k\rvert/\epsilon_{k-1}^2$
0	2	1	2.6795e-01	
1	1.75000000000000	6.2500e-02	1.7949e-02	0.2500
2	1.73214285714286	3.1888e-04	9.2050e-05	0.2857
3	1.73205081001473	8.4727e-09	2.4459e-09	0.2887
4	1.73205080756888	-4.4409e-16	0	

Both the column of function values and the column of errors are typical for an iteration with quadratic convergence: the exponents of the numbers are roughly doubled in each step. The values in the last column converge to the asymptotic error constant

$$C = \frac{2}{2 \cdot 2x^*} = \frac{1}{2\sqrt{3}} \simeq 0.2887 \ .$$

The computation was made in MATLAB with unit roundoff $\mu \simeq$ `1.01e-16`, and x_4 is identical with the floating point representation of $\sqrt{3}$.

We return to the computation of square roots in Section 4.7. ∎

As a *rule of thumb* (not to be confused with a general mathematical truth) we say that when Newton-Raphson's method is used, *the number of correct decimals is doubled in every iteration step.*

The following can be shown:

> Newton-Raphson's method always converges to a simple root if the initial approximation x_0 is chosen close enough to the root x^*.

If an interval $\mathcal{I}$ containing the root x^* is known, it may be quite easy to verify that the condition for convergence of Newton-Raphson's method is satisfied,

$$|\varphi'(x)| = \left| \frac{f(x)f''(x)}{\big(f'(x)\big)^2} \right| \leq m < 1 \quad \text{for } x \in \mathcal{I} \ .$$

Normally, however, one does not bother to check convergence beforehand, since a divergent iteration will show up very quickly. There are important special cases, where it is quite easy to prove that Newton-Raphson's method converges, see Section 4.7.

Neither the bisection method nor the secant method is a fixed point iteration, and we cannot use the above results directly. For the bisection method, however, we know that the upper bound on the error is halved in each step, so the bisection method is equivalent to an iteration method with linear convergence and asymptotic error constant $C = 0.5$.

A closer analysis shows that the sequence of approximations generated by the secant method can be expressed as in Definition 4.4.2 with

$$p = \frac{1+\sqrt{5}}{2} \simeq 1.618, \qquad C = \big(\frac{f''(x^*)}{2f'(x^*)}\big)^{1/p} \ .$$

Thus, the convergence is better than linear, but not as fast as quadratic.

The general theory for iteration methods can be found in the references given at the end of the chapter.

4.5. Error Estimation and Stopping Criteria

When approximations of a root x^* are computed by an iteration method, there will be rounding errors, but Theorem 4.4.1 shows that as long as these errors do not bring us outside the interval $\mathcal{I}$, they do not affect the final accuracy. In this respect iteration methods are *self-correcting*.

The order of convergence of an iteration method tells about the asymptotic behaviour, ie the behaviour when x_k is close to the root, and this may take a large number of steps. In practice one has to stop after a finite number of steps, and a relevant question is: Given an approximation $\overline{x}$ to the simple root x^*, how far is $\overline{x}$ from x^*? Again, we use the mean value theorem:

$$f(\overline{x}) = f(\overline{x}) - f(x^*) = f'(\xi)(\overline{x} - x^*) \ ,$$

where ξ lies between $\overline{x}$ and x^*. Since x^* is a simple root, we have $f'(x^*) \neq 0$, and if $\overline{x}$ is close to x^* (and f' is continuous), then also $f'(\xi) \neq 0$, and

$$|\overline{x} - x^*| = \frac{|f(\overline{x})|}{|f'(\xi)|} \ .$$

Assuming that $|f'(x)| \geq M > 0$ in a neighbourhood around x^* that includes $\overline{x}$, we can make the following estimate

$$|\overline{x} - x^*| \leq \frac{|f(\overline{x})|}{M} \ .$$

In practice we compute an approximation $\widetilde{f}(\overline{x})$ to $f(\overline{x})$. If the absolute error in the approximation is bounded by δ, ie $|\widetilde{f}(\overline{x}) - f(\overline{x})| \leq \delta$, then

$$|f(\overline{x})| \leq |\widetilde{f}(\overline{x})| + \delta \ ,$$

and we can use that in the error estimate. The estimate is independent of the method used to get $\overline{x}$, and is therefore called the method-independent error estimate. We summarize:

Theorem 4.5.1. Method-independent error estimate.
Let $\overline{x}$ be an approximation to a simple root x^* and $\widetilde{f}(\overline{x})$ be an approximation to $f(\overline{x})$. Then

$$|\overline{x} - x^*| \leq \frac{|\widetilde{f}(\overline{x})| + \delta}{M} \ ,$$

where $|\widetilde{f}(\overline{x}) - f(\overline{x})| \leq \delta$ and $|f'(x)| \geq M > 0$ for all x in a neighbourhood of x^* that includes $\overline{x}$.

Example. We have solved the equation $f(x) = x - \cos x = 0$ with a number of methods in the previous sections, and $\overline{x} = 0.73908513$ appears to be correct to 8 decimals. We want to verify this.

Computed in MATLAB with unit roundoff $\mu = 1.11{\cdot}10^{-16}$, we get $\widetilde{f}(\overline{x}) = -5.3809{\cdot}10^{-9}$, and $\delta \simeq 10^{-16}$, which is negligible compared to $|\widetilde{f}(\overline{x})|$. Further, $f'(x) = 1 + \sin x$, $f'(\overline{x}) \simeq 1.6736$, and $M = 1.6$ should be a safe lower bound in a reasonable region around $\overline{x}$. With these values the above error estimate gives

$$|\overline{x} - x^*| \leq \frac{5.39{\cdot}10^{-9}}{1.6} \leq 3.4{\cdot}10^{-9} < 0.5{\cdot}10^{-8} \ .$$

Thus, the value $x^* = 0.73908513$ is correct to 8 decimals. ∎

The accuracy that can be obtained depends on the accuracy with which we can compute the function values. We assume that the computed approximation $\widetilde{f}(x)$ of the value $f(x)$ can be written

$$\widetilde{f}(x) = f(x) + \eta(x) \ , \tag{4.5.1}$$

where $|\eta(x)| \leq \delta$ for x close to the root. Since $\eta(x)$ has contributions from rounding errors, it is not even continuous in general. Figure 4.5 shows a function evaluated in IEEE double precision close to a root. The evaluation was done by means of Horner's rule (see Section 4.6), and it is seen that with this precision and this algorithm we cannot be sure to find the root with an error smaller than about 0.008.

From the method-independent error estimate we see that the smallest error bound is achieved if $\overline{x}$ is determined such that $\widetilde{f}(\overline{x}) = 0$. Then we have

$$|\overline{x} - x^*| \leq \frac{\delta}{M} \ . \tag{4.5.2}$$

The ratio $\epsilon = \dfrac{\delta}{M}$ is called the *attainable accuracy*.

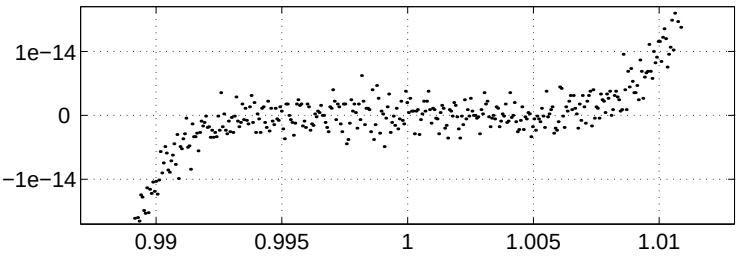


Figure 4.5. *The polynomial* $p(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$ *evaluated in IEEE double precision.*

The attainable accuracy depends both on δ and on the value of the derivative $f'(x)$ in the neighbourhood of the root. This is illustrated in Figure 4.6. If the function values are computed with the same accuracy at both roots, the root α_2 can be computed with higher accuracy than α_1 because the derivative and therefore M has a larger value at α_2 than at α_1. If M is big, the root is said to be *well-conditioned*, and if M is small, the root is *ill-conditioned*.

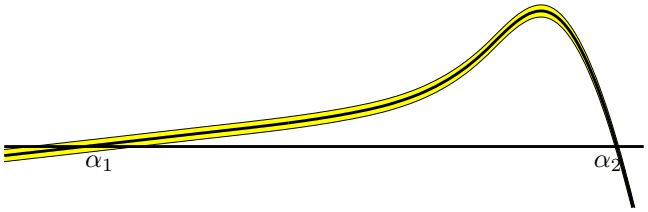


Figure 4.6. *The root* α_1 *is ill-conditioned,* α_2 *is well-conditioned.*

The above argument shows that multiple roots are ill-conditioned, since the derivative is zero at a multiple root. A Taylor expansion around a double root gives

$$f(x) = f(x^*) + f'(x^*)(x - x^*) + \tfrac{1}{2}f''(\eta)(x - x^*)^2 = \tfrac{1}{2}f''(\eta)(x - x^*)^2 \ .$$

If the maximal absolute error of the function value is δ, we get the attainable error for a double root,

$$|\overline{x} - x^*| \leq \sqrt{\frac{2\delta}{M_2}} \ , \tag{4.5.3}$$

where $|f''(x)| \geq M_2 > 0$ for all x in a neighbourhood of x^* that includes $\overline{x}$.

Example. The function $\eta(x)$ in (4.5.1) may include a perturbation of the function f. As an example consider the polynomial

$$\begin{aligned} p(x) &= (x-1)(x-2)\cdots(x-12) \\ &= x^{12} - 78x^{11} + \cdots - 1486442880x + 12! \ . \end{aligned}$$

If the coefficient of x^{11} is changed to $-78 + 2^{-16} \simeq -77.99998474121094$, then some of the roots change drastically. Rounded to 5 decimals the roots of the perturbed polynomial are

1.00000	3.99974	$7.32564 \pm 0.38569i$
2.00000	5.00627	$9.47493 \pm 0.90536i$
3.00000	5.94262	$11.72511 \pm 0.37134i$

The roots $6, 7, \ldots, 12$ are ill-conditioned. ∎

Based on the above discussion we can formulate criteria for stopping an iteration method. Ideally, we want to stop a converging process when we are sufficiently close to the root, ie when $|x_k - x^*| \leq \epsilon$, where ϵ is the desired accuracy (which must not be chosen smaller than the attainable accuracy). However, we do not know x^* but have to use computable quantities. Good choices of *stopping criteria* are

$$|x_{k+1} - x_k| \leq \tau_1 \ , \tag{4.5.4}$$

$$|f(x_k)| \leq \tau_2 \ , \tag{4.5.5}$$

$$k = k_{\max} \ , \tag{4.5.6}$$

where τ_1 and τ_2 are small and $k_{\max}$ is large.

In connection with Newton-Raphson's method we see that if we can neglect rounding errors and use $|f'(x)|$ as an approximation to M, then the absolute value of the step

$$x_{k+1} - x_k = h_k = -f(x_k)/f'(x_k)$$

is equivalent to the right-hand side in the method-independent error estimate. Therefore, if $|h_k| \leq \tau_1$, then $|x_k - x^*| \lesssim \tau_1$.

For a fixed point method with linear convergence a very small value for $|x_{k+1} - x_k|$ does not guarantee that x_k is very close to x^*:

$$\begin{aligned} x_{k+1} - x_k &= \varphi(x_k) - \varphi(x^*) + \varphi(x^*) - x_k \\ &= \varphi'(\xi_k)(x_k - x^*) - (x_k - x^*) \ , \end{aligned}$$

so that

$$x_k - x^* = \frac{x_k - x_{k+1}}{1 - \varphi'(\xi_k)} \ .$$

For a convergent iteration we know that $|\varphi'(\xi_k)| \simeq |\varphi'(x^*)| < 1$, but if $\varphi'(x^*)$ is close to 1, then $|x_k - x^*|$ is much bigger than $|x_{k+1} - x_k|$.

For an ill-conditioned root like α_1 in Figure 4.6 the criterion (4.5.5) is likely to stop the iteration. If high accuracy is required, one should use $\tau_2 = 2\delta$.

Finally, the criterion (4.5.6) should be used together with (4.5.4) and/or (4.5.5) in every implementation of a fixed point iteration method. It is a "safe guard" to prevent an infinite loop, that might be caused by the iteration method being divergent; by errors in the implementation of f or f' or by the choice of τ_1 and τ_2 so small that rounding errors prevent (4.5.4) and (4.5.5) from ever being satisfied.

Example. Below we give a MATLAB implementation of Newton-Raphson's method with the stopping criteria (4.5.4) and (4.5.6).

```
function [x,k] = newton(fdf,x0,tol,kmax)
% Solve f(x)=0 by Newton-Raphson's method.
% f(x) and f'(x) given by  [f,df] = fdf(x)
% Starting point x0.
% Iterate until correction is smaller than tol
% or the number of steps exceeds kmax
k = 0;   x = x0;  % initialize
[f, df] = feval(fdf, x);   h = f/df;
while  (abs(h) > tol) & (k < kmax)
  k = k+1;   x = x - h;
  [f, df] = feval(fdf, x);   h = f/df;
end
```

With `fdf` defined on page 70 we get

```
>> [x, k] = newton(@fdf, 0.7, 1e-10, 100)
x = 0.73908513321516
k = 3
```

The result has an error less than $0.3 \cdot 10^{-15}$. ∎

Example. The function $f(x) = \dfrac{1}{x} - 1$ has a root $x^* = 1$ in the interval $[0.5, 10]$. Below we give results from the first three steps with Newton-Raphson's method (with $x_0 = 10$) and the secant method (with $x_0 = 0.5$, $x_1 = 10$):

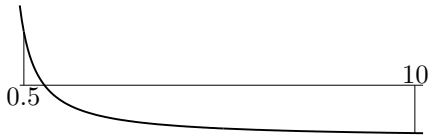


Figure 4.7. $f(x) = 1/x - 1$.

	N-R method	secant method
k	x_k	x_k
0	10	0.500
1	−80	10.000
2	−6560	5.500
3	−43046720	−39.500
4		183.250

Both iterations seem to diverge; also see Exercise E2 at the end of this chapter.

MATLAB has a function `fzero` that uses a hybrid method to find a root. The algorithm combines the robustness of the bisection method with the speed of more accurate methods, among which the secant method is found. The algorithm is described in the book by Brent, and we shall give a slightly simplified description of it.

The algorithm works with three sequences of approximations to the root, $\{a_k\}$, $\{b_k\}$ and $\{c_k\}$, such that b_k and c_k form a bracket,

$$f(b_k) \cdot f(c_k) < 0 \ .$$

The iteration stops when $|b_k - c_k| \leq \tau$ or if a zero function value is encountered.

The interval end points are ordered such that

$$|f(b_k)| \leq |f(c_k)| \ .$$

Either b_k or c_k is the latest approximation, and a_k is the previous approximation. There are two candidates for the next approximation, d_k.

$$\text{midpoint step}: \quad m_k = b_k + \tfrac{1}{2}(c_k - b_k) \ ,$$

$$\text{secant step}: \quad s_k = b_k - f(b_k)\frac{b_k - a_k}{f(b_k) - f(a_k)} \ .$$

d_k must be in the bracket and should be closest to the end point with the smaller function value. If this is not satisfied by s_k, then $d_k = m_k$, otherwise

$$\begin{array}{lll} \textbf{if} \ |s_k - b_k| < \tau & \textbf{then} & d_k = b_k + \text{sign}(c_k - b_k) \cdot \tau \\ & \textbf{else} & d_k = s_k \end{array}$$

The special step taken when the secant method predicts a very small step is introduced because when we have one-sided convergence — as eg in Figure 4.4 on page 72 — then $|c_k - x^*|$ may be large even if $|b_k - x^*|$ is very small, and we hope that the special step brings us to the other side of the root, in which case $|b_{k+1} - c_{k+1}| = \tau$, and the iteration stops.

Now, the function is evaluated at d_k. If $\widetilde{f}(d_k) = 0$, we have found a root and iteration stops. Otherwise, the bracket is updated.

```
function  y = recip(x)
y = 1/x - 1;
```

```
>> opts = optimset('Display','iter', 'TolX',1e-10);
>> x = fzero(@recip, [0.5,10], opts)
```

The tolerance is set to $\tau = 10^{-10}$ and we get the following trace of the iteration and the computed root

```
 Func-count      x          f(x)         Procedure
    1            0.5            1        initial
    2             10         -0.9        initial
    3            5.5    -0.818182        interpolation
    4              3    -0.666667        bisection
    5           1.75    -0.428571        bisection
    6          1.125    -0.111111        bisection
    7       0.953125     0.0491803        interpolation
    8        1.00586   -0.00582524        interpolation
    9        1.00027  -0.000274583        interpolation
   10              1  7.54371e-008        interpolation
   11              1 -2.07194e-011        interpolation
   12              1  1.79281e-010        interpolation
Zero found in the interval: [0.5, 10].

x = 1.00000000002072
```

■

4.6. Algebraic Equations

An algebraic equation is an equation $p(x) = 0$, where $p(x)$ is a polynomial

$$p(x) = a_1 x^n + a_2 x^{n-1} + \cdots + a_n x + a_{n+1} \ .$$

The *fundamental theorem of algebra* states that an nth degree polynomial has exactly n roots (multiple roots are counted with their multiplicity), and if the coefficients $a_1, \ldots, a_{n+1}$ are real, then the complex roots are pairwise conjugate.

Newton-Raphson's method can of course be used also for the solution of algebraic equations. In every iteration step we have to compute the value of the polynomial and its derivative. It is inefficient to compute each term separately and then add them up. If the value of x^i is used to compute x^{i+1}, then this method requires $2n-1$ multiplications and n additions. Instead we can write the polynomial in the form (illustrated by $n = 5$)

$$p(x) = ((((a_1x + a_2)x + a_3)x + a_4)x + a_5)x + a_6 \ ,$$

and $p(x)$ can be computed recursively. In the general case we compute $p(x_0) = b_{n+1}$ by the recurrence

$$\begin{aligned} b_1 &= a_1 \\ b_i &= b_{i-1}x_0 + a_i \ , \quad i = 2, 3, \ldots, n+1 \ . \end{aligned} \tag{4.6.1}$$

This method is called *Horner's rule*. It involves n multiplications and n additions, so the work is reduced from the $3n$ flops mentioned above to $2n$ flops. Also, the effect of rounding errors is normally smaller with Horner's rule.

Example. The following MATLAB function implements Horner's rule for evaluating a polynomial.

```
function  p = horner(a, x)
% Horner's rule to compute
% p = a_1*x^n + ... + a_n*x + a_(n+1)
p = a(1);
for  i = 2 : length(a)
  p = p*x + a(i);
end
```

For the polynomial $p(x) = 3x^4 - 2x^2 + x + 1$ you should verify the following value of $p(2)$:

```
>> p = horner([3 0 -2 1 1],2)
p = 43
```

The standard MATLAB function `polyval` has the same inputs and also uses Horner's rule. ∎

We derived Horner's rule by putting parentheses in the expression. There is an alternative derivation, which is useful when we want to compute both $p(x_0)$ and the derivative $p'(x_0)$. We can write

$$\begin{aligned} p(x) &= (x - x_0)q(x) + r \\ &= (x - x_0)(b_1x^{n-1} + b_2x^{n-2} + \cdots + b_n) + r \ . \end{aligned}$$

By differentiation we get

$$p'(x) = q(x) + (x - x_0)q'(x) \ ,$$

so that

$$p(x_0) = r, \qquad p'(x_0) = q(x_0) \ .$$

To get the values of the coefficients in $q(x)$ we perform the multiplication

$(x-x_0)q(x)$ and identify coefficients in $p(x)$ and $(x-x_0)q(x)+r$. This gives

$$\begin{aligned} a_1 &= b_1 \\ a_2 &= b_2 - b_1 x_0 \\ &\vdots \\ a_i &= b_i - b_{i-1} x_0 \\ &\vdots \\ a_{n+1} &= r - b_n x_0 \end{aligned}$$

If we reorder and introduce $b_{n+1} = r$, we get (4.6.1). The value of the derivative $p'(x_0)$ can be found by applying Horner's rule to the polynomial $q(x)$ of degree $n-1$.

Example. The function `horner1` evaluates both $p(x)$ and $p'(x)$ without storing the coefficients $\{b_i\}$.

```
function  [p, dp] = horner1(a, x)
% Horner's rule used to compute p and p'
p = a(1);  dp = 0;
for  i = 2 : length(a)
  dp = dp*x + p;
  p = p*x + a(i);
end
```

As in `horner` the variable `p` is successively overwritten by b_i. Similarly, `dp` is successively overwritten by $c_i = c_{i-1}x + b_i$, cf (4.6.1) applied to $q(x)$.

Consider the polynomial $p(x) = x^3 - 1.1x^2 + 2x - 2$. The command

```
>> [p, dp] = horner1([1 -1.1 2 -2],1)
```

returns `p = -0.1`, `dp = 2.8`. (Use explicit differentiation and verify that $p'(1) = 2.8$). With the MATLAB function

```
function  [p, dp] = pdp(x)
a = [1 -1.1 2 -2];
[p dp] = horner1(a,x);
```

we can use `newton` from page 82 to find a root of p. With the starting point $x_0 = 1$ we get

```
>> [x,k] = newton(@pdp,1,1e-8,100)
x =  1.03487386861150
k =  3
```

■

Horner's rule is sometimes referred to as *synthetic division*. From the relation

$$p(x) = (x - x_0)q(x) + p(x_0) \ ,$$

we see that if x_0 is a root of the polynomial, $p(x_0) = 0$, then $q(x)$ is the quotient polynomial, $q(x) = p(x)/(x - x_0)$. Thus, when a root x_0 of the polynomial has been found, eg by Newton-Raphson's method, we can divide by $x - x_0$ using Horner's rule, and then continue by determining the roots of the quotient polynomial $q(x)$ (see Computer Exercise C2). This process is called *deflation*. A careful rounding error analysis shows that in order to obtain good accuracy for all roots, they should be removed in order of increasing magnitude.

Example. The MATLAB command `r = roots(a)` returns all zeros of the polynomial given by the coefficient vector a. Assuming that $a_1 \neq 0$, this polynomial has the same zeros as

$$\widetilde{p}(x) = x^n + c_2 x^{n-1} + \cdots + c_n x + c_{n+1}, \quad c_i = a_i/a_1 \ ,$$

(with $n =$ `length(a)` $- 1$). This is the characteristic polynomial of a so-called *companion matrix*. In the case $n = 5$ this matrix is given by

$$C = \begin{pmatrix} -c_2 & -c_3 & -c_4 & -c_5 & -c_6 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \ .$$

The roots of $\widetilde{p}(x)$ (and therefore the roots of $p(x)$) are computed as the eigenvalues of C, using a very efficient and accurate program from LAPACK. ∎

4.7. Computer Implementation of Square Root

In this section we shall describe one possible way of implementing the square root function $\sqrt{a}$ in a computer with binary arithmetic. The purpose is to show how theory can be used to derive a practical algorithm for an important application.

The standard for floating point arithmetic, described in Chapter 2, prescribes that the square root function is implemented together with the arithmetic operations. It may happen, however, when a microprocessor is embedded in a larger system (eg for process control), that the whole standard is not provided. In such a case a systems programmer may have to supply the standard functions needed.

We assume that the computer has binary arithmetic and that we want to compute the square root of normalized binary numbers with $t+1$ digits in the significand,

$$A = 1.b_1 b_2 \dots b_t \cdot 2^e \ .$$

If the exponent e is odd, we shift the significand one step to the left, so that the number is of the form

$$A = a \cdot 2^{2k}, \quad a = c_1 c_0 . d_1 d_2 \dots d_t \ .$$

Then

$$\sqrt{A} = \sqrt{a} \cdot 2^k \ .$$

The exponent of the result is obtained by shifting one step to the right, and we see that we need a method to compute the square root of a binary number a that satisfies

$$1 \le a < 4 \ , \qquad 1 \le \sqrt{a} < 2 \ .$$

A commonly used approach is to apply Newton-Raphson's method to the equation $f(x) = x^2 - a = 0$. This gives the iteration

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right) \ . \tag{4.7.1}$$

We first show a theorem on monotone convergence.

Theorem 4.7.1. For any x_0, $0 < x_0 < \infty$, the iteration (4.7.1) generates a decreasing sequence

$$x_1 \ge x_2 \ge \cdots \ge \sqrt{a} \ ,$$

that converges to $\sqrt{a}$.

Proof. Let $\epsilon_k = x_k - \sqrt{a}$. Then

$$x_{k+1} = \frac{1}{2}\left(\sqrt{a} + \epsilon_k + \frac{a}{\sqrt{a} + \epsilon_k}\right) = \frac{2a + 2\sqrt{a}\epsilon_k + \epsilon_k^2}{2(\sqrt{a} + \epsilon_k)} = \sqrt{a} + \epsilon_{k+1} \ ,$$

with

$$\epsilon_{k+1} = \frac{\epsilon_k^2}{2(\sqrt{a} + \epsilon_k)} = \frac{\epsilon_k^2}{2x_k} \ .$$

For $k = 0$ we see that $\epsilon_1 = \epsilon_0^2/(2x_0) \ge 0$, and by induction: all $\epsilon_k \ge 0$, which is equivalent to $x_k \ge \sqrt{a}$, $k = 1, 2, \dots$. Next,

$$x_k - x_{k+1} = \frac{1}{2}\left(x_k - \frac{a}{x_k}\right) = \frac{1}{2x_k}(x_k^2 - a) \ge 0 \ ,$$

so the sequence is decreasing. The iteration function $\varphi(x) = \frac{1}{2}(x + a/x)$ satisfies

$$0 \le \varphi'(x) = \frac{1}{2}\left(1 - \frac{a}{x^2}\right) \le \tfrac{1}{2}$$

for $x \ge \sqrt{a}$, and Theorem 4.4.1 tells us that the iteration converges to a unique root x^*, that can be found by the fixed point condition

$$x^* = \frac{1}{2}\left(x^* + \frac{a}{x^*}\right) \ .$$

This equation is equivalent to $(x^*)^2 = a$, so that $x^* = \sqrt{a}$. □

The rate of convergence can be estimated by the relation found in the proof,

$$x_{k+1} - \sqrt{a} = \frac{(x_k - \sqrt{a})^2}{2x_k} \ .$$

Since $a \ge 1$ we naturally choose $x_0 \ge 1$, so we have $x_k \ge 1$, $k = 0, 1, \ldots$, and

$$x_{k+1} - \sqrt{a} \le \tfrac{1}{2}(x_k - \sqrt{a})^2 \ .$$

This confirms that the application of Newton-Raphson's method gives quadratic convergence, and implies that

$$\begin{aligned}
x_1 - \sqrt{a} &\le \frac{1}{2}(x_0 - \sqrt{a})^2 \ , \\
x_2 - \sqrt{a} &\le \frac{1}{2}(x_1 - \sqrt{a})^2 = \frac{1}{2^3}(x_0 - \sqrt{a})^4 \ , \\
x_3 - \sqrt{a} &\le \frac{1}{2^7}(x_0 - \sqrt{a})^8 \ , \\
&\vdots
\end{aligned}$$

Thus, we get very fast convergence if we choose a good approximation x_0. Now, for a given a in $[1, 4]$, how can one choose a good initial approximation $x_0(a)$ that is quickly obtainable?

Since computer memory is quite cheap, we can make a table of initial approximation as follows: The first four bits in a is one of the following 12 combinations

01.00	01.01	01.10	01.11
10.00	10.01	10.10	10.11
11.00	11.01	11.10	11.11

In a table with 12 entries we store the square roots of $c_1c_0.d_1d_21$, where

$c_1c_0.d_1d_2$ is one of the above combinations. For a given a we use the first four bits to get the address and use the table value for x_0. Eg if $a = (10.010\ldots)_2$ or $a = (10.011\ldots)_2$, we get $x_0 = \sqrt{(10.011)_2}$. The largest initial error $|x_0 - \sqrt{a}|$ can be estimated by means of the mean value theorem

$$|\sqrt{a+h} - \sqrt{a}| = \frac{|h|}{2\sqrt{\xi}} \leq \frac{|h|}{2} \ ,$$

since $a \geq 1$. The maximal value of h is $(0.001)_2 = 2^{-3}$, so that

$$\begin{aligned} |x_0 - \sqrt{a}| &\leq 2^{-4} \simeq 6.3 \cdot 10^{-2} \ , \\ |x_1 - \sqrt{a}| &\leq 2^{-9} \simeq 2.0 \cdot 10^{-3} \ , \\ |x_2 - \sqrt{a}| &\leq 2^{-19} \simeq 1.9 \cdot 10^{-6} \ , \\ |x_3 - \sqrt{a}| &\leq 2^{-39} \simeq 1.8 \cdot 10^{-12} \ . \end{aligned}$$

We have $t+1$ bits in the significand, and therefore we want to compute the square root with an error less than 2^{-t-1}. With $t = 23$ (as in the floating point standard) and with the table size discussed, three iteration step suffice to give the desired accuracy, 2^{-24}. If we use 6 bits as address in the table, we need a table with 48 entries, and the maximal error is

$$\begin{aligned} |x_0 - \sqrt{a}| &\leq 2^{-6} \simeq 1.6 \cdot 10^{-2} \ , \\ |x_1 - \sqrt{a}| &\leq 2^{-13} \simeq 1.2 \cdot 10^{-4} \ , \\ |x_2 - \sqrt{a}| &\leq 2^{-27} \simeq 7.5 \cdot 10^{-9} \ . \end{aligned}$$

In this case the required accuracy is obtained after two iteration steps.

4.8. Systems of Nonlinear Equations

In this section we give a short introduction to the problem of solving a system of n nonlinear equations in n unknowns,

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \ , \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \ , \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \ . \end{aligned}$$

Example. Throughout this section we use the example (with $n = 2$)

$$f_1(x_1, x_2) = \ 4x_1^2 + 9x_2^2 - 36 \ = 0 \ ,$$
$$f_2(x_1, x_2) = 16x_1^2 - 9x_2^2 - 36 = 0 \ .$$

Figure 4.8 shows the curves along which $f_1(x_1, x_2) = 0$ (an ellipse) and $f_2(x_1, x_2) = 0$ (a hyperbola). A solution to the system is an intersection

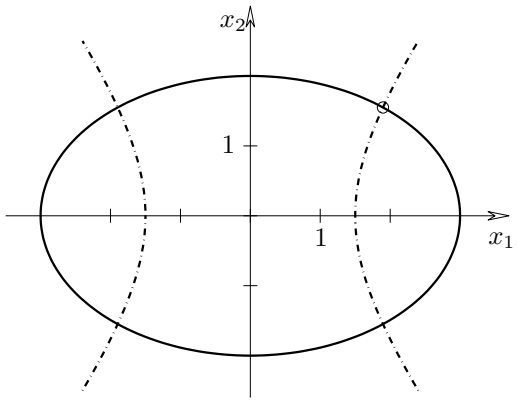


Figure 4.8. *Nonlinear system of equations.*

between the two curves. It is easy to solve this system analytically, and the solution in the first quadrant (marked by a circle in the figure) is

$$x_1 = \sqrt{3.6} \simeq 1.897367, \qquad x_2 = \sqrt{2.4} \simeq 1.549193 \ .$$

■

In the presentation we shall use concepts from linear algebra that are defined in Chapter 8. We recommend to read that chapter before this section.

With the notation

$$x = (x_1, x_2, \ldots, x_n)^T, \qquad f(x) = \big(f_1(x), f_2(x), \ldots, f_n(x)\big)^T \ ,$$

we can write the system in the usual form,

$$f(x) = 0 \ ,$$

where now f is a vector valued function and 0 is the zero vector in $\mathbb{R}^n$. Several of the methods described earlier in this chapter for a scalar equation can be generalized to systems of nonlinear equations.

First, consider Newton-Raphson's method, with an approximation $x^{[k]} = (x_1^{[k]}, \ldots, x_n^{[k]})^T$ to the root x^*. In Section 4.3 we derived the method by approximating the scalar f by the first two terms in the Tay-

lor expansion. Similarly, we can expand each component of f,

$$f_i(x^{[k]} + h) \simeq f_i(x^{[k]}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j}(x^{[k]})h_j \ , \quad i = 1, \ldots, n \ .$$

We let $J(x)$ denote the $n \times n$ matrix of partial derivatives, $(J(x))_{ij} = \frac{\partial f_i}{\partial x_j}(x)$, and get

$$f(x^{[k]} + h) \simeq f(x^{[k]}) + J(x^{[k]})h \ .$$

J is called the *Jacobian* of f. We want $f(x^{[k]} + h) = 0$, and in the same way as before we get an approximation to $h = x^* - x^{[k]}$ by solving the *linear system of equations*

$$J(x^{[k]})h^{[k]} = -f(x^{[k]}) \ , \tag{4.8.1}$$

and the next approximation is

$$x^{[k+1]} = x^{[k]} + h^{[k]} \ .$$

We can write the vector version of Newton-Raphson's method in the form

$$x^{[k+1]} = x^{[k]} - \left(J(x^{[k]})\right)^{-1} f(x^{[k]}), \quad k = 0, 1, 2, \ldots \ .$$

This shows a clear analogy with the one-dimensional version, Definition 4.3.1. It should be emphasized that the inverse of J is not computed explicitly, but the system in (4.8.1) is solved via an LU factorization of J, see Chapter 8.

The analysis of Newton-Raphson's method for systems of nonlinear equations is more complicated than for the one-dimensional version, but it can be shown that if f is twice continuously differentiable in a neighbourhood of x^*; if $J(x^*)$ is nonsingular; and if $x^{[0]}$ is sufficiently close to x^*, then Newton-Raphson's method converges, and the convergence is quadratic. This means that there exists a constant C such that

$$\|x^{[k+1]} - x^*\|_2 \simeq C\|x^{[k]} - x^*\|_2^2$$

for $\|x^{[k]} - x^*\|_2$ sufficiently small. Here, $\|\cdot\|_2$ is the so-called *Euclidean norm*, defined by

$$\|d\|_2 = \sqrt{d_1^2 + d_2^2 + \cdots + d_n^2} \ .$$

Example. For the function of of the previous example,

$$f_1(x_1, x_2) = 4x_1^2 + 9x_2^2 - 36 \ ,$$
$$f_2(x_1, x_2) = 16x_1^2 - 9x_2^2 - 36 \ ,$$

the Jacobian is

$$J(x) = \begin{pmatrix} 8x_1 & 18x_2 \\ 32x_1 & -18x_2 \end{pmatrix} \ .$$

The following MATLAB function computes both $f(x)$ and $J(x)$,

```
function  [f, J] = f_J(x)
f = [4*x(1)^2 + 9*x(2)^2 - 36
  16*x(1)^2 - 9*x(2)^2 - 36];
J = [8*x(1)  18*x(2)
  32*x(1)  -18*x(2)];
```

It is straightforward to modify the one-dimensional `newton` from Example 4.5.1 to n-dimensional problems. The linear system `J`$h =$ `f` is solved by the command[2] $h =$ `J\f`, and `norm(h)` returns $\|h\|_2$, defined above.

```
function [x,k] = newtonsys(fdf,x0,tol,kmax)
% Solve system f(x)=0 by Newton-Raphson's method.
% f(x) and J(x) given by  [f,J] = fdf(x)
% Starting point x0.
% Iterate until norm of correction is smaller than tol
% or the number of steps exceeds kmax
k = 0;   x = x0;  % initialize
[f, J] = feval(fdf, x);   h = J\f;
while  (norm(h) > tol) & (k < kmax)
  k = k+1;   x = x - h;
  [f, J] = feval(fdf, x);   h = J\f;
end
```

We use the starting point $x^{[0]} = (1, 1)^T$:

```
>> [x k] = newtonsys(@f_J,[1; 1],1e-10,100)
x = 1.89736659610103
    1.54919333848297
k = 5
```

We have printed results during the iteration (see the next page). There is quadratic convergence. This is clearly seen in the column with function values.

---

[2] `\` is pronounced *backslash*.

k	$x_1^{[k]}$	$x_2^{[k]}$	$\|f(x^{[k]})\|_2$	$\|h^{[k]}\|_2$
0	1.00000000000000	1.00000000000000	3.70e+01	1.48e+00
1	2.30000000000000	1.70000000000000	2.52e+01	3.95e-01
2	1.93260869565217	1.55588235294118	2.10e+00	3.56e-02
3	1.89768792487896	1.54920771711331	1.98e-02	3.22e-04
4	1.89736662330576	1.54919333854969	1.70e-06	2.72e-08
5	1.89736659610103	1.54919333848297	7.11e-15	1.07e-16

■

As in the one-dimensional case other fixed point iterations can be constructed by reformulating $f(x) = 0$ to

$$x = \varphi(x) \ ,$$

with $\varphi(x) = (\varphi_1(x), \ldots, \varphi_n(x))^T$. Starting from an initial approximation $x^{[0]}$ a sequence of approximations $x^{[1]}, x^{[2]}, \ldots$ is computed by

$$x^{[k+1]} = \varphi(x^{[k]}) \ , \quad k = 0, 1, 2, \ldots \ .$$

Theorem 4.4.1 about convergence criteria can be generalized: Assume that $x^* = \varphi(x^*)$, and that the partial derivatives

$$d_{ij}(x) = \frac{\partial \varphi_i}{\partial x_j}(x) \ , \quad i, j = 1, \ldots, n$$

exist for $x \in \mathcal{I} = \{x \big| \ \|x - x^*\| \leq \delta\}$. Let $D(x)$ be the $n \times n$ matrix with elements $d_{ij}(x)$. A sufficient condition for the fixed point iteration to converge for any $x^{[0]} \in \mathcal{I}$ is that

$$\|D(x)\| \leq m < 1 \ , \quad x \in \mathcal{I} \ ,$$

where $\|\cdot\|$ is an induced matrix norm, defined in Section 8.10. If this condition is satisfied, we have linear convergence,

$$\|x^{[k+1]} - x^*\| \leq m \|x^{[k]} - x^*\| \ .$$

There is no simple generalization to $n > 1$ dimensions of the bisection method or the secant method.

Example. For the nonlinear system from the two previous examples,

$$4x_1^2 + 9x_2^2 - 36 = 0 \ ,$$
$$16x_1^2 - 9x_2^2 - 36 = 0 \ ,$$

we seek the root $x^* = (x_1^*, x_2^*) = (\sqrt{3.6}, \sqrt{2.4}) \simeq (1.8974, 1.5492)$. We shall consider two fixed point iterations obtained by solving each equation for one

of the unknowns. First,

$$x_1 = \varphi_1(x) = \tfrac{1}{2}\sqrt{36 - 9x_2^2}\ ,$$
$$x_2 = \varphi_2(x) = \tfrac{1}{3}\sqrt{16x_1^2 - 36}\ .$$

With the starting point $x^{[0]} = (1,1)^T$ we get

k	$x_1^{[k]}$	$x_2^{[k]}$
1	2.5981	2.4037
2	$2.0000i$	4.0000
3	$5.1962i$	$1.7638i$
4	4.0000	$6.6332i$

There is no sign of convergence. The matrix of partial derivatives is

$$D(x) = \begin{pmatrix} 0 & \frac{-18x_2}{4\sqrt{36-9x_2^2}} \\ \frac{32x_1}{6\sqrt{16x_1^2-36}} & 0 \end{pmatrix}\ , \qquad D(x^*) \simeq \begin{pmatrix} 0 & -1.84 \\ 2.18 & 0 \end{pmatrix}\ .$$

$\|D(x^*)\| \simeq 2.18 > 1$, so the sufficient condition for convergence is not satisfied.

Next, consider the fixed point method given by

$$x_1 = \varphi_1(x) = \tfrac{1}{4}\sqrt{36 + 9x_2^2}\ ,$$
$$x_2 = \varphi_2(x) = \tfrac{1}{3}\sqrt{36 - 4x_1^2}\ ,$$

$$D(x) = \begin{pmatrix} 0 & \frac{18x_2}{8\sqrt{36+9x_2^2}} \\ \frac{-8x_1}{6\sqrt{36-4x_1^2}} & 0 \end{pmatrix}\ , \qquad \|D(x^*)\| \simeq 0.544\ .$$

The sufficient condition for convergence is satisfied in some region around x^*, and with $x^{[0]} = (1,1)^T$ we get

k	$x_1^{[k]}$	$x_2^{[k]}$	$\|x^{[k]} - x^*\|_2$
1	1.6771	1.8856	0.4021
2	2.0616	1.6583	0.1971
3	1.9486	1.4530	0.1090
4	1.8540	1.5207	0.0519
$\vdots$			

The error is roughly halved in every iteration step. ∎

Exercises

E1. Illustrate a divergent fixed point iteration, cf. Figure 4.3.

E2. On some computers division is performed via solution of the equation $f(x) = 1/x - a$ by means of Newton-Raphson's method. Analyze this algorithm on the lines of the square root algorithm of Section 4.7, with special focus on the relation between the size of the table and the number of iterations needed.

E3. Find the positive root of the equation

$$f(x) = \int_0^x \cos t \, dt - 0.5 = 0$$

with 5 correct decimals. Hint: Solve the equation $p(x) = 0$, where $p(x)$ is a polynomial obtained by integrating a partial sum of the Maclaurin expansion of $\cos t$. Use the method-independent error estimate for the original equation to find the necessary degree of the approximating polynomial.

E4. Show that the polynomial $p(x) = x^3 - 3x - 2$ has the roots $-1, -1$ and 2, and estimate the roots of the perturbed polynomial $\overline{p}(x) = x^3 - 3x - 2 + \delta$, where $|\delta|$ is small.

Computer Exercises

C1. The fixed point iteration $x_{k+1} = \varphi(x_k)$ does not converge if $|\varphi'(x)| > 1$ in an interval around the desired root x^*. This does not necessarily imply that the iteration diverges towards infinity. Try the iteration $x_{k+1} = 1 - \lambda x_k^2$ for $\lambda = 0.7,\ 0.9,\ 2$ and illustrate graphically how x_k depends on k. For $\lambda = 2$ the iteration exhibits a chaotic behaviour.

C2. Modify the Matlab function `horner` from `incbox`, cf page 85, so that the first line is

```
function b = qhorner(a, x)
```

`b` should hold the b_i defined by (4.6.1). Use `qhorner` to find the quotient polynomial $p(x)/(x-1)$, where $p(x) = x^3 - 4x^2 + 7x - 4$, and compute all the roots of $p(x)$.

C3. Write a program that implements Newton-Raphson's method for computing $1/a$ as described in Exercise E2 above. How large should the table be in order to ensure full accuracy after three iteration steps? First write a program that generates the table, and then an efficient code (without loops or function calls) for the Newton iterations.

C4. Use `roots` to find the zeros of

$$\overline{p}(x) = x^3 - 3x - 2 + \delta \ ,$$

for $\delta = \pm 10^{-3}$ and $\delta = \pm 10^{-1}$, and compare with the results of Exercise E4.

C5. We shall determine the smallest positive root x^* of

$$f(x) = \cot 3x - \frac{x^2 - 1}{2x} \ .$$

(a) Plot the function for $0 < x \leq 4$ and use the graph to determine a bracket for x^*. The bracket should have length 0.2.

(b) Starting with this bracket, how many bisection steps are needed to determine an approximation $\overline{x}$ to x^*, such that $|\overline{x} - x^*| \leq 0.5 \cdot 10^{-6}$?

(c) Use Newton-Raphson's method with $x_0 = 0.5$ to compute x_1, x_2, x_3 and x_4.

(d) The differences $d_k = x_k - x_{k-1}$ can be considered as estimates of the errors $x^* - x_{k-1}$. Use these estimates to make it probable that the sequence of iterates has quadratic convergence.

C6. Use Newton-Raphson's method to find two of the solutions of the nonlinear system of equations

$$\begin{aligned} f_1(x) &= x_1^2 + x_1 x_2^3 - 9 \ = 0 \ , \\ f_2(x) &= 3x_1^2 x_2 - x_2^3 - 4 = 0 \ , \end{aligned}$$

and discuss the convergence.

(The exercise is inspired by an example in Shampine et al, *Fundamentals of numerical computing,* John Wiley and Sons, New York, 1999. Also see http://www.imm.dtu.dk/∼hbn/demos/NonlinSystem.pdf)

References

Isaac Newton (1642 – 1727) was really the first to use what today is known as Newton-Raphson's method. He used the method to solve $x - a \sin x = M$ for given a and M, in order to determine the position of a planet in an elliptical orbit. Joseph Raphson (1648 – 1715) independently discovered the method for the case where $f(x)$ is a polynomial. The history of the method is described in

T.J. Ypma, *Historical Development of the Newton-Raphson Method*, SIAM Review **37** (1995), 531–551.

The classical theory of solution of nonlinear equations can be found in the following books,

A.S. Householder, *The Numerical Treatment of a Single Nonlinear Equation*, McGraw-Hill, New York, 1963.

A. Ostrowski, *Solution of Equations and Systems of Equations*, 2nd ed., Academic Press, New York, 1966.

J. Traub, *Iterative Methods for the Solution of Equations*, Prentice-Hall, Englewood Cliffs, New Jersey, 1964.

A detailed description of the method sketched in Example 4.5.1 can be found in

R.P. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

James Hardy Wilkinson (1919 – 1986) studied perturbation theory for the solution of algebraic equations in

J.H. Wilkinson, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

Numerical solution of systems of nonlinear equations are treated in

J.M. Ortega, W.C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York, 1970.

J.E. Dennis, R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Chapters 5–8, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

Chapter 5

Interpolation

5.1. Introduction

For some function f, assume that we know its value $f_i = f(x_i)$ in $n+1$ different points $x_0, x_1, \ldots, x_n$. We want to determine a function P such that

$$P(x_i) = f_i, \quad i = 0, 1, \ldots, n \ .$$

The function P is said to *interpolate* f in the points $x_0, x_1, \ldots, x_n$. As an example, Figure 5.1 shows a polynomial P that interpolates $f(x) = \sin x$ in the points 1.1, 1.2, 1.3. In the interval $1.1 \leq x \leq 1.3$ we cannot see the difference between the function and the interpolating polynomial.

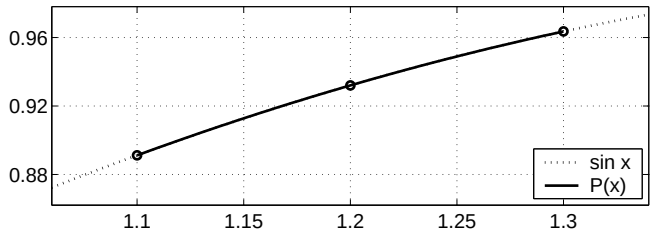


Figure 5.1. *Three points are interpolated by a polynomial.*

An interpolating function P can be used to estimate the value of f at a point x. If x is in the interval formed by $x_0, x_1, \ldots, x_n$, we speak about *interpolation*, otherwise about *extrapolation*.

It is possible to use other interpolating functions, eg rational functions or trigonometric functions. However, since polynomials are easy to evaluate, differentiate and integrate, we shall only discuss polynomials and functions consisting of different polynomials in different subintervals, so-called spline functions.

Often, one can get a good approximation of f by a low degree interpolating polynomial. However, many numerical methods (eg for integration and for the solution of differential equations) are based on interpolating polynomials, and therefore we shall derive formulas for interpolating polynomials of arbitrary degree.

If only approximations of the function values f_i are known (eg from measurements), then it is not advisable to construct an approximation function by means of interpolation. It is better to use an approximation method that can reduce the influence of the measurement errors; see Chapter 9.

5.2. Interpolation by Polynomials

Example. Assume that we know the values of a function f in three points (we use the function $f(x) = \sin x$)

i	x_i	$f_i = f(x_i)$
0	1.1	0.8912
1	1.2	0.9320
2	1.3	0.9636

We shall construct a polynomial that interpolates f in these points. What is the degree of the polynomial?

In general, a straight line cannot pass through three given points, but there is a second degree polynomial and an infinity of different third degree polynomials with this property.

In order to determine a polynomial P of degree ≤ 2, that interpolates the three points in the table, we can write it in the form

$$P(x) = a_0 + a_1 x + a_2 x^2 \ .$$

The computation of the coefficients is easier, however, if we use the alternative formulation

$$P(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) \ .$$

The requirement that $P(x_i) = f_i$, $i = 0, 1, 2$, leads to the following system of equations for determining the coefficients $c_0,\ c_1,\ c_2$,

$$\begin{aligned} P(x_0) &= c_0 &&= f_0 \ , \\ P(x_1) &= c_0 + c_1(x_1 - x_0) &&= f_1 \ , \\ P(x_2) &= c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) &&= f_2 \ . \end{aligned}$$

The coefficient c_0 is given by the first equation. This value is inserted in the second equation, and we get

$$c_1 = (f_1 - c_0)/(x_1 - x_0) \ .$$

Next, c_0 and c_1 are inserted in the third equation, and we get[1)]

$$c_2 = \big(f_2 - c_0 - c_1(x_2 - x_0)\big)/\big((x_2 - x_0)(x_2 - x_1)\big) \ .$$

With the points from the above table we get

$$c_0 = 0.8912, \quad c_1 = 0.4083, \quad c_2 = -0.4656 \ ,$$

and

$$P(x) = 0.8912 + 0.4083(x - 1.1) - 0.4656(x - 1.1)(x - 1.2) \ .$$

This polynomial is plotted in Figure 5.1 on page 99.

Note that the first term in $P(x)$ is the zero order polynomial that interpolates f in $x_0 = 1.1$. The first two terms interpolate f in x_0 and $x_1 = 1.2$. The third term vanishes at these two points, so the second degree polynomial also interpolates f in the first two points, and the coefficient c_2 is computed so that $P(1.3)$ attains the required value. ∎

The same technique for construction of interpolating polynomials of successively higher degree is used in the proof of the following theorem, which is the basis of polynomial interpolation.

Theorem 5.2.1. Let $x_0, x_1, \ldots, x_n$ be arbitrary, distinct points. For arbitrary values $f_0, f_1, \ldots, f_n$ there is a unique polynomial P of degree at most n, such that

$$P(x_i) = f_i, \quad i = 0, 1, \ldots, n \ .$$

Proof. We first use induction to prove existence. For $n = 0$ we can take the polynomial $P_0(x) = f_0$. This has degree zero, and $P_0(x_0) = f_0$.

Now, assume that P_k is a polynomial of degree at most k, such that

$$P_k(x_i) = f_i, \quad i = 0, 1, ..., k \ .$$

We shall show that we can construct P_{k+1} of degree at most $k+1$, which interpolates f in the points x_i, $i = 0, 1, ..., k, k+1$. Put

$$P_{k+1}(x) = P_k(x) + c(x - x_0)(x - x_1) \cdots (x - x_k) \ .$$

For $c \neq 0$ this is a polynomial of degree $k+1$, and for any c it satisfies

[1)] This algorithm for solving the linear system of equations is called *forward substitution*; see Section 8.2.

$$P_{k+1}(x_i) = P_k(x_i) = f_i, \quad i = 0, 1, ..., k \ .$$

Since the points $x_0, x_1, \ldots, x_{k+1}$ are distinct, we can determine c so that also the condition $P_{k+1}(x_{k+1}) = f_{k+1}$ is satisfied:

$$c = \frac{f_{k+1} - P_k(x_{k+1})}{(x_{k+1} - x_0)(x_{k+1} - x_1) \cdots (x_{k+1} - x_k)} \ .$$

Thus, P_{k+1} is a polynomial of degree $\leq k+1$, which interpolates f in the points $x_0, x_1, \ldots, x_{k+1}$.

The proof of uniqueness is made by contradiction: Assume that P and Q are two polynomials of degree at most n, both of which interpolate the given points,

$$P(x_i) = Q(x_i) = f_i, \quad i = 0, 1, \ldots, n \ .$$

This means that $P-Q$ is a polynomial of degree $\leq n$ with $n+1$ distinct zeros, $x_0, x_1, \ldots, x_n$. According to the fundamental theorem of algebra such a polynomial is identically zero, ie $P = Q$. □

How large is the error, when the function f is approximated by an interpolating polynomial P ? To be able to answer this question, we must of course know more about f than its values at some discrete points.

Theorem 5.2.2. Let f be a function with $n+1$ continuous derivatives in the interval formed by the points $x, x_0, x_1, \ldots, x_n$. If P is the unique polynomial of degree $\leq n$, that satisfies

$$P(x_i) = f(x_i), \quad i = 0, 1, \ldots, n \ ,$$

then

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!} (x - x_0)(x - x_1) \cdots (x - x_n) \ ,$$

for some $\xi(x)$ in the interval formed by the points $x, x_0, x_1, \ldots, x_n$.

Proof. We use *Rolle's theorem*: if a function g is continuous in the interval $[a, b]$, differentiable in the open interval $]a, b[$, and $g(a) = g(b)$, then there is at least one point η in $]a, b[$ such that $g'(\eta) = 0$.

We choose an arbitrary point $\hat{x} \neq x_i$, $i = 0, 1, \ldots, n$, and write the error $f(\hat{x}) - P(\hat{x})$ in the form

$$f(\hat{x}) - P(\hat{x}) = A(\hat{x} - x_0)(\hat{x} - x_1) \cdots (\hat{x} - x_n) \ .$$

To determine the constant A we introduce the auxiliary function

$$\psi(x) = f(\hat{x}) - P(\hat{x}) - A(x - x_0)(x - x_1)\cdots(x - x_n) \ .$$

We see that $\psi(\hat{x}) = 0$ and

$$\psi(x_i) = f(x_i) - P(x_i) - A \cdot 0 = 0, \quad i = 0, 1, \ldots, n \ .$$

Thus, ψ has (at least) $n+2$ distinct roots, $\hat{x}, x_0, x_1, \ldots, x_n$.

According to the assumptions, ψ is $n+1$ times continuously differentiable, and Rolle's theorem shows that ψ' has a zero in each subinterval between two roots of ψ. Therefore, ψ' has $n+1$ distinct roots. Similarly, in each subinterval between two neighbouring roots of ψ', there is a root of ψ'', so that ψ'' has n distinct roots. Repeating this argument, we finally see that $\psi^{(n+1)}$ has a zero in the interval formed by $\hat{x}, x_0, x_1, \ldots, x_n$:

$$\psi^{(n+1)}(\xi) = 0 \ .$$

Differentiating the expression for ψ, we get

$$\psi^{(n+1)}(x) = f^{(n+1)}(x) - A \cdot (n+1)! \ ,$$

so $A = f^{(n+1)}(\xi)/(n+1)!$, where ξ depends on $\hat{x}$. Since $\hat{x}$ is arbitrary, the theorem is proved. □

The error expression in the theorem is easy to remember: The factor $(x-x_0)\cdots(x-x_n)$ ensures that the error is zero at all interpolation points, and if all x_i tend to x_0, then the error term tends to the remainder term in the degree n Taylor expansion of f around x_0.

Example. Figure 5.2 shows the error $\sin x - P(x)$, where P is the second degree polynomial computed in the previous example. The error in the three

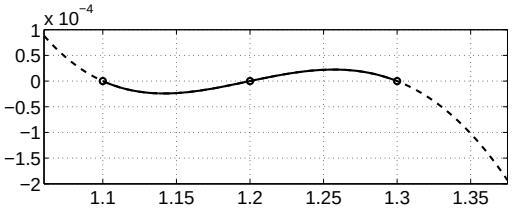


Figure 5.2. *Error in interpolation of* $\sin x$ *by a second degree polynomial.*

interpolation points 1.1, 1.2, 1.3 is zero, and $|\sin x - P(x)| \lesssim 0.25 \cdot 10^{-4}$ for $1.1 \le x \le 1.3$, but the error becomes larger when we extrapolate. ■

5.3. Linear Interpolation

In this section we shall study the influence of different types of errors in connection with linear interpolation, ie when the function f is approximated in the interval $[x_0, x_1]$ by the straight line through the points (x_0, f_0) and (x_1, f_1),

$$P(x) = f_0 + \frac{x - x_0}{x_1 - x_0}(f_1 - f_0) \ . \tag{5.3.1}$$

Example. Find an approximation of $f(1.14)$, when the following function values are known

x_i	f_i
1.1	0.8912
1.2	0.9320

We get

$$P(1.14) = 0.8912 + \frac{1.14 - 1.1}{1.2 - 1.1}(0.9320 - 0.8912)$$

$$= 0.8912 + \frac{0.04}{0.1} \cdot 0.0408 \doteq 0.9075 \ .$$

(We use $\doteq$ to denote correct rounding). ∎

The approximation 0.9075 of the value $f(1.14)$ has four error contributions, cf Chapter 2,

R_{X}	from error in the interpolation argument x,
R_{XF}	from errors in the given function values f_0 and f_1,
R_{T}	from the approximation of f by a straight line,
R_{C}	from rounding errors during computation.

The contributions R_{X} and R_{C} are examined as described in Chapter 2. Here, we shall examine the other two error contributions.

First, we study the truncation error $R_{\mathrm{T}} = f(x) - P(x)$ for $x_0 \le x \le x_1$, see Figure 5.3.

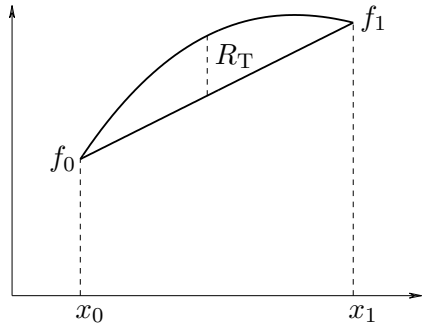


Figure 5.3. *Truncation error in linear interpolation.*

Theorem 5.3.1. Let the function f be twice continuously differentiable in the interval $x_0 \le x \le x_1 = x_0 + h$ and let $P(x)$ be the polynomial that interpolates the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$. For $x_0 \le x \le x_1$ the truncation error can be estimated as

$$|R_T| = |f(x) - P(x)| \le \frac{h^2}{8} \max_{x_0 \le \xi \le x_1} |f''(\xi)| \ .$$

Proof. From Theorem 5.2.2 we see that

$$R_T = \frac{f''(\xi(x))}{2!} (x - x_0)(x - x_1) \ ,$$

where $x_0 < \xi(x) < x_1$. Put $x = x_0 + uh$ with $0 \le u \le 1$, and use the fact that $x_1 = x_0 + h$. Then

$$R_T = \frac{f''(\xi(x))}{2!} h^2 u(u-1) \ .$$

Since $\max_{0 \le u \le 1} |u(u-1)| = \frac{1}{4}$, the theorem follows. □

Note that the truncation error grows with $|f''|$, ie the curvature of f. This can also be seen in Figure 5.3.

How can we estimate f'' if we only know the value of f in certain points? A possible way is to approximate f by an interpolating polynomial of degree 2, and approximate f'' by the second derivative of that polynomial.

Example. Assume that the function in the previous example is also known at a third point,

$$f(1.3) = 0.9936 \ .$$

In the example on page 100 we computed the second degree polynomial that

interpolates f in the points $1.1, 1.2, 1.3$:

$$P(x) = 0.8912 + 0.4083(x - 1.1) - 0.4656(x - 1.1)(x - 1.2) \ .$$

This leads to $P''(x) = -0.4656 \cdot 2 = -0.9312$, and since $h = 1.2 - 1.1 = 0.1$, we get the estimate

$$|R_{\mathrm{T}}| \lesssim \frac{0.1^2}{8} \cdot 0.9312 \leq 0.0012 \ .$$

The example has been constructed with $f(x) = \sin x$, so $f''(x) = -\sin x$, and $-0.9320 \leq f''(x) \leq -0.8912$ for $x \in [1.1, 1.2]$. Thus, our method gives a satisfactory estimate of f'' in this case.

The true truncation error is

$$|R_{\mathrm{T}}| = |\sin 1.14 - 0.9075| \simeq 0.0011 \ .$$

This shows that the estimate of the error is quite accurate. ∎

Next, consider the effect of errors in the given function values, R_{XF}. The following theorem shows that those errors are not enhanced in linear interpolation.

Theorem 5.3.2. Let $\overline{f}_0$ and $\overline{f}_1$ be given approximations of $f(x_0)$ and $f(x_1)$, respectively. The induced error in linear interpolation satisfies

$$|R_{\mathrm{XF}}| \leq \epsilon = \max\left\{|\overline{f}_0 - f(x_0)|, |\overline{f}_1 - f(x_1)|\right\} \ .$$

Proof. Let $f_i = f(x_i)$, $i = 0, 1$, and $u = (x - x_0)/(x_1 - x_0)$. From (5.3.1) we see that the true and the perturbed values of the interpolant are

$$\begin{aligned} P(x) &= f_0 + u(f_1 - f_0) = (1 - u)f_0 + uf_1 \ , \\ \overline{P}(x) &= \overline{f}_0 + u(\overline{f}_1 - \overline{f}_0) = (1 - u)\overline{f}_0 + u\overline{f}_1 \ . \end{aligned}$$

Thus, the error is

$$R_{\mathrm{XF}} = \overline{P}(x) - P(x) = (1 - u)(\overline{f}_0 - f_0) + u(\overline{f}_1 - f_1) \ .$$

Now, we introduce ϵ and use the fact that both u and $1 - u$ are non-negative. Therefore

$$|R_{\mathrm{XF}}| \leq (1 - u)\epsilon + u\epsilon = \epsilon \ ,$$

and the proof is finished. □

Example. In the previous example, assume that the given values are correct to the four decimals given. Then $|R_{\mathrm{XF}}| \leq \epsilon \leq 0.5 \cdot 10^{-4}$. ∎

5.4. Newton's Interpolation Formula

Theorem 5.2.1 tells us that the interpolating polynomial of a function is uniquely determined, as soon as we have chosen the points $x_0, x_1, \ldots, x_n$. There are, however, several ways to write this polynomial. In this section we shall derive a formulation that dates back to Newton, and which gives simple computations in many practical interpolation problems.

In the derivation we use the same technique as in the proof of Theorem 5.2.1, where we generated a sequence of polynomials $P_0, P_1, \ldots, P_n$ by the recursion

$$\begin{aligned} &P_0(x) = c_0 \ , \\ &P_k(x) = P_{k-1}(x) + c_k(x - x_0)\cdots(x - x_{k-1}), \quad k = 1, \ldots, n \ . \end{aligned} \tag{5.4.1}$$

The polynomial P_n interpolates f in $x_0, x_1, \ldots, x_n$, when the coefficients are computed by

$$\begin{aligned} c_0 &= f_0 \ , \\ c_k &= \frac{f_k - P_{k-1}(x_k)}{(x_k - x_0)\cdots(x_k - x_{k-1})} \ , \quad k = 1, \ldots, n \ , \end{aligned}$$

where $f_i = f(x_i)$. We can give explicit expressions for the coefficients:

$$\begin{aligned} c_0 &= f_0 \ , \\ c_1 &= \frac{f_1 - f_0}{x_1 - x_0} \ , \\ c_2 &= \frac{f_2 - f_0 - \frac{x_2 - x_0}{x_1 - x0}(f_1 - f_0)}{(x_2 - x_0)(x_2 - x_1)} \ , \\ &\text{etc} \end{aligned} \tag{5.4.2}$$

The expressions for the following coefficients become increasingly complicated. In order to get simpler formulas for the coefficients, let us redefine the recursion (5.4.1). Remember that

$$P_{k-1} \ \text{ interpolates } f \text{ in } x_0, x_1, \ldots, x_{k-1} \ ,$$

and introduce the polynomial $\vec{P}_{k-1}$ of degree at most $k-1$, such that

$$\vec{P}_{k-1} \ \text{ interpolates } f \text{ in } \quad x_1, \ldots, x_{k-1}, x_k \ .$$

Then the polynomial P_k defined by (5.4.1) can be written in the form

$$P_k(x) = P_{k-1}(x) + \frac{x - x_0}{x_k - x_0}\left(\vec{P}_{k-1}(x) - P_{k-1}(x)\right) \ . \tag{5.4.3}$$

We must prove this statement: Both P_{k-1} and $\vec{P}_{k-1}$ are polynomials of degree $\leq k-1$. Therefore, P_k is a polynomial of degree $\leq k$. Further,

$$\begin{aligned} P_k(x_0) &= P_{k-1}(x_0) + 0 = f_0 \ , \\ P_k(x_i) &= P_{k-1}(x_i) + \frac{x_i - x_0}{x_k - x_0}(f_i - f_i) = f_i, \ i = 1, \ldots, k-1 \ , \\ P_k(x_k) &= P_{k-1}(x_k) + \vec{P}_{k-1}(x_k) - P_{k-1}(x_k) = f_k \ . \end{aligned}$$

Thus, P_k is the unique polynomial of degree $\leq k$, which interpolates f in $x_0, x_1, \ldots, x_k$, and we have proved that (5.4.3) is correct.

The coefficient c_k in (5.4.1) depends on $x_0, x_1, \ldots, x_k$ and the corresponding values of f. We introduce the notation

$$c_k = f[x_0, x_1, \ldots, x_k] \ .$$

This is the coefficient of x^k in $P_k(x)$. Accordingly, the coefficients of x^{k-1} in $P_{k-1}(x)$ and $\vec{P}_{k-1}(x)$ are $f[x_0, x_1, \ldots, x_{k-1}]$ and $f[x_1, x_2 \ldots, x_k]$, respectively, and from (5.4.3) we get

$$c_k = f[x_0, x_1, \ldots, x_k] = \frac{f[x_1, x_2 \ldots, x_k] - f[x_0, x_1, \ldots, x_{k-1}]}{x_k - x_0} \ .$$

This quotient between two differences is called a divided difference.

Definition 5.4.1. *The kth divided difference of f with respect to the points $x_0, x_1, \ldots, x_k$* is given by

$$\begin{aligned} f[x_i] &= f(x_i) \ , \\ f[x_0, x_1, \ldots, x_k] &= \frac{f[x_1, x_2 \ldots, x_k] - f[x_0, x_1, \ldots, x_{k-1}]}{x_k - x_0} \ . \end{aligned}$$

Divided differences are conveniently arranged in a table like the following,

$$\begin{array}{llll} x_0 & f_0 & & \\ & & f[x_0, x_1] & \\ x_1 & f_1 & & f[x_0, x_1, x_2] \\ & & f[x_1, x_2] & & f[x_0, x_1, x_2, x_3] \\ x_2 & f_2 & & f[x_1, x_2, x_3] \\ & & f[x_2, x_3] & \\ x_3 & f_3 & & \end{array}$$

Eg, we have

$$f[x_0, x_1] = \frac{f_1 - f_0}{x_1 - x_0} \ ,$$
$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} \ ,$$
$$f[x_0, x_1, x_2, x_3] = \frac{f[x_1, x_2, x_3] - f[x_0, x_1, x_2]}{x_3 - x_0} \ .$$

It follows from above that the coefficients in the interpolating polynomial

$$\begin{aligned} P_3(x) = c_0 \ &+ c_1(x - x_0) + c_2(x - x_0)(x - x_1) \\ &+ c_3(x - x_0)(x - x_1)(x - x_2) \end{aligned} \tag{5.4.4}$$

are given by

$$c_0 = f_0, \quad c_1 = f[x_0, x_1], \quad c_2 = f[x_0, x_1, x_2], \quad c_3 = f[x_0, x_1, x_2, x_3] \ .$$

It is easy to see that these expressions for c_0 and c_1 agree with (5.4.2), while the agreement for c_2 is less obvious. However, from (5.4.2) we get

$$\begin{aligned} c_2 &= \frac{1}{x_2 - x_1} \left(\frac{f_2 - f_0}{x_2 - x_0} - \frac{f_1 - f_0}{x_1 - x_0} \right) \\ &= \frac{f[x_0, x_2] - f[x_1, x_0]}{x_2 - x_1} = f[x_1, x_0, x_2] \ , \end{aligned}$$

since $f[x_0, x_1] = f[x_1, x_0]$. We shall show that $f[x_1, x_0, x_2] = f[x_0, x_1, x_2]$: These two divided differences are the coefficients of x^2 in the second order polynomials that interpolate the three points (x_1, f_1), (x_0, f_0), (x_2, f_2) and (x_0, f_0), (x_1, f_1), (x_2, f_2), respectively. The order, in which the three points are taken, is of no consequence, so the two polynomials are identical, ie $f[x_1, x_0, x_2] = f[x_0, x_1, x_2]$. By the same argument we can show that the value of $f[x_0, x_1, \ldots, x_k]$ does not change if the arguments $x_0, x_1, \ldots, x_k$ are given in a different order.

Example. Determine a polynomial that interpolates the points $(-1, 6)$, $(0, 1)$, $(2, 3)$, $(5, 66)$.

We get the table shown on the next page. The resulting 3rd degree polynomial is

$$P(x) = 6 - 5(x + 1) + 2(x + 1)x + \tfrac{1}{3}(x + 1)x(x - 2) \ .$$

Check that this polynomial does indeed interpolate the given data.

x	$f(x)$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$
-1	6			
		$\frac{1-6}{0-(-1)} = -5$		
0	1		$\frac{1-(-5)}{2-(-1)} = 2$	
		$\frac{3-1}{2-0} = 1$		$\frac{4-2}{5-(-1)} = \frac{1}{3}$
2	3		$\frac{21-1}{5-0} = 4$	
		$\frac{66-3}{5-2} = 21$		
5	66			

■

The results are summarized in the following theorem.

Theorem 5.4.2. Newton's interpolation polynomial.

$$P_n(x) = f_0 + f[x_0,x_1](x-x_0) + f[x_0,x_1,x_2](x-x_0)(x-x_1) + \cdots + f[x_0,x_1,\ldots,x_n](x-x_0)(x-x_1)\cdots(x-x_{n-1})$$

of degree $\le n$ satisfies $P_n(x_i) = f_i$, $i = 0, 1, \ldots, n$.

The coefficients are easily computed, and once they are known, values of the polynomial P_n can be computed by a generalization of Horner's rule (Section 4.6). The idea is illustrated by the following reformulation of (5.4.4)

$$P_3(x) = c_0 + (x-x_0)\big(c_1 + (x-x_1)\big(c_2 + (x-x_2)c_3\big)\big) \ .$$

This can be computed as $P_3(x) = b_0$, given by the recursion

$$b_3 = c_3 \ ,$$
$$b_j = b_{j+1}(x-x_j) + c_j, \quad j = 2,1,0 \ .$$

The algorithm easily generalizes to arbitrary n.

Example. The following MATLAB function `intpolc` uses divided differences to compute the coefficients in an interpolating polynomial. Remember that MATLAB indexes from 1, so that x_i is stored in `x(i+1)`, and if the vectors `x` and `f` have length m, then the interpolating polynomial has degree at most $n = m-1$. The function `intpval` implements a generalization of the above algorithm for evaluation of the interpolating polynomial. If the argument `t` is a vector, then the output `p` is a vector of the same type with $p_i = P(t_i)$, $i = 1, 2, \ldots,$ `length(t)`.

```
function  c = intpolc(x,f)
% Coefficients c in Newton's form of the polynomial
% interpolating (x1,f1), ..., (xm,fm)
% f(k:m) is successively overwritten by divided
% differences of order k-1
m = length(x);            % number of interpolation points
for  k = 2 : m
  f(k:m) = (f(k:m) - f(k-1:m-1)) ./ (x(k:m) - x(1:m+1-k));
end
c = f;  % return coefficients

function  p = intpval(x,c,t)
% Value p = P(t) of interpolating polynomial
m = length(x);  % number of interpolation points and coeffs
p = c(m) * ones(size(t));
for  k = m-1 : -1 : 1
  p = p .* (t - x(k)) + c(k);
end
```

With the data from the previous example we get

```
>> x = [-1 0 2 5];
>> c = intpolc(x,[6 1 3 66])
c = 6   -5    2    0.3333

>> p = intpval(x,c,[-1 5 1])
p = 6   66   -0.6667
```

Thus, we get the same coefficients as in the previous example, and the results from `intpval` show that $P(x_i) = f_i$ for two of the interpolation points, and the interpolated value in $x = 1$ is $-2/3$. ∎

In applications one is interested not only in getting an approximation of the function value, but also in getting an estimate of the truncation error of the approximation. We did that for linear interpolation in the example on page 105, and that approach generalizes.

According to (5.4.1) and Theorem 5.4.2 we have

$$P_k(x) = P_{k-1}(x) + f[x_0, x_1, \ldots, x_k](x - x_0)(x - x_1)\cdots(x - x_{k-1}) \ ,$$

and $P_k(x_k) = f(x_k)$. Therefore,

$$f(x_k) - P_{k-1}(x_k) = f[x_0, x_1, \ldots, x_k](x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1}) \ .$$

The right hand side is the truncation error for $x = x_k$ when f is approximated by the interpolating polynomial P_{k-1}, with interpolation points $x_0, x_1, \ldots, x_{k-1}$. Theorem 5.2.2 gives another expression for this error: if f is k times continuously differentiable, then

$$f(x_k) - P_{k-1}(x_k) = \frac{f^{(k)}(\xi)}{k!}\,(x_k - x_0)(x_k - x_1)\cdots(x_k - x_{k-1}) \ .$$

Comparing the two expressions for the truncation error, we get the following theorem.

Theorem 5.4.3. If the function f is k times continuously differentiable in the interval formed by the points $x_0, x_1, \ldots, x_k$, then there is a point ξ in this interval, such that

$$f[x_0, x_1, \ldots, x_k] = \frac{f^{(k)}(\xi)}{k!} \ .$$

Therefore, if $f^{(k)}$ does not vary too much in the interval, then it is reasonable to estimate $f^{(k)}(\xi(x))/k!$ by the divided difference. Note that $f[x_0, x_1, \ldots, x_k] = c_k$ is the coefficient of the highest power of x in $P_k(x)$. In other words, the estimate is based on approximating $f^{(k)}$ by the kth derivative of P_k, just as we did for $k = 2$ in the example on page 105.

The result of this discussion can be formulated in another way: P_{k-1} is obtained from P_k by neglecting the term of highest degree, and this term can be used to estimate the error in the approximation P_{k-1}.

When a function is approximated by Newton's interpolation polynomial, then the truncation error can be estimated by the first neglected term.

Example. The MATLAB function `polyfit` can be used to find the interpolating polynomial in the form

$$P_n(x) = d_1 x^n + d_2 x^{n-1} + \cdots + d_n x + d_{n+1} \ .$$

The coefficients are computed by setting up and solving a linear system of equations, which express that

$$x_i^n d_1 + x_i^{n-1} d_2 + \cdots x_i d_n + d_{n+1} = f_i, \quad i = 0, 1, \ldots, n \ .$$

With the data from the example on page 109 we get

```
>> x = [-1 0 2 5];  f = [6 1 3 66];
>> d = polyfit(x,f, 3)
d = 0.3333    1.6667   -3.6667    1.0000
```

The last input parameter tells that we want a degree 3 polynomial. See page 273 about the choice of a degree that is smaller than `length(x)-1`.

Earlier we found the following expression for the interpolating polynomial

$$P(x) = 6 - 5(x+1) + 2(x+1)x + \tfrac{1}{3}(x+1)x(x-2) \ .$$

The result for `d` corresponds to the polynomial

$$Q(x) = \tfrac{1}{3}x^3 + \tfrac{5}{3}x^2 - \tfrac{11}{3}x + 1 \ .$$

The reader should verify that $P(x) = Q(x)$.

When the coefficients are known, the MATLAB function `polyval` can be used to evaluate $P_n(x)$ (by means of Horner's rule).

```
>> p = polyval(d,[-1 5 1])
p = 6   66   -0.6667
```

We recognize the results we got earlier.

It should be mentioned that the computation by means of Newton's interpolation polynomial is less sensitive to rounding errors. This aspect is discussed in an example on page 249. ∎

5.5. Neville's Method

In this section we shall show that the evaluation of an interpolating polynomial of degree $\leq n$ can be performed as a series of linear interpolations. This approach is widely used in computer graphics.

According to (5.4.3) the value $P_k(x)$ can be expressed as

$$P_k(x) = P_{k-1}(x) + \frac{x - x_0}{x_k - x_0}\left(\vec{P}_{k-1}(x) - P_{k-1}(x)\right) \ ,$$

and by comparison with (5.3.1) we wee that this can be interpreted as a linear interpolation between the "points" $(x_0, P_{k-1}(x))$ and $(x_k, \vec{P}_{k-1}(x))$. This is the background for *Neville's method*. We shall give a description of it, where the notation emphasizes the interpolation points.

Let $P_{01}(x)$ denote the polynomial of degree ≤ 1, which interpolates the function f in x_0 and x_1:

$$P_{01}(x) = f_0 + \frac{x - x_0}{x_1 - x_0}\big(f_1 - f_0\big) = \frac{(x - x_0)f_1 - (x - x_1)f_0}{x_1 - x_0} \ .$$

Next, let $P_{012}(x)$ denote the polynomial of degree ≤ 2, which interpolates the function f in x_0, x_1 and x_2. The polynomial can be computed by linear interpolation between $(x_0, P_{01}(x))$ and $(x_2, P_{12}(x))$,

$$P_{012}(x) = \frac{(x - x_0)P_{12}(x) - (x - x_2)P_{01}(x)}{x_2 - x_0} \ .$$

Let us verify this: Since P_{01} and P_{12} are polynomials of degree ≤ 1, the

right hand side is a polynomial of degree ≤ 2, and we just need to show that $P_{012}(x_j) = f_j$, $j = 0, 1, 2$:

$$\begin{aligned} P_{012}(x_0) &= \frac{-(x_0 - x_2)P_{01}(x_0)}{x_2 - x_0} = P_{01}(x_0) = f_0 \ , \\ P_{012}(x_1) &= \frac{(x_1 - x_0)P_{12}(x_1) - (x_1 - x_2)P_{01}(x_1)}{x_2 - x_0} \\ &= \frac{(x_1 - x_0 - x_1 + x_2)f_1}{x_2 - x_0} = f_1 \ , \\ P_{012}(x_2) &= P_{12}(x_2) = f_2 \ . \end{aligned}$$

One can continue like this and construct interpolating polynomials of higher degree by successive linear interpolation:

Theorem 5.5.1. Let $P_{i_0 i_1 \ldots i_k}$ be the polynomial of degree $\leq k$ that satisfies

$$P_{i_0 i_1 \ldots i_k}(x_{i_j}) = f_{i_j}, \quad j = 0, 1, \ldots, k \ .$$

The polynomial can be computed by the recursion

$$\begin{aligned} P_i(x) &= f_i \ , \\ P_{i_0 i_1 \ldots i_k}(x) &= \frac{(x - x_{i_0})P_{i_1 \ldots i_k}(x) - (x - x_{i_k})P_{i_0 \ldots i_{k-1}}(x)}{x_{i_k} - x_{i_0}} \ . \end{aligned}$$

The theorem is easily proved by induction.

Neville's method is conveniently arranged in a table like the following.

$$\begin{array}{l|llllll} x - x_0 & x_0 & f_0 = P_0(x) & & & & \\ & & & P_{01}(x) & & & \\ x - x_1 & x_1 & f_1 = P_1(x) & & P_{012}(x) & & \\ & & & P_{12}(x) & & P_{0123}(x) & \\ x - x_2 & x_2 & f_2 = P_2(x) & & P_{123}(x) & & \\ & & & P_{23}(x) & & & \\ x - x_3 & x_3 & f_3 = P_3(x) & & & & \end{array}$$

Eg, we have

$$P_{123}(x) = \frac{(x - x_1)P_{23}(x) - (x - x_3)P_{12}(x)}{x_3 - x_1} \ .$$

Example. Compute the value for $x = 1$ of the polynomial that interpolates the points $(-1, 6)$, $(0, 1)$, $(2, 3)$, $(5, 66)$.

$1-x_i$	x_i	f_i			
2	-1	6			
			$\frac{2\cdot 1-1\cdot 6}{0+1}=-4$		
1	0	1		$\frac{2\cdot 2+1\cdot(-4)}{2+1}=0$	
			$\frac{1\cdot 3+1\cdot 1}{2-0}=2$		$\frac{2\cdot(-2)+4\cdot 0}{5+1}=-\frac{2}{3}$
-1	2	3		$\frac{1\cdot(-18)+4\cdot 2}{5-0}=-2$	
			$\frac{-1\cdot 66+4\cdot 3}{5-2}=-18$		
-4	5	66			

We naturally get the same result as in the example on page 110. ∎

Neville's method is useful for computing the value of an interpolating polynomial at a particular point. If an explicit expression for the polynomial is needed or if values of the polynomial are desired at several points, then it is better to use Newton's interpolating polynomial.

5.6. Lagrange's Interpolating Polynomial

Lagrange's interpolating polynomial is another way of representing the interpolating polynomial of a given function. This form is often used in textbooks for the proof of Existence, Theorem 5.2.1. In Chapter 7 we shall see that the Lagrange formulation can be used in the derivation of formulas for numerical integration. We shall only give a traditional formulation, which is neither efficient nor accurate for practical computation. It is possible, however, to reformulate the method[2)] so that it is competitive with the computation via Newton's interpolation polynomial.

Lagrange's interpolating polynomial of degree $\leq n$, which interpolates the function f in $x_0, x_1, \ldots, x_n$, is given by

$$P_n(x) = \sum_{i=0}^{n} f(x_i) L_i(x) \ ,$$

where L_i is the degree n polynomial

$$L_i(x) = \frac{(x-x_0)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_n)}{(x_i-x_0)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_n)} \ .$$

[2)] See J.-P. Berrut and L.N. Trefethen, *Barycentric Lagrange interpolation*, to appear in SIAM Review 2004.

Example. For $n=2$ the three polynomials are $L_0(x) = \dfrac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}$,
$L_1(x) = \dfrac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}$ and $L_2(x) = \dfrac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$. ∎

As an exercise, show that $P_n(x_i) = f(x_i)$ and that the degree of the polynomial is at most n.

5.7. Hermite Interpolation

It sometimes happens that we know both the function values $f(x_i)$ and the derivatives $f'(x_i)$ at the points $x_0, x_1, \ldots, x_n$. Then we can determine a polynomial P_{2n+1} of degree at most $2n+1$, such that

$$P_{2n+1}(x_i) = f(x_i) \quad \text{and} \quad P'_{2n+1}(x_i) = f'(x_i), \quad i = 0, 1, \ldots, n \ .$$

The polynomial is called the *Hermite interpolant*. We shall only discuss the case with two points, $n=1$. To prepare for the use in Section 5.11 we denote them x_{i-1} and $x_i = x_{i-1} + h_i$, and we let q_i denote the polynomial of degree ≤ 3. We write the polynomial in the form

$$q_i(x) = a_i + b_i u + c_i u^2 + d_i u^3, \quad u = \frac{x - x_{i-1}}{h_i} \ . \tag{5.7.1}$$

The coefficients must satisfy the following linear system of equations[3)]

$$\begin{aligned}
q_i(x_{i-1}) &= a_i &&&&&&= f_{i-1} \ , \\
q_i(x_i) &= a_i + b_i + c_i + d_i &&&&&= f_i \ , \\
h_i q'_i(x_{i-1}) &= b_i &&&&&= h_i f'_{i-1} \ , \\
h_i q'_i(x_i) &= b_i + 2c_i + 3d_i &&&&&= h_i f'_i \ ,
\end{aligned}$$

where f_j and f'_j are the given values of f and f'. This system has the unique solution (verify that!)

$$\begin{aligned}
a_i &= f_{i-1} \ , \\
b_i &= h_i f'_{i-1} \ , \\
c_i &= 3(f_i - f_{i-1}) - h_i(2f'_{i-1} + f'_i) \ , \\
d_i &= 2(f_{i-1} - f_i) + h_i(f'_{i-1} + f'_i) \ .
\end{aligned} \tag{5.7.2}$$

[3)] Remember that $'$ denotes differentiation with respect to x, and $\dfrac{d}{dx} = \dfrac{1}{h_i}\dfrac{d}{du}$.

It can be shown that the following estimate of the approximation error holds.

Theorem 5.7.1. Let the function f be four times continuously differentiable in the interval $[x_{i-1}, x_i]$, and let q_i be the cubic Hermite interpolant. Then

$$\max_{x_{i-1}\le x\le x_i} |f(x) - q_i(x)| \le \tfrac{1}{384} M_i h_i^4 + \tfrac{1}{4} E_i' h_i + E_i \ ,$$

where

$$h_i = x_i - x_{i-1} \ , \qquad M_i = \max_{x_{i-1}\le x\le x_i} |f^{(4)}(x)| \ ,$$
$$E_i = \max_{j=i-1,i} |f(x_j) - f_j| \ , \quad E_i' = \max_{j=i-1,i} |f'(x_j) - f_j'| \ .$$

If the given f_j agree with $f(x_j)$, then $E_i = 0$, and if $f_j' = f'(x_j)$, then $E_i' = 0$.

Example. We shall use cubic Hermite interpolation to approximate $f(x) = \sin x$ for $x \in [1.1, 1.3]$.

x_j	$f_j = \sin x_j$	$f_j' = \cos x_j$
1.1	0.8912	0.4536
1.3	0.9636	0.2675

When we insert these values (with $h_i = 0.2$) in (5.7.1) and (5.7.2), we get

$$q_i(x) = 0.8912 + 0.09072u - 0.01789u^2 - 0.0004827u^3, \quad u = \frac{x - 1.1}{0.2} \ .$$

The error is shown in Figure 5.4. Note that both the error function and its

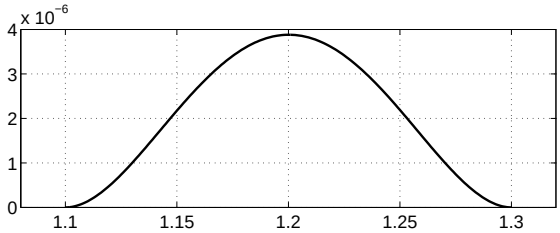


Figure 5.4. *Error in cubic Hermite interpolation of* $\sin x$.

derivative is zero at the interpolation points. It is seen that

$$\max_{1.1\le x\le 1.3} |\sin x - q_i(x)| \lesssim 4 \cdot 10^{-6} \ .$$

This is in good agreement with Theorem 5.7.1: We have $E_i = E_i' = 0$, and with $f^{(4)}(x) = \sin x$ we get the upper bound $4.01 \cdot 10^{-6}$ on the truncation error. ∎

5.8. Runge's Phenomenon

When a function f is approximated by interpolation in an interval, then the error in the interpolation points is zero, and it is tempting to believe that if we increase the number of interpolation points in such a way that they get closer and closer, then the truncation error becomes smaller. This, however, is not always the case. The classical counter-example was constructed by Runge.

Example. Interpolate the function

$$f(x) = \frac{1}{1+25x^2} \ , \qquad -1 \le x \le 1 \ ,$$

by a polynomial P_n of degree $\le n$, using equidistant points,

$$x_i = -1 + \frac{2i}{n} \ , \quad i = 0, 1, \ldots, n \ .$$

Figure 5.5 shows the function and the interpolating polynomial P_n for $n = 4$, 6, 10. As n increases, the error gets smaller close to $x = 0$, but close to $x = \pm 1$ the error increases with n.

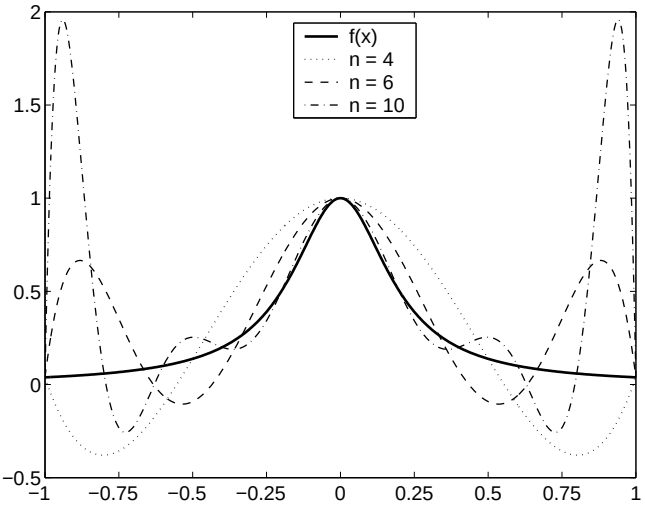


Figure 5.5. *Runge's phenomenon.*

For this function and this distribution of interpolation points it can be shown that

$$\max_{-1 \le x \le 1} |f(x) - P_n(x)| \to \infty \quad \text{for} \quad n \to \infty \ .$$

■

In Chapter 9 we shall see that in many cases the error can be reduced by other choices of interpolation points. In general, however, there is no guarantee that we can get a good approximation with a high degree polynomial and suitable interpolation points. It can be shown that, for any choice of interpolation points there is a continuous function such that the maximal error in polynomial interpolation tends to infinity, as the number of interpolation points tends to infinity.

Rule of thumb: only interpolate with polynomials of low degree.

5.9. Spline Interpolation

Runge's phenomenon shows that we cannot be sure to get a better approximation by increasing the degree of an interpolating polynomial. Also, polynomials are not suited for approximation of functions that change their character in different parts of the interval. This is the case for many functions that describe a physical phenomenon or the shape of an object.

An example is shown in Figure 5.6. If we approximate this function

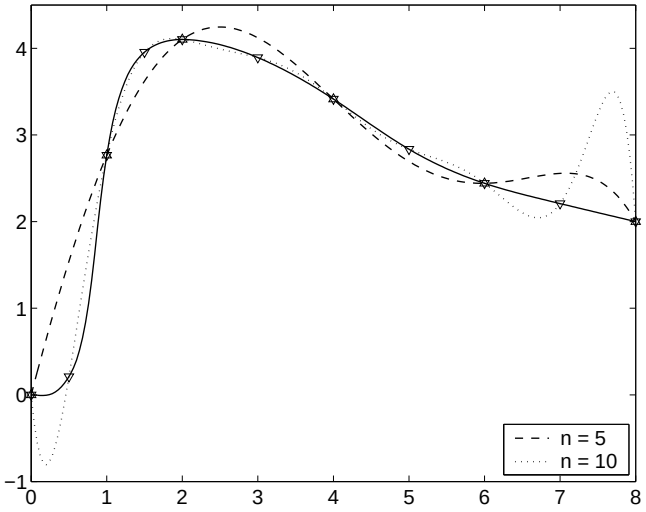
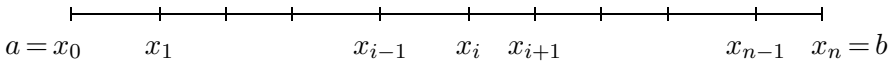


Figure 5.6. *Approximation with interpolating polynomials of degree 5 and 10.*

by an interpolating polynomial, we get a poor approximation by a low degree polynomial, and with a high degree polynomial we get large oscillations in part of the interval. It is better to approximate by different, low degree polynomials in different parts of the interval.

Let $[a, b]$ be the interval under consideration. We subdivide it by introducing points $a = x_0 < x_1 < \cdots < x_n = b$.



In each of the n subintervals $[x_{i-1}, x_i]$ we use a different polynomial. The simplest choice is to use a straight line in each subinterval. Then the approximation is a polygonal curve, ie a nonsmooth curve with discontinuities in the first derivative at the points $x_1, \ldots, x_{n-1}$. In many applications one gets good approximations by using third degree polynomials in each subinterval, determined so that the approximating function and its first two derivatives are continuous.

A function s is called a *spline*[4)] of degree $2m+1$ if s is composed of polynomials of degree $2m+1$ in such a way that s and its first $2m$ derivatives are continuous. For $m = 0$ and $m = 1$ one has respectively a *linear spline* and a *cubic spline*. If $s(x_i) = f(x_i)$, $i = 0, 1, \ldots, n$, we have an interpolating spline.

Example. The name "spline" comes from ship building. A physical spline is a thin, elastic ruler, that is fixed in certain points.

Such a ruler was used by ship builders to mark the shape of a hull. The ruler will assume a shape that minimizes its strain energy.

In the mid 1940s some researchers became interested in finding a mathematical model of the physical spline.

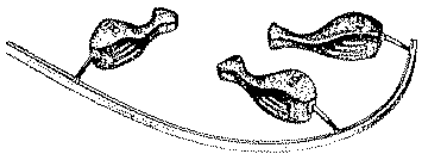


Figure 5.7. *Physical spline.*

Assume that the shape of the physical spline is unimodal in a certain coordinate system, and that it can be described by $y = f(x)$ for $a \le x \le b$. Then the strain energy of the ruler is

[4)] Strictly speaking: a *spline function*.

$$E_p = A \int_a^b \frac{(f''(x))^2}{(1+(f'(x))^2)^{5/2}}\, dx \ ,$$

where A is a constant that depends on the material and the cross section of the ruler. If $(f'(x))^2 \ll 1$ or f' is almost constant, then

$$E_p \simeq B \int_a^b (f''(x))^2\, dx \ ,$$

where B may be another constant. It turns out that this integral is minimal if $f(x) = s(x)$, the "natural" cubic spline, which interpolates the fixed points; see Theorem 5.11.3. ∎

Cubic splines have many applications in computer graphics and computer aided design of curves and surfaces in eg car and aeroplane industry. We start by describing linear splines to give a simple introduction to the concept of piecewise polynomials.

5.10. Linear Spline Functions

Definition 5.10.1. Let $a = x_0 < x_1 < \cdots < x_n = b$ be given points, so-called *knots*. A function s is said to be a *linear spline* on the interval $[a, b]$, if

s is continuous in $[a, b]$, and

s is a straight line in each knot interval $[x_{i-1}, x_i]$, $i = 1, \ldots, n$.

Thus, a linear spline s is composed of n straight lines,

$$s_i(x) = a_i + b_i(x - x_{i-1}), \quad x_{i-1} \le x \le x_i \ .$$

This is illustrated in Figure 5.8.

There are $2n$ coefficients to be determined, but the continuity requirement gives $n-1$ conditions: In each interior knot x_i the value $s_i(x_i)$ must be equal to $s_{i+1}(x_i)$. This gives the conditions

$$a_i + b_i(x_i - x_{i-1}) = a_{i+1}, \quad i = 1, 2, \ldots, n-1 \ .$$

There remain $2n-(n-1) = n+1$ *degrees of freedom*. They can be used to make the spline take given values in the knots, $s(x_i) = f_i$, $i = 0, 1, \ldots, n$. This gives

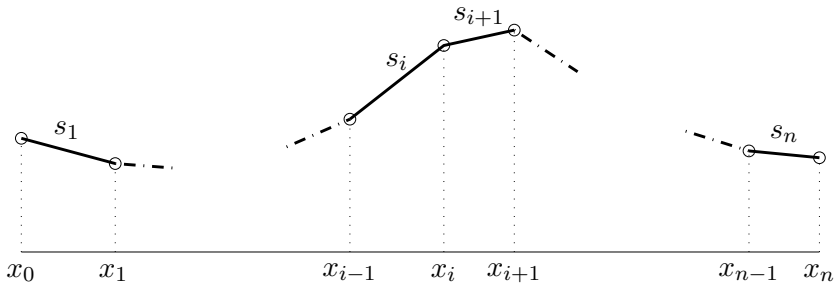


Figure 5.8. *Linear spline.*

$$a_i = f_{i-1}, \qquad b_i = \frac{f_i - f_{i-1}}{x_i - x_{i-1}} \ , \quad i = 1, 2, \ldots, n \ .$$

An alternative way of representing an interpolating linear spline is

$$s(x) = \sum_{i=0}^{n} f_i l_i(x) \ , \quad x_0 \le x \le x_n \ ,$$

where $l_i(x)$ is a linear spline function with the property (cf Figure 5.9)

$$l_i(x_j) = \delta_{ij} = \begin{cases} 0, & i \ne j \ , \\ 1, & i = j \ . \end{cases}$$

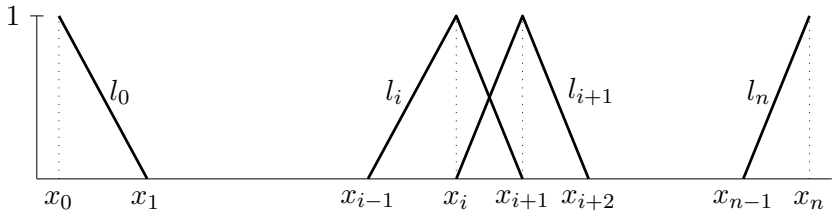


Figure 5.9. *Basis functions for linear splines.*

It is easily seen that the linear spline $l_i(x)$ is unique. We have

$$l_0(x) = \begin{cases} (x_1 - x)/(x_1 - x_0), & x_0 \le x \le x_1 \ , \\ 0, & x_1 \le x \le x_n \ . \end{cases}$$

For $i = 1, 2, \ldots, n-1$:

$$l_i(x) = \begin{cases} 0, & x_0 \le x \le x_{i-1} \ , \\ (x - x_{i-1})/(x_i - x_{i-1}), & x_{i-1} \le x \le x_i \ , \\ (x_{i+1} - x)/(x_{i+1} - x_i), & x_i \le x \le x_{i+1} \ , \\ 0, & x_{i+1} \le x \le x_n \ , \end{cases}$$

and

$$l_n(x) = \begin{cases} 0, & x_0 \le x \le x_{n-1} \ , \\ (x - x_{n-1})/(x_n - x_{n-1}), & x_{n-1} \le x \le x_n \ . \end{cases}$$

The functions $\{l_i\}_{i=0}^n$ are basis functions for the space of all linear splines with the given knots. They are so-called *linear B-splines*. Note that each of them is nonzero only in two consecutive knot intervals.

Now, assume that $f_i = f(x_i)$, $i = 0, 1, \ldots, n$, where f is a given, twice continuously differentiable function. In the ith interval we can use the analysis of linear interpolation, Section 5.3: Let

$$h_i = x_i - x_{i-1}$$

be the length of the interval, then Theorem 5.3.1 gives the following estimate of the *local error*,

$$\max_{x_{i-1} \le x \le x_i} |f(x) - s(x)| \le \frac{h_i^2}{8} \max_{x_{i-1} \le x \le x_i} |f''(x)| \ . \tag{5.10.1}$$

It follows that the *global error* can be estimated by

$$\max_{x_0 \le x \le x_n} |f(x) - s(x)| \le \frac{h^2}{8} \max_{x_0 \le x \le x_n} |f''(x)| \ , \qquad h = \max\{h_i\} \ .$$

The placement of the knots affects the error. From the estimate (5.10.1) of the local error we see that the knots should be close in regions where $|f''|$ is large (ie where f varies rapidly), while the knots may be further apart in regions where $|f''|$ is small (ie where f is almost a straight line).

Example. Figure 5.10 shows the function from Figure 5.6 and the interpolation by two linear splines with $n = 5$. The spline with equidistant knots has

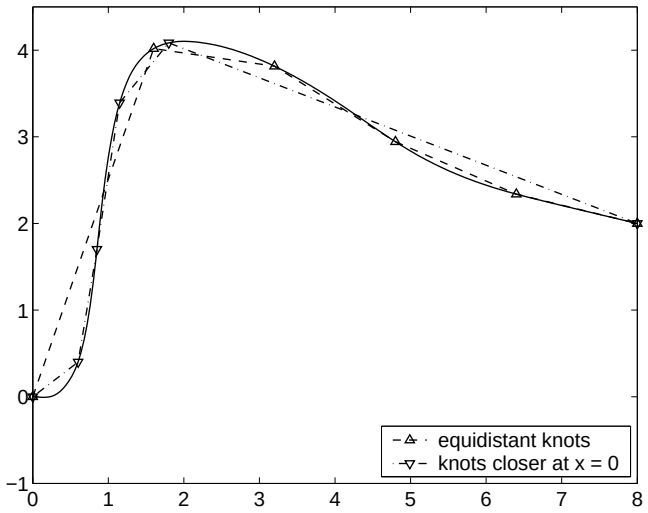


Figure 5.10. *Approximation with interpolating linear spline.*

maximum truncation error 1.106. The function varies little for $x \gtrsim 2.5$, and if we take the knots as `[0 0.60 0.85 1.15 1.80 8]`, then the maximum truncation error is reduced to 0.237. ∎

5.11. Cubic Splines

Definition 5.11.1. Let $a = x_0 < x_1 < \cdots < x_n = b$ be given knots. A function s is said to be a *cubic spline* on the interval $[a, b]$, if

s, s' and s'' are continuous in $[a, b]$, and

s is a polynomial of degree ≤ 3 in each knot interval $[x_{i-1}, x_i]$, $i = 1, \ldots, n$.

The spline s is composed of n cubic polynomials,

$$s(x) = s_i(x), \quad x_{i-1} \leq x \leq x_i \ .$$

This is illustrated in Figure 5.11.

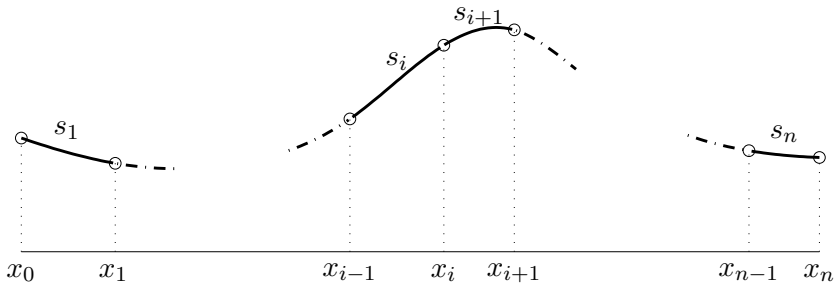


Figure 5.11. *Cubic spline.*

Each s_i has four coefficients, so there are $4n$ coefficients to be determined, but the continuity requirements give $3(n-1)$ conditions,

$$\left.\begin{aligned} s_i(x_i) &= s_{i+1}(x_i) \\ s_i'(x_i) &= s_{i+1}'(x_i) \\ s_i''(x_i) &= s_{i+1}''(x_i) \end{aligned}\right\} \quad i = 1, \ldots, n-1 \ .$$

There remain $4n - 3(n-1) = n+3$ degrees of freedom. If we choose to make the spline interpolate in the knots, $s(x_i) = f(x_i)$, $i = 0, 1, \ldots, n$, then we have $n+1$ conditions. Common choices for the two remaining conditions are given in the following theorem.

Theorem 5.11.2. Let $\{x_i\}_{i=0}^n$ be given points with $a = x_0 < x_1 < \cdots < x_n = b$. A cubic spline s with knots $x_0, x_1, \ldots, x_n$ is uniquely determined by the interpolation conditions

$$s(x_i) = f(x_i), \quad i = 0, 1, \ldots, n \ ,$$

supplied with any of the following choices of two extra conditions:

"Natural spline" $\quad s_1''(x_0) = 0, \quad s_n''(x_n) = 0 \ .$

Correct boundary conditions: $\quad s_1'(x_0) = f'(x_0), \quad s_n'(x_n) = f'(x_n) \ .$

"Not-a-knot": $\quad s_1^{(3)}(x_1) = s_2^{(3)}(x_1), \quad s_{n-1}^{(3)}(x_{n-1}) = s_n^{(3)}(x_{n-1}) \ .$

Periodic boundary conditions: $\quad s_1^{(r)}(x_0) = s_n^{(r)}(x_n), \quad r = 1, 2 \ .$

Before we prove the theorem, we discuss the extra conditions.

The term *"Natural spline"* refers to properties of a physical spline: The ruler is straight outside the interval of fixed points, ie the second derivative is zero for $x \le a$ and $x \ge b$.

Normally, *correct boundary conditions* give significantly better approximation of f close to the endpoints (except if $f''(x)$ is close to zero for x close to a and b). Therefore, if $f'(a)$ and $f'(b)$ (or good approximations of them) are available, then we recommend to use them.

The third derivative of s_i is constant in each knot interval, and normally $s^{(3)}(x)$ jumps when we pass a knot. The *"not-a-knot"* condition is that the first and last knot intervals are $[x_0, x_2]$ and $[x_{n-2}, x_n]$, respectively, but x_1 and x_{n-1} are still used as interpolation points. Thus, the degrees of freedom reduces to $(n-2)+3 = n+1$, which is equal to the number of conditions of the form $s(x_i) = f(x_i)$.

The *periodic boundary conditions* are used for interpolation of periodic functions. We already have periodicity of the knot values of s itself from the conditions $s(x_0) = f(a) = f(b) = s(x_n)$, and the extra conditions ensure that the first and second derivative also are periodic.

Now, we shall prove Theorem 5.11.2. The proof is constructive: it gives an algorithm for computing the spline.

The interpolation property and the continuity of s are satisfied if we prescribe that

$$s_i(x_{i-1}) = f_{i-1}, \quad s_i(x_i) = f_i, \quad i = 1, 2, \ldots, n \ . \tag{5.11.1}$$

Further, let $\{s'_j\}$ denote the values of the derivative of s at the knots. The continuity of s' is satisfied if we prescribe that

$$s'_i(x_{i-1}) = s'_{i-1}, \quad s'_i(x_i) = s'_i, \quad i = 1, 2, \ldots, n \ .$$

We do not know $\{s'_j\}$, but assuming that we did, this is the problem discussed in Section 5.7 about Hermite interpolation. The polynomial

$$\begin{aligned} s_i(x) &= a_i + b_i u + c_i u^2 + d_i u^3 \ , \\ u &= \frac{x - x_{i-1}}{h_i}, \quad h_i = x_i - x_{i-1} \end{aligned} \tag{5.11.2}$$

is determined by the values at x_{i-1} and x_i of s and s'. From (5.7.2) and (5.11.1) we get

$$\begin{aligned} a_i &= f_{i-1} \ , \\ b_i &= h_i s'_{i-1} \ , \\ c_i &= 3(f_i - f_{i-1}) - h_i(2s'_{i-1} + s'_i) \ , \\ d_i &= 2(f_{i-1} - f_i) + h_i(s'_{i-1} + s'_i) \ . \end{aligned} \tag{5.11.3}$$

The values of the $\{s'_j\}$ are determined by the continuity of $s''(x)$ and the two extra conditions: From (5.11.2) we get

$$s''_i(x) = \frac{2c_i + 6d_i u}{h_i^2} \ ,$$

and the condition $s''_i(x_i) = s''_{i+1}(x_i)$ at the interior knots is equivalent to

$$\frac{2c_i + 6d_i}{h_i^2} = \frac{2c_{i+1}}{h_{i+1}^2} \ , \quad i = 1, 2, \ldots, n-1 \ .$$

We insert the values from (5.11.3),

$$\frac{-6(f_i - f_{i-1}) + h_i(2s'_{i-1} + 4s'_i)}{h_i^2} \ = \ \frac{6(f_{i+1} - f_i) - h_{i+1}(4s'_i + 2s'_{i+1})}{h_{i+1}^2} \ ,$$

and after multiplication by $\frac{1}{2}h_i h_{i+1}$ and reordering we get

$$h_{i+1}s'_{i-1} + 2(h_i + h_{i+1})s'_i + h_i s'_{i+1} = r_i, \quad i = 1, 2, \ldots, n-1 \ , \tag{5.11.4a}$$

where

$$r_i = 3\left(h_{i+1}\frac{f_i - f_{i-1}}{h_i} + h_i\frac{f_{i+1} - f_i}{h_{i+1}}\right) \ . \tag{5.11.4b}$$

This is a system of $n-1$ linear equations in the $n+1$ unknowns $\{s'_j\}_{j=0}^n$.

A natural spline must further satisfy the two equations

$$s''(x_0) = \frac{2c_1}{h_1^2} = \frac{2}{h_1}\left(3\frac{f_1 - f_0}{h_1} - 2s'_0 - s'_1\right) = 0 \ ,$$

$$s''(x_n) = \frac{2(c_n + 3d_n)}{h_n^2} = \frac{2}{h_n}\left(-3\frac{f_n - f_{n-1}}{h_n} + s'_{n-1} + 2s'_n\right) = 0 \ ,$$

or

$$\begin{aligned} 2s'_0 + s'_1 &= r_0, \qquad r_0 = 3(f_1 - f_0)/h_1 \ , \\ s'_{n-1} + 2s'_n &= r_n, \qquad r_n = 3(f_n - f_{n-1})/h_n \ . \end{aligned} \tag{5.11.5}$$

When we combine these two equations with (5.11.4), we have $n+1$ linear equations in the $n+1$ unknowns $\{s'_j\}_{j=0}^n$. We can write the system in matrix form,

$$\begin{pmatrix} 2 & 1 & & & \\ h_2 & 2(h_1{+}h_2) & h_1 & & \\ & \ddots & \ddots & \ddots & \\ & & h_n & 2(h_{n-1}{+}h_n) & h_{n-1} \\ & & & 1 & 2 \end{pmatrix} \begin{pmatrix} s'_0 \\ s'_1 \\ \vdots \\ s'_{n-1} \\ s'_n \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_{n-1} \\ r_n \end{pmatrix} . \tag{5.11.6}$$

If we know the slopes at the endpoints, then (5.11.5) is replaced by

$$s'_0 = r_0 = f'(a) \ ,$$
$$s'_n = r_n = f'(b) \ .$$

The modifications of the first and last rows of the matrix in (5.11.6) are straightforward.

In both cases the matrix is tridiagonal and diagonally dominant, cf Chapter 8. It can be shown that such a matrix is nonsingular, which means that the system has a unique solution.

Thus, we have proved Theorem 5.11.2 in the cases where the extra conditions are chosen as the natural spline or correct boundary conditions. It can be shown that also the "not–a–knot" and the periodic boundary conditions lead to a well-defined set $\{s'_j\}_{j=0}^n$, and thereby a unique interpolating cubic spline.

Example. The solution to the system (5.11.6) can be found by Gaussian elimination without pivoting, and the amount of work involved in the solution is approximately $8n$ flops, cf Section 8.8.

This property is, however, not exploited in the following MATLAB function `splint1`, which can be used to compute an interpolating cubic spline. It returns the coefficients in the piecewise polynomial (5.11.2). There is a choice between "natural" and correct boundary conditions.

```
function  p = splint1(x,f,df)
% Compute coefficients for interpolating cubic spline
% Call    p = splint1(x,f)     % Natural spline
% or      p = splint1(x,f,df)  % Correct boundary conds.
% Input
% x:  Vector with knots.  Length n+1.
% f:  Vector with nodal function values.  Length n+1.
% df: If present, then a vector with endpoint derivatives,
%     df(1) = f'(x(1)),  df(2) = f'(x(n+1))
% Output
% p : n*4 array with p(i,:) = [ai bi ci di].
```

```
n = length(x) - 1;     % number of knot intervals
h = diff(x);           % n-vector with knot spacings
v = diff(f)./h;        % n-vector with divided differences
% Set up matrix A and right hand side r
A = zeros(n+1,n+1);   r = zeros(n+1,1);
for  i = 1 : n-1
  A(i+1,i:i+2) = [h(i+1)  2*(h(i)+h(i+1))  h(i)];
  r(i+1) = 3*(h(i+1)*v(i) + h(i)*v(i+1));
end
if  nargin == 3                 % Correct boundary conds.
  A(1,1) = 1;  r(1) = df(1);
  A(n+1,n+1) = 1;  r(n+1) = df(2);
else                            % Natural spline.
  A(1,1:2) = [2 1];  r(1) = 3*v(1);
  A(n+1,n:n+1) = [1 2];  r(n+1) = 3*v(n);
end
ds = A\r;                       % Solve  A*ds = r
% Compute coefficients
p = zeros(n,4);
for  i = 1:n
  p(i,1) = f(i);
  p(i,2) = h(i)*ds(i);
  p(i,3) = 3*(f(i+1) - f(i)) - h(i)*(2*ds(i) + ds(i+1));
  p(i,4) = 2*(f(i) - f(i+1)) + h(i)*(ds(i) + ds(i+1));
end
```

The function `splint2` can be used to compute values of the spline for multiple arguments, given in a vector `t`. The evaluation is made by Horner's rule.

```
function  s = splint2(x,p,t)
% Compute values of cubic spline with knots x and coefficients
% p, see SPLINT1.
if  any((t < x(1)) | (t > x(end)))
  error('arguments must be in knot range')
end
s = zeros(size(t));              % vector of same type as  t
for  i = 1 : length(x)-1         % run through knot intervals
  k = find((x(i) <= t) & (t <= x(i+1)));
  if  ~isempty(k)                      % arguments in interval
    u = (t(k) - x(i))/(x(i+1) - x(i));  % Normalized arguments
    s(k) = p(i,1) + u.*(p(i,2) + u.*(p(i,3) + u*p(i,4)));
  end
end
```

We shall use these two functions to compute two cubic splines that interpolate the Runge function from page 118, $f(x) = 1/(1+25x^2)$, in 11 equidistant

points on $[-1, 1]$. The two cubic splines are respectively a natural spline and a spline with correct boundary conditions, $f'(x) = -50x/(1 + 25x^2)^2$.

```
x = linspace(-1,1,11);  fx = 1./(1 + 25*x.^2);
pn = splint1(x,fx);                 % natural spline
dfe = [50 -50]/26^2;                % endpoint slopes
pc = splint1(x,fx,dfe);             % correct bound. conds.
% Error curves
t = linspace(-1,1,201);  ft = 1./(1 + 25*t.^2);
plot(t,splint2(x,pn,t)-ft,'--', ...
   t,splint2(x,pc,t)-ft,'-', x,0*x,'o')
```

The resulting error plot is shown in Figure 5.12.

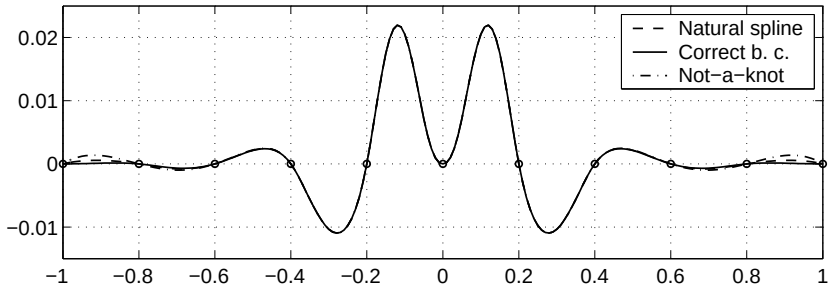


Figure 5.12. *Error function* $s(x) - 1/(1 + 25x^2)$.

In the plot we have added the error curve corresponding to the cubic spline given by the standard MATLAB function `spline`

```
>> s = spline(x,fx,t);
```

The vector `s` contains values of the interpolating cubic spline, computed with the "not-a-knot" condition. It should be mentioned that if we had used the command

```
>> s = spline(x,[dfe(1) fx dfe(2)],t);
```

then `s` would contain values of the interpolating cubic spline, computed with the correct boundary conditions.

The figure shows that the correct boundary conditions give the best approximation, but except for the two extreme knot intervals at both ends, it is not possible to see the difference between the three cubic splines. ∎

Finally, we give some general results about properties of interpolating cubic spline functions. First, we look at the similarity with physical splines:

Theorem 5.11.3. Among all functions g that are twice continuously differentiable on $[a,b]$, and that interpolate f in the points $a = x_0 < x_1 < \cdots < x_n = b$, the interpolating natural cubic spline minimizes the integral

$$\int_a^b \big(g''(x)\big)^2 dx \ .$$

Proof. Let s denote the interpolating natural cubic spline, and consider

$$\begin{aligned}\int_a^b \big(g''(x) - s''(x)\big)^2 dx = \int_a^b \big(g''(x)\big)^2 dx - \int_a^b \big(s''(x)\big)^2 dx \\ -2\int_a^b \big(g''(x) - s''(x)\big)s''(x)\, dx \ .\end{aligned} \tag{5.11.7}$$

We will show that the last integral is zero. To do that, we look at the contribution from the ith interval and use partial integration.

$$\begin{aligned} I_i &= \int_{x_{i-1}}^{x_i} \big(g''(x) - s''(x)\big)s''(x)\, dx \\ &= \big[(g'(x) - s'(x))s''(x)\big]_{x_{i-1}}^{x_i} - \int_{x_{i-1}}^{x_i} \big(g'(x) - s'(x)\big)s^{(3)}(x)\, dx \\ &= \big[(g'(x) - s'(x))s''(x) - 6d_i(g(x) - s(x))\big]_{x_{i-1}}^{x_i} \\ &= \big(g'(x_i) - s'(x_i)\big)s''(x_i) - \big(g'(x_{i-1}) - s'(x_{i-1})\big)s''(x_{i-1}) \ . \end{aligned}$$

During the reformulation we used that $s^{(3)}(x)$ is equal to the constant $6d_i$ for $x_{i-1} < x < x_i$, and that

$$g(x_j) - s(x_j) = f(x_j) - f(x_j) = 0, \quad j = i{-}1, i \ ,$$

because of the interpolation property. Thus, we have

$$\begin{aligned} \int_a^b \big(g''(x) - s''(x)\big)s''(x)\, dx &= I_1 + I_2 + \cdots + I_n \\ &= \big(g'(x_1) - s'(x_1)\big)s''(x_1) - \big(g'(x_0) - s'(x_0)\big)s''(x_0) \\ &\quad + \big(g'(x_2) - s'(x_2)\big)s''(x_2) - \big(g'(x_1) - s'(x_1)\big)s''(x_1) \\ &\quad + \cdots \\ &\quad + \big(g'(x_n) - s'(x_n)\big)s''(x_n) - \big(g'(x_{n-1}) - s'(x_{n-1})\big)s''(x_{n-1}) \\ &= \big(g'(x_n) - s'(x_n)\big)s''(x_n) - \big(g'(x_0) - s'(x_0)\big)s''(x_0) = 0 \ , \end{aligned}$$

since $s''(x_0) = s''(x_n) = 0$. Now, we insert this result in (5.11.7) and reorder the terms:

$$\int_a^b \big(g''(x)\big)^2 dx = \int_a^b \big(s''(x)\big)^2 dx + \int_a^b \big(g''(x) - s''(x)\big)^2 dx$$
$$\geq \int_a^b \big(s''(x)\big)^2 dx \ .$$

The minimum is obtained when $g''(x) = s''(x)$ for $a \leq x \leq b$, ie for $g(x) = s(x) + A + Bx$. However, the interpolation conditions imply that $A = B = 0$, so that $g(x) = s(x)$. □

From the proof it follows that if g is required not only to interpolate the given points, but also to satisfy the boundary conditions $g'(a) = f'(a)$ and $g'(b) = f'(b)$, then the same integral is minimized, when $g(x) = s(x)$, the interpolating cubic spline with correct boundary conditions.

Theorem 5.11.3 implies that an interpolating cubic splines cannot have large oscillations. If it had, then s' would take large positive and negative values, and according to the mean value theorem also $|s''|$ would be large, and the integral of $(s''(x))^2$ would not be small.

The local truncation error can be estimated by Theorem 5.7.1 with $E_i = 0$,

$$\max_{x_{i-1} \leq x \leq x_i} |f(x) - s_i(x)| \leq \tfrac{1}{384} M_i h_i^4 + \tfrac{1}{4} E_i' h_i \ ,$$

where

$$M_i = \max_{x_{i-1} \leq x \leq x_i} |f^{(4)}(x)| \ , \quad E_i' = \max_{j=i-1,i} |f'(x_j) - s'(x_j)| \ .$$

Ignoring the term with E_i' we get the following recommendation,

Rule of thumb no. 1. The knots should be closely spaced in regions where $|f^{(4)}(x)|$ is large, while the knot spacing may be larger in regions, where $|f^{(4)}(x)|$ is small, ie where f can be well approximated by a cubic polynomial.

This is analogous to the recommendation about linear splines given in connection with (5.10.1). It can be shown that undesirable oscillations may be induced if the knot spacing varies wildly, and therefore we also give another recommendation:

Rule of thumb no. 2. The knots should preferably be distributed so that $\frac{1}{2} \leq h_{i+1}/h_i \leq 2$.

Example. As in the previous example we interpolate $f(x) = 1/(1+25x^2)$ in $[-1,1]$ by a cubic spline with 11 knots. Figure 5.13 shows the error when we use correct boundary conditions and the interior knots distributed according to the first rule of thumb.

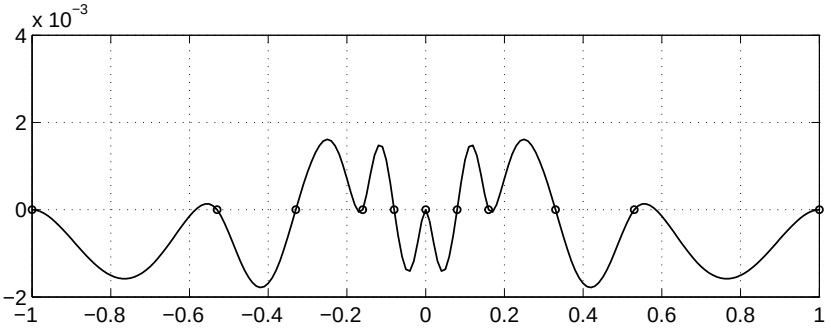


Figure 5.13. *Error function $s(x) - 1/(1+25x^2)$. Non-equidistant knots.*

As compared with equidistant knots, the maximum error is reduced from approximately 0.022 to approximately 0.0018. ∎

The final theorem in this section gives information about the global approximation properties of cubic splines. The proof can be found in the paper by Beatson given in the references.

Theorem 5.11.4. Let the function f be four times continuously differentiable in $[a,b]$, and let s be the cubic spline that interpolates f at the points

$$a = x_0 < x_1 < \cdots < x_n = b \ ,$$

with correct boundary conditions. Then

$$\max_{a\le x\le b} |s^{(r)}(x) - f^{(r)}(x)| < K_r(\beta) M h^{4-r}, \quad r = 0,1,2,3 \ ,$$

where

$$h = \max_i \ h_i, \quad \beta = \frac{h}{\min_i \ h_i}, \quad M = \max_{a\le x\le b} |f^{(4)}(x)| \ ,$$

and

$$K_0(\beta) = \tfrac{5}{384} \ , \qquad K_1(\beta) = \tfrac{1}{216}(9+\sqrt{3}) \ ,$$

$$K_2(\beta) = \tfrac{1}{12}(1+3\beta) \ , \quad K_3(\beta) = \tfrac{1}{2}(1+\beta^2) \ .$$

The theorem shows that, as the largest knot spacing tends to zero, the interpolating spline and its first three derivatives converge uniformly to f and its derivatives. In particular, we have $|f(x) - s(x)| = O(h^4)$ when we use correct boundary conditions. If we use a natural spline or the not–a–knot condition, we will get $|f(x) - s(x)| = O(h^2)$ for x close to the endpoints x_0 and x_n.

5.12. Cubic B-Splines

In many programs that use cubic splines, the spline s is expressed as a linear combination of basis splines B_{i3}, so-called *cubic B-splines*,

$$s(x) = \sum_i c_i B_{i3}(x) \ .$$

In this section we shall give an introduction to this formulation.

In Section 5.10 we saw that a linear spline can be expressed as a linear combination of linear basis splines $l_i(x)$. Each of these is a linear spline, which is nonzero only in two consecutive knot intervals. We say that l_i has its *support* in the interval $[x_{i-1}, x_{i+1}]$.

Similarly, each B_{i3} is a cubic spline, which is nonzero only in a small number of consecutive knot intervals. How many intervals do we need? To answer that question, let $B_{i3}(x)$ be nonzero in the open interval $]x_i, x_{i+q}[$. There are $4q$ parameters in the piecewise cubic polynomials. They must be determined such that B_{i3}, B'_{i3} and B''_{i3} are continuous across the interior knots $x_{i+1}, \ldots, x_{i+q-1}$, and such that B_{i3} and its first two derivatives also are continuous across the knots x_i and x_{i+q} :

$$B_{i3}(x_k) = B'_{i3}(x_k) = B''_{i3}(x_k) = 0, \quad k = i, i+q \ .$$

Thus, we have $3(q+1)$ conditions on the $4q$ parameters, and in order to get a nontrivial solution we must have

$$4q > 3(q+1) \ .$$

The smallest integer value of q that satisfies this demand is $q = 4$. This leaves one degree of freedom, which is used to normalize the basis splines. The normalization is made so that for every $x \in [x_0, x_n]$ the sum of all $B_{i3}(x)$ is equal to one.

In conclusion, $B_{i3}(x)$ is nonzero only in the open interval $]x_i, x_{i+4}[$ (ie B_{i3} has its support in $[x_i, x_{i+4}]$). In other words: For a given x in $[x_i, x_{i+1}]$ there are at most[5)] four nonzero basis splines, $B_{i-3,3}(x)$, $B_{i-2,3}(x)$, $B_{i-1,3}(x)$ and $B_{i,3}(x)$. Thus, we can write

$$s(x) = \sum_{i=-3}^{n-1} c_i B_{i3}(x), \quad x_0 \le x \le x_n \ .$$

The number of terms, $n+3$, is recognized as the number of degrees of freedom of a cubic spline with knots $x_0, x_1, \ldots, x_n$. The B-splines are illustrated in Figure 5.14.

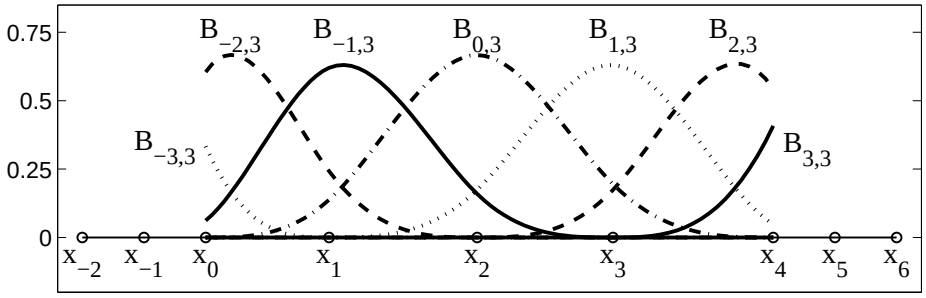


Figure 5.14. *Cubic B-splines. Knots indicated by circles.* $n = 4$.

The B-splines can be computed recursively:

$$B_{i,0} = \begin{cases} 1, & x_i \le x < x_{i+1} \ , \\ 0, & \text{otherwise,} \end{cases} \tag{5.12.1a}$$

and for $r = 1, 2, 3$:

$$B_{i,r}(x) = \frac{x - x_i}{x_{i+r} - x_i} B_{i,r-1}(x) + \frac{x_{i+r+1} - x}{x_{i+r+1} - x_{i+1}} B_{i+1,r-1}(x) \ . \tag{5.12.1b}$$

Especially, for $r = 1$ we get

$$B_{i,1}(x) = \begin{cases} 0, & x < x_i \ , \\ (x - x_i)/(x_{i+1} - x_i), & x_i \le x < x_{i+1} \ , \\ (x_{i+2} - x)/(x_{i+2} - x_{i+1}), & x_{i+1} \le x < x_{i+2} \ , \\ 0, & x \ge x_{i+2} \ . \end{cases}$$

We recognize this as the linear basis spline $l_{i+1}(x)$ from Section 5.10.

[5)] At a knot there are only three nonzero B-splines.

For a given $x \in [x_i, x_{i+1}[$ the only nonzero basis spline of degree zero is $B_{i,0}(x)$. According to (5.12.1b) this leads to nonzero $B_{i-1,1}(x)$ and $B_{i,1}(x)$, etc. The computation of the four nonzero $B_{j,3}(x)$ is illustrated in Figure 5.15.

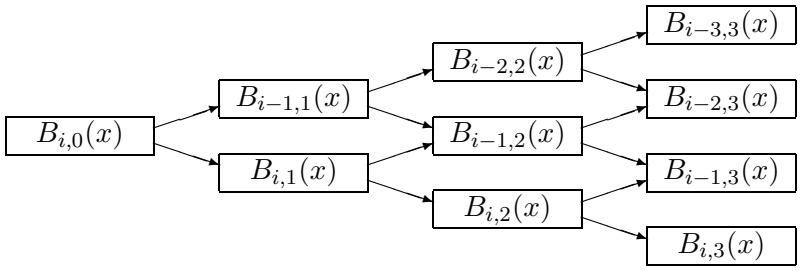


Figure 5.15. *Nonzero values in the recurrence for cubic B-splines when* $x \in [x_i, x_{i+1}[$.

In the recurrence (5.12.1b) we only need to include the terms with $B_{j,r-1}(x) \neq 0$, eg

$$B_{i-3,3}(x) = \frac{x_{i+1} - x}{x_{i+1} - x_{i-2}} B_{i-2,2}(x) \ .$$

When we take that into account, we can see that the computation for $x \in [x_i, x_{i+1}[$ involves knots $x_{i-2}, \ldots, x_{i+3}$. Since $0 \leq i \leq n-1$, this shows we need two extra knots at both ends. They must satisfy

$$x_{-2} \leq x_{-1} \leq x_0, \qquad x_{n+2} \geq x_{n+1} \geq x_n \ .$$

Often, the extra knots are chosen so that they coincide with x_0 and x_n, respectively.

Suppose that we want to interpolate m points $\{(t_i, f_i)\}_{i=1}^m$ by a cubic spline. When we express this in terms of B-splines, we no longer have to use the knots given by the values of the first coordinate. Let $n = m-3$, choose knots $x_0, x_1, \ldots, x_n$ such that all $t_i \in [x_0, x_n]$, and determine the spline by the conditions

$$\sum_{j=-3}^{m-4} B_{j3}(t_i) c_j = f_i, \quad i = 1, 2, \ldots, m \ .$$

This is a system of m linear equations in the m unknown coefficients $\{c_j\}_{j=-3}^{m-4}$. Each equation involves only four unknowns, so the matrix of the system is banded, cf Section 8.8. It can be shown that the matrix is nonsingular if the so-called *Schoenberg-Whitney* conditions

$$t_i < x_i < t_{i+4}, \quad i = 1, 2, \ldots, m-4$$

are satisfied. Especially, there must be at least one data point in each of the two extreme knot intervals $[x_0, x_1[$ and $]x_{n-1}, x_n]$.

Example. Figure 5.16 illustrates the same problem as the two previous examples: interpolate $f(x) = 1/(1 + 25x^2)$ in $[-1, 1]$ by a cubic spline with 11 knots.

We take the same knots as in Figure 5.13, but now the spline is determined by interpolating 13 equidistant points, $t_i = -1 + (i-1)/6$, $i = 1, 2, \ldots, 13$.

The approximation is almost as good as in Figure 5.13, where we used nodal values of the function, supplied with correct boundary conditions.

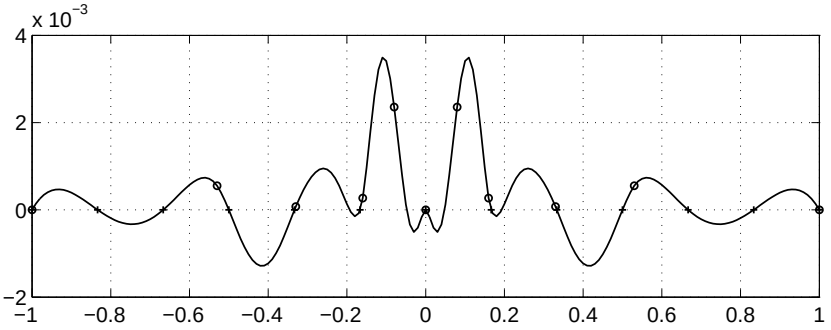


Figure 5.16. *Error function* $s(x) - 1/(1 + 25x^2)$.
Knots and data points marked by o *and* $+$ *, respectively.*

■

Exercises

E1. We want to compute $f(a) = \sqrt{a}$, and we have very high requirements concerning speed.

(a) One possible method is to interpolate linearly in an equidistant table. Which table size is needed if we require that $|R_{\mathrm{XF}} + R_{\mathrm{T}}|$ shall be smaller than 2μ? The computer is using the floating point system $(2, 23, -126, 127)$.

(b) Another method is to perform one iteration with Newton-Raphson's method applied to the equation $f(x) = x^2 - a = 0$.
The initial approximation x_0 is taken from a table (see Section 4.7). Which table size is needed if we require that the error after one iteration is smaller than 2μ?

(c) The computational work is approximately the same in (a) and (b). Which method requires the smallest table?

E2. We know that P is a polynomial of degree ≤ 5, and know the following values

x	-2	-1	0	1	2	3
$P(x)$	-5	1	1	1	7	25

.

What is the degree of P ?

E3. The polynomial

$$p(x) = 2 - (x+1) + x(x+1) - 2x(x+1)(x-1)$$

interpolates the first four points in the table

x	-1	0	1	2	3
$f(x)$	2	1	2	-7	10

Use Newton's interpolation polynomial to determine the term that should be added to $p(x)$, so that the resulting polynomial interpolates all the points in the table.

E4. Let $f(x) = x^m$, where m is a natural number. Show that

$$f[x_0, x_1, \ldots, x_n] = \begin{cases} 1, & n = m \ , \\ 0, & n > m \ . \end{cases}$$

E5. The two expressions

$$f(x_0) + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] \\ +(x - x_0)(x - x_1)(x - x_2)f[x_0, x_1, x_2, x_3]$$

and

$$f(x_1) + (x - x_1)f[x_1, x_3] + (x - x_1)(x - x_3)f[x_1, x_3, x_2] \\ +(x - x_1)(x - x_3)(x - x_2)f[x_1, x_3, x_2, x_0]$$

represent the same polynomial. Explain why.

E6. The following table with correctly rounded values is given

x	$\sin x$	$\cos x$	$\cot x$
0.001	0.001000	1.000000	1000.0
0.002	0.002000	0.999998	499.999
0.003	0.003000	0.999996	333.332
0.004	0.004000	0.999992	249.999
0.005	0.005000	0.999988	199.998

Compute cot(0.0015) as accurately as possible

(a) by interpolation in the table for $\cot x$,

(b) by interpolation in the tables for $\cos x$ and $\sin x$.

(c) Estimate the error in (b).

(d) Explain the difference between the results in (a) and (b).

The argument 0.0015 is assumed to be exact.

E7. Derive a method for estimating $\int_a^b f(x)\,dx$ by interpolating f by a linear spline with the knots $x_i = a + \frac{i}{n}(b-a)$, $i = 0, 1, \ldots, n$.

E8. Show that the interpolating linear spline with knots $x_0, x_1, \ldots, x_n$ is the function that minimizes

$$\int_{x_0}^{x_n} \big(g'(x)\big)^2 dx$$

among all functions g such that $g(x_i) = f_i$, $i = 0, 1, \ldots, n$, and such that the integral is bounded.

E9. For $r = 1, 2, 3$ show that the B-spline $B_{ir}(x)$ has support $[x_i, x_{i+r+1}]$ and that $B_{ir}(x) > 0$ for $x_i < x < x_{i+r+1}$.

Computer Exercises

C1. On page 111 we gave a MATLAB function `intpolc` for computing the coefficients in Newton's interpolation polynomial. Compare `intpolc` with the function

```
function c = intpolc1(x,f)
m = length(x);   % number of interpolation points
for  k = 2 :  m
   f(k:m) = (f(k:m) - f(k-1)) ./ (x(k:m) - x(k-1));
end
c = f;   % return coefficients
```

Explain why the two functions give the same result.

C2. Consider the MATLAB functions `splint1` and `splint2` from pages 128 – 129. How should the output `p` from `splint1` be modified, so that `splint2(x,p,t)` returns $s'(t)$?

C3. Consider the function

$$f(x) = \frac{x}{0.25 + x^2}$$

in the interval $-2 \le x \le 2$.

(a) Approximate f by *different* interpolation polynomials in different parts of the interval. (In MATLAB you can use `polyfit`, cf the example on page 112). Determine the polynomials P, Q and R such that

$$\begin{aligned} P(-2+0.5i) &= f(-2+0.5i), \quad i = 0,1,2,3\ , \\ Q(-0.5+i/3) &= f(-0.5+i/3), \quad i = 0,1,2,3\ , \\ R(0.5+0.5i) &= f(0.5+0.5i), \quad i = 0,1,2,3\ , \end{aligned}$$

and let

$$p(x) = \begin{cases} P(x), & -2 \le x \le -0.5 \\ Q(x), & -0.5 \le x \le 0.5 \\ R(x), & 0.5 \le x \le 2 \end{cases}$$

Plot f and p in the same figure. Plot another figure with the error function $p - f$. Is this a good approximation?

(b) Use the MATLAB function `spline` to interpolate f in the points $x_i = -2 + 0.5i$, $i = 0, 1, \ldots, 8$, with not–a–knot conditions, cf page 130. Plot f and the spline s in the same figure, and the error function $s - f$ in another figure.

(c) Repeat (b) with the knot spacing changed to 1/3.

It is obvious that the error is largest close to $x = 0$, where the function varies fast. One way of getting a better distribution of the knots is to use the fact that on the interval $[-2, 2]$ the function f has the parametric representation

$$x = \tfrac{1}{2}\cot\theta, \quad y = \sin 2\theta, \quad \alpha \le \theta \le \pi - \alpha\ ,$$

where $\frac{1}{2}\cot\alpha = 2$.

(d) Repeat (b) with the knots given by

$$x_i = \tfrac{1}{2}\cot\theta_i, \quad \theta_i = \alpha + i\frac{\pi - 2\alpha}{12},\ i = 0, 1, \ldots, 12\ .$$

Do we now get a good approximation for x close to 0? Why is the error large for x close to the endpoints of the interval?

(e) Repeat (d) with the not–a–knot conditions changed to the correct boundary conditions, cf page 130. Do we now get a good approximation in the entire interval?

C4. We consider an algorithm for cubic spline interpolation of a function f in the interval $[a, b]$. The spline should be computed with correct boundary conditions, and the knots x_j should be inserted adaptively, until a specified accuracy τ is obtained.

Ideally, we want

$$\max_{a \le t \le b} |f(t) - s(t)| \ \le \ \tau \ ,$$

and approximate this by

$$\max_{0 \le i \le m} |f_i - s(t_i)| \ \le \ \tau \ ,$$

where the "test points" t_i are equidistant in $[a, b]$ and $f_i = f(t_i)$.

The algorithm is specified as follows.

Initialize:
$t_i = a + \frac{i}{m}(b-a), \quad f_i = f(t_i), \quad i = 0, 1, \ldots, m$
$n = 2, \quad x_j = a + \frac{j}{2}(b-a), \quad j = 0, 1, 2$
Iterate:
repeat
Compute s
Find E_n and k : $\quad E_n = |f_k - s(t_k)| = \max_i |f_i - s(t_i)|$
if $E_n \le \tau$ **then** STOP
else $n := n{+}1$; insert t_k as a new knot
end

(a) Implement the algorithm in MATLAB with appropriate illustrations of its performance, eg plot the error $s-f$ in each step of the iteration. The MATLAB function `spline` can be used to compute s, cf page 130.

(b) Test your implementation with $m = 200$ and

$$f(x) = \frac{x}{0.25 + x^2} \ , \qquad [a, b] = [0, 2] \ .$$

References

Interpolation is one of the oldest areas of research in numerical analysis, and has major contributions from a number of people, whose names are still connected to special formulas or types of interpolation, see Short Biographies at the end of the book.

There is an extensive classical theory, where different representations of interpolating polynomials are derived using operator calculus, see eg

C.-E. Fröberg, *Numerical Mathematics, Theory and Computer Applications*, The Benjamin/Cummings Publishing Company, Menlo Park, 1985.

The theoretical aspects of splines are thoroughly discussed in

> L.L. Schumaker, *Spline Functions: Basic Theory*, John Wiley, New York, 2001.

Practically oriented presentations of splines and related subjects are given in the following books. In computer graphics Neville's method is often referred to as the de Casteljau algorithm.

> C. de Boor, *A Practical Guide to Splines, Revised Edition*, Springer Verlag, New York, 2001.
>
> P. Dierckx, *Curve and Surface Fitting wIth Splines*, Oxford University Press, 1995.
>
> G. Farin, *Curves and Surfaces for Computer Aided Geometric Design*, Fourth ed., Academic Press, San Diego. 1997.

The proof of Theorem 5.11.4 is given in

> R.K. Beatson, *On the Convergence of some Cubic Spline Interpolation Schemes*, SIAM J. Num. Anal. **23**, pp 903–912, 1986.

Chapter 6

Differentiation and Richardson Extrapolation

6.1. Introduction

Suppose that a function is known only at some discrete points, as in Figure 6.1, and that we want to estimate the derivative of the function.

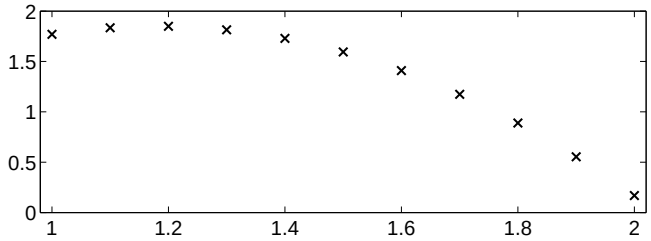


Figure 6.1. *Only the function values marked by × are known.*

If we want to compute the derivative for many values of the argument, or if the arguments of the known function values are not equidistant, then we recommend to compute an interpolating cubic spline and differentiate that. In this chapter, however, we assume that the function values are known at equidistant points, and that approximations to the derivative are only required at a small number of points. Then it is simpler to interpolate the function by a polynomial of low degree, and differentiate that polynomial. We shall see that this is equivalent to approximating the derivative by a difference quotient. Actually, we already used this technique in Chapter 5 when we estimated the truncation error by the first

neglected term; in Theorem 5.4.3 the kth derivative $f^{(k)}$ was estimated by means of a divided difference.

Numerical differentiation can also be useful in cases where we know a formula for the function. If the derivative involves a large computational work, it may be preferable instead to compute an approximation with the desired accuracy.

In Chapter 10 we shall use formulas for numerical differentiation to derive methods for numerical solution of differential equations.

When we have discussed different difference approximations, we shall use one of them to illustrate Richardson extrapolation. This is a powerful technique, that can be used for estimating the truncation error and for reducing it. In the next chapter we shall use this technique in connection with numerical methods for integration.

6.2. Difference Approximations of Derivatives

Assume that the function f is known at the points $x-h$, x and $x+h$. We want to compute an approximation of $f'(x)$, ie the slope of the tangent of the curve $y = f(x)$ at the point x. Figure 6.2 shows three secants, whose slopes can be used as approximations of $f'(x)$.

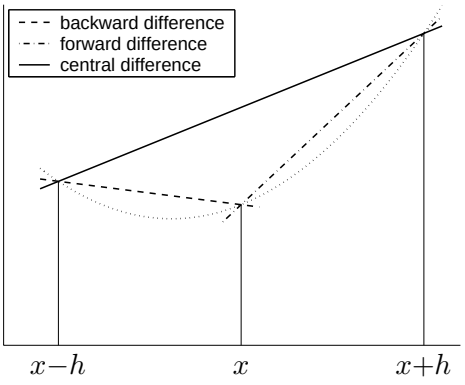


Figure 6.2. *Different approximations of* $f'(x)$.

In the figure h is large in order to enhance the difference between the three approximations. In practice we use a much smaller h-value in order to get a good approximation.

If f is approximated by the straight line through the points $(x, f(x))$ and $(x+h, f(x+h))$, we get the so-called *forward difference* approximation $D_+(h)$,

$$f'(x) \simeq D_+(h) = \frac{f(x+h) - f(x)}{h} \ .$$

Similarly, the *backward difference* approximation $D_-(h)$ is the slope of the straight line through the points $(x, f(x))$ and $(x-h, f(x-h))$,

$$f'(x) \simeq D_-(h) = \frac{f(x) - f(x-h)}{h} \ .$$

Intuitively, we get a better approximation of $f'(x)$ if we use function values at points that are symmetric around x. The *central difference* approximation $D_0(h)$ is the slope of the straight line through the points $(x-h, f(x-h))$ and $(x+h, f(x+h))$,

$$f'(x) \simeq D_0(h) = \frac{f(x+h) - f(x-h)}{2h} \ .$$

The same idea can be used to get approximations of higher derivatives. For instance, if f is approximated by a second degree polynomial through the three points $(x-h, f(x-h))$, $(x, f(x))$ and $(x+h, f(x+h))$, and this polynomial is differentiated twice, one gets

$$f''(x) \simeq \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \ . \tag{6.2.1}$$

6.3. The Error in Difference Approximations

When the derivative of a function is approximated by a difference quotient, there is a *truncation error*, R_{T}. This error can be estimated using a Taylor expansion of the function.

Let f be a twice continuously differentiable function, whose derivative is approximated by a forward difference. Then

$$\begin{aligned} R_{\mathrm{T}} = D_+(h) - f'(x) &= \frac{1}{h}\big(f(x+h) - f(x)\big) - f'(x) \\ &= \frac{1}{h}\big(f(x) + hf'(x) + \tfrac{1}{2}h^2 f''(\xi) - f(x)\big) - f'(x) \\ &= \tfrac{1}{2}hf''(\xi) \ , \end{aligned}$$

where ξ is a point in the open interval $]x, x+h[$. We know neither ξ nor f'', but we see that the truncation error is[1] $O(h)$ as $h \to 0$. If f'' is almost constant for arguments close to x, then R_{T} will be approximately halved if h is halved.

If we keep more terms in the Taylor expansion of the function, then we find[2]

$$\begin{aligned} R_{\mathrm{T}} &= D_+(h) - f'(x) \\ &= \tfrac{1}{2} f''(x)\, h + \tfrac{1}{3!} f^{(3)}(x)\, h^2 + \tfrac{1}{4!} f^{(4)}(x)\, h^3 + \cdots \\ &= a_1 h + a_2 h^2 + a_3 h^3 + \cdots \ . \end{aligned} \tag{6.3.1}$$

In the next section we shall see that we can exploit the knowledge that R_{T} has this form, even though we do not know the coefficients $a_k = f^{(k+1)}(x)/(k+1)!$. Also, if $a_1 \neq 0$ and none of the $|a_k/a_1|$ is very large, then the first term dominates (ie $R_{\mathrm{T}} \simeq a_1 h$) when h is sufficiently small.

Similarly, it can be shown that

$$D_0(h) - f'(x) = b_1 h^2 + b_2 h^4 + b_3 h^6 + \cdots \ . \tag{6.3.2}$$

If $b_1 \neq 0$ and none of the $|b_k/b_1|$ is very large, then $R_{\mathrm{T}} \simeq b_1 h^2$ when h is sufficiently small. Also note that except if $|a_1| \ll |b_1|$, we can expect to get a more accurate approximation by using $D_0(h)$ instead of $D_+(h)$.

The central difference approximation to the second derivative given in (6.2.1) also has a $O(h^2)$ truncation error,

$$\frac{f(x-h) - 2f(x) + f(x+h)}{h^2} - f''(x) = c_1 h^2 + c_2 h^4 + c_3 h^6 + \cdots \ .$$

Example. We will study the truncation error in the case $f(x) = e^x$ (with $f'(x) = e^x$). Below we give a MATLAB script that computes the forward and central difference approximations for $x = 1$ and $h = 0.4, 0.2, ..., 0.025$.

```
e = exp(1);  h = 0.4;
for  i = 1 : 5
  fd(i) = (exp(1+h) - e)/h;
  cd(i) = (exp(1+h) - exp(1-h))/(2*h);
  h = h/2;
end
```

[1] The "big O" concept is discussed at the end of Section 1.1.

[2] We assume that f is sufficiently many times differentiable in a neighbourhood of x.

The table gives the computed results and the errors

h	$D_+(h)$	$D_+(h)-e$	$D_0(h)$	$D_0(h)-e$
0.400	3.3422953	0.6240	2.7913515	0.07307
0.200	3.0091755	0.2909	2.7364400	0.01816
0.100	2.8588420	0.1406	2.7228146	0.00453
0.050	2.7873858	0.0691	2.7194146	0.00113
0.025	2.7525453	0.0343	2.7185650	0.00028

We see that when h is halved, the errors $D_+(h)-e$ are approximately halved; h is so large, that the contribution from $a_2h^2+a_3h^3+\cdots$ cannot be ignored, but as h decreases, we get closer to the asymptotic relation $D_+(\frac{1}{2}h)-e=\frac{1}{2}(D_+(h)-e)$.

The truncation error for the central difference approximation is much smaller, and also for the largest h-values we have almost equality in the relation

$$D_0(\tfrac{1}{2}h)-e \simeq (\tfrac{1}{2})^2(D_+(h)-e)=\tfrac{1}{4}(D_+(h)-e)\ .$$

The linear, respectively quadratic, behaviour of the error is illustrated in Figure 6.3.

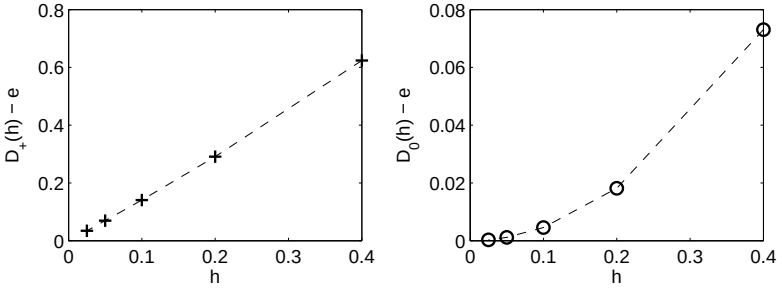


Figure 6.3. *Errors in approximation of f' by forward and central differences.*

■

Since $\lim_{h\to 0} R_{\mathrm{T}}=0$ for all our difference approximations, it is tempting to believe that it is possible to get an arbitrarily good approximation of a derivative, by taking h sufficiently small. The next example shows that this is **not** the case.

Example. Again we study the error in numerical differentiation of $f(x)=e^x$ at $x=1$. Below we give a MATLAB script that computes the errors in forward and central difference approximations with $h=0.5,0.25,\ldots,2^{-32}$.

```
e = exp(1);  h = .5;
R = zeros(32,3);          % to store h and errors
for  i = 1 : 32
```

```
  fd = (exp(1+h) - e)/h;            % forward diff.
  cd = (exp(1+h) - exp(1-h))/(2*h); % central diff.
  R(i,:) = [h  abs(fd-e)  abs(cd-e)];
  h = h/2;
end
loglog(R(:,1),R(:,2),'+', R(:,1),R(:,3),'o')
```

The results are shown in Figure 6.4. The reason why we use log-log scale is

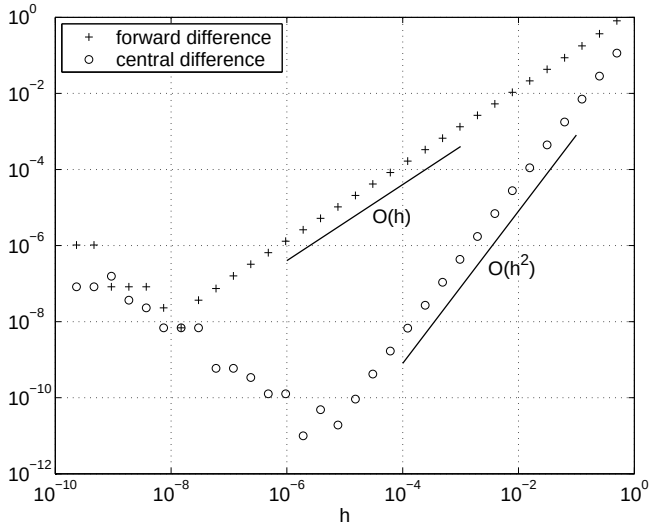


Figure 6.4. *Errors in approximation of f' by $D_+(h)$ and $D_0(h)$.*

that we expect the error $\psi(h) = |D(h) - f'(x)|$ to have the form $\psi(h) \simeq A \cdot h^p$, for some constants A and p. Then

$$\log \psi(h) \simeq \log A + p \cdot \log h \ . \tag{6.3.3}$$

Thus, in a log-log scale the points $(h, \psi(h))$ lie close to a straight line with slope p.

For the large h-values the results confirm our expectations. The forward difference approximation has errors corresponding to $p = 1$ (as indicated by the line labeled "$O(h)$"), and the errors in the central difference approximations correspond to $p = 2$.

When h gets small, we see that the errors grow when h is reduced further. Let h_{opt} denote the h-value that gives the smallest error. The value depends on the method (and also on the function f and the argument x). In Figure 6.4 we see that $h_{\text{opt}} \simeq 10^{-5}$ with $\psi(h_{\text{opt}}) \simeq 10^{-11}$ for the central difference approximation, and $h_{\text{opt}} \simeq 10^{-8}$, $\psi(h_{\text{opt}}) \simeq 10^{-8}$ for the forward approximation. ∎

We shall explain the behaviour observed in the example. There are two dominating error sources when we use a difference quotient to approximate a derivative: the truncation error R_{T} and the effect R_{XF} of errors in the function values. We already discussed the former, and now we will look at the latter in connection with the central difference approximation

$$D_0(h) = \frac{f(x+h) - f(x-h)}{2h} \ .$$

In general, the function values $f(x\pm h)$ cannot be represented exactly in the computer, and instead we get the floating point numbers $\overline{f}(x\pm h)$. We shall assume that

$$\overline{f}(x\pm h) = f(x\pm h)(1+\delta_\pm), \quad |\delta_\pm| \le K\mu \ ,$$

where μ is the unit roundoff of the computer and $K \ge 1$ is small[3)]. This means that instead of $D_0(h)$ we get the approximation

$$\begin{aligned} \overline{D}_0(h) &= \frac{\overline{f}(x+h) - \overline{f}(x-h)}{2h} \\ &= \frac{f(x+h) - f(x-h)}{2h} + \frac{\delta_+ \cdot f(x+h) - \delta_- \cdot f(x-h)}{2h} \\ &= D_0(h) + R_{\mathrm{XF}} = f'(x) + R_{\mathrm{T}} + R_{\mathrm{XF}} \ . \end{aligned}$$

In the worst case the two contributions to the numerator of R_{XF} have opposite sign, so we get the following estimate

$$|R_{\mathrm{XF}}| \le \frac{K\mu\big(|f(x+h)| + |f(x-h)|\big)}{2h} \simeq K|f(x)|\,\frac{\mu}{h} \ .$$

When we combine the result of this analysis with the expression (6.3.2) for the truncation error, we see that for small h we can expect that the total error has a form like

$$|\overline{D}_0(h) - f'(x)| \simeq Ah^2 + B\,\frac{\mu}{h} = \psi(h) \ .$$

This is illustrated in Figure 6.5. The two contributions are respectively decreasing and increasing, as $h \to 0$, and the sum has a minimum at

$$h_{\mathrm{opt}} = \sqrt[3]{\frac{\mu B}{2A}} \ . \qquad \psi(h_{\mathrm{opt}}) = 1.5\,\sqrt[3]{2A\mu^2B^2} \ .$$

We know neither A nor B, but assuming that both are of the order of magnitude 1, we get $h_{\mathrm{opt}} \sim \mu^{1/3}$ and the minimum error $\sim \mu^{2/3}$.

[3)] In MATLAB this relation is satisfied with $K = 1$ for standard functions like `sin`, `exp`, `log`, etc.

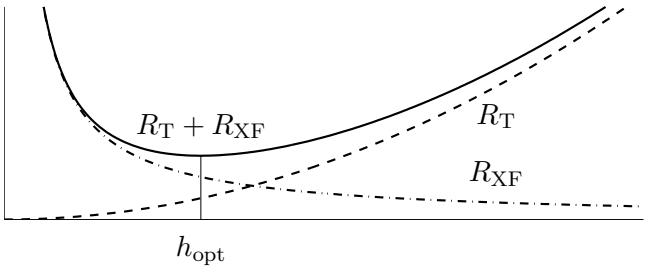


Figure 6.5. *Errors in difference approximation of f' as functions of h.*

Example. For the problem treated in the previous example, the assumptions $A \sim 1$ and $B \sim 1$ are satisfied. The computation was made with $\mu = 2^{-53} \simeq 10^{-16}$, so $\mu^{1/3} \simeq 10^{-5}$ and $\mu^{2/3} \simeq 10^{-11}$. These numbers agree with the observed values for h_{opt} and the minimum error.

A similar analysis of the forward difference approximation shows that

$$|\overline{D}_+(h) - f'(x)| \simeq A_+ h + B_+ \frac{\mu}{h} \ .$$

If both A_+ and B_+ are of the order of magnitude 1, we get $h_{\text{opt}} \sim \mu^{1/2}$ and this is also the order of magnitude of the minimum error. Again we see a good agreement with Figure 6.4.

Note that the computed errors exhibit a rugged behaviour when $h \lesssim h_{\text{opt}}$. This is because there is partial cancellation between the error contributions in R_{XF} and in $R_T + R_{XF}$. Except for this we see that the results in Figure 6.4 for $h < h_{\text{opt}}$ correspond to the slope $p = -1$ in (6.3.3). ∎

6.4. Richardson Extrapolation

Assume that a function $F(h)$ can be computed for different values of $h \neq 0$, and that we want to find the limit of the function as $h \to 0$.

This is the type of problem we have when we use a central difference approximation to estimate the derivative of a function f. We can compute

$$F(h) = \frac{f(x+h) - f(x-h)}{2h}$$

for different values of h, and we want to compute

$$f'(x) = \lim_{h \to 0} F(h) \ .$$

L.F. Richardson has shown how a good estimate of $F(0)$ can be computed if we know the behaviour of F as $h \to 0$ and have computed the value of F for two different h-values. We shall use the central difference approximation to illustrate the method. From the expression (6.3.2) for the truncation error we know that

$$F(h) = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + b_1 h^2 + O(h^4) \ .$$

We do not know the value of b_1 ($b_1 = \frac{1}{6} f^{(3)}(x)$), but we know that it is independent of h. If F is also computed for $2h$, then

$$F(2h) = f'(x) + 4b_1 h^2 + O(h^4) \ ,$$

and by subtraction we get

$$F(2h) - F(h) = 3b_1 h^2 + O(h^4) \ .$$

This shows that the dominating error term in $F(h)$ is

$$b_1 h^2 = \frac{F(2h) - F(h)}{3} + O(h^4) \ ,$$

and we see that

$$f'(x) = F(h) - b_1 h^2 + O(h^4) = F(h) + \frac{F(h) - F(2h)}{3} + O(h^4) \ .$$

Thus, without extra evaluations of f we have an approximation with truncation error proportional to h^4 instead of h^2.

Example. In the example on page 146 we found the following approximations of $f'(1)$ when $f(x) = e^x$,

h	$F(h) = D_0(h)$
0.4	2.7913515
0.2	2.7364400

The truncation error $R_{\mathrm{T}} = F(0.2) - f'(1)$ is estimated as

$$\tau = \big(F(0.4) - F(0.2)\big)/3 = 0.0183038 \ ,$$

and we get

$$f'(1) \simeq F(0.2) - \tau = 2.7181362 \ .$$

The error in this approximation is $-1.46 \cdot 10^{-4}$, while the true error in the approximation $F(0.2)$ is $F(0.2) - f'(1) = 1.82 \cdot 10^{-2}$. ∎

We want to generalize this procedure, and introduce the notation

$$F_1(h) = F(h) \ ,$$
$$F_2(h) = F_1(h) + \tfrac{1}{3}\left(F_1(h) - F_1(2h)\right) \ .$$

According to the derivation, we have eliminated the h^2-term in the truncation error (6.3.2), so that

$$F_2(h) = f'(x) + \widetilde{b}_2 h^4 + O(h^6) \ .$$

If we have computed $F_2(h)$, for different values of h, we can estimate the term $\widetilde{b}_2 h^4$ in a similar way:

$$\widetilde{b}_2 h^4 = \frac{F_2(2h) - F_2(h)}{15} + O(h^6) \ ,$$
$$f'(x) = F_3(h) + O(h^6) \ , \quad F_3(h) = F_2(h) + \frac{F_2(h) - F_2(2h)}{15} \ .$$

Example. Again, we consider $f(x) = e^x$ and look at approximations of $f'(1)$. By means of the above procedure we get the following results[4)], where Δ_k denotes the difference $D_k(h) = F_k(h) - F_k(2h)$.

h	$F_1(h) = D_0(h)$	$\Delta_1(h)/3$	$F_2(h)$	$\Delta_2(h)/15$	$F_3(h)$
0.40	2.7913515				
0.20	2.7364400	$-1.83 \cdot 10^{-2}$	2.7181362		
0.10	2.7228146	$-4.54 \cdot 10^{-3}$	2.7182728	$9.11 \cdot 10^{-6}$	2.7182819
0.05	2.7194146	$-1.13 \cdot 10^{-3}$	2.7182813	$5.67 \cdot 10^{-7}$	2.7182818

Notice that $\Delta_1(2h)/\Delta_1(h) \simeq 4$ and $\Delta_2(2h)/\Delta_2(h) \simeq 16$, ie the errors in the approximations are reflected in the differences.

The error in $F_3(0.05)$ can be estimated by

$$F_3(0.05) - f'(x) \simeq \frac{F_3(0.10) - F_3(0.05)}{2^6 - 1} = 5.409 \cdot 10^{-10} \ .$$

The true error is $F_3(0.05) - e = 5.397 \cdot 10^{-10}$. ∎

The above principle for estimating $F(0)$ can be used more generally, when F has been computed for two arguments, h and qh, and it is known that the truncation error in F is proportional to h^p.

[4)] The values were computed in MATLAB with approximately 16 decimal digits, but we only display 8 digits

Richardson extrapolation. If

$$F(h) = F(0) + ch^p + O(h^r) \ , \quad r > p \ ,$$

with a known p and an unknown c, which are independent of h, then

$$F(h) + \frac{1}{q^p - 1}\big(F(h) - F(qh)\big) = F(0) + O(h^r) \ .$$

If we know a complete expansion of the truncation error, we can perform repeated Richardson extrapolation. Assume that

$$F(h) = F(0) + a_1 h^{p_1} + a_2 h^{p_2} + \cdots \tag{6.4.1}$$

with known exponents $p_1, p_2, \ldots$, but unknown $a_1, a_2, \ldots$. Further, assume that F has been computed for arguments $\ldots, q^3h, q^2h, qh, h$.

Put $F_1(h) = F(h)$ and compute

$$F_{k+1}(h) = F_k(h) + \frac{1}{q^{p_k} - 1}\big(F_k(h) - F_k(qh)\big) \quad k = 1, 2, \ldots \tag{6.4.2}$$

In this extrapolation the h^{p_k}-term is eliminated from the expansion, so that

$$F_{k+1}(h) = F(0) + \widetilde{a}_{k+1} h^{p_{k+1}} + \widetilde{a}_{k+2} h^{p_{k+2}} + \cdots \ .$$

The values are conveniently arranged in a scheme like

$$\begin{array}{lllll} F_1(q^3h) & & & & \\ F_1(q^2h) & F_2(q^2h) & & & \\ F_1(qh) & F_2(qh) & F_3(qh) & & \\ F_1(h) & F_2(h) & F_3(h) & F_4(h) & \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

The entries are computed row by row, and extrapolations are performed until two consecutive values in the same column agree to the required accuracy. The following can be shown.

If h is sufficiently small, then the difference between two adjacent values in the same column gives an upper bound for the truncation error.

Example. Repeated Richardson extrapolation can be implemented in MATLAB as follows.

```
function  [y, info, R] = richextr(F,h0,q,p,tol)
% Richardson extrapolation of values given by
%   F(h), h = h0, h0/q, h0/q^2, ...
% Assume that  F(h) = F(0) + a1*h^p(1) + ... + am*h^p(m) ,
% where  m = length(p) .
% Stop when two adjacent values in the same column differ
% less than  tol .
% info = [estimated error, row number, column number]

% Initialize the scheme
m = length(p);  R = zeros(m+2,m+1);
info = [inf  0  0];                          % best so far
h = h0;  R(1,1) = feval(F,h); for  i = 1 : m+1
  h = h/q;   R(i+1,1) = feval(F,h);
  for  k = 1 : min(i,m)
    d = R(i+1,k) - R(i,k);
    if  abs(d) <= tol          % desired accuracy obtained
      y = R(i+1,k);
      info = [abs(d)  i+1  k];
      if  nargout > 2            % return active part of R
        R = R(1:i+1,1:i);
      end
      return
    end

    if  abs(d) < info(1)              % update best so far
      info = [abs(d)  i+1  k];
    end
    R(i+1,k+1) = R(i+1,k) + d/(q^p(k) - 1);  % extrapolate
  end
end
% Required accuracy not obtained.  Return best estimate
y = R(info(2),info(3));
```

We shall use `richextr` to estimate $f'(1)$ when $f(x) = e^x$, but now we start with the forward difference approximation. In MATLAB this can be implemented by

```
function  Df = Dforw(h)
Df = (exp(1+h) - exp(1))/h;
```

The exponents in (6.4.1) are $p_i = i$, and we get

```
>> [y,info] = richextr(@Dforw,0.5,4,[1:8],1e-8)
y = 2.718281828
info = 1.38e-09    6    4
```

The first element in `info` is the estimate $|y - e| \leq 1.38 \cdot 10^{-09}$. The true error is $|y - e| = 6.69 \cdot 10^{-12}$. By comparison with Figure 6.4 on page 148

we see that the use of successive Richardson extrapolation has given us approximately 3 more correct digits than it was possible to get by means of the forward difference approximation.

The last two elements in `info` show that the process stops in the 6th row ($h = 0.5/4^5 \simeq 4.88 \cdot 10^{-4}$) because $|F_4(h) - F_4(4h)| \leq 10^{-8}$. ∎

In practice we cannot avoid errors in the entry values of the extrapolation scheme. Instead of the values $F_1(h) = F(h)$ we get $\overline{F}_1(h) = F(h) + \epsilon_h$. Suppose that all $|\epsilon_h| \leq \epsilon$. Then

$$\overline{F}_2(h) = F_1(h) + \epsilon_h + \frac{F_1(h) + \epsilon_h - F_1(qh) - \epsilon_{qh}}{q^{p_1} - 1} = F_2(h) + \epsilon_h^{[2]} \ ,$$

where

$$|\epsilon_h^{[2]}| \leq \left| \epsilon_h + \frac{\epsilon_h - \epsilon_{qh}}{q^{p_1} - 1} \right| \leq \left(1 + \frac{2}{q^{p_1} - 1} \right) \epsilon \ = \ \frac{q^{p_1} + 1}{q^{p_1} - 1} \, \epsilon \ .$$

Similarly,

$$\overline{F}_3(h) = F_3(h) + \epsilon_h^{[3]} \ , \quad |\epsilon_h^{[3]}| \leq \frac{q^{p_2} + 1}{q^{p_2} - 1} \, \frac{q^{p_1} + 1}{q^{p_1} - 1} \, \epsilon \ ,$$

etc. This shows that the effect of the errors may grow as we proceed with the extrapolations. The growth is modest, however. It can be shown that all $|\epsilon_h^{[j]}| \leq 2\epsilon$ in the case $q = 2$ and $p_i = 2i$.

Richardson extrapolation has another interpretation. We know $F(h)$ for certain nonzero h-values and seek $F(0)$. It is natural to approximate F by an interpolating polynomial, and use this polynomial to estimate $F(0)$. Assume, eg, that we know that

$$F(h) = b_0 + b_1 h^2 + b_2 h^4 + \cdots \ ,$$

and that we have computed $F(4h_0)$, $F(2h_0)$ and $F(h_0)$. We can find the polynomial of the form $c_0 + c_1 h^2 + c_2 h^4$, which interpolates the three points $\big((4h_0)^2, F(4h_0)\big)$, $\big((2h_0)^2, F(2h_0)\big)$ and $\big(h_0^2, F(h_0)\big)$. The value corresponding to $h = 0$ can be computed by Neville's method, Section 5.5. Let $x = h^2$, then the table on page 114 takes the form

$0 - x$	x			
$-(4h_0)^2$	$(4h_0)^2$	$F(4h_0)$		
			$P_{01}(0)$	
$-(2h_0)^2$	$(2h_0)^2$	$F(2h_0)$		$P_{012}(0)$
			$P_{12}(0)$	
$-h_0^2$	h_0^2	$F(h_0)$		

The values of the elements in the last two columns are computed as prescribed in Theorem 5.5.1:

$$P_{01}(0) = \frac{-(4h_0)^2 F(2h_0) + (2h_0)^2 F(4h_0)}{(2h_0)^2 - (4h_0)^2} = \frac{4F(2h_0) - F(4h_0)}{3} = F_2(2h_0) \,,$$

$$P_{12}(0) = \frac{-(2h_0)^2 F(h_0) + h_0^2 F(2h_0)}{h_0^2 - (2h_0)^2} = \frac{4F(h_0) - F(2h_0)}{3} = F_2(2h_0) \,,$$

$$P_{012}(0) = \frac{-(4h_0)^2 P_{12}(0) + h_0^2 P_{01}(0)}{h_0^2 - (4h_0)^2} = \frac{16F_2(h_0) - F_2(2h_0)}{15} = F_3(2h_0) \,.$$

We recognize the results from Richardson extrapolation.

There are other extrapolation methods, which are based on interpolation by rational functions. In some applications they give better approximations than methods based on polynomial interpolation.

Finally, we give two examples with applications of the methods discussed in this chapter.

Example. In Chapter 4 we demonstrated that Newton-Raphson's method is efficient for computing a zero of a function f. Especially the quadratic convergence is useful if we want high accuracy. The success, however, depends on correct implementation of the derivative. We shall describe a method for checking the implementation of f'.

For a given x and step length h we compute a forward difference approximation, a backward difference approximation with half the step length and an extrapolated approximation,

$$\begin{aligned} D_+ &= \big(f(x+h) - f(x)\big)/h \,, \\ D_- &= \big(f(x) - f(x-\tfrac{1}{2}h)\big)/(\tfrac{1}{2}h) \,, \\ D_E &= (D_+ + 2D_-)/3 \,. \end{aligned}$$

According to the analysis in Section 6.3 we have

$$\begin{aligned} D_+ &= f'(x) + \tfrac{1}{2}\,hf''(x) + \tfrac{1}{6}\,h^2 f^{(3)}(x) + O(h^3) \,, \\ D_- &= f'(x) - \tfrac{1}{4}\,hf''(x) + \tfrac{1}{24}\,h^2 f^{(3)}(x) + O(h^3) \,, \\ D_E &= f'(x) + \tfrac{1}{12}\,h^2 f^{(3)}(x) + O(h^3) \,. \end{aligned}$$

Now, let $d(x)$ denote the computed derivative, and assume that it has an error $\psi(x)$,

$$d(x) = f'(x) - \psi(x) \,.$$

Then

$$\delta_+ = D_+ - d(x) = \psi(x) + \tfrac{1}{2}\,hf''(x) + O(h^2)\ ,$$
$$\delta_- = D_- - d(x) = \psi(x) - \tfrac{1}{4}\,hf''(x) + O(h^2)\ ,$$
$$\delta_E = D_E - d(x) = \psi(x) + \tfrac{1}{12}\,h^2 f^{(3)}(x) + O(h^3)\ .$$

Thus, if $\psi(x) = 0$ and h is sufficiently small, then we can expect to get $\delta_- \simeq -\frac{1}{2}\,\delta_+$ and $|\delta_E| \ll |\delta_-|$. If $\psi(x) \neq 0$, then all three δ-values will be almost equal to this error.

The following MATLAB function implements this algorithm

```
function  [df,delta] = checkder(fdf,x,h)
% Check implementation of derivative, as provided by
%   [f,df] = fdf(x)
% delta = [F-df  B-df  E-df], where F, B, E are
% forward, backward and extrapolated approximations of df
[f,df] = feval(fdf,x);
xp = x + h;
F = (feval(fdf,xp) - f)/(xp -x);
xm = x - h/2;
B = (f - feval(fdf,xm))/(x - xm);
E = (F + 2*B)/3;
delta = [F B E] - df;
```

Generally, $\underline{fl}[x+h] \neq x+h$, so in the forward difference approximation we divide by $\overline{h}$, which is the difference between the floating point numbers `xp` and `x`. Similarly in the backward difference approximation.

We shall try the algorithm on the problem from the example on page 70.

```
function  [f, df] = fdf(x)
f = x - cos(x);
df = 1 + sin(x);
```

We choose $x = 1$ and $h = 10^{-5}$. (Based on the discussion at the end of Section 6.3 we recommend to use $h \simeq \sqrt[3]{\mu} \cdot |x|$).

```
>> [df, delta] = checkder(@fdf,1,1e-5)
df = 1.8415
delta = 2.7015e-006 -1.3508e-006 -1.4526e-011
```

We see that $\delta_- = -1.3508 \cdot 10^{-6} \simeq -\frac{1}{2}\,\delta_+ = -\frac{1}{2}(2.7015 \cdot 10^{-6})$, and that $|\delta_E| = 1.4526 \cdot 10^{-11}$ is orders of magnitude smaller, so we conclude that the derivative seems to be correctly implemented.

If we had made a sign error,

```
function  [f, df] = fdf2(x)
f = x - cos(x);
df = 1 - sin(x);
```

then we would get

```
>> [df, delta] = checkder(@fdf2,1,1e-5)
df = 0.1585
delta = 1.6829    1.6829    1.6829
```

The three elements in `delta` are equal (within the 5 displayed digits) and nonzero. This shows that there is an error.

We return to this application in a computer exercise. The exercise demonstrates that in general it may be necessary to try several values of x before one can draw a sound conclusion about the correctness of the implementation of f'.

It should be mentioned that the algorithm can easily be generalized to checking the elements of a Jacobian, cf Section 4.8. ∎

Example. In image analysis a common problem is to identify objects. This involves a search for sharp edges. As a simple example consider the image shown in Figure 6.6. This is a 128×128 matrix `A`. Each element in `A`

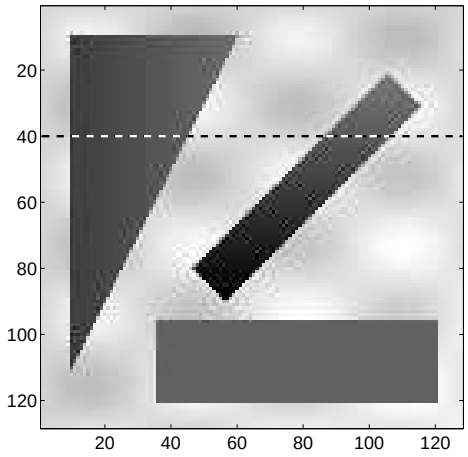


Figure 6.6. *Simple image.*

represents a *pixel* (a picture element), and the value `A(i,j)` is the light intensity of the pixel. The image is available in incbox as exc6_4.mat, and the plot is obtained by the commands

```
>> load exc6_4
>> imagesc(A),  axis image,  colormap gray
```

In Figure 6.7 we show the variation along the dotted line, which is row 40 in `A`. Note that smaller values in `A` correspond to darker pixels in Figure 6.6.

```
>> f = A(40,:);  plot(f, '.')
```

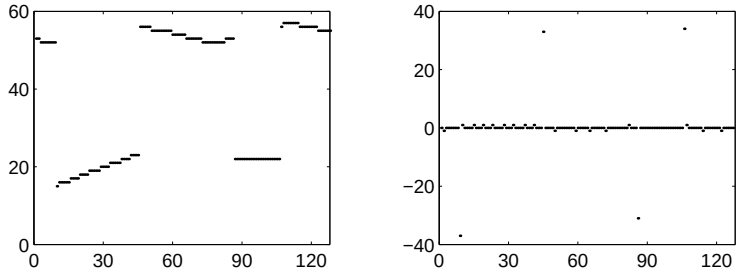


Figure 6.7. *Left: Variation along the dashed line in Figure 6.6. Right: Forward differences.*

An edge is an abrupt change in pixel value. Think of the pixel values along the dotted line as points from a differentiable function $f(x)$. Then a large value of $|f'(x)|$ signifies an edge. We can estimate f' by a forward difference,

$$f'(x_j) \simeq D_j = \frac{f_{j+1} - f_j}{h} \ , \quad j = 1, 2, \ldots, 127 \ .$$

We have $h = 1$, and all the D_j can be computed and plotted by the commands

```
>> D = f(2:128) - f(1:127);  plot(D, '.')
```

The result is shown in the right part of Figure 6.7. Note how the large values of $|D_j|$ stand clearly out from the rest. These values can, eg, be identified as the elements in `D`, which are larger than three times the standard deviation of all the elements,

```
>> thr = 3*std(D)
thr = 18.1107

>> e = find(abs(D) > thr)
e =  9    45    86   106
```

Thus, there is an edge between the 9^{th} and the 10^{th} pixel, another between the 45^{th} and the 46^{th} pixel, etc.

We return to this problem in a computer exercise. ∎

Exercises

E1. The third derivative $f^{(3)}(x)$ can be approximated by

$$\frac{1}{2h^3}\big(f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)\big) \ .$$

Show that the truncation error is

$$R_{\mathrm{T}} = a_1 h^2 + a_2 h^4 + a_3 h^6 + \cdots \ .$$

E2. Assume that $F(h) = F_0 + c_3 h^3 + O(h^4)$ with $c_3 \neq 0$. Which combination of $F(h)$ and $F(h/3)$ gives the best approximation of F_0 ?

E3. Consider the following approximation of $f'(x)$,

$$D(h) = \frac{1}{12h}\big(f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)\big) \ .$$

(a) Show that the truncation error is

$$R_{\mathrm{T}} = D(h) - f'(x) = a_1 h^4 + a_2 h^6 + a_3 h^8 + \cdots \ .$$

(b) Assume that we know the following, correctly rounded function values, and want to approximate $f'(0.5)$ as accurately as possible.

x	$f(x)$
0.1	0.000167
0.3	0.004480
0.4	0.010582
0.425	0.012679
0.450	0.015034
0.475	0.017662
0.5	0.020574
0.525	0.023787
0.550	0.027313
0.575	0.031165
0.6	0.035358
0.7	0.055782
0.9	0.116673

Use the above $D(h)$ and Richardson extrapolation to approximate $f'(0.5)$, and estimate the error in the approximation. Does the Richardson extrapolation pay off in this case?

Computer Exercises

C1. We shall study the error when the derivative of the function

$$f(x) = \frac{1}{1 + (x-2)^3}$$

is approximated by forward and central differences, respectively.

(a) Make tables of the errors $D_+(h) - f'(x)$ and $D_0(h) - f'(x)$ for all combinations of

$$h = 0.1, 0.01, 0.001, 0.0001, \qquad x = 0.4, 0.8, 1.2, \ldots, 2.8$$

Compare the errors $D_+(h) - f'(x)$ and $D_0(h) - f'(x)$ for fixed x and h.

How are the errors affected when h is divided by 10 ?

(b) The second derivative of f is

$$f''(x) = 6(x-2)(2(x-2)^3 - 1)/\big(1 + (x-2)^3\big)^3 \ ,$$

so $f''(2) = f''(2 + 2^{-1/3}) = 0$. Use this fact to explain the relatively good accuracy in the approximation by $D_+(h)$ at $x = 2$ and $x = 2.8$.

Why is the approximation so poor at $x = 0.8$ and $x = 1.2$?

C2. Let $f(x) = e^x$. Use `richextr` (cf page 153; available from incbox) to find an approximation of $f'(1)$, starting with the central difference approximation.

Use $h_0 = 1/16$, `tol = 1e-12` and $q = 2, 4, 8, \ldots, 256$, and discuss the results.

C3. Let $f(x) = x - \cos x$. Use the MATLAB functions `checkder`, `fdf` and `fdf2` (cf the example starting on page 156; `checkder` is available from incbox) to answer the following questions

(a) Use `fdf` with $x = 1$, and make a plot like Figure 6.4 for the three error estimates δ_+, δ_- and δ_E.

(b) Comment on the results from

```
>> [df, delta] = checkder(@fdf,pi/2,1e-5)
```

(c) Comment on the results from

```
>> [df, delta] = checkder(@fdf2,pi,1e-5)
```

This example demonstrates that in general the check should be performed for several values of x before one can draw a sound conclusion about the correctness of the implementation of f'.

C4. Load the matrix `A` in exc6_4.mat (provided in incbox), and use the technique indicated in the example on page 158 to find the triangle in Figure 6.6.

References

The idea behind Richardson extrapolation was first proposed in

> L.F. Richardson, *The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam*, Philos. Trans. Roy. Soc. London, Ser. A, **210**, pp 307–357, 1910.

The usual reference for it is 17 years younger,

> L.F. Richardson and J.A. Gaunt, *The deferred approach to the limit*, Philos. Trans. Roy. Soc. London, Ser. A, **226**, pp 229–361, 1927.

Extrapolation is used to enhance accuracy in connection with numerical methods for a number of problems. In the next chapter we use Richardson extrapolation in connection with numerical integration. The following two papers give a good overview of different extrapolation methods and give many references.

> D.C. Joyce, *Survey of extrapolation processes in numerical analysis*, SIAM Review **13**, pp 435–490, 1971.

> D.A. Smith, W.F. Ford and A. Sidi, *Extrapolation methods for vector sequences*, SIAM Review **29**, pp 199–233, 1987.

Chapter 7

Integration

7.1. Introduction

Numerical integration (or quadrature) is the computation of an approximation of $\int_a^b f(x)\,dx$. We shall derive methods for this.

Such methods are needed, of course, if the integrand f is known only in discrete points. Also when an explicit formula for f is known, it is sometimes not possible to compute the integral exactly because a primitive function cannot be found or is not directly computable. This is the case, eg, for $\int_0^1 e^{-x^2}\,dx$ and $\int_a^{\pi/2} \sqrt{1+\cos^2 x}\,dx$. Finally — even if a primitive function is known, the computation of it may be so costly that approximate, numerical methods for computing the integral are to be preferred.

We construct methods for numerical integration by approximating f by a function that is easily integrated: a polynomial. There are two possible ways to obtain good accuracy:

1. approximate f by a single interpolating polynomial of high degree,
2. approximate f by different low degree polynomials in small subintervals of the interval of integration.

We shall see that the second approach is to be preferred.

7.2. The Trapezoidal Rule

The trapezoidal rule is based on approximating the integrand f by the straight line through the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$ (where $x_0 = a$ and $x_1 = b$), see Figure 7.1. This means that the integral is approximated

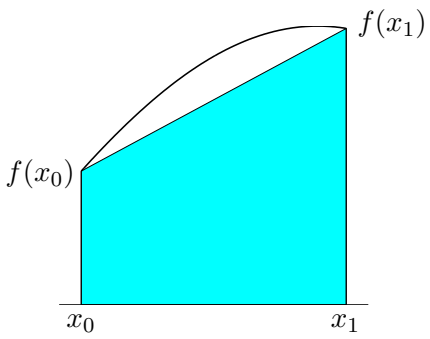


Figure 7.1. *The trapezoidal rule.*

by the area of a trapezoid,

$$\int_{x_0}^{x_1} f(x)\,dx = \frac{h}{2}\left(f(x_0)+f(x_1)\right) + R_{\mathrm{T}}\ .$$

By use of Theorem 5.2.2 we see that the truncation error is

$$R_{\mathrm{T}} = \int_{x_0}^{x_1} \frac{f''(\xi(x))}{2!}\,(x-x_0)(x-x_1)\,dx\ ,$$

where $\xi(x)$ is a point in the open interval $]x_0, x_1[$. To simplify the expression we make a change of variable, $x = x_0 + th$. Since $x_1 = x_0 + h$, we get

$$\begin{aligned} R_{\mathrm{T}} &= h\int_0^1 \frac{f''(\xi(x_0+th))}{2!}\,th\,(t-1)h\,dt \\ &= \tfrac{1}{2}h^3\int_0^1 f''(\xi(x_0+th))\,t(t-1)\,dt\ . \end{aligned} \tag{7.2.1}$$

This can be simplified further when we use the following theorem.

Theorem 7.2.1. Mean value theorem of integral calculus. If the function φ is continuous and the function ψ is continuous and does not change sign in the closed interval $[a, b]$, then there is a point $\hat{x}$ inside the interval, such that

$$\int_a^b \varphi(x)\psi(x)\,dx = \varphi(\hat{x})\int_a^b \psi(x)\,dx\ .$$

Returning to (7.2.1), we see that $t(t-1)$ has constant sign for $0 \le t \le 1$. Therefore, if f'' is continuous in $[x_0, x_1]$, then the mean value theorem gives

$$R_{\mathrm{T}} = \tfrac{1}{2}h^3 f''(\xi(x_0+\hat{t}h))\int_0^1 t(t-1)\,dt\ ,$$

where $0<\hat{t}<1$. We put $\eta = x_0 + \hat{t}h$, compute the integral, and get

$$R_{\mathrm{T}} = -\tfrac{1}{12}h^3 f''(\eta), \qquad x_0 < \eta < x_1 \ . \tag{7.2.2}$$

This expression shows that the truncation error is small if h is small. To achieve that, we can divide the interval of integration $[a,b]$ into subintervals $\{[x_{i-1},x_i]\}_{i=1}^m$, each of length $h=(b-a)/m$, cf Figure 7.2.

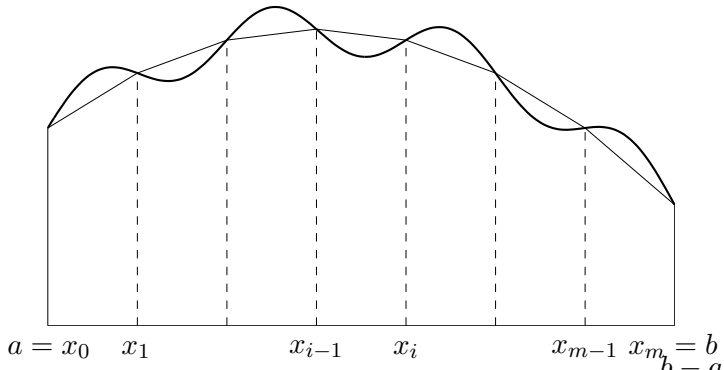


Figure 7.2. *The trapezoidal rule.* $x_i = a + ih, \quad h = \dfrac{b-a}{m}$.

The integral is the sum of the contributions from the subintervals,

$$\int_a^b f(x)\,dx = \sum_{i=1}^m \int_{x_{i-1}}^{x_i} f(x)\,dx \ ,$$

and each contribution is approximated by the trapezoidal rule,

$$\int_{x_{i-1}}^{x_i} f(x)\,dx = \tfrac{1}{2}h(f_{i-1}+f_i) - \tfrac{1}{12}h^3 f''(\eta_i) \ ,$$

where $f_j = f(x_j)$ and $\eta_i \in]x_{i-1},x_i[$. The term $-\frac{1}{12}h^3 f''(\eta_i)$ is called the *local truncation error*.

When we sum the contributions, we note that $\frac{1}{2}hf_i$, $i=1,\ldots,m-1$ appears both in the ith and the $(i+1)$st contribution, and we get

$$\int_a^b f(x)\,dx = h\left(\tfrac{1}{2}f_0 + f_1 + \cdots + f_{m-1} + \tfrac{1}{2}f_m\right) + R_{\mathrm{T}} \ ,$$

where the *total truncation error* is the sum of the local errors,

$$R_{\mathrm{T}} = -\tfrac{1}{12}h^3 \sum_{i=1}^m f''(\eta_i) \ .$$

If f'' is continuous in $]a,b[$, there is a number η in this interval, such that

$$\sum_{i=1}^{m} f''(\eta_i) = m \cdot f''(\eta) \ .$$

We insert this in the above expression for R_{T}, use the fact that $mh = b-a$, and get

$$R_{\mathrm{T}} = -\frac{b-a}{12}\, h^2 f''(\eta) \ .$$

Thus, the local truncation error is $O(h^3)$ and the total truncation error is $O(h^2)$.

We summarize:

The trapezoidal rule

$$\int_a^b f(x)\,dx = T(h) + R_{\mathrm{T}} \ ,$$

$$T(h) = h\left(\tfrac{1}{2}f_0 + f_1 + \cdots + f_{m-1} + \tfrac{1}{2}f_m\right) \ ,$$

where $h = (b-a)/m$ and $f_j = f(a+jh)$. If f'' is continuous, then

$$R_{\mathrm{T}} = -\frac{b-a}{12}\, h^2 f''(\eta), \quad a < \eta < b \ .$$

Example. The following MATLAB function implements the trapezoidal rule.

```
function  T = trapezrule(f,a,b,m)
% Approximate integral by trapezoidal rule
x = linspace(a,b,m+1);                    % grid points
T = (feval(f,a) + feval(f,b))/2;    % endpoint contrib.
for  i = 1 : m-1
  T = T + feval(f,x(i+1));    % interior point contrib.
end
T = (b-a)/m * T;                      % multiply by  h
```

We use this to approximate $I = \int_0^1 1/(1+x)\,dx$ for $h = 1, 1/2, 1/4$ and $1/8$.

```
function  f = intfun(x)
f = 1/(1+x);

>> for  k = 0 : 3
  disp(trapezrule(@intfun,0,1,2^k))
end
```

The exact value is $I = \log 2 \simeq 0.693147$. The results (rounded to 6 digits) and the associated errors are given below.

h	$T(h)$	$T(h) - \log 2$
1	0.750000	$5.69 \cdot 10^{-2}$
0.5	0.708333	$1.52 \cdot 10^{-2}$
0.25	0.697024	$3.88 \cdot 10^{-3}$
0.125	0.694122	$9.75 \cdot 10^{-4}$

Since $R_{\mathrm{T}} = O(h^2)$, the error should roughly be divided by 4 when h is halved. This is indeed seen to be the case. ∎

7.3. Newton-Cotes' Quadrature Formulas

The Newton-Cotes quadrature formulas are derived by replacing the integrand f by an interpolating polynomial and integrating that. The polynomial interpolates f on the grid

$$x_i = a + ih, \quad i = 0, 1, \ldots, n; \qquad h = \frac{b-a}{n} \ .$$

This means that $P_n(x)$ of degree $\leq n$ is determined so that $P_n(x_i) = f(x_i)$, $i = 0, 1, \ldots, n$. We get

$$\int_a^b f(x)\, dx = \int_a^b P_n(x)\, dx + R_{\mathrm{T}} \ .$$

In the case $n = 1$ we get the trapezoidal rule. For arbitrary n it follows from Theorem 5.2.2 that

$$R_{\mathrm{T}} = \int_a^b \frac{f^{(n+1)}(\xi(x))}{(n+1)!}\, (x - x_0)(x - x_1) \cdots (x - x_n)\, dx \ .$$

This shows that if the integrand is a polynomial of degree at most n, then $R_{\mathrm{T}} = 0$, ie in this case the quadrature formula gives the exact result. It is also easy to show that the quadrature formula expresses the approximate integral as a linear combination of the function values on the grid,

$$\int_a^b P_n(x)\, dx = \sum_{i=0}^{n} A_i f(x_i) \ . \tag{7.3.1}$$

To see this, we represent $P_n(x)$ for a moment by Lagrange's interpolating polynomial,

$$P_n(x) = \sum_{i=0}^{n} f(x_i) L_i(x) \ ,$$

where $L_i(x)$ is a certain degree n polynomial (given on page 115). It follows that $A_i = \int_a^b L_i(x)\,dx$.

Thus, the coefficients A_i in (7.3.1) can be derived by integration of the polynomials $L_i(x)$. There are other ways of computing the A_i, and we shall discuss the "*method of unknown coefficients*". Put

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n} A_i f(x_i) + R_{\mathrm{T}} \ ,$$

and require that

$$R_{\mathrm{T}} = 0 \quad \text{for} \quad f(x) = p_k(x), \quad k = 0, 1, \ldots, n \ ,$$

where p_k is a polynomial of degree k. This leads to a linear system of equations for the coefficients.

We illustrate the technique by using it to derive *Simpson's rule*: Put $n = 2$ and

$$\int_{x_0}^{x_2} f(x)\,dx = A_0 f(x_0) + A_1 f(x_1) + A_2 f(x_2) + R_{\mathrm{T}} \ ,$$

with $x_0 = a$, $x_1 = a + h$, $x_2 = b = a + 2h$. In order to get a simple system of equations we use the polynomials $p_k(x) = (x - x_1)^k$, and get

$$\begin{aligned}
\textstyle\int_{x_0}^{x_2} 1\,dx &= \quad 2h \ = \quad A_0 + A_1 + \ A_2 \quad , \\
\textstyle\int_{x_0}^{x_2} (x - x_1)\,dx &= \quad 0 \ = -hA_0 \quad\quad + \ hA_2 \quad , \\
\textstyle\int_{x_0}^{x_2} (x - x_1)^2\,dx &= \ \tfrac{2}{3}h^3 = \ h^2 A_0 \quad\quad + \ h^2 A_2 \quad .
\end{aligned}$$

This system has the solution

$$A_0 = A_2 = \frac{h}{3}, \quad A_1 = \frac{4h}{3} \ .$$

It is easy to show that also any third degree polynomial is integrated exactly, but not a fourth degree polynomial. It can be shown that the truncation error is $R_{\mathrm{T}} = -\frac{1}{90} h^5 f^{(4)}(\eta)$, where $x_0 < \eta < x_2$.

Thus, we have derived *Simpson's rule*:

$$\begin{aligned}
\int_{x_0}^{x_2} f(x)\,dx &= \frac{h}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right) + R_{\mathrm{T}} \ , \\
R_{\mathrm{T}} &= -\tfrac{1}{90}\, h^5 f^{(4)}(\eta), \quad x_0 < \eta < x_2 \ .
\end{aligned} \tag{7.3.2}$$

It is tempting to believe that one gets successively better approximations to the integral if m is increased. The following example illustrates that this is not always true.

Example. Consider the two integrals

$$I = \int_0^1 \frac{1}{1+25x^2}\,dx\ , \qquad J = \int_{-1}^1 \frac{1}{1+25x^2}\,dx\ .$$

The exact values are $I = \frac{1}{10}(\arctan 5 - \arctan(-5)) \simeq 0.274680$ and $J = 2I$. We have used Newton-Cotes' quadrature formulas for $n = 1, 2, 3, \ldots, 9$ to compute approximations I_n and J_n (we used tabulated values for the coefficients A_i for $n > 2$) and got the errors shown in the table below.

n	$I_n - I$	$J_n - J$
1	0.24455	−0.472
2	−0.00965	0.810
3	−0.01464	−0.133
4	−0.01316	−0.075
5	−0.00856	−0.088
6	−0.00241	0.225
7	−0.00124	0.030
8	0.00055	−0.249
9	0.00052	−0.070

The poor accuracy for small values of n is not surprising: with so few points it is not possible to get a good approximation to the integrand $(1+25x^2)^{-1}$. The accuracy of the approximations I_n improves steadily as n increases from 3 to 9, but this is **not** the case with the approximations J_n. The reason for this disappointing behaviour can be seen from the figure on page 118, illustrating Runge's phenomenon: the interpolating polynomial oscillates between the interpolation points, and the amplitude of the oscillations increases with n. The effect of the oscillations is damped by integration, but still we see that we, eg, get a poorer approximation by using a degree 8 polynomial than we do by using a polynomial of degree 7 or 6. ∎

It can be shown that there are continuous functions such that approximations I_n of $I = \int_a^b f(x)\,dx$ computed by Newton-Cotes' quadrature formulas do not converge to I as $n \to \infty$. Therefore, instead of using one high degree interpolating polynomial on the interval $[a, b]$ we subdivide the integration interval and use a low degree interpolating polynomial in each subinterval. If we use first degree polynomials, we get the trapezoidal rule. If the number of subintervals is even, $m = 2q$, we can use Simpson's rule (7.3.2) in each of the subintervals $[x_0, x_2]$, $[x_2, x_4], \ldots, [x_{m-2}, x_m]$ and get

$$\int_a^b f(x)\,dx = \sum_{i=1}^{m/2} \int_{x_{2i-2}}^{x_{2i}} f(x)\,dx$$
$$= \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{m-2} + 4f_{m-1} + f_m\right) + R_{\mathrm{T}} \ ,$$

where

$$R_{\mathrm{T}} = -\frac{h^5}{90} \sum_{i=1}^{m/2} f^{(4)}(\eta_i) \ , \quad x_{2i-2} < \eta_i < x_{2i} \ .$$

If $f^{(4)}$ is continuous, we can proceed as we did with the trapezoidal rule, and we get

Simpson's rule

$$\int_a^b f(x)\,dx = S(h) + R_{\mathrm{T}} \ ,$$

$$S(h) = \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{m-2} + 4f_{m-1} + f_m\right) \ ,$$

where m is even, $h = (b-a)/m$ and $f_j = f(a+jh)$.
If $f^{(4)}$ is continuous, then

$$R_{\mathrm{T}} = -\frac{b-a}{180}\, h^4 f^{(4)}(\eta), \quad a < \eta < b \ .$$

Example. As in the example on page 166 we consider the integral

$$\int_0^1 \frac{1}{1+x}\,dx = \log 2 \ .$$

We use Simpson's rule with $h = 0.5,\ 0.25,\ 0.125$, and get

h	$S(h)$	$S(h) - \log 2$
0.5	0.694444	$1.30 \cdot 10^{-3}$
0.25	0.693254	$1.07 \cdot 10^{-4}$
0.125	0.693155	$7.35 \cdot 10^{-6}$

Since $R_{\mathrm{T}} = O(h^4)$, the error should roughly be divided by 16 when h is halved. This is indeed seen to be the case. ∎

7.4. Romberg's Method: Trapezoidal Rule with Richardson Extrapolation

As in Section 7.2 we approximate

$$I = \int_a^b f(x)\,dx$$

by

$$T(h) = h\left(\tfrac{1}{2}f_0 + f_1 + \cdots + f_{m-1} + \tfrac{1}{2}f_m\right) ,$$

where

$$h = (b-a)/m ,$$
$$f_j = f(x_j) = f(a+jh), \quad j = 0, 1, \ldots, m .$$

We have shown that the truncation error is $O(h^2)$ if f'' is continuous. One can show even more:

> If f is $(2k+2)$ times continuously differentiable, then
> $$T(h) = \int_a^b f(x)\,dx + a_1h^2 + a_2h^4 + \cdots + a_kh^{2k} + O(h^{2k+2}) ,$$
> where the coefficients $a_1, \ldots, a_k$ are independent of h.

When this condition is satisfied, we can use Richardson extrapolation as described in Section 6.4. Put $T_1(h) = T(h)$ and assume that we also have computed $T_1(2h)$, $T_1(4h)$, $\ldots$. Then the values computed by

$$T_{r+1}(h) = T_r(h) + \frac{T_r(h) - T_r(2h)}{2^{2r}-1} , \quad r = 1, 2, \ldots, k \tag{7.4.1}$$

satisfy

$$T_r(h) = I + R_{\mathrm{T}}, \quad R_{\mathrm{T}} = \begin{cases} O(h^{2r}), & r \le k+1 , \\ O(h^{2k+2}), & r > k+1 . \end{cases}$$

This procedure is called *Romberg's method*.

Example. As in the example on page 166 we consider the integral

$$\int_0^1 \frac{1}{1+x}\,dx = \log 2 \simeq 0.693147 ,$$

and get the following results from Romberg's method. The values were computed with approximately 16 digits accuracy, but we show only 6. We use the notation $\Delta_r(h) = 10^6(T_r(h) - T_r(2h))$.

h	$T_1(h)$	$\frac{\Delta_1(h)}{3}$	$T_2(h)$	$\frac{\Delta_2(h)}{15}$	$T_3(h)$	$\frac{\Delta_3(h)}{63}$	$T_4(h)$
1	0.750000						
1/2	0.708333	−13889	0.694444				
1/4	0.697024	−3770	0.693254	−79	0.693175		
1/8	0.694122	−967	0.693155	−7	0.693148	0	0.693147

The truncation error in the value $T_3(1/8) \simeq 0.693148$ is estimated as $-10^{-6} \cdot \Delta_3(1/8)/63 \simeq 4.24 \cdot 10^{-7}$. The true error is $T_3(1/8) - \log 2 \simeq 7.21 \cdot 10^{-7}$. ∎

If we compare the above results for $T_2(h)$ with the results in the example on page 170, we see that $T_2(h) = S(h)$. This is no coincidence: If $x_0, x_1, x_2, \ldots, x_{m-2}, x_{m-1}, x_m$ is the grid corresponding to the step length h, then $x_0, x_2, \ldots, x_{m-2}, x_m$ is the grid corresponding to the step length $2h$, and

$$T(h) = h\big(\tfrac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{m-2} + f_{m-1} + \tfrac{1}{2}f_m\big) ,$$
$$T(2h) = 2h\big(\tfrac{1}{2}f_0 \quad + \quad f_2 + \cdots + f_{m-2} \quad + \quad \tfrac{1}{2}f_m\big) .$$

Then

$$T_2(h) = T(h) + \frac{T(h) - T(2h)}{3} = \frac{4T(h) - T(2h)}{3}$$
$$= \frac{h}{3}\big(f_0 + 4f_1 + 2f_2 + \cdots + 2f_{m-2} + 4f_{m-1} + f_m\big) = S(h) .$$

Thus, for $h = (b-a)/2$, $T_2(h)$ is equivalent to the Newton-Cotes formula for $n = 2$. For $r > 2$ there is no Newton-Cotes formula, which is equivalent to $T_r(h)$.

It can be shown that the entries in each column of the Romberg scheme converge to the integral I as $h \to 0$. As in the general Richardson extrapolation procedure the truncation error of each element $T_r(h)$ in the Romberg scheme is bounded by $|T_r(h) - T_r(2h)|$, ie by the difference between the element and the nearest value above it in the same column.

Example. In the general Richardson extrapolation procedure (6.4.2) we assume that the step lengths are h_0, h_0/q, h_0/q^2, ..., for some constant q. Romberg's method is always used with $q = 2$, as discussed above. One reason for this is, that then we can reuse the function values for the grid with step length $2h$ when we compute $T(h)$: from the above proof of $T_2(h) = S(h)$ it follows that

$$T(h) = \tfrac{1}{2}T(2h) + h\,(f_1 + f_3 + \cdots + f_{m-1}) .$$

This is exploited in the following MATLAB implementation of Romberg's method (the command `R(i,1) = R(i-1,1)/2 + h*s;`). The default use of the MATLAB function corresponds to the assumptions in the frame on page 171. In the example on page **??** we illustrate the use when this assumption is not satisfied.

```
function  [I, info, R] = romberg(f,a,b,tol,p)
% Romberg's method for the integral of  f  over  [a,b].
% Assume that  T(h) = T(0) + a1*h^p(1) + ... + aq*h^p(q) ,
% where  q = length(p) .
% Default:  p = [2 4 ... 14]
% Stop when two adjacent values in the same column differ
% less than  tol .
% info = [estimated error, m, r], where m+1 and r-1 are the
%    number of function evaluations and extrapolations.

% Initialize the scheme
if  nargin < 5
  p = 2 : 2 : 14;
end
q = length(p);  R = zeros(q+2,q+1);
h = b - a;  m = 1;
R(1,1) = h*(feval(f,a) + feval(f,b))/2;
mdif = inf;   % initialize min difference
for  i = 2 : q+2
  % First element in next row
  h = h/2;  m = 2*m;
  s = 0;
  for  j = 1 : 2 : m
    s = s + feval(f, a+j*h);
  end
  R(i,1) = R(i-1,1)/2 + h*s;
  % Extrapolate
  jmax = min(i-1,q);
  for  j = 1 : jmax
    R(i,j+1) = R(i,j) + (R(i,j) - R(i-1,j))/(2^p(j) - 1);
  end
  % Check accuracy
  [md, jb] = min(abs(R(i,1:jmax) - R(i-1,1:jmax)));
  if  md < mdif  % better result
    info = [md m jb];  I = R(i,jb);
    if  md <= tol
      R = R(1:i,1:jmax+1);        % return active part of R
      return
    end
  end
end
```

Applied to the problem $I = \int_0^1 1/(1+x)\,dx = \log 2 \simeq 0.693147$, we get (`intfun` is defined on page 166)

```
>> [I, info] = romberg(@intfun,0,1,1e-8)
I = 0.69314718056362
info = 1.35e-09    32    5
```

The true error is $y - \log 2 \simeq 1.35 \cdot 10^{-11}$, so the estimate given by `info(1)` is very pessimistic. `info(2:3)` tell us that the result was obtained after the use of $h = 1, 1/2, \ldots, 1/16, 1/32$ (involving a total of 33 evaluations of the integrand) because $|T_5(h) - T_5(2h)| \leq 10^{-8}$. ∎

So far, we have discussed the truncation error R_{T}. In practice we have to accept that instead of the values $f_j = f(x_j)$ we get approximate values $\overline{f}_j$, and

$$\overline{T}(h) = h(\tfrac{1}{2}\overline{f}_0 + \overline{f}_1 + \cdots + \overline{f}_{m-1} + \tfrac{1}{2}\overline{f}_m) \ .$$

Assume that $|\overline{f}_j - f_j| \leq \epsilon$, $j = 0, 1, \ldots, m$. Then

$$\begin{aligned} |\overline{T}(h) - T(h)| &\leq h\big(\tfrac{1}{2}|\overline{f}_0 - f_0| + \sum_{j=1}^{m-1} |\overline{f}_j - f_j| + \tfrac{1}{2}|\overline{f}_m - f_m|\big) \\ &\leq h\epsilon(\tfrac{1}{2} + m - 1 + \tfrac{1}{2}) = hm\epsilon = (b-a)\epsilon \ . \end{aligned}$$

In contrast to the analysis on page 156 of general Richardson extrapolation, it can be shown that the effect of errors in the function values is not increased during the extrapolations in the Romberg scheme.

If the function values have absolute errors no larger than ϵ, then these lead to errors R_{XF} in the entries of the Romberg scheme, that can be estimated as

$$|R_{\mathrm{XF}}| \leq (b-a)\epsilon \ .$$

Finally, consider the effect of rounding errors. We only have to consider the summation error in the computation of $T(h)$,

$$R_{\mathrm{C}} = fl[T(h)] - T(h) \ .$$

It follows from Theorem 2.7.2 that we can use the above result when we put

$$\epsilon = m \cdot 1.06\mu \cdot \max|f_j| \ ,$$

where μ is the unit round off. Thus,

$$|R_{\mathrm{C}}| \leq mC\mu, \qquad C = 1.06(b-a) \cdot \max|f_j| \ .$$

Example. In the example on page 171 the integrand is $f(x) = 1/(1+x)$ and the interval is $[0, 1]$. Therefore $0.5 \le f_j \le 1$.

Assume that the function values are rounded to 6 decimals, ie $\epsilon = 0.5 \cdot 10^{-6}$. Then $|R_{\rm XF}| \le (1-0) \cdot 0.5 \cdot 10^{-6} = 0.5 \cdot 10^{-6}$.

The MATLAB precision is $\mu = 2^{-53} \simeq 10^{-16}$, so we get

$$C = 1.06 \cdot 1 \cdot 1 = 1.06, \qquad |R_{\rm C}| \lesssim 1.06 \cdot 10^{-16} \cdot 8 \simeq 10^{-15} \ .$$

This is negligible. Finally, the truncation error of the element 0.693148 can be bounded by $|R_{\rm T}| \le |0.693148 - 0.693175| = 2.7 \cdot 10^{-5}$.

Thus, the total error in the approximation 0.693148 of the integral I can be estimated as

$$|0.693148 - I| \le |R_{\rm T}| + |R_{\rm XF}| + |R_{\rm C}| < 3 \cdot 10^{-5} \ . \qquad \blacksquare$$

Example. Figure 7.3 shows the errors for different values of h when we use Romberg's method to approximate

$$I = \int_0^{4.5\pi} \sin x \, dx = 1 \ .$$

The unit round off is $\mu = 2^{-53} \simeq 10^{-16}$. As motivated in connection with Figure 6.4 on page 148 we use a log-log scale.

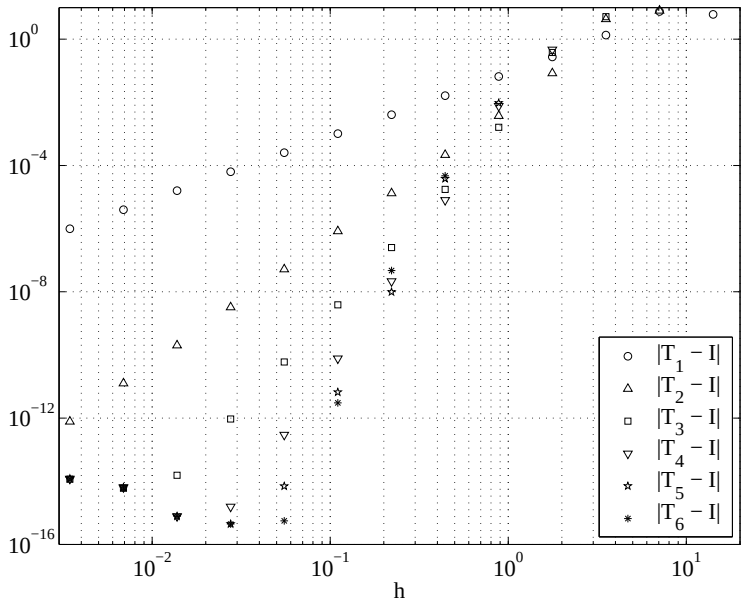


Figure 7.3. *Errors in approximation of* $I = \int_0^{4.5\pi} \sin x \, dx = 1$ *by Romberg's method.*

First, let us examine the errors $|T_1(h) - I|$: For large values of h the approximation is so crude that we cannot ignore the contribution $a_2h^4 + \cdots$ in the frame on page 171. As h decreases, the term a_1h^2 dominates more and more, and in the log-log scale the points get close to a straight line corresponding to the $O(h^2)$ behaviour of the error. The smallest h-value is $4.5\pi/m$ with $m = 2^{12} = 4096$. The corresponding error is approximately 10^{-6}.

The values $T_2(h)$ are computed under the assumption that $T_1(h) \simeq I + a_1h^2$ is satisfied with "almost equality". We get the expected $O(h^4)$ behaviour for $h \lesssim 1$. The error for $m = 4096$ is approximately 10^{-12}, ie we get approximately 6 more correct digits than we did with $T_1(h)$ for the same h-value.

As the number r of extrapolations increases, we get smaller ranges of h-values where we can observe the expected $O(h^{2r})$ behaviour. To explain that, consider $T_5(h)$: the $O(h^2)$ assumption about the trapezoidal rule must be satisfied to "almost equality" for the step lengths $h, 2h, \ldots, 16h$; the $O(h^4)$ assumption about $T_2(h)$ must be satisfied to "almost equality" for the step lengths $h, 2h, \ldots, 8h$; etc. This leaves a small range of h-values before rounding errors dominate, and we do not really get the improvement that we hoped for. This aspect is even more pronounced for the results for T_6. However, it should be mentioned that the smallest of all the errors is $|T_6(4.5\pi/256) - I| \simeq 5.6 \cdot 10^{-16}$. ∎

7.5. Difficulties with Numerical Integration

The use of the trapezoidal rule or Simpson's method assumes that the integrand f has a finite value for all x in the closed interval $[a, b]$, since the methods involve the endpoint values $f(a)$ and $f(b)$. Further, if some of the low order derivatives of f have singularities at one of the endpoints, then we get poorer accuracy and do not get the $O(h^2)$ and $O(h^4)$ convergence that was illustrated in the examples on page 166 and 170.

In many cases the integral can be rewritten into a form that is better suited for numerical computation. It is a good rule to *use numerical integration only when the integral has been simplified as far as possible by mathematical reductions.*

We shall use the integral

$$I = \int_0^{0.64} \frac{\arctan x}{\sqrt{x}}\, dx \tag{7.5.1}$$

to demonstrate different ways of handling an endpoint singularity in the derivative. For $|x| \le 1$ we have

$$\arctan x = x - \tfrac{1}{3}\,x^3 + \tfrac{1}{5}\,x^5 - \cdots = x + O(x^3)\ . \tag{7.5.2}$$

Hence, the integrand has the form

$$f(x) = \frac{\arctan x}{\sqrt{x}} = \sqrt{x} + O(x^{5/2})\ ,$$

and f' has a singularity at $x = 0$.

Sometimes a *change of variables* will get rid of a troublesome singularity. With the substitution $t = \sqrt{x}$ we get[1)]

$$I = \int_0^{0.8} 2\arctan t^2\ dt\ ,$$

so this transformation got rid of the singularity.

Example. The original and the transformed versions of the integrand can be implemented in MATLAB as follows.

```
function  y = intf1(x)
if  x == 0,  y = 0;
else,        y = atan(x)/sqrt(x); end

function  y = intf2(x)
y = 2*atan(x^2);
```

We use `trapezrule` from page 166,

```
T1 = trapezrule(@intf1,0,0.64,m)
T2 = trapezrule(@intf2,0,0.80,m)
```

for $m = 20, 40, 80$, and get

m	`T1`	$\lvert\texttt{T1} - I\rvert$	`T2`	$\lvert\texttt{T2} - I\rvert$
20	0.322785	$1.16 \cdot 10^{-3}$	0.324249	$3.03 \cdot 10^{-4}$
40	0.323533	$4.14 \cdot 10^{-4}$	0.324022	$7.57 \cdot 10^{-5}$
80	0.323799	$1.47 \cdot 10^{-4}$	0.323965	$1.89 \cdot 10^{-5}$

The results for the transformed integral behave as we saw on page 166: a doubling of m (ie a halving of the step length h) leads to an error which is approximately 4 times smaller. With the original formulation the "gain factor" is only approximately $2^{1.5} \simeq 2.828$. Also note that for the same m-value the results with the transformed integral are considerably more accurate than with the original formulation.

[1)] Remember that also the integration bounds must be changed.

With Simpson's rule we similarly get

m	S1	$\lvert\texttt{S1} - I\rvert$	S2	$\lvert\texttt{S2} - I\rvert$
20	0.323482	$4.65 \cdot 10^{-4}$	0.323946	$1.57 \cdot 10^{-7}$
40	0.323782	$1.64 \cdot 10^{-4}$	0.323946	$9.81 \cdot 10^{-9}$
80	0.323888	$5.81 \cdot 10^{-5}$	0.323946	$6.13 \cdot 10^{-10}$

The results with the transformed integral behave as we saw on page 170, corresponding to $R_{\mathrm{T}} = O(h^4)$. With the original formulation we get the same poor gain ratio as with the trapezoidal rule, $R_{\mathrm{T}} = O(h^{1.5})$. ∎

Partial integration can sometimes simplify the integral.

$$I = \int_0^{0.64} \frac{\arctan x}{\sqrt{x}}\,dx = \left[2\sqrt{x}\arctan x\right]_0^{0.64} - \int_0^{0.64} \frac{2\sqrt{x}}{1+x^2}\,dx$$
$$= 1.6\arctan 0.64 - \int_0^{0.8} \frac{4t^2}{1+t^4}\,dt\ .$$

In the rewriting of the remaining integral we again used the change of variables $t = \sqrt{x}$, to get rid of the singularity of the derivative at $x = 0$. The last integral can easily be computed numerically (and it is even possible to give an explicit expression for the primitive function).

"Subtraction of the singularity" means that a function with the same type of singularity and with a known primitive function is subtracted from the integrand. In our example we get

$$I = \int_0^{0.64} \frac{\arctan x - x}{\sqrt{x}}\,dx \ + \ \int_0^{0.64} \frac{x}{\sqrt{x}}\,dx$$
$$= \int_0^{0.64} \frac{\arctan x - x}{\sqrt{x}}\,dx \ + \ \frac{1.024}{3}\ .$$

The new integrand is of the form $-\frac{1}{3}x^{5/2} + O(x^{9/2})$. This implies that its third and higher derivatives are singular at $x = 0$, and Simpson's rule can be expected to give poor accuracy (but better than the original formulation). Also, for small values of x there is cancellation in the numerator of the integrand. We return to this formulation in the example starting on page 182.

Series expansion and termwise integration sometimes works well. In our case we get

$$I = \int_0^{0.64} \sum_{r=0}^{\infty} (-1)^r \frac{x^{(2r+1)-1/2}}{2r+1}\,dx \ = \ \sum_{r=0}^{\infty} (-1)^r \frac{0.64^{2r+1.5}}{(2r+1)(2r+1.5)}\ .$$

Using the remainder term estimate from Section 3.2 we see that if we eg stop the summation after $r=9$, then the error is at most $1.51 \cdot 10^{-7}$.

If the integrand itself has a singularity, we cannot use a numerical method based on the trapezoidal rule, but there are methods for numerical integration that do not use endpoint values of the integrand. Sometimes one of the above reformulations may be used to get rid of the singularity.

Example. In the integral

$$I = \int_0^1 \frac{\arctan x}{x^{3/2}}\, dx$$

the integrand has a singularity at $x=0$. The change of variables $t = \sqrt{x}$ removes it:

$$I = \int_0^1 \frac{2\arctan t^2}{t^2}\, dt\ .$$

We leave it as an exercise to show that also subtraction of the singularity will make the integral amenable to the use of the trapezoidal rule. ∎

Another type of difficulty arises if the interval of integration is infinite. Sometimes a change of variables can be used to change the infinite interval into a finite interval; eg the interval $[0,\infty]$ is changed into $[0,1]$ by the substitutions $x=(1-t)/t$ or $x=-\log t$.

In some cases one gets a good approximation by *"cutting off the tail"*. If, eg, $I = \int_0^\infty e^{-x^2}\, dx$ is approximated by $I = \int_0^a e^{-x^2}\, dx$, we make a truncation error

$$R_{\mathrm{T}} = \int_a^\infty e^{-x^2}\, dx < \frac{1}{a}\int_a^\infty x e^{-x^2}\, dx = \frac{1}{2a}e^{-a^2}\ .$$

For $a=6$ we have $R_{\mathrm{T}} < 2\cdot 10^{-17}$. The exact value is $I = \frac{1}{2}\sqrt{\pi} \simeq 0.886$, so $R_{\mathrm{T}} < \mu I$, where $\mu = 2^{-53}$ is the unit roundoff in IEEE double precision.

Finally, we shall discuss the use of Romberg's method when applied to a problem like (7.5.1). Romberg's method assumes that the truncation error for the trapezoidal method has an expansion in even powers of h, cf page 171. This condition is not satisfied, eg, if some of the derivatives of f have singularities at one of the endpoints. As examples of this, consider $f(x) = (x-a)^\alpha g(x)$ or $f(x) = (b-x)^\alpha g(x)$ with $0<\alpha<1$. It can be shown that if g is $(k+1)$ times continuously differentiable in $[a,b]$, then the truncation error of $T(h)$ is

$$R_{\mathrm{T}} = \sum_{i=1}^{k} a_i h^{2i} + \sum_{j=1}^{2k} b_j h^{j+\alpha} + O(h^{2k+1})\ .$$

The integrand in (7.5.1) has the form

$$f(x) = x^{1/2} g(x), \quad g(x) = \frac{\arctan x}{x} \ ,$$

and according to (7.5.2) there is no problems about singularity in any derivative of g. Thus, we have a problem of the type discussed above, with $\alpha = 0.5$, and the truncation error for the trapezoidal rule is

$$R_{\mathrm{T}} = b_1 h^{1.5} + a_1 h^2 + b_2 h^{2.5} + b_3 h^{3.5} + a_2 h^4 + b_4 h^{4.5} + \cdots \ .$$

This shows that $R_{\mathrm{T}} = O(h^{1.5})$ as we saw in the example on page 177.

Example. We use `romberg` from page 172 on the two versions of the integral,

$$I = \int_0^{0.64} \frac{\arctan x}{\sqrt{x}} \, dx \ = \ \int_0^{0.8} 2 \arctan t^2 \, dt \ ,$$

as implemented in `intf1` and `intf2` on page 177.

```
>> [I1, info1] = romberg(@intf1,0,0.64,1e-6)
I1 = 0.32392208774273
info1 = 1.57e-05    256   8

>> [I2, info2] = romberg(@intf2,0,0.8,1e-6)
I2 = 0.32394633528981
info2 = 7.43e-07     16   3
```

So, with the original formulation we use 257 function evaluations, and do not achieve the desired accuracy. After the transformation we get the desired results after 17 function evaluations.

`romberg` is prepared to handle the form of R_{T} given above:

```
>> p = [1.5 2 2.5 3.5 4 4.5 5.5 6];
>> [I3, info3] = romberg(@intf1,0,1,1e-6,p)
I3 = 0.32394627972287
info3 = 5.57e-07     32   3
```

Thus, the desired accuracy is obtained after 33 function evaluations. ∎

7.6. Adaptive Quadrature

Our goal is to compute a sufficiently accurate approximation of the integral, using as few function evaluations as possible. If the integrand has a large fourth derivative in part of the interval of integration, then a small step length h is needed in this part, in order to get a satisfactory accuracy

with Simpson's rule. In other parts of the interval it may be possible to get the same accuracy with a much larger h-value. In order not to make unnecessarily many function evaluations one should adjust the step length h to the behaviour of the integrand. This is done in adaptive quadrature.

We shall describe a method based on the *"divide-and-conquer"* principle: If the length of an interval is too large, then split it in the middle and apply the method separately in the two half intervals. More specific, let $[a, b]$, $h = b - a$ and $m = (a + b)/2$ denote an interval, its length and its midpoint. Use Simpson's rule with step length $h/2$ to find an approximation $I_1 \simeq \int_a^b f(x)\,dx$ and with step length $h/4$ to find $I_L \simeq \int_a^m f(x)\,dx$ and $I_R \simeq \int_m^b f(x)\,dx$. Next, compute

$$I_2 = I_L + I_R, \qquad I_3 = I_2 + \frac{I_2 - I_1}{15} \ .$$

Simpson's rule has error $O(h^4)$, and the value I_3 is obtained by Richardson extrapolation. The difference $I_2 - I_3$ is used as an estimate of the truncation error in I_2, and I_3 is accepted as an even better approximation if

$$|I_2 - I_3| \le \tau \ ,$$

where τ is the desired absolute accuracy. If this condition is not satisfied, then the process is repeated on the two subintervals $[a, m]$ and $[m, b]$ with $\tau := \frac{1}{2}\tau$.

Example. The following MATLAB function implements this algorithm for adaptive quadrature. The divide-and-conquer idea is conveniently handled by a recursive function, `adpi`, and `adaptint` itself merely acts as an initializer and caller of `adpi`. When this function is called with the interval $[a, b]$, we already have computed the function values $f(a)$, $f(b)$ and $f(m)$, and the value I_1, and these values are part of the input so that we avoid unnecessary recomputation of them.

```
function  [I, fcnt] = adaptint(f,a,b,tol)
% Integral of function f over interval [a,b]
% tol : desired absolute accuracy.
% fcnt: number of function evaluations

% Initialize.  End- and midpoint values of integrand
ff = feval(f,[a  (a+b)/2  b]);  fcnt = 3;
% Initial Simpson approximation
I1 = (b-a) * (ff(1) + 4*ff(2) + ff(3)) / 6;
% Recursive computation
[I,fcnt] = adpi(f,a,b,tol,ff,I1,fcnt);
```

```
% Auxiliary function
function  [I,fcnt] = adpi(f,a,b,tol,ff,I1,fcnt)
% Check contribution from (sub)interval [a,b]
h = (b-a)/2;  m = (a+b)/2;
% Mid point values in half intervals
fm = feval(f, [(a+m)/2  (m+b)/2]);  fcnt = fcnt + 2;
IL = h*(ff(1) + 4*fm(1) + ff(2))/6;   % Left half interval
IR = h*(ff(2) + 4*fm(2) + ff(3))/6;  % Right half interval
% Refined approximation with extrapolation
I2 = IL + IR;
I = I2 + (I2 - I1)/15;
% Check accuracy
if  abs(I-I2) > tol
  % Refine both subintervals
  [IL,fcnt] = adpi(f,a,m,tol/2,[ff(1) fm(1) ff(2)],IL,fcnt);
  [IR,fcnt] = adpi(f,m,b,tol/2,[ff(2) fm(2) ff(3)],IR,fcnt);
  I = IL + IR;
end
```

■

Example. The standard MATLAB function `quad` is basically the same algorithm as `adaptint`. We shall use it to approximate the integral (7.5.1),

$$I = \int_0^{0.64} \frac{\arctan x}{\sqrt{x}}\, dx \ .$$

Both `adaptint` and `quad` call a function `f` that implements the integrand. They use multiple arguments, so we cannot use the implementation `intf1` given on page 177, but we can, eg, use the following implementation, which also plots the computed points $(x, f(x))$.

```
function  y = intf3(x)
y = zeros(size(x));
i = find(x > 0);       % indices of nonzero x-values
y(i) = atan(x(i)) ./ sqrt(x(i));
plot(x,y,'.')

>> [Iq, fcq] = quad(@intf3,0,0.64,1e-6)
Iq = 0.59579899706727
fcq = 37
```

The distribution of the 37 points needed is shown in Figure 7.4 below. It is clearly seen that very small steps are needed close to the left hand endpoint, where the derivative of the integrand has a singularity.

In Section 7.5 we showed that by subtraction of the singularity the integral can be rewritten to

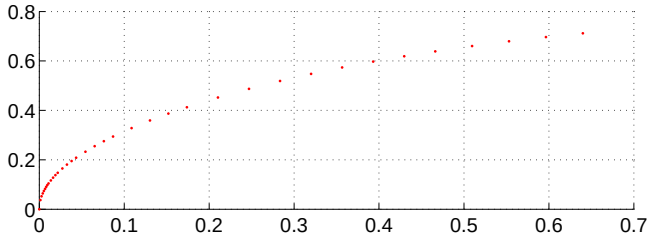


Figure 7.4. *Adaptive quadrature.* $f(x) = \arctan x/\sqrt{x}$.

$$I \;=\; \int_0^{0.64} \frac{\arctan x - x}{\sqrt{x}}\, dx \;+\; \frac{1.024}{3}\;.$$

If we use this formulation with `quad`, we get

```
function  y = intf4(x)
y = zeros(size(x));
i = find(x > 0);
y(i) = (atan(x(i)) - x(i)) ./ sqrt(x(i));

>> [Iss, fcs] = quad(@intf4,0,0.64,1e-6);
>> Iss = Iss + 1.024/3
Iss = 0.32394636665948
fcs = 13
```

The errors are $|\texttt{Iq} - I| = 3.63 \cdot 10^{-6}$ and $|\texttt{Iss} - I| = 3.85 \cdot 10^{-8}$. This illustrates again that it pays to simplify the problem before one uses a numerical method; with less than half the number of function evaluations we get a result whose error is reduced by a factor about 100. ∎

Exercises

E1. Use Romberg's method with $h = 1$ and $h = \frac{1}{2}$ to approximate $\int_0^1 x^3\, dx$. Explain why the result agrees with the exact value.

E2. Use "subtraction of the singularity" to remove the singularity at $x = 0$ in the integral

$$\int_0^1 \frac{\arctan x}{x^{3/2}}\, dx\;.$$

E3. Show that the substitution $x = \sin t$ makes the integral

$$\int_0^1 \sqrt{\frac{1 - \frac{1}{4}x^2}{1 - x^2}}\, dx$$

suitable for numerical integration.

E4. Suggest different methods for the numerical computation of

$$\int_0^1 \frac{\sin x}{\sqrt{1 - x^2}}\, dx \ .$$

E5. Suggest a suitable method for approximating

$$\int_0^\infty e^{-x} \cos x^2\, dx$$

with four correct decimals.

E6. Suggest a suitable method for the numerical computation of

$$I = \int_0^\infty \frac{e^{-x}}{1 + xe^{-x}}\, dx \ .$$

Computer Exercises

C1. Use the MATLAB function `romberg` (available from `incbox`) with `tol = 5e-5` to compute the following integrals. In some of the cases you must rewrite the integral before you can use `romberg`.

(a) $\displaystyle\int_0^1 e^{-x^2}\, dx$,

(b) $\displaystyle\int_0^{\pi/2} \sqrt{1 + \cos^2 x}\, dx$,

(c) $\displaystyle\int_0^1 \frac{\cos x}{\sqrt{x}}\, dx$,

(d) $\displaystyle\int_0^{10} \frac{x^3}{e^x - 1}\, dx$,

(e) $\displaystyle\int_0^1 \frac{32}{1 + 1024x^2}\, dx$. (The exact value is $\arctan 32$). Do the extrapolated values have better accuracy than the results from the trapezoidal rule? What is the result from `quad` ?

(f) $I = \int_a^{a+1} (\cos(8\pi x) + 1)\, dx$, $a = 0.125, 0.25$.
The exact value is $I = 1$ for any a. Explain your results!
What are the results from `quad` ?

C2. We shall compute the integral

$$I = \int_0^4 \frac{1}{1 + 5xe^{x^2}}\, dx \ .$$

Sketch the integrand and use `romberg` with `tol = 5e-6` on suitable subintervals. Compare with the results from `quad` .

C3. Use `adaptint` and `quad` to approximate the integral

$$I = \int_0^{10} \left(\sqrt{x + 10^{-8}} + \frac{4}{1 + 2(x-9)^2} \right) dx$$

for `tol` $= 10^{-1}, 10^{-2}, \ldots, 10^{-10}$.

Discuss the results. The true value of the integral is

$$I = \frac{2}{3}\big((10 + 10^{-8})^{3/2} - 10^{-12}\big) + 2\sqrt{2}\big(\arctan(\sqrt{2}) + \arctan(9\sqrt{2})\big) \ .$$

References

For a more extensive treatment of numerical integration, including extrapolation methods, we refer to

J. Stoer, R. Bulirsch, *Introduction to Numerical Analysis*, 3rd Edition, Springer Verlag, 2002.

and the monograph

P.J. Davis, P. Rabinowitz, *Methods of Numerical Integration*, 2nd Edition, Academic Press, New York, 1984.

An up-to-date discussion of adaptive quadrature is given in

W. Gander, W. Gautschi, *Adaptive Quadrature – Revisited*, BIT **40**, pp 84–101, 2000.

Chapter 8

Linear Systems of Equations

8.1. Introduction

Linear systems of equations arise very often in technical and natural sciences computation. The system may eg come from discretization of a boundary value problem for an ordinary or a partial differential equation, see Chapters 1 and 10. Such problems occur eg in structural mechanics and fluid dynamics, and the number of unknowns may be very large, occasionally hundreds of thousands.

In this chapter we study the numerical solution of linear systems of equations with n unknowns,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \ , \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \ , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \ . \end{aligned}$$

All the coefficients a_{ij} and right hand side elements b_i are assumed to be real. We often use matrix notation for the problem,

$$Ax = b \ ,$$

where A is the $n \times n$ coefficient matrix and b is the $n \times 1$ right hand side vector

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} , \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \ .$$

It may be convenient to interpret the matrix A as being built by n column vectors:

$$A = [a_{:1}\ a_{:2}\ \cdots\ a_{:n}]\ , \qquad a_{:j} = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{nj} \end{pmatrix}\ .$$

Then we can write the system $Ax = b$ in the form

$$x_1 a_{:1} + x_2 a_{:2} + \cdots x_n a_{:n} = b\ ,$$

and the problem of solving the system of equations is seen to be equivalent to finding a linear combination of the vectors $a_{:1}, \ldots, a_{:n}$, which is equal to the vector b. This shows that the system is solvable for any b if and only if the column vectors form a *basis* of the vector space $\mathbb{R}^n$. Put another way, the column vectors should be *linearly independent*. When this condition is satisfied, the matrix A is said to be *nonsingular*, and the system $Ax = b$ has a unique solution. Unless otherwise stated, we assume in this chapter that A is nonsingular.

There are two classes of methods for solving linear systems of equations, *direct* and *iterative methods*. With a direct method the solution is found via a finite number of simple arithmetic operations, and in the absence of rounding errors the computed x would be the exact solution to the system. An iterative method generates a sequence of vectors $x^{[0]}, x^{[1]}, x^{[2]}, \ldots$, that converge to the solution. It is outside the scope of this chapter to discuss this class of methods.

The frequent occurrence of linear systems of equations makes it important to be able to solve them fast and accurately, and high quality computer software for this task has been developed in the recent decades. In a programming language like Fortran or C one calls subroutines from a subroutine library. The best known and by far the best library for linear algebra is LAPACK, which is available for free via the Internet[1)]. Today there exist interactive environments for numerical calculations, eg Matlab, Maple and Matematica, where extensive matrix computations are performed by simple commands. The matrix algorithms in Matlab are taken from LAPACK.

[1)] URL: http://www.netlib.org/lapack/

Example. The following MATLAB script builds a matrix and right hand side

$$A = \begin{pmatrix} 2 & 3 & 4 \\ 1 & 3 & 1 \\ 1 & 1 & 8 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix},$$

and solves the system $Ax = b$.

```
A = [2 3 4; 1 3 1; 1 1 8];
b = [1; 2; 3];
format long    % display 15 digits
x = A\b

x = -2.58823529411765
     1.35294117647059
     0.52941176470588
```

■

As demonstrated, it is easy to solve a linear system of equations – at least in an environment like MATLAB. Therefore, it is relevant to ask whether it is necessary to know what happens when the MATLAB command `x = A\b` is executed. In technical and natural sciences applications the systems are often very large, and the linear system may be just a step in the solution of a nonlinear problem (by means of Newton-Raphson's method, see Section 4.8). Further, it sometimes happens that the matrix is singular or close to being singular, in which case rounding errors may have dominating effect. In order to be able to make good use of the powerful software (as available from the program library or in the interactive environment) in such a complex setting, it is necessary to know the properties of the algorithms.

Gaussian elimination is the basic direct method for solving linear systems of equations, and this is the method implemented[2)] in `x = A\b`. The method consists of a series of simple transformations that result in a triangular matrix. The same transformations are applied to the right hand side:

$$\left(\, A \;\middle|\; b \,\right) \;\rightarrow\; \left(\, U \;\middle|\; c \,\right),$$

where U is upper (or right) triangular. The transformations do not change the solution, so $Ax = b$ has the same solution as $Ux = c$, and x is easily computed by back substitution, see Section 8.2. In Section 8.3 we study the transformation $\left(\, A \;\middle|\; b \,\right) \rightarrow \left(\, U \;\middle|\; c \,\right)$.

[2)] In the case of a square linear system. This is discussed in examples on pages 219 and 257.

8.2. Triangular Systems

An $n\times n$ matrix $U = (u_{ij})$ is said to be *upper* (or *right*) *triangular* if

$$u_{ij} = 0 \ , \quad i > j \ ,$$

and an $n\times n$ matrix $L = (l_{ij})$ is said to be *lower* (or *left*) *triangular* if

$$l_{ij} = 0 \ , \quad i < j \ .$$

The matrix U is nonsingular if and only if all the diagonal elements u_{ii} are non-zero. In that case the system

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1n}x_n &= c_1 \\ u_{22}x_2 + \cdots + u_{2n}x_n &= c_2 \\ &\vdots \\ u_{nn}x_n &= c_n \end{aligned}$$

can be solved by *back substitution*:

$$\begin{aligned} x_n &= c_n/u_{nn} \\ x_i &= \big(c_i - \sum_{j=i+1}^{n} u_{ij}x_j\big)/u_{ii} \ , \quad i = n{-}1, n{-}2, \ldots, 1 \ . \end{aligned} \tag{8.2.1}$$

Example. In MATLAB this can be expressed as

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

∎

Similarly, the matrix L is nonsingular if all the $l_{ii} \neq 0$. Then the system

$$\begin{aligned} l_{11}y_1 &= d_1 \\ l_{11}y_1 + l_{12}y_2 &= d_2 \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + \cdots + l_{nn}y_n &= d_n \end{aligned}$$

can be solved by *forward substitution*

$$y_1 = d_1/l_{11}$$
$$y_i = \big(d_i - \sum_{j=1}^{i-1} l_{ij}y_j\big)/l_{ii} \ , \quad i = 2, 3, \ldots, n \ . \tag{8.2.2}$$

It is important to know how much work is needed to solve a system of equations. A measure of the work is the number of *flops* (floating point arithmetic operations) needed. From (8.2.1) it follows that the computation of x_i involves $n - i$ multiplications and additions and one division. Thus, the total number of flops[3)] needed is

$$n + 2\sum_{i=1}^{n-1}(n-i) = n + 2\sum_{k=1}^{n-1} k \ = \ n + 2\,\frac{n(n-1)}{2} = n^2 \ .$$

It is easy to see that forward substitution involves the same amount of work. As we shall see in Section 8.6, Gaussian elimination involves matrices L, where all the $l_{ii} = 1$. Such matrices are said to be *unit lower triangular*. Then there is no division involved in (8.2.2), and the work reduces to $n^2 - n$ flops. For large values of n this difference is insignificant, and we conclude:

> The solution of a triangular system of equations with n unknowns needs approximately n^2 flops.

For comparison, this is also the work involved in the matrix-vector multiplication $v = Cu$, if C is triangular.

Example. The MATLAB code in the previous example can be shortened considerably by noting that the expression for x_i in (8.2.1) has the form

$$x_i = (c_i - d_i)/u_{ii} \ ,$$

where d_i is the *inner product* of the vectors $[u_{i,i+1}, \ldots, u_{in}]$ and $[x_{i+1}, \ldots, x_n]$. Assuming that `x` is a column vector, this inner product is obtained by the simple command `U(i,i+1:n)*x(i+1:n)`, and the five lines

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

[3)] Up to the mid 1990*s* a "flop" was defined as (*one multiplication and one addition*) **or** (*one division*). On modern computers the execution times for these three operations are (almost) identical, and it seems to be internationally agreed to give up the distinction. So nowadays each floating point operation is counted, and "new" flop counts are about twice the "old" counts.

can be replaced by the single line

```
x(i) = (c(i) - U(i,i+1 : n) * x(i+1 : n)) / U(i,i);
```

This is an example of MATLAB *vectorization*. It does not change the number of flops, but it gives shorter MATLAB programs that are easier to understand (once you get used to it).

The vectorization is incorporated in the following implementation of the back substitution algorithm as a MATLAB function

```
function  x = backsub(U, c)
% Back substitution to solve Ux = c
n = length(c);
x(n,1) = c(n) / U(n,n);  % col. vector of length n
for  i = n-1 : -1 : 1
  x(i) = ( c(i) - U(i,i+1:n)*x(i+1:n) ) / U(i,i);
end
```

■

8.3. Gaussian Elimination

We start this section by looking at a small problem:

Example. Consider the system $Ax = b$, where A is a 3×3 matrix,

$$\begin{pmatrix} 5 & -5 & 10 \\ 2 & 0 & 8 \\ 1 & 1 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -25 \\ 6 \\ 9 \end{pmatrix} \qquad \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

The aim of Gaussian elimination is to transform the coefficient matrix to an upper triangular matrix, and this is achieved as follows: First x_1 is eliminated from Equations (2) and (3) by subtracting appropriate multiples of Equation (1). Omitting the x_j we get

$$\left(\begin{array}{ccc|c} 5 & -5 & 10 & -25 \\ 0 & 2 & 4 & 16 \\ 0 & 2 & 3 & 14 \end{array}\right) \qquad \begin{matrix} (1) \\ (2') = (2) - \frac{2}{5} \cdot (1) \\ (3') = (3) - \frac{1}{5} \cdot (1) \end{matrix}$$

Another way of describing these operations is: The elements in positions $(2,1)$ (ie 2nd row, 1st column) and $(3,1)$ are zeroed by subtracting multiples of the first row. We shall use this terminology henceforward.

Next, we zero the element in position $(3,2)$ by subtracting a multiple of the current 2nd row:

$$\left(\begin{array}{ccc|c} 5 & -5 & 10 & -25 \\ 0 & 2 & 4 & 16 \\ 0 & 0 & -1 & -2 \end{array}\right) \qquad \begin{matrix} (1) \\ (2') \\ (3'') = (3') - \frac{2}{2} \cdot (2') \end{matrix}$$

Now, we have achieved an upper triangular coefficient matrix, and back substitution gives the solution

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -5 \\ 4 \\ 2 \end{pmatrix} .$$

■

In the general case with n unknowns we start with

$$\left(A \mid b\right) = \left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right) .$$

First, we assume that $a_{11} \neq 0$ and zero elements in rows 2 through n of the first column by subtracting multiples of the first row:

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a'_{22} & \cdots & a'_{2n} & b'_2 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & a'_{n2} & \cdots & a'_{nn} & b'_n \end{array}\right) .$$

The elements with a prime were changed in this step, but we do not give details until later.

Next, we assume that $a'_{22} \neq 0$ and zero elements under the main diagonal in the second column by subtracting multiples of the second row, etc. After $k-1$ steps the matrix has the form

$$\left(\begin{array}{ccccccc|c} a_{11} & a_{12} & & & \cdots & & a_{1n} & b_1 \\ & a_{22} & & & \cdots & & a_{2n} & b_2 \\ & & \ddots & & & & \vdots & \vdots \\ & & & a_{kk} & a_{k,k+1} & \cdots & a_{kn} & b_k \\ & & & \vdots & \vdots & & \vdots & \vdots \\ & & & a_{ik} & a_{i,k+1} & \cdots & a_{in} & b_i \\ & & & \vdots & \vdots & & \vdots & \vdots \\ & & & a_{nk} & a_{n,k+1} & \cdots & a_{nn} & b_n \end{array}\right) .$$

For the sake of readability the notation does not show that the elements have been changed as compared to the original a_{ij} and b_i.

Now, assume that $a_{kk} \neq 0$. The elements in column k under the main diagonal are zeroed by subtracting multiples of row k. The result is

$$\left(\begin{array}{ccccccc|c}
a_{11} & a_{12} & & & \cdots & & a_{1n} & b_1 \\
 & a_{22} & & & \cdots & & a_{2n} & b_2 \\
 & & \ddots & & & & \vdots & \vdots \\
 & & & a_{kk} & a_{k,k+1} & \cdots & a_{kn} & b_k \\
 & & & \vdots & \vdots & & \vdots & \vdots \\
 & & & 0 & a'_{i,k+1} & \cdots & a'_{in} & b'_i \\
 & & & \vdots & \vdots & & \vdots & \vdots \\
 & & & 0 & a'_{n,k+1} & \cdots & a'_{nn} & b'_n
\end{array}\right),$$

where the multipliers and transformed elements are

$$\left.\begin{array}{rcl}
m_{ik} & = & a_{ik}/a_{kk} \\
a'_{ij} & = & a_{ij} - m_{ik}a_{kj}, \quad j = k{+}1, \ldots, n \\
b'_i & = & b_i - m_{ik}b_k
\end{array}\right\} \quad i = k{+}1, \ldots, n\ . \quad (8.3.1)$$

The computation

$$a'_{ik} = a_{ik} - m_{ik}a_{kk} = a_{ik} - \frac{a_{ik}}{a_{kk}}\, a_{kk} = 0$$

shows that this transformation does indeed put the desired zeros in the kth column.

The row used to get zeros in the kth column is called the kth *pivot row*, and a_{kk} is the kth *pivot element* (or just *pivot*).

The algorithm consists in repeating (8.3.1) for $k = 1, 2, \ldots, n{-}1$. Then the system $Ax = b$ has been changed to $Ux = c$, where the elements of the upper triangular U are the final values of $a_{ij},\ j \geq i$, and the elements in c are the final values of the elements in b.

Example. The following MATLAB program implements the elimination.

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

After executing the program the elements of U are found in the upper triangle of `A` and `b` contains c. ∎

We now consider the work involved in the transformation from $Ax = b$ to $Ux = c$ when there are n unknowns: The kth step of the transformation involves $n-k$ rows, and for each row we need one division (to get m_{ik}) and $(n-k+1)$ multiplications and additions. We are interested only in the order of magnitude of the work, and this amounts to about $2(n-k)^2$ flops in the kth step, and a total of

$$2\sum_{k=1}^{n-1}(n-k)^2 = 2\sum_{\nu=1}^{n-1}\nu^2 \ .$$

We need the following lemma

Lemma 8.3.1.
$$S_n \ \equiv \ \sum_{\nu=1}^{n-1}\nu^2 \ = \ \frac{n(n-1)(2n-1)}{6} \ .$$

Proof. By induction: The formula is obviously true for $n = 1$. Assume that it is true for $n = N$. Then

$$\begin{aligned} S_{N+1} &= S_N + N^2 = \frac{N}{6}\big(2N^2{-}3N{+}1 + 6N\big) \\ &= \frac{N}{6}(2N+1)(N+1) = \ \frac{(N{+}1)N(2(N{+}1)-1)}{6}, \end{aligned}$$

and the lemma is proved. □

The term with n^3 dominates for large values of n, so we have shown the following rule of thumb.

The transformation to triangular form of a linear $n{\times}n$ system of equations by Gaussian elimination requires approximately $\frac{2}{3}\,n^3$ flops.

Example. Suppose that one flop takes 2 ns (ie $2 \cdot 10^{-9}$ seconds) on a certain computer. The following table gives the execution time for Gaussian elimination and back substitution for varying n. The values are based on the estimates given in our rules of thumb.

n	Elimination	Back substitution
100	0.002	10^{-5}
1000	2	0.001
10 000	2000($\approx$ 33min)	0.1

Time in seconds when one flop takes 2 ns.

■

These timings are not to be taken too seriously, since they do not take into account that the operating system "steals" some time and that the computer architecture (the cache and memory) plays an important role when the matrix is large. Very big matrices cannot be held in cache (a matrix of order 10 000 needs about 800 Mbytes of memory). Nevertheless the estimates give quite reliable information about the ratio between execution times for Gaussian elimination and the solution of triangular systems, and also about how the execution time grows with the size of the problem. If, eg, a system of order n is solved in 2 seconds on a certain computer, then it takes about $3^3 \cdot 2 = 54$ seconds ≈ 1 minute to solve a system of order $3n$.

The estimates on pages 191 and 195 assume that (most of) the elements in the matrix are nonzero. In many applications the matrix may be very large, but most of the elements are zero. Such a matrix is said to be *sparse*, and in Section 8.8 we show how the work can be drastically reduced in the case of *band matrices*.

The presentation of Gaussian elimination assumed that all the pivots were nonzero. This condition is unrealistic, and in the next section we show how the algorithm can easily be modified to handle zero pivots. Also, the modified algorithm gives results that are less affected by rounding errors.

We round off this section by presenting some enhancements of the Matlab implementation given in the example on page 194.

Example. The transformation in (8.3.1) of the matrix elements can be expressed as $v'_{i:} = v_{i:} - m_{ik}v_{k:}$, where $v_{i:}$ is the row vector with elements $[a_{i,k+1}, \ldots, a_{in}]$. $v'_{i:}$ overwrites $v_{i:}$, and in Matlab we can use the command

```
>> A(i,k+1:n) = A(i,k+1:n) - M(i,k)*A(k,k+1:n)
```

Also note that the multiplier m_{ik} can overwrite a_{ik}, the element that is zeroed. Therefore we do not need the array `M`. The partly vectorized Matlab program is

```
for  k = 1 : n-1
  for  i = k+1 : n
    A(i,k) = A(i,k)/A(k,k);  % multiplier
    jj = k+1 : n;
    A(i,jj) = A(i,jj) - A(i,k)*A(k,jj);
    b(i) = b(i) - A(i,k)*b(k);
  end
end
```

The vectorization can be carried a step further when we realize that the change of matrix elements in (8.3.1) can be expressed as[4)]

$$\begin{pmatrix} a'_{k+1,k+1} & \cdots & a'_{k+1,n} \\ \vdots & & \vdots \\ a'_{n,k+1} & \cdots & a'_{n,n} \end{pmatrix}$$

$$= \begin{pmatrix} a_{k+1,k+1} & \cdots & a_{k+1,n} \\ \vdots & & \vdots \\ a_{n,k+1} & \cdots & a_{n,n} \end{pmatrix} - \begin{pmatrix} m_{k+1,k} \\ \vdots \\ m_{nk} \end{pmatrix} \begin{pmatrix} a_{k,k+1} & \cdots & a_{kn} \end{pmatrix} .$$

With this formulation we get a very compact program, which we present as a MATLAB function:

```
function  [U, c] = gauss1(A, b)
% Gaussian elimination applied to Ax = b
n = size(A,1);
b = b(:);  % ensure that b is a column vector
for  k = 1 : n-1
  ii = k+1 : n;
  A(ii,k) = A(ii,k)/A(k,k);  % multipliers
  A(ii,ii) = A(ii,ii) - A(ii,k)*A(k,ii);  % modify A
  b(ii) = b(ii) - b(k)*A(ii,k);  % modify b
end
U = triu(A);  c = b;  % return transformed problem
```

If `A` and `b` hold the matrix and right hand side of the system $Ax = b$, then the commands

```
>> [U,c] = gauss1(A,b);  x = backsub(U,c);
```

will return the solution in `x`. ∎

[4)] A product of the form (*column vector*)·(*row vector*) is a so-called *rank-one matrix*. We shall make further use of such matrices in Section 8.15. This product is also called an *outer product* in contrast to the *inner product* (introduced in the example on page 191) which is a scalar, computed as (*row vector*) · (*column vector*).

8.4. Pivoting

In the previous section we assumed that all the pivot elements in Gaussian elimination were nonzero. The system

$$\left(\begin{array}{cc|c} 0 & 1 & 1 \\ 1 & 1 & 2 \end{array}\right)$$

demonstrates that this assumption is not realistic. The algorithm from the previous section must be modified so that we start by interchanging rows 1 and 2:

$$\left(\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1 & 1 \end{array}\right) ,$$

and we are ready for back substitution.

The interchange of rows should also be done for a system of the form

$$\left(\begin{array}{cc|c} \epsilon & 1 & 1 \\ 1 & 1 & 2 \end{array}\right) , \tag{8.4.1}$$

where $|\epsilon|$ is nonzero, but small. The reason is that otherwise rounding errors may lead to severe loss of accuracy.

Example. Let $\epsilon = 10^{-5}$. The multiplier is $m_{21} = 10^5$, and the upper triangular system is

$$\left(\begin{array}{cc|c} 10^{-5} & 1 & 1 \\ 0 & 1-10^5 & 2-10^5 \end{array}\right) .$$

Suppose we use the floating point number system $(10, 3, -9, 9)$ (radix 10 and 3 digits in the fraction, see Section 2.4). Then

$$\begin{aligned} 1 - 10^5 &= (0.00001 - 1.000)10^5 \\ &= -0.99999 \cdot 10^5 = -9.9999 \cdot 10^4 \\ &\doteq -10.000 \cdot 10^4 = -1.000 \cdot 10^5 . \end{aligned}$$

("$\doteq$" reads "is rounded to". We have assumed that the registers hold 6 digits). Also $2 - 10^5 \doteq -10^5$, so the computed result of the elimination is

$$\left(\begin{array}{cc|c} 10^{-5} & 1 & 1 \\ 0 & -10^5 & -10^5 \end{array}\right) ,$$

with the solution

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} .$$

If we start by interchanging the two rows, we get $m_{21} = 10^{-5}$ and the reduced matrix

$$\left(\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1-10^{-5} & 1-2\cdot 10^{-5} \end{array}\right) \doteq \left(\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1 & 1 \end{array}\right) ,$$

with the solution

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \ .$$

Next, consider the same problem in more realistic settings: The table below shows the solution for different values of ϵ, as computed in MATLAB in IEEE double precision, ie with the floating point number system $(2, 52, -1022, 1023)$, and displayed with `format long`.

As an exercise, find the exact solution to the system and verify that in all cases the solution computed with row interchanges agrees with the computer representation of the exact solution. The quality of the solution computed without row interchange decreases as the parameter ϵ in (8.4.1) becomes smaller.

ϵ	without row interchange	with row interchange
10^{-8}	$\begin{pmatrix} 1.0000\,00005\,02476 \\ 0.9999\,99990\,00000 \end{pmatrix}$	$\begin{pmatrix} 1.0000\,00010\,00000 \\ 0.9999\,99990\,00000 \end{pmatrix}$
10^{-11}	$\begin{pmatrix} 1.0000\,00082\,74037 \\ 0.9999\,99999\,99000 \end{pmatrix}$	$\begin{pmatrix} 1.0000\,00000\,001000 \\ 0.9999\,99999\,99000 \end{pmatrix}$
10^{-14}	$\begin{pmatrix} 0.9992\,00722\,16264 \\ 0.9999\,99999\,99999 \end{pmatrix}$	$\begin{pmatrix} 1.0000\,00000\,00001 \\ 0.9999\,99999\,99999 \end{pmatrix}$
10^{-17}	$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$

■

In the first part of this simple example it is easy to see what caused the loss of accuracy: In the floating point computation of $1 - 10^5$ and $2 - 10^5$ we lost the information in the equation $x_1 + x_2 = 2$; the subtraction $a - 10^5$ gives the result -10^5 for all a in the range $-5 < a < 5$. Generally, if very large elements are generated during the elimination, then we loose part of the information given by the original coefficients. In the example the reduced matrix obtained after row interchange had elements of the same order of magnitude as the original matrix, and there was no loss of accuracy.

Row interchanges during Gaussian elimination is called *pivoting*. In order to reduce the loss of accuracy the row interchanges are made so that the pivot element is as large as possible (and thereby the maximum multiplier is as small as possible). Consider step k of the elimination:

$$\begin{pmatrix} a_{11} & a_{12} & & & \cdots & & a_{1n} \\ & a_{22} & & & \cdots & & a_{2n} \\ & & \ddots & & & & \vdots \\ & & & a_{kk} & a_{k,k+1} & \cdots & a_{kn} \\ & & & \vdots & \vdots & & \vdots \\ & & & a_{ik} & a_{i,k+1} & \cdots & a_{in} \\ & & & \vdots & \vdots & & \vdots \\ & & & a_{nk} & a_{n,k+1} & \cdots & a_{nn} \end{pmatrix} .$$

Column k, from a_{kk} and down, is searched to find the element of largest magnitude. More precisely, find the row index ν such that

$$|a_{\nu k}| = \max_{k \le i \le n} |a_{ik}| \ .$$

If $\nu > k$, then rows k and ν are interchanged, and the elimination proceeds. This process is called *partial pivoting*. It follows from (8.3.1), that with partial pivoting the multipliers satisfy

$$|m_{ik}| = \left| \frac{a_{ik}}{a_{kk}} \right| \ \le \ 1 \ . \tag{8.4.2}$$

In case of *complete pivoting* the kth pivot is found as the largest element in the submatrix

$$\begin{pmatrix} a_{kk} & \cdots & a_{kn} \\ \vdots & & \vdots \\ a_{nk} & \cdots & a_{nn} \end{pmatrix} ,$$

and both row and column interchanges are used to bring this element to position (k, k). This method is rarely used today.

Example. It is easy to change the function `gauss1` from page 197 so that it uses partial pivoting:

```
function  [U, c] = pgauss(A, b)
% Gaussian elimination with partial pivoting, applied to Ax = b
n = size(A,1);
b = b(:);                    % ensure that b is a column vector
for  k = 1 : n-1
  [A, b] = pivot(A, b, k);
  ii = k+1 : n;
  A(ii,k) = A(ii,k)/A(k,k);                        % multipliers
  A(ii,ii) = A(ii,ii) - A(ii,k)*A(k,ii);             % modify A
```

```
  b(ii) = b(ii) - b(k)*A(ii,k);                        % modify b
end
U = triu(A);  c = b;                % return transformed problem
```

The function `pivot` finds the pivot row and performs the interchanges in the matrix and right hand side.

```
function  [A, b] = pivot(A, b, k)
% Find k'th pivot row and interchange
n = size(A,1);
[piv, q] = max(abs(A(k:n,k)));
if  q > 1  % interchange
  pk = k-1 + q;                                % row index
  A([k pk],:) = A([pk k],:);      % swap rows pk and k
  b([k pk]) = b([pk k]);      % swap elements pk and k
end
```

■

The purpose of pivoting is to avoid that matrix elements become too large during the elimination, with associated loss of accuracy. Therefore, in general, one should use partial pivoting in connection with Gaussian elimination for solving a system $Ax = b$. There are, however, exceptions to this rule, viz if the matrix is

a) symmetric and positive definite, or

b) diagonally dominant.

The symmetric matrix A is said to be *positive definite* if

$$x^T A\, x > 0$$

for all vectors $x \neq 0$. The matrix is *diagonally dominant* if

$$|a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \ , \quad i = 1, 2, \ldots, n \ ,$$

with strict inequality for at least one i.

Both of these classes of matrices occur in important applications, eg in structural mechanics and in the discretization of boundary value problems for differential equations, and in connection with cubic splines we derived (5.11.6), which has a diagonally dominant matrix. It can be shown that Gaussian elimination without pivoting is *stable* for such matrices: The maximum element in the original A is larger than all elements generated during the elimination. It may happen that some of the multipliers are larger than 1, but there is no "unnecessary" loss of accuracy. We return to symmetric, positive definite matrices in Section 8.7.

Gaussian elimination with partial pivoting is illustrated by the following example, which we shall also use later.

Example. Suppose that we want to solve a system of equations $Ax = b$ with

$$A = \begin{pmatrix} 0.6 & 1.52 & 3.5 \\ 2 & 4 & 1 \\ 1 & 2.8 & 1 \end{pmatrix} .$$

In the first step we have to interchange rows 1 and 2:

$$\begin{pmatrix} 2 & 4 & 1 \\ 0.6 & 1.52 & 3.5 \\ 1 & 2.8 & 1 \end{pmatrix} .$$

The multipliers are $m_{21} = 0.3$, $m_{31} = 0.5$, and the result of the first step is

$$\begin{pmatrix} 2 & 4 & 1 \\ 0 & 0.32 & 3.2 \\ 0 & 0.8 & 0.5 \end{pmatrix} .$$

Now we have to interchange rows 2 and 3, since $|a_{32}| > |a_{22}|$,

$$\begin{pmatrix} 2 & 4 & 1 \\ 0 & 0.8 & 0.5 \\ 0 & 0.32 & 3.2 \end{pmatrix} ,$$

and with $m_{32} = 0.4$ we get the upper triangular matrix

$$U = \begin{pmatrix} 2 & 4 & 1 \\ 0 & 0.8 & 0.5 \\ 0 & 0 & 3 \end{pmatrix} .$$

■

The observant reader will have noticed that we did not include the right hand side in the example. However, we can save the information about the elimination process – which rows were interchanged and the multipliers, so it is possible to apply the same transformations to the right hand side. We formalize this approach in Section 8.6.

In the introduction to this chapter we saw that the system $Ax = b$ has a unique solution if the column vectors of A are linearly independent. We now ask: Does Gaussian elimination with partial pivoting always work, if this assumption is satisfied? (On page 198 we saw that this is not the case when we do not use pivoting).

From the description of the method it follows that the process breaks down in step k if all $a_{ik} = 0$ for $i \geq k$. This, however, can occur only if column $a_{:k}$ is a linear combination of $a_{:1}, \ldots, a_{:,k-1}$, as we will now show: Consider the system

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1,k-1} & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2,k-1} & a_{2k} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n,k-1} & a_{nk} \end{array}\right) .$$

By means of Gaussian elimination with partial pivoting we know that this matrix is transformed to

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1,k-1} & a_{1k} \\ & a_{22} & \cdots & a_{2,k-1} & a_{2k} \\ & & \ddots & \vdots & \vdots \\ & & & a_{k-1,k-1} & a_{k-1,k} \\ & & & 0 & 0 \\ & & & \vdots & \vdots \\ & & & 0 & 0 \end{array}\right) ,$$

where all $a_{ii} \neq 0,\ i = 1, \ldots, k-1$. This implies that we can find $\{\alpha_i\}_{i=1}^{k-1}$ so that $\alpha_1 a_{:1} + \cdots + \alpha_{k-1} a_{:,k-1} - a_{:k} = 0$, ie the first k columns are **not** linearly independent.

Thus, under the assumption of linearly independent columns in A, Gaussian elimination with partial pivoting has all the pivots nonzero. It should be mentioned that this result neglects effects of rounding errors.

8.5. Permutations, Partitioned Matrices and Gauss Transformations

In the next section we show that Gaussian elimination is equivalent to a factorization of the matrix. First, however, we must introduce some elementary transformation matrices.

A row interchange is equivalent to a multiplication from the left by a permutation matrix. The simplest such matrix P_{rs} is obtained by interchanging rows r and s in the unit matrix I. It differs from I in four positions; it has zeros in positions (r, r) and (s, s) and ones in positions (r, s) and (s, r). For $n = 5$ the permutation matrix P_{24} is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} .$$

We say that P_{rs} is an *simple permutation matrix*. If we multiply a vector x by P_{rs}, then the elements x_r and x_s are interchanged. Similarly,

$$P_{rs}A = P_{rs}[a_{:1}\ a_{:2}\ \cdots\ a_{:n}] \ = \ [P_{rs}a_{:1}\ P_{rs}a_{:2}\ \cdots\ P_{rs}a_{:n}]$$

shows that rows r and s are interchanged. With $r < s$ we

$$P_{rs}A = \begin{pmatrix} \vdots & \vdots & & \vdots \\ a_{r-1,1} & a_{r-1,2} & \cdots & a_{r-1,n} \\ a_{s,1} & a_{s,2} & \cdots & a_{s,n} \\ a_{r+1,1} & a_{r+1,2} & \cdots & a_{r+1,n} \\ \vdots & \vdots & & \vdots \\ a_{s-1,1} & a_{s-1,2} & \cdots & a_{s-1,n} \\ a_{r,1} & a_{r,2} & \cdots & a_{r,n} \\ a_{s+1,1} & a_{s+1,2} & \cdots & a_{s+1,n} \\ \vdots & \vdots & & \vdots \end{pmatrix} .$$

Finally, we note that a simple permutation matrix is symmetric and is its own inverse[5)]:

$$P_{rs} = P_{rs}^T = P_{rs}^{-1} \ . \tag{8.5.1}$$

The general definition of a *permutation matrix* is a matrix P such that Px is just a reordering of the components in x. Therefore, a product of elementary permutation matrices is a permutation matrix, and actually all permutation matrices can be constructed this way:

Proposition 8.5.1. Any permutation matrix can be written as a product of elementary permutation matrices.

Proof. Px defines an ordering of the elements in x. Let $(Px)_1 = x_{k_1}$, then $P_{1k_1}x$ brings x_{k_1} to the desired position. Next, let $(Px)_2 = (P_{1k_1}x)_{k_2}$, then $P_{2k_2}P_{1k_1}x$ has the correct elements in first two positions, etc. □

[5)] The last statement is verified by showing that $P_{rs}P_{rs} = I$; do that!

Proposition 8.5.2. Permutation matrices are orthogonal:

$$P^T P = P P^T = I \ .$$

Proof. Let $P = P_a P_b$, the product of two elementary permutation matrices. By use of (8.5.1) we get

$$P^T P = P_b^T P_a^T P_a \, P_b = P_b^T I \, P_b = I \ .$$

The proof is similar for PP^T and for more than two factors. □

The multiplication PA gives a permutation of the rows in A. The corresponding permutation of the columns in A is obtained by multiplication by P^T from the right, $A\,P^T$.

It is often convenient to *partition* a matrix, ie split it into blocks (submatrices). To illustrate this, consider the 3×3 matrix

$$A = \left(\begin{array}{cc|c} 7 & 5 & 0 \\ 3 & 6 & 1 \\ \hline 3 & 1 & 6 \end{array}\right) = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \ ,$$

with

$$A_{11} = \begin{pmatrix} 7 & 5 \\ 3 & 6 \end{pmatrix}, \quad A_{12} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad A_{21} = \begin{pmatrix} 3 & 1 \end{pmatrix}, \quad A_{22} = \begin{pmatrix} 6 \end{pmatrix} \ .$$

Similarly, let

$$B = \left(\begin{array}{cc|c} 9 & 5 & 4 \\ 0 & 2 & 1 \\ \hline 6 & 5 & 0 \end{array}\right) = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \ .$$

The matrix product AB can be interpreted as if the blocks were scalars:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix} .$$

With the given matrices A and B you should check that this expression gives the same result as you get with the ordinary scheme for multiplying two 3×3 matrices.

In general, if two quadratic matrices A and B are partitioned the same way,

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NN} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1N} \\ B_{21} & B_{22} & \cdots & B_{2N} \\ \vdots & \vdots & & \vdots \\ B_{N1} & B_{N2} & \cdots & B_{NN} \end{pmatrix},$$

then also the product $C = AB$ can be partitioned in this way, and

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \ ,$$

provided that all these products are defined – ie the number of columns in A_{ik} must equal the number of rows in B_{kj}.

In the next section we shall see that Gaussian elimination with partial pivoting can be expressed by permutation matrices and *Gauss transformations*. The Gauss transformation L_k is a lower triangular matrix that differs from the unit matrix I only in the kth column:

$$L_k = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & 1 & & \\ & & & m_{k+1} & 1 & \\ & & & \vdots & & \ddots \\ & & & m_n & & 1 \end{pmatrix} .$$

(The element m_i corresponds to the multiplier m_{ik} in Gaussian elimination). If we multiply a vector x by L_k, we get

$$y = L_k x = \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} + m_{k+1} x_k \\ \vdots \\ x_n + m_n x_k \end{pmatrix} ,$$

or, in other words,

$$y_i = \begin{cases} x_i & , \ i = 1, \ldots, k \ , \\ x_i + m_i x_k & , \ i = k{+}1, \ldots, n \ . \end{cases}$$

Therefore, $x = L_k^{-1} y$ must satisfy

$$x_i = \begin{cases} y_i & , \ i = 1, \ldots, k \ , \\ y_i - m_i y_k & , \ i = k{+}1, \ldots, n \ , \end{cases}$$

showing that it is simple to invert the Gauss transformation:

$$L_k^{-1} = \begin{pmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & -m_{k+1} & 1 & & \\ & & & \vdots & & \ddots & \\ & & & -m_n & & & 1 \end{pmatrix} .$$

A Gauss transformation can be used to zero elements in a vector: Let x be a vector with $x_k \neq 0$ and let L_k be given by

$$m_i = \frac{x_i}{x_k} \quad i = k{+}1, \ldots, n \ .$$

Then

$$L_k^{-1} x = \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} - m_{k+1} x_k \\ \vdots \\ x_n - m_n x_k \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{pmatrix} .$$

Finally, we give an example that illustrates the concepts introduced in this section, and generates two results, which are used in the next section.

Example. The Gauss transformation L_1 can be partitioned,

$$L_1 = \begin{pmatrix} 1 & 0 \\ m & I \end{pmatrix} ,$$

where the column vector m holds the $\{m_i\}_{i=2}^n$. Let

$$P = \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{P} \end{pmatrix} ,$$

where $\widetilde{P}$ is a permutation matrix of order $n{-}1$. Then

$$PL_1P^T = \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{P} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ m & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{P}^T \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \widetilde{P}m & \widetilde{P}\widetilde{P}^T \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \widetilde{P}m & I \end{pmatrix} .$$

Thus, the only difference between L_1 and PL_1P^T is that the elements in positions $(2,1), \ldots, (n,1)$ are permuted as defined by $\widetilde{P}$.

Next, consider the product $L_1 L_k$ of two Gauss transformations with $k > 1$:

$$L_1 L_k = \begin{pmatrix} 1 & 0 \\ m & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{L}_k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ m & \widetilde{L}_k \end{pmatrix} .$$

Here, $\widetilde{L}_k$ is the lower right $(n-1)\times(n-1)$ submatrix of L_k. The calculation shows that the difference between L_k and $L_1 L_k$ is that m appears under the main diagonal in the first column. ∎

8.6. LU Factorization

We shall show that Gaussian elimination with partial pivoting applied to a nonsingular matrix A is equivalent to the factorization

$$P\,A = L\,U\ , \tag{8.6.1}$$

where P is a permutation matrix, L is a unit lower triangular matrix, and U is an upper triangular matrix. We start by a small example.

Example. In the example on page 202 we applied Gaussian elimination with partial pivoting to the matrix

$$A = \begin{pmatrix} 0.6 & 1.52 & 3.5 \\ 2 & 4 & 1 \\ 1 & 2.8 & 1 \end{pmatrix} .$$

We shall go through the the elimination, using the concepts from the previous section.

First, we swap rows 1 and 2 by multiplying with a permutation matrix P_1 (equal to the elementary permutation matrix P_{12}) and zero the elements in positions $(2,1)$ and $(3,1)$ by multiplying with an inverse Gauss transformation L_1^{-1}:

$$A_1 = L_1^{-1} P_1 A, \quad P_1 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad L_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0.3 & 1 & 0 \\ 0.5 & 0 & 1 \end{pmatrix} .$$

In Matlab (with `A1` representing A_1 etc)

```
>> A1 = inv(L1)*P1*A
A1 =  2.0000    4.0000    1.0000
           0    0.3200    3.2000
           0    0.8000    0.5000
```

Next, P_2 corresponds to swapping rows 2 and 3, and L_2 is constructed to

zero the element in position $(3, 2)$:

$$A_2 = L_2^{-1} P_2 A_1, \quad P_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \quad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.4 & 1 \end{pmatrix} .$$

```
>> A2 = inv(L2)*P2*A1
A2 =  2.0000    4.0000    1.0000
           0    0.8000    0.5000
           0         0    3.0000
```

Let U denote the upper triangular matrix A_2, then

$$U = L_2^{-1} P_2 A_1 = L_2^{-1} P_2 L_1^{-1} P_1 A .$$

This is equivalent to $A = P_1^T L_1 \, P_2^T L_2 U$, and by multiplying by $P_2 P_1$ and using that $P_k^{-1} = P_k^T$ since P_k is orthogonal, we get

$$P_2 P_1 A = P_2 L_1 \, P_2^T L_2 U .$$

With regard to (8.6.1) we have found U, and

$$P = P_2 P_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

is the permutation matrix; it gives the final ordering of the rows. It remains to show that $L = P_2 L_1 P_2^T L_2$ is a unit lower triangular matrix. This, however, was done in the example on page 207: First we showed that the matrix $P_2 L_1 P_2^T$ is identical with L_1, except that the subdiagonal elements 0.3 and 0.5 are swapped,

```
>> P2*L1*P2'
ans =  1.0000         0         0
       0.5000    1.0000         0
       0.3000         0    1.0000
```

Next, we showed that multiplying this matrix with L_2 gives a matrix, which is identical to L_2, except for the first column, which is identical to the first column in $P_2 L_1 P_2^T$.

```
>> L = (P2*L1*P2')*L2
L =  1.0000         0         0
     0.5000    1.0000         0
     0.3000    0.4000    1.0000
```

Thus, we have shown that (8.6.1) is true in this case. ∎

The example demonstrates for a special case that Gaussian elimination with partial pivoting is equivalent to LU factorization of the original matrix with the rows permuted. Now we consider the general case:

Theorem 8.6.1. LU factorization. Every nonsingular $n \times n$ matrix A can be factored

$$P\,A = L\,U\ ,$$

where P is a permutation matrix, L is a unit lower triangular matrix, and U is an upper triangular matrix.

Proof. By induction. The theorem is trivially true for $n = 1$. Assume that it is true for $n = N-1$, and consider an $N \times N$ matrix A. The first step of Gaussian elimination with partial pivoting can be formulated as

$$A_1 = L_1^{-1} P_1 A\ ,$$

where P_1 is a permutation matrix and L_1 is a Gauss transformation,

$$L_1 = \begin{pmatrix} 1 & 0 \\ m & I \end{pmatrix}\ , \quad m = \begin{pmatrix} m_{21} \\ \vdots \\ m_{n1} \end{pmatrix}\ .$$

The result is

$$A_1 = \begin{pmatrix} u_{11} & u_1^T \\ 0 & \widetilde{A}_2 \end{pmatrix}\ ,$$

where $u_1^T = \begin{pmatrix} u_{12} & \cdots & u_{1n} \end{pmatrix}$ and $\widetilde{A}_2$ has order $N-1$. The induction assumption tells us that

$$\widetilde{P}_2 \widetilde{A}_2 = \widetilde{L}_2 \widetilde{U}_2\ ,$$

where all the matrices have order $N-1$. Now define the $N \times N$ matrices

$$P_2 = \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{P}_2 \end{pmatrix}, \quad L_2 = \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{L}_2 \end{pmatrix}, \quad U = \begin{pmatrix} u_{11} & u_1^T \\ 0 & \widetilde{U}_2 \end{pmatrix}\ .$$

Then

$$P_2 A_1 = \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{P}_2 \end{pmatrix} \begin{pmatrix} u_{11} & u_1^T \\ 0 & \widetilde{A}_2 \end{pmatrix} = \begin{pmatrix} u_{11} & u_1^T \\ 0 & \widetilde{P}_2 \widetilde{A}_2 \end{pmatrix}$$

$$= \begin{pmatrix} u_{11} & u_1^T \\ 0 & \widetilde{L}_2 \widetilde{U}_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{L}_2 \end{pmatrix} \begin{pmatrix} u_{11} & u_1^T \\ 0 & \widetilde{U}_2 \end{pmatrix} = L_2 U\ .$$

Combining this with $A_1 = L_1^{-1} P_1 A$ and $P_2^T P_2 = I$ we get

$$P_2 P_1 A = P_2 L_1 A_1 = P_2 L_1 P_2^T P_2 A_1 = P_2 L_1 P_2^T L_2 U\ .$$

Now we set $P = P_2 P_1$, and as in the example it is seen that $L = P_2 L_1 P_2^T L_2$ is a unit lower triangular matrix with

$$l_{:1} = \begin{pmatrix} 1 \\ \widetilde{P}_2 m \end{pmatrix} ,$$

and columns $l_{:2}, \ldots, l_{:n}$ equal to the same columns in L_2.
Thus, we have shown that also the $N \times N$ matrix A satisfies $PA = LU$, and the theorem is proved. □

The LU factorization is unique in the following sense:

Theorem 8.6.2. If the pivoting sequence is given (ie the permutation matrix P is given), then the factors L and U are unique.

Proof. The result is a direct consequence of the fact that LU factorization is equivalent to simple Gaussian elimination applied to the matrix PA. □

Once the LU factorization has been found, it is an easy task to solve the system $Ax = b$: It is equivalent to

$$PAx = LUx = Pb ,$$

and letting $y = Ux$, the system is solved in two steps:

$$\begin{aligned} &1. \quad \text{solve} \qquad Ly = Pb , \\ &2. \quad \text{solve} \qquad Ux = y . \end{aligned} \tag{8.6.2}$$

Both L and U are triangular, so the work is about $2n^2$ flops. For comparison, this is the same as the work needed to compute the matrix-vector product Az.

Example. We want to solve the system

$$\begin{pmatrix} 0.6 & 1.52 & 3.5 \\ 2 & 4 & 1 \\ 1 & 2.8 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8.3 \\ -3.8 \\ -2.3 \end{pmatrix} .$$

The LU factorization for the coefficient matrix was found in the previous example, and combining this with (8.6.2) we get

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.3 & 0.4 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 8.3 \\ -3.8 \\ -2.3 \end{pmatrix} = \begin{pmatrix} -3.8 \\ -2.3 \\ 8.3 \end{pmatrix} ,$$

which is solved by forward substitution,

$$y = \begin{pmatrix} -3.8 \\ -0.4 \\ 9.6 \end{pmatrix} .$$

Next, we use back substitution to solve the system

$$\begin{pmatrix} 2 & 4 & 1 \\ 0 & 0.8 & 0.5 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -3.8 \\ -0.4 \\ 9.6 \end{pmatrix}, \qquad x = \begin{pmatrix} 1.5 \\ -2.5 \\ 3.2 \end{pmatrix}.$$

■

Example. The MATLAB function `pgauss` from page 200 can easily be modified so that it returns the LU factorization of a matrix A. The permutations are represented by the vector `p`. This is initialized to $\begin{pmatrix} 1 & 2 & \dots & n \end{pmatrix}^T$, and the elements are interchanged together with the rows in A.

```
function  [L,U, p] = lufac(A)
% LU factorization of the matrix A, corresponding to Gaussian
% elimination with partial pivoting.
% p holds the final row ordering
n = size(A,1);
p = 1 : n;                                    % initial row ordering
for  k = 1 : n-1
  [A, p] = pivot(A, p, k);
  ii = k+1 : n;
  A(ii,k) = A(ii,k)/A(k,k);                            % multipliers
  A(ii,ii) = A(ii,ii) - A(ii,k)*A(k,ii);                % modify A
end
U = triu(A);                               % Resulting upper triangle
L = eye(n) + tril(A,-1);  % Multipliers in strictly lower triangle
```

The solution of the two triangular systems in (8.6.2) is implemented in the functions `luforw` (given below) and `backsub` from page 192.

```
function  b = luforw(L,p, b)
% Use information from LUFAC to transform right-hand side
n = size(L,1);
b = b(:);              % ensure that b is a column vector
b = b(p);                              % final row ordering
for  k = 1 : n-1
  ii = k+1 : n;
  b(ii) = b(ii) - b(k)*L(ii,k);             % transform b
end
```

For the matrix of the examples on pages 202, 208 and 211 we get the following results, that agree with the cited examples.

```
>> [L, U, p] = lufac([.6 1.52 3.5; 2 4 1; 1 2.8 1])
L = 1.0000         0         0
    0.5000    1.0000         0
    0.3000    0.4000    1.0000
U = 2.0000    4.0000    1.0000
         0    0.8000    0.5000
         0         0    3.0000
```

```
p =  2     3     1
>> y = luforw(L,p, [8.3 -3.8 -2.3])
y = -3.8000
    -0.4000
     9.6000
>> x = backsub(U, y)
x =  1.5000
    -2.5000
     3.2000
```

■

Example. MATLAB has a built-in function `lu` that computes the LU factorization of a matrix.

```
>> [L,U,P] = lu([.6 1.52 3.5; 2 4 1; 1 2.8 1])
```

returns the same `L` and `U` as in the previous example, while the row interchanges are represented by the permutation matrix

```
P =  0     1     0
     0     0     1
     1     0     0
```

Next, the solution via the two triangular systems in (8.6.2) can be implemented as follows,

```
>> b = [8.3; -3.8; -2.3];
>> y = L\(P*b);   x = U\y
x =  1.5000
    -2.5000
     3.2000
```

In the command `B\z` MATLAB recognizes if B is triangular, in which case the solution is found via forward or back substitution, as appropriate. In examples on pages 219 and 257 we discuss what happens if B is not triangular.

■

In some applications one has to solve a series of systems with the same coefficient matrix but different right hand sides,

$$Ax^{[k]} = b^{[k]}, \quad k = 1, 2, \ldots, K \ . \tag{8.6.3}$$

Instead of starting from scratch with each of the systems (which would cost about $K \cdot \frac{2}{3}\, n^3$ flops) one only has to factorize the matrix once, and then use (8.6.2) for each of the right hand sides. This reduces the work to about $\frac{2}{3}\, n^3 + 2Kn^2$ flops.

8.7. Symmetric, Positive Definite Matrices

We already claimed that in the case of a symmetric, positive definite matrix (an *spd* matrix) one does not get "unnecessary" loss of accuracy by using Gaussian elimination without pivoting. In this section we first show that a symmetric factorization of the matrix can be computed (without pivoting), ie all the "natural" pivots a_{kk} are nonzero. Next, we show that (in a certain sense) the matrix elements do not grow during the process. Because symmetry is preserved, the work involved in the factorization is halved as compared to a general matrix.

We start with a lemma about *spd* matrices:

Lemma 8.7.1. If A is an *spd* matrix, then all its diagonal elements are positive and the largest element is on the diagonal:

$$a_{kk} > 0, \quad k = 1, \ldots, n \qquad \text{and} \qquad \max_{i,j} |a_{ij}| = \max_{k} \{a_{kk}\} \ .$$

Proof. An *spd* matrix satisfies

$$x^T A\, x > 0 \quad \text{for all} \quad x \neq 0 \ . \tag{8.7.1}$$

Let e_k denote the vector given as the kth column vector in the unit matrix I,

$$(e_k)_i = \begin{cases} 0 \ , & i \neq k \ . \\ 1 \ , & i = k \ . \end{cases}$$

If we use this vector for x in the condition (8.7.1), we get

$$0 < e_k^T A\, e_k = e_k^T a_{:k} = a_{kk} \ ,$$

and we have proved the first part of the lemma. Next,

$$0 < (e_i{+}e_j)^T A\, (e_i{+}e_j) = a_{ii} + a_{ij} + a_{ji} + a_{jj} = a_{ii} + a_{jj} + 2a_{ij} \ .$$

Here we used the symmetry: $a_{ij} = a_{ji}$. Similarly we get

$$0 < (e_i{-}e_j)^T A\, (e_i{-}e_j) = a_{ii} + a_{jj} - 2a_{ij} \ .$$

Combining the last two inequalities we see that

$$|a_{ij}| < \tfrac{1}{2}(a_{ii} + a_{jj}) \leq \max\{a_{ii}, a_{jj}\} \ .$$

□

Next, we prove the main result of this section:

Theorem 8.7.2. LDL^T factorization. Every *spd* matrix A can be factored symmetrically without pivoting: $A = LDL^T$, where L is a unit lower triangular matrix and D is a diagonal matrix with positive diagonal elements.

Proof. By induction. The theorem is trivially true for $n = 1$. Assume that it is true for $n = N-1$, and partition the $N \times N$ matrix A,

$$A = \begin{pmatrix} a_{11} & a_1^T \\ a_1 & A_2 \end{pmatrix}, \quad a_1 = \begin{pmatrix} a_{21} \\ \vdots \\ a_{n1} \end{pmatrix} .$$

Since A is *spd*, Lemma 8.7.1 guarantees that $a_{11} > 0$, so it can be used as the first pivot. The corresponding Gauss transformation is

$$L_1 = \begin{pmatrix} 1 & 0 \\ m_1 & I \end{pmatrix}, \quad m_1 = \frac{1}{a_{11}} a_1 ,$$

and the transformed matrix is

$$A_1 = L_1^{-1} A = \begin{pmatrix} a_{11} & a_1^T \\ 0 & \widetilde{A}_2 \end{pmatrix} , \tag{8.7.2}$$

with

$$\widetilde{A}_2 = A_2 - m_1 a_1^T = A_2 - \frac{1}{a_{11}} a_1 a_1^T . \tag{8.7.3}$$

We can rewrite A_1,

$$A_1 = \begin{pmatrix} a_{11} & a_1^T \\ 0 & \widetilde{A}_2 \end{pmatrix} = \begin{pmatrix} a_{11} & 0 \\ 0 & \widetilde{A}_2 \end{pmatrix} \begin{pmatrix} 1 & m_1^T \\ 0 & I \end{pmatrix} ,$$

and combining this with (8.7.2) we get

$$A = L_1 \begin{pmatrix} a_{11} & 0 \\ 0 & \widetilde{A}_2 \end{pmatrix} L_1^T . \tag{8.7.4}$$

Obviously, the $(n-1)\times(n-1)$ matrix $\widetilde{A}_2$ is symmetric. Further, let x be a vector such that

$$L_1^T x = \begin{pmatrix} 0 \\ y \end{pmatrix}$$

for some nonzero vector $y \in \mathbb{R}^{n-1}$. Such an x exists and is nonzero because the unit upper triangular matrix L_1^T is nonsingular. Using this x in the condition (8.7.1), we see that

$$0 < x^T A\, x = y^T \widetilde{A}_2\, y \ .$$

This shows that $\widetilde{A}_2$ is *spd*, and according to the assumption it has a factorization $\widetilde{A}_2 = \widetilde{L}_2 \widetilde{D}_2 \widetilde{L}_2^T$, where $\widetilde{L}_2$ is unit lower triangular and $\widetilde{D}_2$ is diagonal with positive diagonal elements. When we insert this in (8.7.4), and combine with the expression for the Gauss transformation L_1 we get

$$\begin{aligned} A &= L_1 \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{L}_2 \end{pmatrix} \begin{pmatrix} a_{11} & 0 \\ 0 & \widetilde{D}_2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{L}_2^T \end{pmatrix} L_1^T \\ &= \begin{pmatrix} 1 & 0 \\ m_1 & \widetilde{L}_2 \end{pmatrix} \begin{pmatrix} a_{11} & 0 \\ 0 & \widetilde{D}_2 \end{pmatrix} \begin{pmatrix} 1 & m_1^T \\ 0 & \widetilde{L}_2^T \end{pmatrix} \ . \end{aligned}$$

This is the LDL^T factorization. □

In Section 8.4 we saw that loss of accuracy may occur in Gaussian elimination because some of the matrix elements can become very large during the elimination process. The next theorem shows that such growth cannot happen in Gaussian elimination without pivoting of an *spd* matrix.

Theorem 8.7.3. Let A be an *spd* matrix. All the matrix elements generated during the LDL^T factorization are bounded in magnitude by the largest element in A.

Proof. We only need to show that the largest element in the transformed submatrix $\widetilde{A}_2$, given by (8.7.3), is bounded by the largest element in A. To do so, we use Lemma 8.7.1 and the proof of Theorem 8.7.2: We showed that $\widetilde{A}_2$ is *spd*. Therefore[6)]

$$\max_{ij} |\widetilde{a}_{ij}| = \max_i \{\widetilde{a}_{ii}\} \ ,$$

and from (8.7.3) and the definition of a_1 we get

$$\widetilde{a}_{ii} = a_{ii} - \frac{1}{a_{11}}\, a_{i,1}^2 \le a_{ii} \ ,$$

since $a_{11} > 0$. Further,

$$a_{ii} \le \max_{1 \le k \le n} \{\widetilde{a}_{kk}\} = \max_{ij} |a_{ij}| \ ,$$

and the proof is finished. □

[6)] In this proof the indices in $\widetilde{A}_2$ refer to the indexing in A, ie $2 \le i, j \le n$ for an element in $\widetilde{A}_2$.

Example. The *spd* matrix

$$A = \begin{pmatrix} 4 & 6 & 2 \\ 6 & 18 & -1.5 \\ 2 & -1.5 & 4.25 \end{pmatrix} .$$

has the LU factorization

$$A = LU = \begin{pmatrix} 1 & 0 & 0 \\ 1.5 & 1 & 0 \\ 0.5 & -0.5 & 1 \end{pmatrix} \begin{pmatrix} 4 & 6 & 2 \\ 0 & 9 & -4.5 \\ 0 & 0 & 1 \end{pmatrix} ,$$

and the LDL^T factorization

$$A = LDL^T, \qquad D = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 1 \end{pmatrix} .$$

■

The symmetry of A and of the LDL^T factorization implies that we only need to store and modify elements on the main diagonal and in either the strictly lower or the strictly upper triangle of A. Therefore we have the following rule of thumb:

> The work involved in computing the LDL$^{\mathrm{T}}$ factorization of an $n \times n$ *spd* matrix is approximately $\frac{1}{3}\, n^3$ flops and we only need to store approximately $\frac{1}{2}\, n^2$ elements.

The diagonal elements in D are positive. Therefore the matrix

$$D^{1/2} = \begin{pmatrix} \sqrt{d_{11}} & & \\ & \ddots & \\ & & \sqrt{d_{nn}} \end{pmatrix}$$

also has real elements. It obviously satisfies $D^{1/2} D^{1/2} = D$, and we get

$$A = L\, D\, L^T = (L\, D^{1/2})(D^{1/2} L^T) = C^T C \ .$$

The matrix $C = D^{1/2} L^T$ is an upper triangular matrix. This version of the LDL$^{\mathrm{T}}$ factorization is the called the *Cholesky factorization* of A (and it is normally computed without the detour via the LDL$^{\mathrm{T}}$ factorization). If we know the Cholesky factorization, then the system $Ax = C^T C\, x = b$ is solved in two steps,

$$\begin{aligned} &1. \quad \text{solve} \quad C^T y = b \ , \\ &2. \quad \text{solve} \quad C x = y \ . \end{aligned} \tag{8.7.5}$$

Also the computation of the Cholesky factorization needs roughly $\frac{1}{3}n^3$ flops, and the work involved in the solution of each of the triangular systems in (8.7.5) is n^2 flops.

Example. For the matrix in the previous example the Cholesky factor is

$$C = \begin{pmatrix} 2 & & \\ & 3 & \\ & & 1 \end{pmatrix} \begin{pmatrix} 1 & 1.5 & 0.5 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 \\ 0 & 3 & -1.5 \\ 0 & 0 & 1 \end{pmatrix} .$$

■

Example. MATLAB has a built-in function `chol` that computes the Cholesky factorization.

```
>> A = [4 6 2; 6 18 -1.5; 2 -1.5 4.25];
>> C = chol(A)
C =  2.0000    3.0000    1.0000
          0    3.0000   -1.5000
          0         0    1.0000
```

The matrix

$$A = \begin{pmatrix} 4 & 6 & 2 \\ 6 & 18 & -1.5 \\ 2 & -1.5 & 2 \end{pmatrix} .$$

is not positive definite, and

```
>> A(3,3) = 2;  C = chol(A)
```

returns an error message

```
??? Error using ==> chol
Matrix must be positive definite.
```

The call `[C,p] = chol(A)` returns $p = 0$ if A is *spd*, otherwise $p > 0$. In a MATLAB program this can eg be used as follows,

```
[C, p] = chol(A);
if  p == 0
  x = C \ (C' \ b);  % solve Ax = b
else
  ...  % commands for A not positive definite
end
```

■

In many applications involving symmetric matrices the origin of the problem guarantees that the matrix is positive definite (and if the matrix turns out not to be positive definite, it might indicate an error in the implementation). Eg the differential operator L in the boundary value problem

$$Ly = -y'' + q(x)y = f(x), \qquad y(a) = \alpha, \quad y(b) = \beta \ ,$$

with $q(x) \geq 0$ can be shown to be positive definite, and a suitable discretization of the problem leads to a linear system of equations with an *spd* matrix, see Chapter 10. The normal equations in the least squares method, Section 8.14, have an *spd* coefficient matrix, and, finally, a common problem in Structural Mechanics is to solve $Ax = b$, where A is a *stiffness matrix*, guaranteed by basic physical principles to be *spd*.

Example. The command `A\b` in MATLAB first invokes a check of whether the matrix A is symmetric, in which case a Cholesky factorization $A = C^T C$ is attempted. If this succeeds – showing that A is positive definite – then (8.7.5) is used to find the solution.

If A is not symmetric, but square, or if the Cholesky factorization breaks down, then the LU factorization is computed, and (8.6.2) is used to find the solution. In an example on page 257 we discuss what happens if A is rectangular, ie the number of rows in A is different from the number of columns. ∎

8.8. Band Matrices

In many applications, eg the solution of boundary value problems for differential equations, you get a matrix, where most of the elements are zero. The matrix is said to be banded if the nonzero elements are concentrated close to the main diagonal. More precisely, A is said to a *band matrix* if there exist natural numbers p and q so that

$$a_{ij} = 0 \quad \text{if} \quad j < i-q \quad \text{or} \quad j > i+p \ .$$

Example. A 6×6 band matrix A with $p = 1$, $q = 2$ has the form

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{pmatrix} \ .$$

∎

The *bandwidth* is defined as $w = p+q+1$. From the example we see that this is the maximum number of nonzeros in any row of A.

With a band matrix you only have to save the elements within the band, and when you solve a linear system of equations with such a matrix,

the structure can be exploited so that you can get a large reduction from the $O(n^3)$ flops needed with a full matrix.

First consider a *tridiagonal matrix*, $p = q = 1$. The matrix can suitably be stored in three vectors, a, b and c,

$$A = \begin{pmatrix} a_1 & b_1 & & & & \\ c_2 & a_2 & b_2 & & & \\ & c_3 & a_3 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-1} & a_{n-1} & b_{n-1} \\ & & & & c_n & a_n \end{pmatrix} .$$

In the solution of a tridiagonal system $Ax = f$ it is straightforward to exploit the structure. Assume first that we do not need to pivot (eg because A is diagonally dominant). Then we only have to zero one element in each column during the elimination, and (8.3.1) simplifies to

$$\begin{aligned} m_{k+1,k} &= c_{k+1}/a'_k \\ a'_{k+1} &= a_{k+1} - m_{k+1,k} b_k \\ f'_{k+1} &= f_{k+1} - m_{k+1,k} f_k \ , \end{aligned}$$

with $a'_1 = a_1$. This is repeated for $k = 1, 2, \ldots, n-1$.

Example. As in the full matrix case the modified elements in the upper triangle of the matrix can overwrite the original elements, and the multipliers can be stored in the positions that they are used to zero. Thus, in MATLAB the LU factorization without pivoting of the tridiagonal matrix A above can be expressed as

```
for  k = 2 : n
  c(k) = c(k)/a(k-1);
  a(k) = a(k) - c(k)*b(k-1);
end
```

The solution of $Ax = f$ can be done in two steps: find y by forward substitution, next use back substitution to get x:

```
y(1) = f(1);
for  k = 2 : n
  y(k) = f(k) - c(k)*y(k-1);
end
x(n) = y(n)/a(n);
for  i = n-1 : -1 : 1
  x(i) = ( y(i) - b(i)*x(i+1) )/a(i);
end
```

■

The work involved is $3n$ flops for the factorization and $5n$ flops for the forward and back substitution, so the total is $8n$ flops.

Partial pivoting will to a certain extent destroy the band structure. As an example, consider a tridiagonal matrix of order 5,

$$\begin{pmatrix} \times & \times & & & \\ \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{pmatrix},$$

with $\times$ indicating a nonzero element. Assume that we have to interchange rows 1 and 2:

$$\begin{pmatrix} \times & \times & \times & & \\ \times & \times & & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{pmatrix}.$$

As before, there is only one subdiagonal element to zero, and we get

$$\begin{pmatrix} \times & \times & \times & & \\ 0 & \times & * & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \end{pmatrix}.$$

The $*$ marks a *fill-in*, a new nonzero element, generated when a multiple of the first row was subtracted from the second row. In the worst case rows k and $k+1$ have to be swapped in every step, in which case the upper triangular matrix U has the structure

$$\begin{pmatrix} \times & \times & \times & & \\ & \times & \times & \times & \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{pmatrix}.$$

There will be no fill-ins in the lower triangular factor L. In each column there is only one subdiagonal nonzero, but the band structure may be lost. We shall return to this later.

Also in the general case of a band matrix there will be no fill-ins if Gaussian elimination without pivoting is used. This is illustrated in Figure 8.1.

If A has q nonzeros below the main diagonal and p nonzeros above it, then the matrices L and U have bandwidths $w_L = q+1$ and $w_U = p+1$, respectively.

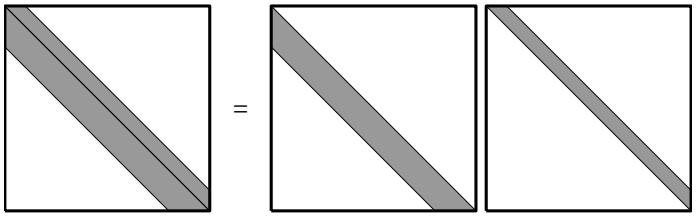


Figure 8.1. *LU factorization of a band matrix. No pivoting.*

If partial pivoting is used in connection with the Gaussian elimination, then the bandwidth of U can increase to $p+q+1$. There is no fill-in in L, but the band structure may be lost.

If the band matrix is *spd* (symmetric, positive definite), then we can use Cholesky factorization, and the Cholesky factor C will be a band matrix.

As we have seen, in the LU factorization of a band matrix both L and U are banded (but the columns in L may be permuted). In contrast, the inverse matrix A^{-1} is normally a full matrix. Therefore, the explicit inversion of a band matrix should be avoided; also see the next section.

Example. The matrix

$$A = \begin{pmatrix} 4 & 2 & & & \\ 2 & 5 & 2 & & \\ & 2 & 5 & 2 & \\ & & 2 & 5 & 2 \\ & & & 2 & 5 \end{pmatrix}$$

is *spd* and has the Cholesky factor

$$C = \begin{pmatrix} 2 & 1 & & & \\ & 2 & 1 & & \\ & & 2 & 1 & \\ & & & 2 & 1 \\ & & & & 2 \end{pmatrix}.$$

The inverse of A is

$$A^{-1} = 2^{-10} \begin{pmatrix} 341 & -170 & 84 & -40 & 16 \\ -170 & 340 & -168 & 80 & -32 \\ 84 & -168 & 336 & -160 & 64 \\ -40 & 80 & -160 & 320 & -128 \\ 16 & -32 & 64 & -128 & 256 \end{pmatrix}.$$

■

A band matrix is a special example of a *sparse* matrix, ie a matrix where most of the elements are zero, but maybe not concentrated close to the diagonal. MATLAB is designed for efficient handling of such matrices.

Example. Consider the matrix A and vector b,

$$A = \begin{pmatrix} 1 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \qquad b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} .$$

The following MATLAB commands set up the matrix and solve the system $Ax = b$.

```
>> A = speye(3);    % sparse representation of unit matrix
>> A(1,2) = 0.5
A = (1,1)       1.0000
    (1,2)       0.5000
    (2,2)       1.0000
    (3,3)       1.0000
>> x = A\[1; 2; 3]  % Solve Ax =  b
x =  0
     2
     3
```

Only the nonzeros are stored, and each of these is represented by a triplet: row number, column number and value of the element.

If `A` is sparse, then `[L, U] = lu(A)` returns sparse matrices `L` and `U`. The factorization is performed without wasting time on zeroing elements that are already zero. ∎

8.9. Computing the Inverse of a Matrix

For a nonsingular matrix A the solution to the system $Ax = b$ can be expressed as $x = A^{-1}b$, where A^{-1} is the inverse matrix. In practice the inverse matrix is normally not needed, and if you meet an expression like $A^{-1}B$ (with an $n \times p$ matrix B), you should realize that this is just shorthand notation for: find the solution to the *matrix equation*

$$A\,X = B\ .$$

This is equivalent to solving the systems

$$Ax_{:j} = b_{:j}\ , \quad j = 1, \ldots, p\ ,$$

where $x_{:j}$ and $b_{:j}$ are column vectors in X and B, respectively. On page 213 we saw that if the LU factorization of A is known, then it can be used to compute the solution at the cost of $2pn^2$ flops. This is the same as is needed for the matrix multiplication $A^{-1}B$, so the difference in work is the difference between the $\frac{2}{3}n^3$ flops needed to find the LU factorization and – as we shall see – $2n^3$ flops needed to compute A^{-1}. Thus, the initial work is tripled. It can also be shown that the effect of rounding errors is considerably larger with the explicit use of the inverse matrix. Summarizing:

Do not compute the inverse matrix unless it is expressively needed.

There are special applications, eg in statistics, where the elements of the inverse matrix give important information, and in such cases it is necessary to compute A^{-1} explicitly. This can be done by solving the matrix equation

$$A\,X = I\ ,$$

where I is the unit matrix with column vectors $e_1, \ldots, e_n$ defined by

$$(e_k)_i = \begin{cases} 0\ , & i \neq k\ . \\ 1\ , & i = k\ . \end{cases}$$

Assume that the LU factorization of A has been computed, and that there is no pivoting. Then $x_{:k}$, the kth column vector in A^{-1} is found in the two steps

$$\begin{aligned} &1.\quad \text{solve} \quad Ly_{:k} = e_{:k}\ , \\ &2.\quad \text{solve} \quad Ux_{:k} = y_{:k}\ . \end{aligned}$$

Generally, the solution of each of these two triangular systems needs $2n^2$ flops. However, we can save operations because of the special form of the right hand side:

$$\begin{pmatrix} l_{11} & & & & \\ \vdots & \ddots & & & \\ l_{k1} & \cdots & l_{kk} & & \\ \vdots & & \vdots & \ddots & \\ l_{n1} & \cdots & l_{nk} & \cdots & l_{nn} \end{pmatrix} \begin{pmatrix} y_{1k} \\ \vdots \\ y_{kk} \\ \vdots \\ y_{nk} \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}\ .$$

The first $k-1$ elements in e_k are zero, and therefore also the corresponding elements in $y_{:k}$ are zero. Essentially, the nonzero elements in $y_{:k}$ are found

by solving the last $n-k+1$ equations in the $n-k+1$ last unknowns, and this needs $(n-k+1)^2$ flops. So the work involved in computing all the columns $y_{:k}$ is

$$\sum_{k=1}^{n}(n-k+1)^2 = \sum_{\nu=1}^{n}\nu^2 \;\approx\; \frac{1}{3}\,n^3 \;,$$

cf Lemma 8.3.1. There is no similar saving in the work needed to solve the systems $Ux_{:k} = y_{:k}$. Each of these needs n^2 flops, and including the $\frac{2}{3}\,n^3$ flops for computing the LU factorization we get the following rule of thumb.

The computation of the inverse of an $n \times n$ matrix needs about $2n^3$ flops.

8.10. Vector and Matrix Norms

In the next section we investigate how the solution to a linear system of equations is affected by perturbations of the elements in the matrix and the right hand side. First, however we need to be able to measure the "size" of a vector and a matrix.

Definition 8.10.1. Vector norm. Let $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$. A vector norm $\|\cdot\|$ is a mapping $\mathbb{R}^n \mapsto \mathbb{R}$ that satisfies the conditions

$$\begin{aligned} \|x\| &\geq 0 \qquad \text{for all } x \;, \\ \|x\| &= 0 \quad \Leftrightarrow \quad x = 0 \;, \\ \|\alpha x\| &= |\alpha|\,\|x\| \;, \\ \|x+y\| &\leq \|x\| + \|y\| \;. \qquad \text{(triangle inequality)} \end{aligned}$$

These conditions are satisfied by the ℓ_p-norms, defined as

$$\|x\|_p = \big(\,|x_1|^p + \cdots + |x_n|^p\,\big)^{1/p} \;, \quad p \geq 1 \;, \tag{8.10.1}$$

and among these the following three norms are most widely used,

$$\|x\|_1 = |x_1| + \cdots + |x_n| \ ,$$
$$\|x\|_2 = \left(x_1^2 + \cdots + x_n^2 \right)^{1/2} = \sqrt{x^T x} \ ,$$
$$\|x\|_\infty = \max_{1 \le i \le n} |x_i| \ .$$

The 2-norm $\|\cdot\|_2$ is called the *Euclidean norm*; it is a generalization to $\mathbb{R}^n$ of the usual vector length in $\mathbb{R}^3$. $\|\cdot\|_\infty$ is called the *maximum norm*.

Example. Let

$$x = \begin{pmatrix} 2 \\ -2 \\ 1 \end{pmatrix}, \qquad y = \begin{pmatrix} 0 \\ -3 \\ 0 \end{pmatrix}, \qquad z = \begin{pmatrix} 0 \\ -2.5 \\ 2 \end{pmatrix} \ .$$

Which of these three vectors in $\mathbb{R}^3$ is the "largest"? The answer depends on the choice of norm:

p	$\|x\|_p$	$\|y\|_p$	$\|z\|_p$
1	$\underline{5}$	3	4.5
2	3	3	$\underline{3.2016}$
∞	2	$\underline{3}$	2.5

The three norms, however, "follow each other" in the sense, that if a vector is "small" (respectively "large") in one norm, then it is also "small" (respectively "large") in the other norms, see Exercise E1. ∎

Example. If `x` contains the vector x, then the MATLAB command `norm(x,p)` returns $\|x\|_p$ as defined in (8.10.1). The command `norm(x)` (with only one input argument) returns $\|x\|_2$. ∎

By means of norms we introduce notions like distance and continuity in $\mathbb{R}^n$. Let $\overline{x}$ be an approximation to the vector x. With a given norm $\|\cdot\|$ we define the

$$\textit{absolute error}: \qquad \|\delta x\| = \|\overline{x} - x\| \ ,$$
$$\textit{relative error}: \qquad \frac{\|\delta x\|}{\|x\|} = \frac{\|\overline{x} - x\|}{\|x\|} \ .$$

Note that δ is not a scalar, but δx denotes a vector.

Example. Given a vector b and the approximation $\overline{b}$,

$$b = \begin{pmatrix} \frac{7}{12} \\ 0.45 \end{pmatrix}, \qquad \overline{b} = \begin{pmatrix} 0.583 \\ 0.45 \end{pmatrix} \ .$$

Then

$$\delta b = \begin{pmatrix} -0.000333\ldots \\ 0 \end{pmatrix} = \begin{pmatrix} -\frac{1}{3}\cdot 10^{-3} \\ 0 \end{pmatrix} ,$$

and

$$\|\delta b\|_\infty = \frac{1}{3}\cdot 10^{-3} , \qquad \frac{\|\delta b\|_\infty}{\|b\|_\infty} = \frac{\frac{1}{3}\cdot 10^{-3}}{\frac{7}{12}} = \frac{4}{7}\cdot 10^{-3} .$$

■

With any vector norm we associate a matrix norm:

Definition 8.10.2. Induced matrix norm. Let $\|\cdot\|$ be a vector norm. The induced matrix norm is

$$\|A\| = \sup_{x\neq 0} \frac{\|Ax\|}{\|x\|} .$$

In some applications it is convenient to use another expression for the matrix norm:

Lemma 8.10.3. The induced matrix norm can be found as

$$\|A\| = \max_{\|x\|=1} \|Ax\| .$$

Proof. Follows from $A(\alpha x) = \alpha Ax$, the vector norm conditions, and Definition 8.10.2. □

Example. We shall illustrate the induced matrix norm of the matrix

$$A = \begin{pmatrix} 7 & -16 \\ 3 & 11 \end{pmatrix} .$$

For $x \in \mathbb{R}^2$ we can identify $x = \begin{pmatrix} x_1 & x_2 \end{pmatrix}^T$ with the point (x_1, x_2) in a Cartesian coordinate system. To the left in Figure 8.2 we show the *unit circle* with respect to the 2-norm, ie the set of points for which $\|x\|_2 = 1$. This is just the circle with radius 1, centered at $(0, 0)$. To the right we show the image of the unit circle,

$$y = \{Ax \mid \|x\|_2 = 1\} .$$

This is an ellipse. The arrows indicate a unit vector v_1 and its image $u_1 = Av_1$, such that

$$\|u_1\|_2 = \max\{\|Ax\|_2 \mid \|x\|_2 = 1\} .$$

(Evidently, the image of $-v_1$ is $-u_1$ and has the same norm). The 2-norm of A is the Euclidean length of u_1.

Figure 8.3 gives the similar picture for the 1-norm, where the unit circle is defined by $|x_1| + |x_2| = 1$.

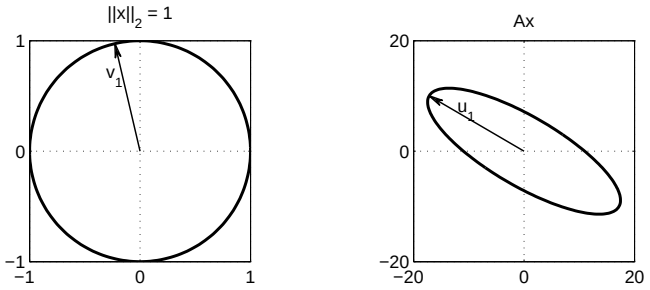


Figure 8.2. *Induced matrix 2-norm.*

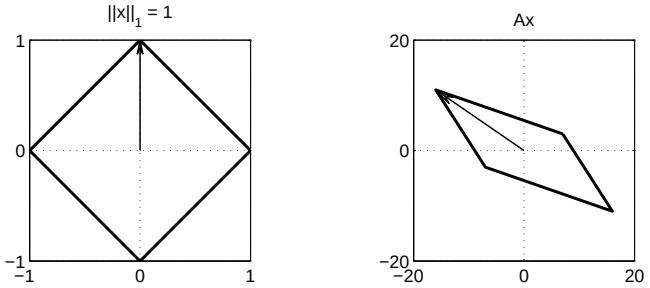


Figure 8.3. *Induced matrix 1-norm.*

With `A` containing the above matrix, we can find the 1-, 2- and maximum norm of A as follows:

```
>> nrms = [norm(A,1)  norm(A,2)  norm(A,inf)]
nrms =  27.0000   19.8870   23.0000
```

Thus, the length of u_1 in Figure 8.2 is 19.8870 . ∎

From the definition of the induced matrix norm it follows immediately that the unit matrix has norm one,

$$\|I\| = 1 \ . \tag{8.10.2}$$

It is also easy to show that the induced matrix norm satisfies the following relations, similar to the vector norm conditions

$$\begin{aligned} \|A\| &\geq 0 \qquad \text{for all } A \ , \\ \|A\| &= 0 \quad \Leftrightarrow \quad A = 0 \ , \\ \|\alpha A\| &= |\alpha| \, \|A\| \ , \quad \alpha \in \mathbb{R} \ , \\ \|A + B\| &\leq \|A\| + \|B\| \ . \end{aligned} \tag{8.10.3}$$

Two more inequalities are provided by the following lemma.

Lemma 8.10.4. Let $\|\cdot\|$ denote a vector norm and its induced matrix norm. Then

$$\|Ax\| \le \|A\|\,\|x\| ,$$
$$\|AB\| \le \|A\|\,\|B\| .$$

Proof. From Definition 8.10.2 we see that

$$\frac{\|Ax\|}{\|x\|} \;\le\; \|A\|$$

for all $x \neq 0$, and since $x \neq 0$ implies that $\|x\| > 0$, we immediately get the first inequality. The second is obtained by applying the first inequality twice on $\|ABx\|$. □

It can be shown that

$$\|A\|_2 = \left(\max_{1\le j\le n} \lambda_j(A^T A) \right)^{1/2} ,$$

ie the square root of the largest eigenvalue of the matrix[7] $A^T A$. Thus, the computation of $\|A\|_2$ for a given matrix A involves a considerable amount of work. It is much easier to compute $\|A\|_\infty$:

Lemma 8.10.5.

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

Proof. We use the formulation from Lemma 8.10.3, ie we consider the product Ax for

$$\|x\|_\infty = \max_{1\le i\le n} |x_i| = 1 .$$

Then the ith component of the vector Ax can be estimated as follows,

$$|(Ax)_i| = \Big| \sum_{j=1}^{n} a_{ij}x_j \Big| \;\le\; \sum_{j=1}^{n} |a_{ij}x_j| = \sum_{j=1}^{n} |a_{ij}|\cdot|x_j| \;\le\; \sum_{j=1}^{n} |a_{ij}| .$$

This shows that the right hand side in the lemma, $r = \max\limits_{1\le i\le n} \Big\{ \sum\limits_{j=1}^{n} |a_{ij}| \Big\}$,

[7] The matrix $A^T A$ is symmetric and positive semidefinite. Such a matrix has real, nonnegative eigenvalues.

is an upper bound for $\|Ax\|_\infty$. We have to show that there exists a vector $\hat{x}$ with $\|\hat{x}\|_\infty = 1$ so that $\|A\hat{x}\|_\infty = r$. Let ν be a row number such that

$$\sum_{j=1}^{n} |a_{\nu j}| = \max_{1\le i\le n} \Big\{ \sum_{j=1}^{n} |a_{ij}| \Big\} ,$$

and let $\hat{x}_j = \operatorname{sign}(a_{\nu j}),\ j=1,\dots,n$. This vector is a unit vector in the maximum norm, and

$$|(A\hat{x})_\nu| = \Big|\sum_{j=1}^{n} a_{\nu j}\hat{x}_j\Big| = \sum_{j=1}^{n} |a_{\nu j}| .$$

□

It can also be shown that

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

Example. The matrix $A = \begin{pmatrix} 7 & -16 \\ 3 & 11 \end{pmatrix}$ from the previous example has

$$\|A\|_\infty = \max\{7{+}16,\ 3{+}11\} = 23 ,$$
$$\|A\|_1 = \max\{7{+}3,\ 16{+}11\} = 27 .$$ ■

Example. The so-called *Frobenius norm* of a matrix is defined as

$$\|A\|_F = \Big(\sum_{i,j=1}^{n} a_{ij}^2 \Big)^{1/2} .$$

This is equivalent to the Euclidean norm of the vector obtained by stacking the column vectors of A on top of each other[8)] . It is a norm – it satisfies (8.10.3) – but it is not an induced matrix norm, and $\|\cdot\|_F$ satisfies neither (8.10.2) nor the inequalities of Lemma 8.10.4. ■

Finally, we shall need the following lemma in the next section.

Lemma 8.10.6. If $\|F\| < 1$, then the matrix $I{+}F$ is nonsingular.

Proof. Suppose that $I{+}F$ is singular. Then $(I{+}F)x = 0$ for some

[8)] A matrix in $\mathbb{R}^{m\times n}$ can also be considered as an element in $\mathbb{R}^{mn}$, since the two linear spaces are isomorphic. In contexts, where one wants to emphasize this isomorphism, the Frobenius norm is often referred to as the Euclidean matrix norm.

nonzero x. But this implies that $\|x\| = \|-Fx\| \leq \|F\| \, \|x\|$, showing that $\|F\| \geq 1$, which is a contradiction. □

8.11. Sensitivity Analysis

In this section we investigate how the solution to a linear system of equations is affected by perturbations of the elements in the matrix and the right hand side. Such perturbations can, eg, be measurement errors or rounding errors when the matrix and right hand side are stored in the computer.

We consider a linear system of equations

$$Ax = b \ ,$$

with a nonsingular matrix A. We say that this is the *exact* system with the *exact* solution x. If the right hand side is perturbed, so is the solution,

$$A(x + \delta x) = b + \delta b \ .$$

We want to estimate the relative error $\|\delta x\|/\|x\|$. To do so we subtract $Ax = b$ from both sides of the perturbed equation and get $A\delta x = \delta b$, or

$$\delta x = A^{-1}\delta b \ .$$

We take the norm and use Lemma 8.10.4:

$$\|\delta x\| = \|A^{-1}\delta b\| \ \leq \ \|A^{-1}\| \cdot \|\delta b\| \ .$$

This is an estimate of the absolute error. Similarly

$$\|b\| = \|Ax\| \ \leq \ \|A\| \cdot \|x\| \ ,$$

which can be rewritten to

$$\frac{1}{\|x\|} \ \leq \ \|A\| \, \frac{1}{\|b\|} \ ,$$

and we see that

$$\frac{\|\delta x\|}{\|x\|} \ \leq \ \frac{\|A^{-1}\| \cdot \|\delta b\|}{\|x\|} \ \leq \ \|A\| \cdot \|A^{-1}\| \cdot \frac{\|\delta b\|}{\|b\|} \ . \tag{8.11.1}$$

This shows that the relative perturbation of the right hand side is multiplied by a factor $\|A\| \cdot \|A^{-1}\|$.

Definition 8.11.1. Condition number. For a nonsingular matrix A the condition number is

$$\kappa(A) = \|A\| \cdot \|A^{-1}\| \ .$$

The condition number depends on the underlying norm and subscripts are used accordingly, eg $\kappa_\infty(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty$.

Example. Let $b = \begin{pmatrix} 7/12 \\ 0.45 \end{pmatrix}$, $\quad b + \delta b = \overline{b} = \begin{pmatrix} 0.583 \\ 0.45 \end{pmatrix}$.

We showed in the second example on page 226 that $\|\delta b\|_\infty / \|b\|_\infty = \frac{4}{7} \cdot 10^{-3}$. The matrix and its inverse are

$$A = \begin{pmatrix} 1/3 & 1/4 \\ 1/4 & 1/5 \end{pmatrix}, \qquad A^{-1} = \begin{pmatrix} 48 & -60 \\ -60 & 80 \end{pmatrix} \ .$$

The maximum-norm condition number for A is

$$\kappa_\infty(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty = \tfrac{7}{12} \cdot 140 \ \simeq \ 81.7 \ ,$$

so (8.11.1) gives the estimate

$$\frac{\|\delta x\|}{\|x\|} \ \leq \ 81.7 \cdot \tfrac{4}{7} \cdot 10^{-3} \ \simeq \ 0.047 \ .$$

The exact solution to the system $Ax = b$ is $x = \begin{pmatrix} 1 & 1 \end{pmatrix}^T$, and $A\overline{x} = \overline{b}$ has the solution $\overline{x} = \begin{pmatrix} 0.984 & 1.020 \end{pmatrix}^T$. Thus, the true relative error is

$$\frac{\|\overline{x} - x\|}{\|x\|} \ = \ 0.02 \ .$$

The estimate is on the safe side by a factor about 2. ∎

From the derivation and the example we see that the condition number is a measure of how sensitive the solution is to perturbations in the right hand side. The following theorem shows that the condition number also reflects the sensitivity to changes in the matrix.

Theorem 8.11.2. Let $Ax = b$ and $(A+\delta A)\overline{x} = b+\delta b$. If A is nonsingular and

$$\|A^{-1}\| \cdot \|\delta A\| \ = \ \kappa(A) \, \frac{\|\delta A\|}{\|A\|} \ = \ \tau \ < \ 1 \ ,$$

then the matrix $A+\delta A$ is also nonsingular, and

$$\frac{\|\overline{x} - x\|}{\|x\|} \ \leq \ \frac{\kappa(A)}{1 - \tau} \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right) \ .$$

Proof. We rewrite the perturbed matrix:

$$A + \delta A = A(I + F) \ , \qquad F = A^{-1}\delta A \ ,$$

and by means of Lemma 8.10.4 and the assumption about τ we get

$$\|F\| = \|A^{-1}\delta A\| \ \leq \ \|A^{-1}\| \cdot \|\delta A\| = \tau \ < \ 1 \ .$$

Therefore, according to Lemma 8.10.6 the matrix $I+F$ is nonsingular, and $(A+\delta A)^{-1} = (I+F)^{-1}A^{-1}$ exists; showing that $A + \delta A$ is nonsingular. Next, let $\overline{x} = x + \delta x$. From the definition of the perturbed problem we find

$$A(x + \delta x) = b+\delta b - \delta A(x + \delta x) \ ,$$

and after subtracting $Ax = b$ and multiplying by A^{-1} on both sides we get

$$\delta x = \ A^{-1}\delta b - A^{-1}\delta A(x + \delta x) \ .$$

Now we take norms and use the norm inequalities and the first expression for τ:

$$\begin{aligned} \|\delta x\| &\leq \|A^{-1}\| \, \|\delta b\| + \|A^{-1}\| \, \|\delta A\| \, \|x + \delta x\| \\ &\leq \|A^{-1}\| \, \|\delta b\| + \tau(\|x\| + \|\delta x\|) \ , \end{aligned}$$

so that $(1-\tau)\|\delta x\| \leq \|A^{-1}\| \, \|\delta b\| + \tau\|x\|$, or

$$\frac{\|\delta x\|}{\|x\|} \ \leq \ \frac{1}{1-\tau}\left(\frac{\|A^{-1}\| \, \|\delta b\|}{\|x\|} + \tau\right) \ .$$

The first term in the parenthesis is estimated as in (8.11.1), and the theorem follows when we insert the second expression for τ. □

From the definition of the induced matrix norm, the identity $I = A \, A^{-1}$, and the norm inequalities it follows that

$$1 = \|I\| = \|A \, A^{-1}\| \ \leq \ \|A\| \cdot \|A^{-1}\| \ ,$$

showing that $\kappa(A) \geq 1$. A matrix with a small condition number is said to be *well-conditioned*. An orthogonal matrix Q (ie Q satisfies $Q^TQ = I$; see Section 8.15) has $\kappa_2(Q) = 1$, so it is as well-conditioned as is possible.

A matrix with a large condition number is said to be *ill-conditioned*. The condition number is used to get an estimate of what accuracy can be expected when you solve a linear system of equations $Ax = b$. If, eg there are errors in the matrix and right hand side of the order of magnitude 10^{-8} and $\kappa(A) = 10^3$, then you can expect the solution to have five

significant decimal digits, but if $\kappa(A) = 10^6$, then you can only expect to get two significant digits.

Example. The matrix $A = \begin{pmatrix} 1 & 2 \\ 2 & 4.001 \end{pmatrix}$ has the condition number $\kappa_\infty(A) \simeq 3.6 \cdot 10^4$, so the matrix is rather ill-conditioned. This is equivalent with the column vectors being almost linearly dependent. In this example it is easy to see that the two column vectors are almost parallel. ∎

Example. In the proof of Theorem 8.11.2 we used the norm inequality $\|Aw\| \leq \|A\| \, \|w\|$ several times. From Figure 8.2 on page 228, we see that if $w \simeq \alpha v_1$ (ie the direction of w is close to the direction of v_1), then this is a good estimate, but it may be a gross overestimate if w is (almost) orthogonal to v_1. $\|A^{-1}\| \, \|z\|$ can be much larger than $\|A^{-1}z\|$.

The conclusion is, that with special combinations of the matrix A, the right hand side b and perturbations δA and δb the error can be as large as predicted by the theorem, but the estimate may be very pessimistic, especially for ill-conditioned problems. ∎

It sometimes happens that we are given a vector $\widetilde{x}$ and ask: how close is $\widetilde{x}$ to the solution of $Ax = b$? The following corollary answers this question.

Corollary 8.11.3. Let the system $Ax = b$ and a vector $\widetilde{x}$ be given. Then

$$\frac{\|\widetilde{x} - x\|}{\|x\|} \leq \kappa(A) \, \frac{\|r\|}{\|b\|} \, , \qquad r = b - A\widetilde{x} \, .$$

r is called the *residual*.

Proof. Obviously, $\widetilde{x}$ is the solution to the perturbed system $A\widetilde{x} = b - r$, and the corollary follows from Theorem 8.11.2 with $\delta A = 0$, $\delta b = -r$. □

Example. Consider the system $Ax = b$,

$$\begin{pmatrix} 57.5 & 43.75 \\ 77 & 47 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 13.75 \\ 30 \end{pmatrix} \, ,$$

and the two vectors

$$x^{(1)} = \begin{pmatrix} 1.02 \\ -1.03 \end{pmatrix} \, , \qquad x^{(2)} = \begin{pmatrix} 1.0006 \\ -1.0004 \end{pmatrix} \, .$$

Which of them is closest to x ? The residuals are

$$r^{(1)} = b - Ax^{(1)} = \begin{pmatrix} 0.1625 \\ -0.1300 \end{pmatrix} , \qquad r^{(2)} = b - Ax^{(2)} = \begin{pmatrix} -0.0170 \\ -0.0274 \end{pmatrix} .$$

Both residuals are small compared to the right hand side b, and since the elements in $r^{(2)}$ are much smaller than the elements in $r^{(1)}$, we expect that $x^{(2)}$ is closer than $x^{(1)}$ to x, but how close? The condition number of A is $\kappa_\infty(A) \simeq 25.0$, and Corollary 8.11.3 gives the error estimates

$$\frac{\|x^{(1)} - x\|_\infty}{\|x\|_\infty} \leq 0.14 , \qquad \frac{\|x^{(2)} - x\|_\infty}{\|x\|_\infty} \leq 0.023 .$$

The exact solution is $x = \begin{pmatrix} 1 & -1 \end{pmatrix}^T$ and the true relative errors are

$$\frac{\|x^{(1)} - x\|_\infty}{\|x\|_\infty} = 0.03 , \qquad \frac{\|x^{(2)} - x\|_\infty}{\|x\|_\infty} = 0.0006 .$$

Thus, the conclusion about $x^{(2)}$ being the better approximation to x is right, but the error estimate is very pessimistic.

Next, consider the problem $Bx = c$ with

$$B = \begin{pmatrix} 95.75 & 64.375 \\ 120.2 & 79.7 \end{pmatrix} , \qquad c = \begin{pmatrix} 31.375 \\ 40.5 \end{pmatrix} .$$

This system has the same solution as $Ax = b$, but a larger condition number, $\kappa_\infty(B) \simeq 405$. With the same approximate solutions we get the residuals

$$r^{(1)} = c - Bx^{(1)} = \begin{pmatrix} 0.0162 \\ -0.0130 \end{pmatrix} , \quad r^{(2)} = c - Bx^{(2)} = \begin{pmatrix} -0.0317 \\ -0.0402 \end{pmatrix} ,$$

and the error estimates

$$\frac{\|x^{(1)} - x\|_\infty}{\|x\|_\infty} \leq 0.17 , \qquad \frac{\|x^{(2)} - x\|_\infty}{\|x\|_\infty} \leq 0.41 .$$

The residuals are still small, compared to the right hand side, but now we cannot see that $x^{(2)}$ is the better of the two approximations. ∎

The matrix A is singular if and only if its determinant $\det(A)$ is zero. Therefore, it is tempting to believe that the value of the determinant can be used to measure whether the matrix is well-conditioned or ill-conditioned. A large (or small) value of $\det(A)$ could be expected to reveal that A is well-conditioned (or ill-conditioned). As the following example demonstrates, however, the determinant is totally unsuited for investigation of the conditioning of a matrix.

Example. Let A_n and D_n be the $n \times n$ matrices

$$A_n = \begin{pmatrix} 1 & -1 & -1 & \cdots & -1 \\ & 1 & -1 & \cdots & -1 \\ & & 1 & \cdots & -1 \\ & & & \ddots & \vdots \\ & & & & 1 \end{pmatrix}, \quad D_n = \begin{pmatrix} 10^{-2} & & & & \\ & 10^{-2} & & & \\ & & 10^{-2} & & \\ & & & \ddots & \\ & & & & 10^{-2} \end{pmatrix}.$$

As n grows, the matrices A_n become increasingly ill-conditioned: $\kappa_\infty(A_n) = n \cdot 2^{n-1}$, but $\det(A_n) = 1$.

The matrices D_n are well-conditioned: $\kappa_\infty(D_n) = 1$, but $\det(D_n) = 10^{-2n}$. ∎

In practice the condition number is not computed in connection with the solution of a linear system of equations $Ax = b$, since this would need the computation of the inverse matrix A^{-1} at a cost of $2n^3$ flops. In Section 8.13 we shall see that the LU factorization of A can be used to get a good estimate of $\kappa(A)$ at a cost of $O(n^2)$ flops.

8.12. Rounding Errors in Gaussian Elimination

We know from the discussion in Chapter 2 that any real number (which is inside the range of a given floating point number system) is represented in the computer with a relative error that is bounded by the unit roundoff μ. This can be expressed as

$$fl[x] = x(1+\epsilon) \quad |\epsilon| \le \mu \ .$$

Therefore, when we represent a matrix A and a vector b, errors arise,

$$fl[a_{ij}] = a_{ij}(1+\epsilon) = a_{ij} + \epsilon a_{ij} \quad |\epsilon| \le \mu \ ,$$

and similar for b. It follows that

$$fl[A] = A + \delta A \ , \qquad \|\delta A\|_\infty \le \epsilon \, \|A\|_\infty \ ,$$
$$fl[b] = b + \delta b \ , \qquad \|\delta b\|_\infty \le \epsilon \, \|b\|_\infty \ .$$

Thus, the exact system $Ax = b$ is represented by the perturbed system

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

Assume for the moment that there are no rounding errors during the solution of the perturbed system. Then $\hat{x}$ is the computed solution, and from Theorem 8.11.2 we see that

$$\frac{\|\hat{x} - x\|_\infty}{\|x\|_\infty} \;\le\; \frac{\kappa_\infty(A)}{1-\tau} \cdot 2\mu \;, \qquad \tau = \mu\kappa_\infty(A) \;, \tag{8.12.1}$$

provided that $\tau < 1$.

This is an example of backward error analysis, cf Chapter 2.

Backward error analysis. The computed solution $\hat{x}$ is the exact solution to a perturbed system

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

The error in $\hat{x}$ can be estimated by means of sensitivity analysis.

Gaussian elimination involves about $\frac{2}{3}n^3$ floating point operations, each of which may be affected by a rounding error, and the result takes part in later operations. Therefore, (8.12.1) cannot be expected to hold, but we have the following theorem. The proof of the theorem is omitted here, it is rather technical, except for the final estimate, which follows from Theorem 8.11.2 when we insert the estimate for $\|\delta A\|_\infty$.

Theorem 8.12.1. Assume that computation is made in a floating point number system with unit roundoff μ and let $\hat{L}$ and $\hat{U}$ be the triangular factors obtained by applying Gaussian elimination on the permuted matrix PA. Further, let $\hat{x}$ be the solution obtained by forward and back substitution in the systems

$$\hat{L}\,\hat{y} = P\,b \;, \qquad \hat{U}\,\hat{x} = \hat{y} \;.$$

Then $\hat{x}$ is the exact solution to a system $(A + \delta A)\hat{x} \;=\; b$, where

$$\|\delta A\|_\infty \;\le\; \mu(n^3 + 3n^2) g_n \|A\|_\infty \;, \quad g_n = \frac{\max\limits_{i,j,k} |\hat{a}_{ij}^{(k)}|}{\max\limits_{i,j} |a_{ij}|} \;.$$

($\hat{a}_{ij}^{(k+1)}$ are the elements formed in the kth step of the Gaussian elimination). If $\tau = \mu\kappa_\infty(A)(n^3 + 3n^2)g_n < 1$, then

$$\frac{\|\hat{x} - x\|_\infty}{\|x\|_\infty} \;\le\; \frac{\tau}{1-\tau} \;.$$

The elements $\hat{a}_{ij}^{(k)}$ are given by (8.3.1):

$$\hat{a}_{ij}^{(k+1)} = fl[\hat{a}_{ij}^{(k)} - \hat{m}_{ik}\hat{a}_{kj}^{(k)}]$$

with $\hat{a}_{ij}^{(1)} = a_{ij}$. The *growth factor* g_n depends on the actual growth of the matrix elements rather than on the size of the multipliers $\hat{m}_{ik}$. With partial pivoting we have $|\hat{m}_{ik}| \leq 1$, and thereby an upper bound on the growth in each step is minimized.

It is simple to modify the code for LU factorization so that it also computes g_n. This way you can get an *a posteriori* (afterward) estimate of the effects of rounding errors.

A priori (beforehand) you can show that if partial pivoting is used, then $g_n \leq 2^{n-1}$. It is possible to construct matrices, where this growth does take place (if $n = 31$, then $g_n = 2^{30} \simeq 10^9$). Therefore, it may be necessary to use complete pivoting, in which case it can be shown that $g_n \leq 1.8\, n^{0.25 \ln n}$, and $g_{31} \leq 34.4$.

In practice, when one uses partial pivoting, g_n is rarely larger than 10.

In Section 8.7 we showed that if A is *spd* and we use Gaussian elimination without pivoting, then $\max_{i,j} |\hat{a}_{ij}^{(k)}|$ does not grow with k, so in this case $g_n = 1$.

Example. In Section 8.4 we considered the matrix $A = \begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix}$, which has the inverse $A^{-1} = \dfrac{1}{\epsilon - 1} \begin{pmatrix} 1 & -1 \\ -1 & \epsilon \end{pmatrix}$.

For small ϵ the condition number is $\kappa_\infty(A) \simeq 4$. This shows that the *problem* of solving $Ax = b$ is well-conditioned.

Gaussian elimination without pivoting corresponds to

$$A = LU, \quad L = \begin{pmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{pmatrix}, \quad U = \begin{pmatrix} \epsilon & 1 \\ 0 & 1 - 1/\epsilon \end{pmatrix},$$

and the growth factor $g_n = |1 - 1/\epsilon|$, which is very big if $|\epsilon|$ is very small, in which case Theorem 8.12.1 tells us that we can expect large errors in the computed solution. Note also, that the LU factors are very ill-conditioned,

$$\kappa_\infty(L) \simeq \kappa_\infty(U) \simeq \frac{1}{\epsilon^2}.$$

The algorithm is unstable.

With partial pivoting we get $L = \begin{pmatrix} 1 & 0 \\ \epsilon & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 1 \\ 0 & 1 - \epsilon \end{pmatrix}$.

There is no growth of matrix elements, $g_n = 1$, and $\kappa_\infty(L) \simeq 1$, $\kappa_\infty(U) \simeq 4$. *This algorithm is stable.* ∎

Example. The matrix $A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$ is symmetric and positive definite (*spd*).

It is well-conditioned, $\kappa_\infty(A) = 3$, and the $\mathrm{LDL}^{\mathrm{T}}$ factorization is

$$A = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 0 & \frac{3}{2} \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{2} \\ 0 & 1 \end{pmatrix} .$$

There is no growth of matrix elements; $g_2 = 1$.

Next, the matrix

$$B = \begin{pmatrix} \epsilon & 1 \\ 1 & 1/\epsilon + 1 \end{pmatrix}$$

is also *spd*, but for small values of ϵ it is ill-conditioned: $\kappa_\infty(B) \approx 1/\epsilon^3$. The $\mathrm{LDL}^{\mathrm{T}}$ factorization is

$$B = \begin{pmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{pmatrix} \begin{pmatrix} \epsilon & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1/\epsilon \\ 0 & 1 \end{pmatrix} ,$$

and again $g_2 = 1$. The problem $Bx = c$ is certainly ill-conditioned, but the algorithm (Gaussian elimination without pivoting) does not lead to *unnecessary* loss of accuracy. ∎

In practice the error estimate in Theorem 8.12.1 is often very pessimistic. It assumes that in every floating point operation the rounding error is maximal, ie

$$fl[a \odot b] = (a \odot b)(1 + \epsilon) \quad \text{with} \quad |\epsilon| = \mu ,$$

and that these errors accumulate in the worst possible way. If we use a stable algorithm for the elimination (Gaussian elimination with pivoting (without pivoting if the matrix is *spd*)), then a more realistic – although not safe – estimate is that the computed solution $\hat{x}$ satisfies

$$(A + \delta A)\hat{x} = b, \qquad \|\delta A\|_\infty \lesssim \mu \|A\|_\infty .$$

Under this assumption the corresponding residual is small:

$$r = b - A\hat{x} = \delta A \hat{x}, \qquad \|r\|_\infty \lesssim \mu \|A\|_\infty \|\hat{x}\|_\infty .$$

Further, from $b - A\hat{x} = A(x - \hat{x}) = r$ we get $\|\hat{x} - x\|_\infty \leq \|A^{-1}\|_\infty \, \|r\|_\infty$, and by inserting the above expression for $\|r\|_\infty$ and ignoring the difference between $\|\hat{x}\|_\infty$ and $\|x\|_\infty$ we get

$$\frac{\|\hat{x} - x\|_\infty}{\|x\|_\infty} \lesssim \mu \cdot \kappa_\infty(A) . \tag{8.12.2}$$

These observations are summarized in the following rule of thumb:

If the unit roundoff and the condition number satisfy $\mu \simeq 10^{-d}$ and $\kappa_\infty(A) \simeq 10^q$, then a stable version of Gaussian elimination can be expected to produce a solution $\hat{x}$ that has about $d-q$ correct decimal digits.

Example. In order to check the validity of (8.12.2) we have used MATLAB (with $\mu = 2^{-53} \simeq 1.11 \cdot 10^{-16}$) to generate a number of problems of varying size and condition. Each problem was generated by the script

```
x = [1:n]';  x(2:2:end) = -x(2:2:end);
A = gallery('randsvd',n,kappa);
E = norm(A,inf)*1e3*ones(n);
B = A + E;  A = B - E;
b = A*x;  xh = A\b;
```

`gallery('randsvd',n,kappa)` returns an $n \times n$ random matrix with condition number $\kappa_2(A) =$ `kappa`, and `A` is modified so that there are no rounding errors in `b`.

In Figure 8.4 we show the observed relative error and the estimate (the left and right hand sides in (8.12.2)) for $n = 50$ and varying condition number.

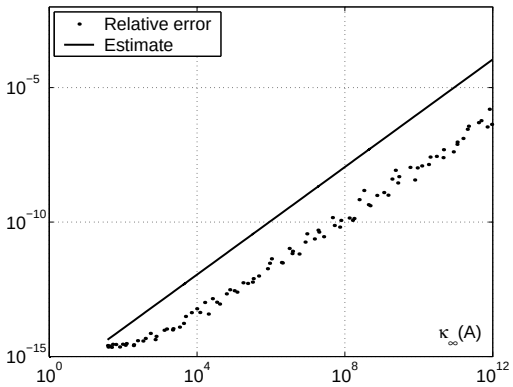


Figure 8.4. *Observed and estimated relative error;* $n = 50$.

In all the 99 cases shown the inequality in (8.12.2) is satisfied. The relative error grows with the condition number, and the rule of thumb is qualitatively correct. For this class of problems, however, we seem to get roughly two more correct digits than predicted.

Next, to see the influence of the size of the problem we have generated 92 more problems with varying n. In all of these we used $\kappa_2(A) =$ `kappa` $= 10^8$.

In order not to confuse the picture by the change from $\|\cdot\|_2$ to $\|\cdot\|_\infty$, we show the ratio between the left- and right hand side of (8.12.2) in Figure 8.5.

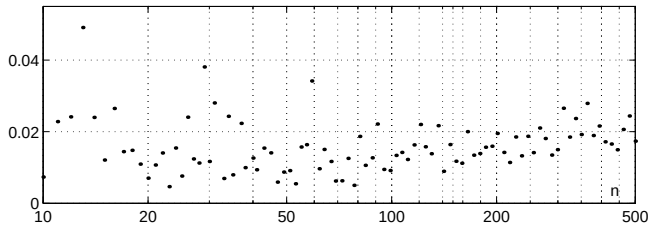


Figure 8.5. *Ratio between observed and estimated relative error;* $\kappa_2(A) = 10^8$.

All these ratios are smaller than one, so (8.12.2) is also satisfied with these problems, and there is no indication that the error grows with n.

It must be emphasized that the statement in the frame on page 240 should only be used as a rule of thumb. If a guaranteed bound on the error is needed, one should use Theorem 8.12.1. ∎

8.13. Estimation of Condition Number

When we solve a linear system of equations, and want to know the accuracy of the solution, we need an estimate of the condition number $\kappa(A) = \|A\| \cdot \|A^{-1}\|$. The computation of the inverse matrix involves about three times as much work as the solution of the system (approximately $2n^3$ and $\frac{2}{3}\,n^3$ flops, respectively), and this can be avoided as follows: For an arbitrary right hand side d we have

$$Ay = d \quad \Rightarrow \quad y = A^{-1}d \quad \Rightarrow \quad \|y\|_\infty \le \|A^{-1}\|_\infty \, \|d\|_\infty \ .$$

(We used Lemma 8.10.4 in the last implication). Therefore,

$$\|A^{-1}\|_\infty \ge \frac{\|y\|_\infty}{\|d\|_\infty}$$

for any $d \neq 0$ and $y = A^{-1}d$. The idea is to choose d with $d_i = \pm 1$ (implying that $\|d\|_\infty = 1$) so that $\|y\|_\infty$ is as large as possible.

First, consider the special case, where $A = L$, a lower triangular matrix. The solution of $Ly = d$ is found by forward substitution, which we

can formulate as

$$y_i = (d_i - s_i)/l_{ii}, \quad s_i = \sum_{j=1}^{i-1} l_{ij} y_j; \quad i = 1, 2, \ldots, n \ .$$

The idea is simple: successively choose $d_i \in \{-1, 1\}$ so that $|y_i|$ is maximized; this is obtained by $d_i = -\text{sign}(s_i)$.

Example. In MATLAB we can use the following function. Note that we cannot make use of the built-in MATLAB function `sign` because it returns `sign(0) = 0`, while we want $\text{sign}(0) = 1$.

```
function  y = lcond(L)
% y = L\d, where norm(d,inf) = 1 and norm(y,inf) is maximal
n = size(L,1);
y = -ones(n,1);                                  % initialize y
for  i = 2 : n
  si = L(i,1:i-1)*y(1:i-1);
  if  si == 0,  y(i) = -1/L(i,i);
  else,         y(i) = -(sign(si) + si)/L(i,i); end
end
```

For the matrix

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & 1 \end{pmatrix} ,$$

the algorithm gives

$$y = -\begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} , \qquad \|y\|_\infty = 8 \ .$$

The inverse matrix is

$$L^{-1} = -\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 2 & 1 & 1 & 0 \\ 4 & 2 & 1 & 1 \end{pmatrix} , \qquad \|L^{-1}\|_\infty = 8 \ .$$

So in this case the algorithm works perfectly. ∎

Normally, the algorithm does not work quite so well, but it is possible to improve it and still needing only $O(n^2)$ flops.

Now, assume that we know the LU factorization of a matrix A, $PA = LU$. The following algorithm can be used to estimate the condition number $\kappa_\infty(A)$:

1. Use `lcond` to find a solution to the lower triangular system $U^T y = d$ with $\|d\|_\infty = 1$ and $\|y\|_\infty$ large.
2. Solve $L^T c = y$ and normalize the solution: $c := (1/\|c\|_\infty)\, c$.
3. Solve $Lw = c$ and $Uz = w$.
4. $\widetilde{\kappa}_\infty = \|A\|_\infty \cdot \|z\|_\infty$.

The algorithm solves the system

$$A^T(Az) = A^T c = d \ ,$$

where d is chosen so that the solution becomes large. The reason for using both A and A^T is that this will further enhance the growth of the solution. The normalization in step 2 is made so that step 3 starts with a unit right hand side, $\|c\|_\infty = 1$.

With $PA = LU$, which is equivalent to $A = P^T LU$, we see that

$$A^T Az = U^T L^T P\, P^T LUz = U^T L^T LUz = d \ ,$$

so that we do not need the permutation matrix in the computation. The algorithm involves the solution of four triangular systems and computation of $\|A\|_\infty$. Each of these five subtasks needs about n^2 flops, so the total work is approximately $5n^2$ flops.

Example. The algorithm can be implemented in MATLAB as follows:

```
function  cnest = lucond(A,L,U)
% Estimate condition number of A whose LU factorization is
% given in L and U
y = lcond(U');          % initialize with lower triangular U'
c = L'\y;  c = c/norm(c,inf);           % solve and normalize
z = U \ (L \ c);                            % solve  LUz = c
cnest = norm(A,inf) * norm(z,inf);
```

■

MATLAB has a number of built-in functions for computing or estimating the condition number. `cond(A)` (equivalent with `cond(A,2)`) computes the Euclidean norm condition number $\kappa_2(A)$ by means of an algorithm that uses $O(n^3)$ flops. `cond(A,1)` and `cond(A,inf)` return respectively $\kappa_1(A)$ and $\kappa_\infty(A)$; in both cases the inverse matrix is computed, so these algorithms also need $O(n^3)$ flops. For *estimating* the condition number you can choose between `rcond(A)` and `condest(A)` that return respectively an approximation to $1/\kappa_1(A)$ and $\kappa_1(A)$. Both of these are based on algorithms similar to the one described above, although more

sophisticated. However, they cannot take the factorization as input, but compute it as part of the process; therefore also these estimator use $O(n^3)$ flops.

Example. We have generated a number of random matrices of order $n=50$ and varying condition. In the table below we give results for the condition number in the three norms and the accuracy of the estimators discussed:

$$\eta_{\texttt{lucond}} = \big(\text{Estimate from } \texttt{lucond}\big)/\texttt{cond(A,inf)}$$
$$\eta_{\texttt{rcond}} = \big(1/(\text{Estimate from } \texttt{rcond})\big)/\texttt{cond(A,1)}$$
$$\eta_{\texttt{condest}} = \big(\text{Estimate from } \texttt{condest}\big)/\texttt{cond(A,1)}$$

$\kappa_2(A)$	$\kappa_1(A)$	$\kappa_\infty(A)$	$\eta_{\texttt{lucond}}$	$\eta_{\texttt{rcond}}$	$\eta_{\texttt{condest}}$
$1.00{\cdot}10^{1}$	$1.25{\cdot}10^{2}$	$1.21{\cdot}10^{2}$	0.134	0.957	1.000
$1.00{\cdot}10^{3}$	$7.39{\cdot}10^{3}$	$6.40{\cdot}10^{3}$	0.256	1.000	0.796
$1.00{\cdot}10^{5}$	$4.82{\cdot}10^{5}$	$5.73{\cdot}10^{5}$	0.309	0.706	1.000
$1.00{\cdot}10^{7}$	$5.78{\cdot}10^{7}$	$5.73{\cdot}10^{7}$	0.441	0.827	0.827
$1.00{\cdot}10^{9}$	$4.23{\cdot}10^{9}$	$4.43{\cdot}10^{9}$	0.188	0.883	0.932
$1.00{\cdot}10^{11}$	$4.27{\cdot}10^{11}$	$4.64{\cdot}10^{11}$	0.511	0.516	1.000
$1.00{\cdot}10^{13}$	$5.03{\cdot}10^{13}$	$5.57{\cdot}10^{13}$	0.437	1.000	1.000
$9.95{\cdot}10^{14}$	$5.33{\cdot}10^{15}$	$4.71{\cdot}10^{15}$	0.369	1.000	1.000

Note how the three condition numbers follow each other. Our simple estimator is not quite as good as the other two, but in all cases it gets the right order of magnitude of the condition number, ie it tells correctly how many digits we can expect to loose because of rounding errors. ∎

8.14. Overdetermined Systems

In this section we present the least squares method in matrix-vector notation. In the next chapter we shall present it in a function theoretical formulation. This implies that there are overlaps between this section and Chapter 9, but they can be read independently. We start with an example.

Example. Suppose that we have made a physics experiment, where we put a series of loads $F_1, F_2, \ldots, F_5$ on an elastic spring and measured its length, $\ell_1, \ell_2, \ldots, \ell_5$.

i	F_i	ℓ_i
1	0.8	7.97
2	1.6	10.2
3	2.4	14.2
4	3.2	16.0
5	4.0	21.2

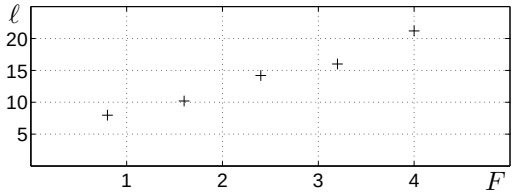


Figure 8.6. *Measured length (ℓ) as function of force (F).*

According to Hooke's law of elasticity there is a linear relation between load and length:

$$\ell = \lambda + \kappa F \ .$$

($k = 1/\kappa$ is the so-called spring constant). Thus, the points in the figure should be on a straight line, but they are not. The reason is that there are measurement errors. If we ignore these, Hooke's law with the given data result in $\lambda + \kappa F_i = \ell_i$, $i = 1, \ldots, 5$:

$$\begin{aligned} \lambda + 0.8\kappa &= 7.97 \\ \lambda + 1.6\kappa &= 10.2 \\ \lambda + 2.4\kappa &= 14.2 \\ \lambda + 3.2\kappa &= 16.0 \\ \lambda + 4.0\kappa &= 21.2 \end{aligned}$$

or, in matrix-vector notation

$$\begin{pmatrix} 1 & 0.8 \\ 1 & 1.6 \\ 1 & 2.4 \\ 1 & 3.2 \\ 1 & 4.0 \end{pmatrix} \begin{pmatrix} \lambda \\ \kappa \end{pmatrix} = \begin{pmatrix} 7.97 \\ 10.2 \\ 14.2 \\ 16.0 \\ 21.2 \end{pmatrix} \ . \qquad \blacksquare$$

In the example we derived an *overdetermined system of linear equations*, ie a linear system with more equations than unknowns. In principle we might use two points to determine the straight line, but the result from taking eg points 3 and 4 would be very different from taking points 4 and 5. The purpose of using more equations than unknowns is to *reduce the effect of measurement errors.*

In general, consider an overdetermined system

$$Ax \ \simeq \ b \ ,$$

where A is an $m \times n$ matrix, $m > n$; x is an n-vector and b is an m-vector. We write "$\simeq$" instead of "$=$" to indicate that maybe it is not possible to satisfy all equations simultaneously, but we want to find x so that the vector $Ax \in \mathbb{R}^m$ is as close as possible to $b \in \mathbb{R}^m$. As in Section 8.1 we can

partition the matrix A into column vectors,

$$A = [a_{:1} \ \cdots \ a_{:n}] \ , \qquad a_{:j} = \begin{pmatrix} a_{1j} \\ \vdots \\ a_{mj} \end{pmatrix} \ ,$$

and we see that the problem is equivalent to finding a linear combination $x_1 a_{:1} + \cdots + x_n a_{:n}$, which is as close as possible to the right hand vector b.

Example. Let $m = 3$ and $n = 2$; Figure 8.7 shows the vectors $a_{:1}$, $a_{:2}$ and b.

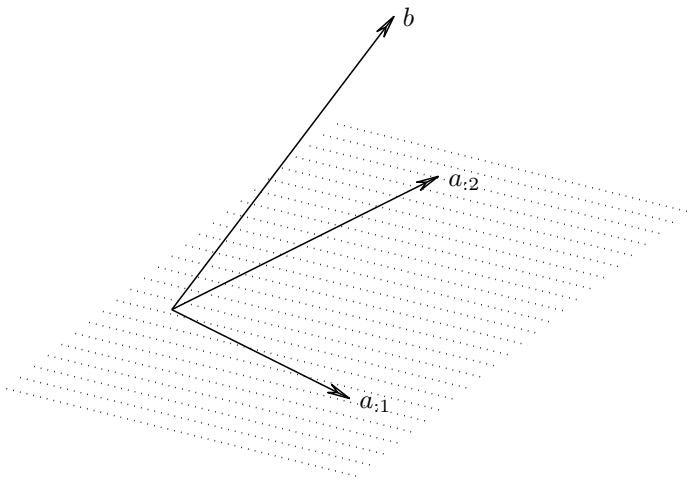


Figure 8.7. *Geometric illustration of an overdetermined system.*

Assuming that $a_{:1}$ and $a_{:2}$ are linearly independent, the vectors generated by $x_1 a_{:1} + x_2 a_{:2}$ for $x_1, x_2 \in \mathbb{R}$ form a plane in $\mathbb{R}^3$, the plane *spanned* by $x_1 a_{:1} + x_2 a_{:2}$. We cannot assume that b is in this plane. ∎

The difference between b and Ax is the *residual*,

$$r = b - Ax \ .$$

This vector depends on the choice of x, and we want to make it as "small" as possible. To quantify this we can use a vector norm, cf Section 8.10. We choose to use the 2-norm, or equivalently

$$\|r\|_2^2 \ = \ r_1^2 + \cdots + r_m^2 \ = \ r^T r \ .$$

A vector $\hat{x}$ that minimizes $\|r\|_2^2$ also minimizes $\|r\|_2$, and this is the desired solution.

Definition 8.14.1. Least squares method. Given an overdetermined system of equations $Ax \simeq b$, where $A \in \mathbb{R}^{m\times n}$ with $m > n$. A least squares solution is a vector $\hat{x}$ that minimizes the Euclidean norm of the residual, ie $\hat{x}$ is a solution to the minimization problem $\min_x \|b - Ax\|_2$.

Example. For the problem in the previous example the residual has minimal length if it is orthogonal to the plane spanned by $a_{:1}$ and $a_{:2}$, cf Figure 8.8.

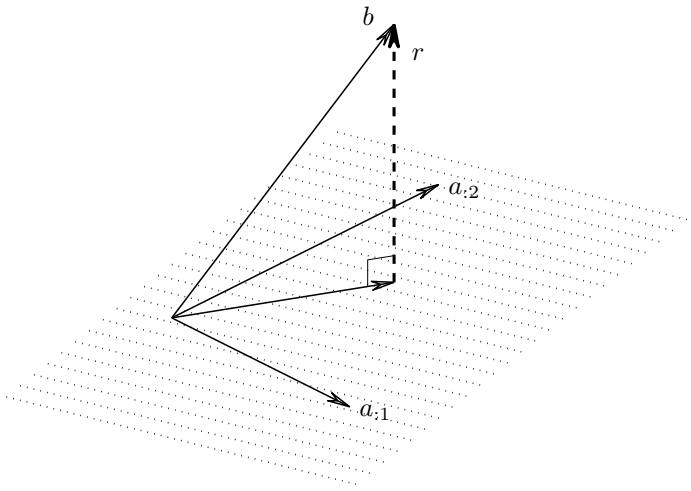


Figure 8.8. *Geometric illustration of least squares solution.*

The condition is that r is orthogonal to all vectors in the plane, and this is satisfied when r is orthogonal to the two vectors spanning it: $a_{:1}^T r = a_{:2}^T r = 0$.

■

This simple example generalizes: The column vectors of A span a subspace of $\mathbb{R}^m$ (the so-called *range of* A), and to get the least squares solution, the residual $r = b - Ax$ should be orthogonal to this subspace. This condition is satisfied if r is orthogonal to each column vector in A:

$$a_{:j}^T r = 0 \ , \quad j = 1, \ldots, n \ ,$$

or, in matrix-vector notation

$$A^T r = A^T (b - Ax) = 0 \ .$$

Now we can formulate the following theorem about the solution of least squares problems.

Theorem 8.14.2. Least squares solution. Consider the overdetermined system $Ax \simeq b$, $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $x \in \mathbb{R}^n$ with $m > n$. A solution $\hat{x}$ of the least squares problem

$$\min_x \|b - Ax\|_2$$

satisfies the *normal equations*

$$A^T A\,\hat{x} = A^T b\ .$$

If the column vectors in A are linearly independent, then the matrix $A^T A$ is positive definite, and the solution $\hat{x}$ is unique.

Proof. We already saw that the geometric generalization lead to

$$A^T(b - A\hat{x}) = A^T b - A^T A\,\hat{x} = 0\ ,$$

and the normal equations follow immediately.
Next, the matrix $A^T A$ is symmetric: $(A^T A)_{ij} = a_{:i}^T a_{:j} = (A^T A)_{ji}$, and if the column vectors in A are linearly independent, then $x \neq 0$ is equivalent to $y = Ax \neq 0$, so that

$$x^T A^T A x = y^T y = y_1^2 + \cdots + y_m^2 \ > \ 0 \quad \text{for } x \neq 0\ .$$

This shows that $A^T A$ is positive definite. Therefore this matrix is nonsingular, so the solution to the normal equations is unique.
Finally, we have to show that $\hat{x}$ minimizes $\|b - Ax\|_2$: Let $x = \hat{x} + u$, for some arbitrary vector u. Then $r = b - A(\hat{x} + u) = \hat{r} - Au$ with $A^T\hat{r} = 0$, and

$$\begin{aligned}\|r\|_2^2 &= (\hat{r} - Au)^T(\hat{r} - Au) = \hat{r}^T\hat{r} - \hat{r}^T Au - u^T A^T\hat{r} + u^T A^T Au \\ &= \hat{r}^T\hat{r} + u^T A^T Au \ > \ \hat{r}^T\hat{r} \quad \text{if } u \neq 0\ .\end{aligned}$$

□

Example. The overdetermined system in the example on page 244 is given by

$$A = \begin{pmatrix} 1 & 0.8 \\ 1 & 1.6 \\ 1 & 2.4 \\ 1 & 3.2 \\ 1 & 4.0 \end{pmatrix}, \qquad b = \begin{pmatrix} 7.97 \\ 10.2 \\ 14.2 \\ 16.0 \\ 21.2 \end{pmatrix}\ .$$

The associated normal equations are

$$\begin{pmatrix} 5 & 12 \\ 12 & 35.2 \end{pmatrix} \hat{x} = \begin{pmatrix} 69.57 \\ 192.776 \end{pmatrix} ,$$

with the solution

$$\hat{x} = \begin{pmatrix} 4.2360 \\ 4.0325 \end{pmatrix} .$$

Thus, the least squares solution corresponds to $\lambda = 4.2360$, $\kappa = 4.0325$ (and the spring constant $k = \kappa^{-1} = 0.2480$). The corresponding straight line is shown in Figure 8.9 together with the data points.

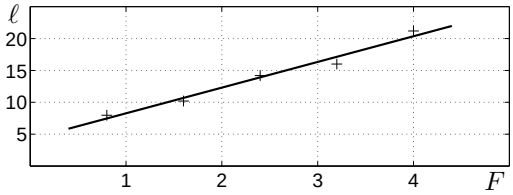


Figure 8.9. *Least squares fit to spring data.*

■

The normal equations can be very ill conditioned, with associated risk of getting a poor determination of the least squares solution. In some cases it is possible to reformulate the problem so that we get a better determination of $\hat{x}$. This is illustrated in the next example.

Example. Assume that we are asked to fit a straight line to the data

t_i	998	999	1000	1001	1002
$f(t_i)$	3.765	4.198	5.123	5.888	6.184

We can express the straight line in the form $f^*(t) = x_1 + x_2 t$, and get the overdetermined system $Ax \simeq b$ with

$$A = \begin{pmatrix} 1 & 998 \\ 1 & 999 \\ 1 & 1000 \\ 1 & 1001 \\ 1 & 1002 \end{pmatrix} , \qquad b = \begin{pmatrix} 3.765 \\ 4.198 \\ 5.123 \\ 5.888 \\ 6.184 \end{pmatrix} .$$

The associated normal equations have the coefficient matrix

$$A^T A = \begin{pmatrix} 5 & 5000 \\ 5000 & 5000010 \end{pmatrix} , \qquad \kappa_\infty(A^T A) \simeq 5 \cdot 10^{11} .$$

Another way of expressing the straight line is $f^*(t) = x_1 + x_2(t - 1000)$. The corresponding system $Ax \simeq b$ has the same b, while

$$A = \begin{pmatrix} 1 & -2 \\ 1 & -1 \\ 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{pmatrix} , \quad A^T A = \begin{pmatrix} 5 & 0 \\ 0 & 10 \end{pmatrix} , \qquad \kappa_\infty(A^T A) = 2 .$$

■

8.15. QR Factorization

An *orthogonal matrix* is a matrix Q that satisfies

$$Q^T Q = Q\, Q^T = I . \tag{8.15.1}$$

Since $(Q^T Q)_{ij} = q_{:i}^T q_{:j}$, this condition shows that the column vectors in Q are *orthonormal*, ie they are mutually orthogonal and have the Euclidean length $\|q_{:j}\|_2 = 1$. Similarly, the condition $QQ^T = I$ implies that also the row vectors in Q are orthonormal.

A vector $y = Qx$ is said to be an *orthogonal transformation* of x if Q an orthogonal matrix. By means of (8.15.1) and the relation $\|z\|_2^2 = z^T z$ we see that $\|y\|_2^2 = \|Qx\|_2^2 = x^T Q^T Q\, x = x^T x = \|x\|_2^2$. In words: *the Euclidean length is invariant under orthogonal transformations.* This property makes orthogonal transformations very useful in matrix computations, eg in the solution of least squares problems. Before we demonstrate that, however, we formulate the basic result of this section as a theorem.

Theorem 8.15.1. QR factorization. Any $m \times n$ matrix A with $m \geq n$ can be factored

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} ,$$

where $Q \in \mathbb{R}^{m \times m}$ is orthogonal and $R \in \mathbb{R}^{n \times n}$ is right (upper) triangular. The matrix Q can be partitioned into $Q = \begin{pmatrix} \widehat{Q} & \overline{Q} \end{pmatrix}$ with $\widehat{Q}$ consisting of the first n column vectors in Q, and the "economy size" (or "thin") version of the factorization is

$$A = \widehat{Q} R .$$

If the column vectors in A are linearly independent, then R is nonsingular.

We give a constructive proof, in the form of an algorithm that computes the factorization. First, however, we look at the last statement. If we replace A by $\widehat{Q}^T R$ in the normal equations matrix, we get[9)]

$$A^T A = R^T \widehat{Q}^T \widehat{Q}\, R = R^T R \ .$$

We saw in the previous section that the matrix $A^T A$ is nonsingular if the column vectors in A are linearly independent. In that case

$$0 \neq \det(A) = \det(R^T) \cdot \det(R) = \big(\det(R)\big)^2 \ ,$$

so R is nonsingular, and we have proved the last statement in the theorem. In the remainder of this section we shall assume that R is nonsingular.

Next, we show how the factorization can be used to solve a least squares problem:

$$\begin{aligned} \|r\|_2^2 = \left\|Q^T r\right\|_2^2 &= \left\| Q^T \left(b - Q \begin{pmatrix} R \\ 0 \end{pmatrix} x \right) \right\|_2^2 \\ &= \left\| \begin{pmatrix} \widehat{Q}^T b \\ \overline{Q}^T b \end{pmatrix} - \begin{pmatrix} Rx \\ 0 \end{pmatrix} \right\|_2^2 = \left\| \widehat{Q}^T b - Rx \right\|_2^2 + \left\| \overline{Q}^T b \right\|_2^2 \ . \end{aligned}$$

(We used the invariance of $\|\cdot\|_2$; that $Q^T Q = I$; the partitioning of Q; and the last reformulation follows from the definition of the vector 2-norm). We see that $\|r\|_2$ is minimized by $\hat{x}$ defined as the solution of the linear system

$$R\hat{x} = \widehat{Q}^T b \ . \tag{8.15.2}$$

Thus, the least squares solution can be computed without forming the normal equations.

The QR factorization is equivalent to

$$Q^T A \ = \ \begin{pmatrix} R \\ 0 \end{pmatrix} \ , \tag{8.15.3}$$

and we now show how this can be constructed through a series of orthogonal transformations, each of which zeros elements in specified positions in the matrix. This is similar to Gaussian elimination for a square matrix, where the upper triangular U is obtained through a series of Gauss transformations. We shall show that we can find orthogonal matrices $Q_1, Q_2, \ldots, Q_p$ so that

[9)] The column vectors in $\widehat{Q}$ ore orthonormal, so $\widehat{Q}^T \widehat{Q} = I$, the unit matrix of order n. When $m > n$, the matrix $\widehat{Q}\widehat{Q}^T \in \mathbb{R}^{m \times m}$ is different from the mth order unit matrix, so $\widehat{Q}$ is **not** an orthogonal matrix.

$$Q_p \cdots Q_2 Q_1 A \;=\; \begin{pmatrix} R \\ 0 \end{pmatrix} \;. \tag{8.15.4}$$

This is equivalent to (8.15.3) if $Q = Q_1^T Q_2^T \cdots Q_p^T$.

Lemma 8.15.2. If $Q_1, Q_2, \ldots, Q_p$ are orthogonal, then the product

$$Q = Q_1^T Q_2^T \cdots Q_p^T$$

is orthogonal.

Proof. We obviously only need to show that the product of two orthogonal matrices is orthogonal:

$$(Q_1^T Q_2^T)^T (Q_1^T Q_2^T) = Q_2 Q_1 Q_1^T Q_2^T = Q_2 Q_2^T = I \;,$$

and similarly $(Q_1^T Q_2^T)(Q_1^T Q_2^T)^T = Q_1^T Q_2^T Q_2 Q_1 = I$. □

All that remains now is to show that we can construct a sequence of orthogonal transformations (ie $\{Q_k\}_{k=1}^p$) so that the resulting matrix is upper triangular, cf (8.15.4).

Example. A *rotation matrix*

$$Q = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

is orthogonal (show this!) and Qx turns the vector x the angle θ clockwise. Suppose that we want to zero the second element in a vector x:

$$Q \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} u \\ 0 \end{pmatrix} \;.$$

By choosing θ so that $\cos\theta = x_1/\sqrt{x_1^2 + x_2^2}$, $\sin\theta = x_2/\sqrt{x_1^2 + x_2^2}$, we get

$$\frac{1}{\sqrt{x_1^2 + x_2^2}} \begin{pmatrix} x_1 & x_2 \\ -x_2 & x_1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \sqrt{x_1^2 + x_2^2} \\ 0 \end{pmatrix} \;.$$

Note that we do not have to choose the angle θ explicitly. ■

In numerical linear algebra rotation matrices are often called *Givens transformations*. They can be embedded in unit matrices of larger dimension and be used to zero elements in vectors and matrices of order larger than two. The complete transformation (8.15.4) can be achieved with $p = \frac{1}{2}n(2m-n-1)$ Givens transformations. Instead of giving details about this, we shall introduce another type of orthogonal transformations in $\mathbb{R}^m$ that simultaneously zero a sequence of elements in a vector or a matrix.

Let $v \neq 0$ be an arbitrary m-vector and define the matrix[10)]

$$Q = I - \frac{2}{v^T v}\, v\, v^T \ . \tag{8.15.5}$$

Q is symmetric and a simple calculation shows that it is also orthogonal. A matrix defined by (8.15.5) is called a reflection matrix or a *Householder matrix*. A transformation with such a matrix is called a *Householder transformation*, and it is done without explicitly forming the matrix Q:

$$\big(I - \frac{2}{v^T v}\, v\, v^T\big) z = z - \beta v, \quad \beta = \frac{2 v^T z}{v^T v} \ . \tag{8.15.6}$$

If $v^T v$ is known, then the cost of performing the transformation Qz this way is approximately $4n$ flops, whereas the matrix vector multiplication would cost $2n^2$ flops.

Now, let x and y be two vectors with $\|x\|_2 = \|y\|_2$, ie $x^T x = y^T y$, and choose $v = x - y$. Then

$$\begin{aligned} v^T v &= x^T x + y^T y - 2x^T y = 2(x^T x - x^T y) \ , \\ v^T x &= x^T x - x^T y \ . \end{aligned}$$

Therefore, the transformation Qx defined by this v gives $\beta = 1$ in (8.15.6) and $Qx = x - (x - y) = y$. This relation can be used to zero elements $2, \ldots, m$ in a vector x, ie, we want $y_i = 0$, $i = 2, \ldots, m$. The invariance of the norm requires that $y_1^2 = \|x\|_2^2$, so we get

$$v = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} - \begin{pmatrix} y_1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} x_1 - y_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} \ , \qquad y_1 \ = \ \pm \|x\|_2 \ .$$

To avoid cancellation we choose $y_1 = -\mathrm{sign}(x_1) \cdot \|x\|_2$. Further, to simplify the transformations we normalize the vector:

$$Q = I - \frac{2}{v^T v}\, v\, v^T = I - 2\, u\, u^T \ , \quad u = \frac{1}{\|v\|_2}\, v \ .$$

The *Householder vector* u has length $\|u\|_2 = 1$.

Example. From the above expressions for $v^T v$ and y it follows that

$$\|v\|_2^2 = v^T v = 2(\|x\|_2^2 + |x_1| \cdot \|x\|_2) \ .$$

[10)] Remember that the inner product $v^T v$ is a scalar and the outer product $v\, v^T$ is a matrix.

Thus, $\|v\|_2$ can be obtained by a simple updating of $\|x\|_2$ instead of the $2m$ flops involved in using the definition $\|v\|_2^2 = v_1^2 + \cdots + v_m^2$. This is exploited in the following MATLAB function for computing the Householder vector for a given x. As mentioned on page 242 we cannot use the MATLAB function `sign` because it returns `sign(0)` $= 0$.

```
function  u = househ(x)
% Compute Householder vector u so that (I - 2u*u')*x = N*e1,
% where  |N| = ||x||2  and e1 is the first unit vector.
v = x;   nx = norm(x);
if  x(1) < 0,  v(1) = x(1) - nx;
else,          v(1) = x(1) + nx; end
u = (1/sqrt(2*nx*(nx + abs(x(1))))) * v;
```

■

For a given Householder vector u the transformation (8.15.6) simplifies to $Qz = z - (2u^T z)u$, and for a matrix we get

$$QZ = Z - 2u(u^T Z) \ .$$

(Note that the jth component in $u^T Z$ is $u^T z_{:j}$, so that $(QZ)_{:j} = z_{:j} - (2u^T z_{:j})u$, as it should be).

Example. A MATLAB function that applies a Householder transformation is simple to construct:

```
function  Y = apphouse(u, X)
% Householder transformation, Y = (I - 2u*u')*X
Y = X - 2*u*(u'*X);
```

To zero the last three elements in $x = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}^T$:

```
>> x = [1:4]';
>> u = househ(x);  y = apphouse(u,x)
y = -5.4772
          0
          0
          0
```

■

Now we have all the tools necessary to generate the QR factorization of a matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$. First use $x = a_{:1}$ to determine a Householder transformation Q_1 that zeros elements $2, \ldots, m$ in the first column:

$$A_2 = Q_1 A = \begin{pmatrix} r_{11} & r_1^T \\ 0 & \widetilde{A}_2 \end{pmatrix} \ .$$

Next, determine $\widetilde{Q}_2 \in \mathbb{R}^{(m-1)\times(m-1)}$ that puts zeros below the main diagonal in the first column of $\widetilde{A}_2$ (ie the second column of A_2), and augment $\widetilde{Q}_2$ to the orthogonal $m\times m$ matrix Q_2

$$Q_2 = \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{Q}_2 \end{pmatrix} .$$

Then

$$A_3 = Q_2 A_2 = \begin{pmatrix} 1 & 0 \\ 0 & \widetilde{Q}_2 \end{pmatrix} \begin{pmatrix} r_{11} & r_1^T \\ 0 & \widetilde{A}_2 \end{pmatrix}$$
$$= \begin{pmatrix} r_{11} & r_1^T \\ 0 & \widetilde{Q}_2 \widetilde{A}_2 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & \cdots \\ 0 & r_{22} & r_2^T \\ 0 & 0 & \widetilde{A}_3 \end{pmatrix} .$$

Note that the transformation with Q_2 changes neither the elements in the first row of A_2 nor the zeros in the first column. The process continues, and after $p = n$ steps we have

$$A_{n+1} = Q_n \cdots Q_2 Q_1 A = \begin{pmatrix} R \\ 0 \end{pmatrix} .$$

We recognize this as (8.15.4), and the proof of Theorem 8.15.1 is finished.

Example. The following MATLAB function implements the QR factorization by use of the functions from the two previous examples. The output is the Householder vectors and the upper triangular matrix R.

```
function  [H, R] = qrfac(A)
% QR factorization of A by Householder transformations.
% Householder vectors are stored in H
[m,n] = size(A);  H = zeros(m,n);    % make room for vectors
for  k = 1 : n
  ii = k:m;   jj = k:n;            % active rows and columns
  H(ii,k) = househ(A(ii,k));            % Householder vector
  A(ii,jj) = apphouse(H(ii,k),A(ii,jj));         % transform
end
R = triu(A(1:n,:));            % upper triangular n*n factor
```

For an overdetermined system of equations, $Ax \simeq b$, we have to apply the same transformations to the right-hand side,

$$Q_n \cdots Q_2 Q_1 b = \begin{pmatrix} \hat{b} \\ \overline{b} \end{pmatrix} ,$$

and then solve the system $R\hat{x} = \hat{b}$. This is implemented in

```
function  x = qrsolv(H,R, b)
% Solve an overdetermined system Ax = b by means of Housholder
% transformation.  H,R  is the output from qrfac(A)
[m,n] = size(H);
for  k = 1 : n
  ii = k:m;  b(ii) = apphouse(H(ii,k),b(ii));
end
x = R \ b(1:n);
```

The following commands set up and solve the system for the elastic spring problem discussed in previous examples.

```
>> A = [ones(1,5); 0.8*(1:5)]';  b = [7.97 10.2 14.2 16 21.2]';
>> [H, R] = qrfac(A);  x = qrsolv(H,R, b)
x =  4.2360
     4.0325
```

■

Example. MATLAB has a built-in function `qr` that computes the QR factorization by means of Householder transformations. The call

```
>> [Qh, R] = qr(A,0)
```

returns the economy size factorization, and this command can be followed by `x = R\(Qh'*b)` to get the least squares solution. ■

Orthogonal transformations are considerably more stable than Gauss transformations: Because the length of a vector is preserved, there cannot be the growth of matrix elements which caused the loss of accuracy in connection with Gaussian elimination.

The work involved in the factorization, `qrfac`, is about $2n^2(m-n/3)$ flops, and for $m=n$ this is twice the work needed by Gaussian elimination. For an overdetermined system we have the choice between using the normal equations or orthogonal transformation. If $m \simeq n$, the cost of the two methods is almost the same, whereas the cost of orthogonal transformation is about twice the cost of the normal equations if $m \gg n$. You can say that this extra cost is more than paid off by the better stability properties of orthogonal transformation.

Example. There is a risk of loosing information when forming the normal equations. For the matrix

$$A = \begin{pmatrix} 1 & 1 \\ \epsilon & 0 \\ 0 & \epsilon \end{pmatrix}$$

the normal equations matrix is $A^TA = \begin{pmatrix} 1+\epsilon^2 & 1 \\ 1 & 1+\epsilon^2 \end{pmatrix}$, and if $|\epsilon| \le \sqrt{\mu}$, then $fl[A^TA] = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, which is singular. Under the same assumption we get

$$fl[\widehat{Q}] = \begin{pmatrix} -1 & \epsilon\sqrt{0.5} \\ -\epsilon & -\sqrt{0.5} \\ 0 & \sqrt{0.5} \end{pmatrix}, \qquad fl[R] = \begin{pmatrix} -1 & -1 \\ 0 & \epsilon\sqrt{2} \end{pmatrix},$$

so information is preserved when we use orthogonal transformation. ■

Example. In an example on page 219 we discussed what happens in MATLAB, when the command `x = A\b` is issued with a square matrix A. If A has more rows than columns (the system $Ax = b$ is overdetermined), then `x` is the least squares solution, computed via orthogonal transformation. (We assume that `A` and `b` have the same number of rows; if not, we get an error message instead).

If $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$ with $m < n$ we have an *underdetermined system of equations*. Assuming that the rows in A are linearly independent, it is possible to permute the columns so that the first m columns are linearly independent. Using the concepts from Section 8.5 we can express this as

$$AP^T x = \begin{pmatrix} A_1 & | & A_2 \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} = b \ ,$$

where P is a permutation matrix, $A_1 \in \mathbb{R}^{m\times m}$ is nonsingular, and we have introduced

$$P^T x = \begin{pmatrix} y \\ z \end{pmatrix} \ .$$

It follows that

$$x = P \begin{pmatrix} y \\ z \end{pmatrix}, \qquad y = A_1^{-1}(b - A_2 z) \ ,$$

for any choice of $z \in \mathbb{R}^{n-m}$. MATLAB returns a so-called *basic solution*, defined by $z = 0$. ∎

Exercises

E1. Let $x \in \mathbb{R}^n$. Show that

(a) $\|x\|_2 \ \leq \ \|x\|_1 \ \leq \ \sqrt{n}\,\|x\|_2$,

(b) $\|x\|_\infty \ \leq \ \|x\|_1 \ \leq \ n\|x\|_\infty$,

(c) $\|x\|_\infty \ \leq \ \|x\|_2 \ \leq \ \sqrt{n}\,\|x\|_\infty$.

E2. Figures 8.2 and 8.3 illustrate the matrix 2- and 1-norms. Sketch a similar illustration of $\|A\|_\infty$ for the same matrix A.

E3. Let Q be an orthogonal matrix. Compute the norm $\|Q\|_2$ and condition number $\kappa_2(Q)$.

Computer Exercises

C1. In connection with the solution of the system $Ax = b$ of order n we have computed the LU factorization, $PA = LU$. A common problem in applications is that we also have to solve a modified problem

$$(A + uv^T)z = d \ ,$$

where u and v are n-vectors.

(a) Show that if $A+uv^T$ is nonsingular, then its inverse can be expressed by the *Sherman-Morrison formula*

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

(b) Write a MATLAB function that uses $O(n^2)$ flops to solve the modified problem.

C2. This exercise is an experimental study of the time required to factorize a banded matrix of order n.

(a) Make a MATLAB program with the following layout

```
for  n = 50 :  50 :  500
   Generate the tridiagonal matrix of order n defined
   by generalizing A in the example on page 222.
   Use tic–toc to determine the execution time t for lu(A)
end
Plot t as a function of n
```

(b) As (a), except that `lu` operates on `full(A)` .

C3. (a) Suppose that we have to solve the system $Ax = b$ with

$$A = \begin{pmatrix} 6.86 & 5.26 & 7.01 & 0.474 \\ 5.88 & 0.919 & 9.10 & 7.36 \\ 8.30 & 6.53 & 7.62 & 3.28 \\ 8.46 & 4.16 & 2.62 & 6.32 \end{pmatrix}, \qquad b = \begin{pmatrix} 7.56 \\ 9.91 \\ 3.65 \\ 2.47 \end{pmatrix} \ ,$$

and let $\widehat{A}$ and $\widehat{b}$ denote the matrix and vector obtained by rounding the elements to floating point numbers in the system $(2, 23, -126, 127)$. Estimate the difference between the solutions to $Ax = b$ and $\widehat{A}x = \widehat{b}$.

(b) Same problem, except that the matrix is changed to (the so-called Hilbert matrix of order 4)

$$A = \begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{pmatrix} \ .$$

References

As the name indicates, Gaussian elimination is a classical method that has been used for centuries to solve linear systems of equations by hand. At the end of the 1940s it became possible to use computers to solve systems with a relatively large number of unknowns,[11] and it was generally believed that rounding errors would accumulate catastrophically and completely dominate the result. These fears proved to be exaggerated, and the method is applicable for very large systems. The first analyses of rounding errors in Gaussian elimination were made by J.H. Wilkinson in the early 1950s and were collected in

> J.H. Wilkinson, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

Wilkinson had participated in the development of the computers in England during the Second World War and was among the first users, specializing in matrix problems. After the war there was a fast development, both in computer technology and in theory and algorithms for linear algebra. Wilkinson summarized the state-of-the-art in numerical linear algebra at the mid 1960s in the book

> J.H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965.

This book had great impact on the development of software for problems with full or banded matrices. The program libraries LINPACK (linear systems of equations) and EISPACK (matrix eigenvalue problems) appeared in the 1970s. The currently best library is LAPACK, which includes programs both for systems of equations and for eigenvalue problems. It is constructed so that programs execute efficiently on computers with memory hierarchy (which is the case with modern computers).

> E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, *LAPACK User's Guide* 3rd edition, SIAM, Philadelphia, PA, 1999. Also see http://www.netlib.org/lapack/

[11] At that time a system of order 20 was huge. On one occasion it took two days to compute the inverse of a 16×16 matrix; see page 283 in Higham's book.

There is a rich literature in numerical linear algebra. Good presentations of the state-of-the-art in numerical linear algebra at the mid 1990s are given in

> J.W. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, PA, 1997.
>
> G.H. Golub and C.F. Van Loan, *Matrix Computations* 3rd edition, Johns Hopkins Press, Baltimore, Maryland, 1996.

Least squares problems are treated in

> Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, PA, 1996.

The reader specially interested in error analysis and theory of matrix algorithms should study

> N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd edition, SIAM, Philadelphia, PA, 2002.

Chapter 9

Approximation

9.1. Introduction

Assume that some function f is known by an analytic expression. The function may be an elementary function like eg $\sin x$, e^x, $\ln x$, or a more complicated function like $\int_0^x e^{-t^2} dt$. For x in some interval $[a, b]$, we want to approximate f by a simpler function f^*, that can be evaluated fast.

An obvious choice is to let f^* be a polynomial. The value of a polynomial can be efficiently computed by Horner's rule, and it is also easy to differentiate and integrate a polynomial. By choosing a sufficiently high degree of the polynomial, we should be able to get a good approximation.

A "good" approximation is one that makes the error function $f - f^*$ small in some sense. We might eg require that the error is zero at certain points, and use interpolation as in Chapter 5. Now, however, it is natural to require that the error is small in the entire interval. We may, eg, determine f^* such that

$$\max_{a \le x \le b} |f(x) - f^*(x)|$$

is minimized. This f^* is said to be the best approximation to f in the *maximum norm* or *Chebyshev norm*, and f^* is said to be the *minimax approximation* of f.

We shall see that it is easier to determine f^* that minimizes

$$\int_a^b \big(f(x) - f^*(x)\big)^2 dx \ ,$$

or – more generally

$$\int_a^b w(x)\big(f(x) - f^*(x)\big)^2 dx \ ,$$

where $w(x)$ is some weight function. This f^* is said to be the best approximation in the *least squares* sense, and is also called the *least squares fit* to f.

Example. We want to approximate $f(x) = \sin(\frac{\pi}{2}x)$ on the interval $[0,1]$ by a straight line, $f^*(x) = c_0 + c_1 x$. If we determine the parameters c_0 and c_1 such that

$$\int_a^b \big(f(x) - f^*(x)\big)^2\,dx = \int_0^1 \big(\sin(\frac{\pi}{2}x) - (c_0 + c_1 x)\big)^2\,dx$$

is minimized, we get $f^*(x) = 0.1148 + 1.0437x$. This approximation is shown by a dotted line in Figure 9.1.

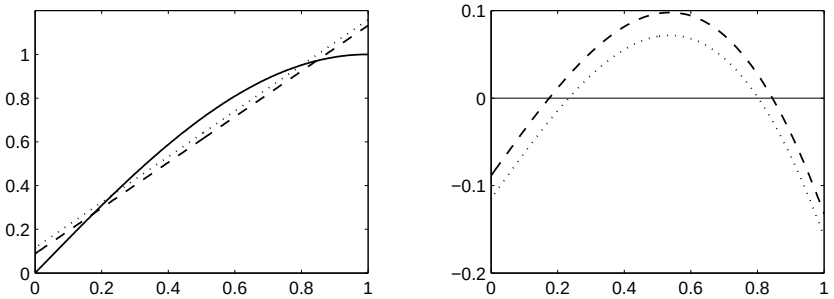


Figure 9.1. *Left:* $f(x) = \sin(\frac{\pi}{2}x)$ *and two linear approximations Right: corresponding error curves* $f - f^*$.

If we minimize

$$\int_a^b w(x)\big(f(x) - f^*(x)\big)^2\,dx = \int_a^b \frac{\big(\sin(\frac{\pi}{2}x) - (c_0 + c_1 x)\big)^2}{\sqrt{x(1-x)}}\,dx\ ,$$

we get $f^*(x) = 0.08857 + 1.0273x$, indicated by the dashed line in the figure. It is clearly seen that close to the endpoints, the second approximation has a smaller error. This is reasonable, since the weight function is large for x close to 0 and 1.

For this problem it is easy to find the minimax approximation, see the example on page 300. This approximation has a maximum error $d \simeq 0.1053$, and the error function oscillates between $\pm d$ in 0, 0.5607 and 1. There is not a big difference between that approximation and the one determined by the weight function $w(x) = 1/\sqrt{x(1-x)}$. ∎

The situation is different in a so-called *data fitting problem*. Here, we are only given approximate values of f in certain points. The function may eg describe the relation between two physical entities, x and $f(x)$,

and by measurements one has determined $f_i = f(x_i) + \epsilon_i$, $i = 1, 2, \ldots, m$, where the errors ϵ_i are (unknown) measurement errors. If there were no measurement errors, we might interpolate the points (x_i, f_i), but we can reduce the effect of measurement errors if we use the method of least squares, as discussed in connection with overdetermined systems of equations, Section 8.14.

In this chapter we shall formulate the approximation problem in a general way, so that the same terminology can be used in the case where f is only known in discrete points, and in the continuous case where we have a formula defining f.

The approximating function f^* should have a similar behaviour as f, and sometimes it is better to use other functions than polynomials, eg trigonometric, exponential or rational functions. We cannot, eg, expect to get a good polynomial approximation to a function close to a discontinuity.

Example. If we demand that $|\tan(\frac{\pi}{4}x) - f^*(x)| \leq 2{\cdot}10^{-6}$ for all $x \in [0, 1]$, then we have to use a polynomial of degree at least nine. The same accuracy can be obtained by proper choice of the three parameters in the rational function

$$f^*(x) = x\big(a_0 + \frac{a_1}{a_2 + x^2}\big) \ .$$

■

Sometimes, a simple trick can be used to reduce a general approximation problem to a problem that is suited for polynomial approximation. Assume, eg, that $f(x) = \cot x$ shall be approximated. Since

$$\cot x = \frac{1}{x}\big(1 - \frac{x^2}{3} - \frac{x^4}{45} - \cdots\big) \quad \text{for} \ \ |x| < \pi \ ,$$

it is natural to determine a polynomial of the type $\sum_{k=0}^{n} a_k x^{2k}$, which approximates $x \cot x$.

Another way of getting a good accuracy by polynomial approximation, is to used *different* polynomials in different parts of the interval, cf Section 5.9.

9.2. Some Important Concepts

We shall formulate the approximation problem in terms that are sufficiently general to describe both the *continuous case*, when f is a continuous function defined on an interval $[a, b]$, and the *discrete case*, when f is known only on a *grid* $G = \{x_1, x_2, \ldots, x_m\}$. In both cases f is an element of a *linear space*, and the approximating function f^* is chosen from a *subspace*. We therefore start by reminding the reader of these two concepts.

Definition 9.2.1. A space $\mathbb{L}$ is said to be *linear* if, for arbitrary elements f and g and an arbitrary real number α, it holds that αf and $f+g$ also belong to $\mathbb{L}$.

Definition 9.2.2. A nonempty subset $\mathbb{U}$ of a linear space $\mathbb{L}$ is said to be a *subspace* if, for arbitrary elements f_1 and f_2 in $\mathbb{U}$ and an arbitrary real number α, also αf_1 and f_1+f_2 belong to $\mathbb{U}$.

The linear spaces that we are going to use are

1. the space of all functions that are continuous on the interval $[a, b]$, ie the space $C[a, b]$,
2. the space of all real valued vectors with m elements, ie the space $\mathbb{R}^m$.

In the continuous case $f \in C[a, b]$ shall be approximated by

$$f^*(x) = c_0\varphi_0(x) + c_1\varphi_1(x) + \cdots c_n\varphi_n(x) \ .$$

In the discrete case we consider a column vector of function values

$$f_{\mathrm{G}} = \big(f(x_1), f(x_2), \ldots, f(x_m)\big)^T \ .$$

The notation refers to the grid $G = \{x_i\}_{i=1}^m$ of arguments. The vector f_{G} shall be approximated by

$$f^*_{\mathrm{G}} = c_0\varphi_{0\mathrm{G}} + c_1\varphi_{1\mathrm{G}} + \cdots c_n\varphi_{n\mathrm{G}} \ .$$

The $\varphi_0, \varphi_1, \ldots, \varphi_n$ are given, continuous, linearly independent functions (see Definition 9.2.7). In case of polynomial approximation we may use $\varphi_k(x) = x^k$, $k = 0, 1, \ldots, n$, but we shall see that it is better to express f^* as a linear combination of *orthogonal* polynomials.

To measure the magnitude of the error, we need the concept of a norm:

Definition 9.2.3. Let f and g be elements in a linear space $\mathbb{L}$. A *norm* $\|\cdot\|$ is a mapping $\mathbb{L} \mapsto \mathbb{R}$ with the properties

$$\begin{aligned} \|f\| &\geq 0 \qquad \text{for all } f\ , \\ \|f\| = 0 &\ \Leftrightarrow\ f = 0\ , \\ \|\alpha f\| &= |\alpha| \cdot \|f\| \qquad \text{for arbitrary } \alpha \in \mathbb{R}\ , \\ \|f+g\| &\leq \|f\| + \|g\| \qquad \text{(triangle inequality)}\ . \end{aligned}$$

We use $\|f-g\|$ to measure the distance between two elements f and g in $\mathbb{L}$, and we can now give a precise formulation of the problem.

The approximation problem. Let $\mathbb{U}$ be a subspace of the normed linear space $\mathbb{L}$. Given $f \in \mathbb{L}$, determine $f^* \in \mathbb{U}$ such that

$$\|f - f^*\| = \min_{g \in \mathbb{U}} \|f - g\|\ .$$

Common definitions of the norm of a function $f \in C[a,b]$ are

$$\begin{aligned} \textit{Euclidean norm} \qquad & \|f\|_2 \ = \ \sqrt{\int_a^b f(x)^2\, dx}\ , \\ \textit{maximum norm} \ \text{or} \ \textit{Chebyshev norm} \qquad & \|f\|_\infty \ = \ \max_{a \leq x \leq b} |f(x)|\ . \end{aligned}$$

Both of these are special cases of the L_p-norm

$$\|f\|_p = \big(\int_a^b |f(x)|\, dx\big)^{1/p}\ .$$

It can be shown that, for $p \geq 1$ this expression does define a norm, ie it has the properties in Definition 9.2.3.

For functions defined on a grid $G = \{x_i\}_{i=1}^m$ the corresponding norm is defined as

$$\|f\|_{p,G} = \big(\sum_{i=1}^m |f(x_i)|^p\big)^{1/p}\ .$$

By comparison with Section 8.10 we see that

$$\|f\|_{p,G} = \|f_{\mathrm{G}}\|_p\ ,$$

where the right hand side is the vector p-norm. In the present context we say that $\|f\|_{p,G}$ is a *seminorm*, because it does not satisfy all the

requirements of Definition 9.2.3: $\|f\|_{p,G} = 0$ does not imply that $f = 0$, only that all the $f(x_i) = 0,\ i = 1, \ldots, m$.

The definition of the norm can be generalized by introducing a weight function. This is a function w such that $w(x) > 0$ for all $a < x < b$. (w is not necessarily defined at the endpoints $x = a$ and $x = b$).

$$\|f\|_{p,w} = \Big(\int_a^b w(x)|f(x)|^p\,dx\Big)^{1/p}\ .$$

In the discrete case we introduce weights $w_i > 0,\ i = 1, \ldots, m$,

$$\|f\|_{p,G,w} = \Big(\sum_{i=1}^{m} w_i|f(x_i)|^p\Big)^{1/p}\ .$$

What effect does weighting have on the approximation f^* ?

In the discrete case f^* is determined so that we minimize

$$\Big(\sum_{i=1}^{m} w_i|f_i - f^*(x_i)|^p\Big)^{1/p}\ ,$$

where the f_i are the known approximations to $f(x_i)$. A large value of w_i means that the error at x_i is given special importance. It may, eg, be suitable to let $\sqrt{w_i}$ be inversely proportional to the estimated error in the measured value f_i. Then f^* is determined so that there is particularly good agreement with f at the points that have been measured with best accuracy. In the continuous case a suitable choice of the weight function w can similarly enforce better agreement between f and f^* in some parts of the interval $[a, b]$ than in other parts. Later, we shall see how this is useful in connection with Chebyshev approximation.

In the remainder of this chapter the notation $\|\cdot\|_p$ is used as abbreviation for $\|\cdot\|_{p,w}$ and – when it is clear from the context – also for $\|\cdot\|_{p,G,w}$. Especially, given $w(x)$, the concept *Euclidean norm* is redefined to mean the weighted norm.

Euclidean norm :

$$\|f\|_2 = \begin{cases} \Big(\int_a^b w(x)f(x)^2\,dx\Big)^{1/2} & \text{(continuous case)}\ , \\ \Big(\sum_{i=1}^{m} w_i f(x_i)^2\Big)^{1/2} & \text{(discrete case)}\ . \end{cases} \tag{9.2.1}$$

This norm can be expressed as a scalar product.

Definition 9.2.4. The *scalar product* (f,g) of $f,g \in \mathbb{L}$ is defined by

$$(f,g) = (g,f) = \begin{cases} \displaystyle\int_a^b w(x)f(x)g(x)\,dx & \text{(continuous case)}\ , \\ \displaystyle\sum_{i=1}^{m} w_i f(x_i)g(x_i) & \text{(discrete case)}\ . \end{cases}$$

In both the continuous and the discrete case it is seen that

$$\|f\|_2 = \sqrt{(f,f)}\ .$$

In the discrete case with all $w_i = 1$ the scalar product is recognized as the scalar product (the inner product) of the vectors of function values,

$$(f,g) = \sum_{i=1}^{m} f(x_i)g(x_i) = f_{\mathrm{G}}^T g_{\mathrm{G}}\ . \tag{9.2.2}$$

The scalar product is also used to define orthogonality:

Definition 9.2.5. f and g in $\mathbb{L}$ are said to be *orthogonal* if $(f,g) = 0$.

Definition 9.2.6. A finite or infinite sequence of functions $\varphi_0, \varphi_1, \ldots$ is called an *orthogonal system* if $(\varphi_i, \varphi_j) = 0$ for $i \neq j$ and $(\varphi_i, \varphi_i) \neq 0$ for all i.

If, in addition, $(\varphi_i, \varphi_i) = 1$ for all i, the sequence is called an *orthonormal system*.

Example. The functions $\varphi_0(x) = 1$, $\varphi_1(x) = x$ and $\varphi_2(x) = x^2 - \frac{1}{3}$ are orthogonal on the interval $[-1,1]$ with respect to the weight function $w(x) = 1$:

$$(\varphi_0,\varphi_1) = \textstyle\int_{-1}^{1} x\,dx = 0, \quad (\varphi_0,\varphi_2) = \int_{-1}^{1}(x^2 - \frac{1}{3})\,dx = 0,$$
$$(\varphi_1,\varphi_2) = \textstyle\int_{-1}^{1} x(x^2 - \frac{1}{3})\,dx = 0\ .$$

You should verify that the functions

$$\widetilde{\varphi}_0(x) = \tfrac{1}{\sqrt{2}}, \quad \widetilde{\varphi}_1(x) = \sqrt{\tfrac{3}{2}}x, \quad \widetilde{\varphi}_2(x) = \sqrt{\tfrac{45}{8}}\left(x^2 - \tfrac{1}{3}\right)$$

are orthonormal.

The functions $\psi_0(x) = 1$, $\psi_1(x) = x$ and $\psi_2(x) = x^2 - \frac{5}{9}$ are orthogonal on the grid $x_1 = -1$, $x_2 = -\frac{1}{3}$, $x_3 = \frac{1}{3}$, $x_4 = 1$ with respect to the weights

$w_i = 1$, $i = 1, \ldots, 4$:

$$\psi_{0G} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad \psi_{1G} = \begin{pmatrix} -1 \\ -\frac{1}{3} \\ \frac{1}{3} \\ 1 \end{pmatrix}, \quad \psi_{2G} = \begin{pmatrix} \frac{4}{9} \\ -\frac{4}{9} \\ -\frac{4}{9} \\ \frac{4}{9} \end{pmatrix},$$

and by means of (9.2.2) we find

$$(\psi_0, \psi_1) = \psi_{0G}^T \psi_{1G} = 0 ,$$

and similarly we see that $(\psi_0, \psi_2) = (\psi_1, \psi_2) = 0$. ∎

It is reasonable to require that f^* is expressed in terms of linearly independent functions. Later, we shall show that functions in an orthogonal system are linearly independent.

Definition 9.2.7. The functions $\varphi_0, \varphi_1, \ldots, \varphi_n$ are said to be *linearly independent* if

$$\|\sum_{j=0}^{n} c_j \varphi_j\| = 0 \quad \text{if and only if all} \quad c_j = 0 .$$

In the continuous case we can omit the norm symbol, since $\|g\| = 0$ if and only if $g = 0$.

In the discrete case we have

$$\|\sum_{j=0}^{n} c_j \varphi_j\|_{p,G,w} = 0$$

if and only if

$$\sum_{j=0}^{n} c_j w_i \varphi_j(x_i) = 0 \quad \text{for every} \quad x_i \in G .$$

Dividing by the factor w_i we see that this condition is equivalent to

$$c_0 \begin{pmatrix} \varphi_0(x_1) \\ \vdots \\ \varphi_0(x_m) \end{pmatrix} + c_1 \begin{pmatrix} \varphi_1(x_1) \\ \vdots \\ \varphi_1(x_m) \end{pmatrix} + \cdots + c_n \begin{pmatrix} \varphi_n(x_1) \\ \vdots \\ \varphi_n(x_m) \end{pmatrix} = 0 ,$$

or

$$\sum_{j=0}^{n} c_j \varphi_{jG} = 0 .$$

Thus, we have shown

Theorem 9.2.8. The functions $\varphi_0, \varphi_1, \ldots, \varphi_n$ are linearly independent on the grid $G = \{x_1, x_2, \ldots, x_m\}$ if and only if the vectors $\{\varphi_{j\mathrm{G}}\}_{j=1}^{n}$ are linearly independent.

The theorem implies that there are at most m linearly independent functions on a grid with m points.

Example. The functions $\varphi_j(x) = x^j$, $j = 0, 1, \ldots, n$ are linearly independent on *any* grid with $n+1$ points. If they were not, then there would exist constants $c_0, c_1, \ldots, c_n$, not all equal to zero, such that $\sum_{j=0}^{n} c_j \varphi_{j\mathrm{G}} = 0$. However, this means that

$$\sum_{j=0}^{n} c_j x_i^j = 0 \quad \text{for} \quad i = 1, 2, \ldots, n+1 \ ,$$

and we would have a polynomial of degree at most n with $n+1$ roots. The fundamental theorem of algebra tells us that this is not possible. ∎

9.3. Least Squares Method

We want to find an approximating function f^* so that the Euclidean norm of the error function $f - f^*$ is minimized.

In the *continuous case* f^* shall be determined so that

$$\|f - f^*\|_{2,w} = \Big(\int_a^b w(x)\big(f(x) - f^*(x)\big)^2 \, dx\Big)^{1/2}$$

is minimized. In the *discrete case* f^* shall be determined so that

$$\|f - f^*\|_{2,G,w} = \Big(\sum_{i=1}^{m} w_i\big(f(x_i) - f^*(x_i)\big)^2\Big)^{1/2}$$

is minimized.

The function f^* is called the *least squares fit* to f. Both the discrete and the continuous case are covered by the following important theorem.

Theorem 9.3.1. Assume that $\varphi_0, \varphi_1, \ldots, \varphi_n$ are linearly independent. Then there is a uniquely determined element $f^* \in \mathbb{U}$,

$$f^* = \sum_{j=0}^{n} c_j^* \varphi_j \ ,$$

such that

$$\|f - f^*\|_2 \leq \|f - g\|_2 \quad \text{for all} \quad g = \sum_{j=0}^{n} c_j \varphi_j \ .$$

f^* is characterized by the *normal equations*

$$(f - f^*, \varphi_k) = 0, \quad k = 0, 1, \ldots, n \ .$$

Before we give the proof, we note that the normal equations express that the difference $f - f^*$ is orthogonal to each φ_k. The normal equations can be formulated as

$$(f^*, \varphi_k) = (f, \varphi_k), \quad k = 0, 1, \ldots, n \ . \tag{9.3.1}$$

Thus, we want $f^* = f$, but have to make do with $(f^*, \varphi_k) = (f, \varphi_k)$ for every φ_k. We insert the expression for f^* in (9.3.1), and get

$$\sum_{j=0}^{n} (\varphi_j, \varphi_k) c_j^* = (f, \varphi_k), \quad k = 0, 1, \ldots, n \ . \tag{9.3.2}$$

Thus, the coefficients $\{c_j^*\}_{j=0}^n$ can be determined by solving a linear system of equations.

Next, we illustrate the theorem in Figure 9.2 (in the case $n = 1$, $w_0 = w_1 = 1$).

The linear subspace $\mathbb{U}$ of $\mathbb{L}$ is spanned by φ_0 and φ_1. According to Theorem 9.3.1 the vector $f - f^*$ is orthogonal to φ_0 and φ_1, and thereby to all vectors in $\mathbb{U}$. This means that f^* is the orthogonal projection of f onto $\mathbb{U}$. The theorem simply says that this is the vector in $\mathbb{U}$, that has the smallest Euclidean distance to f. The dashed lines in the figure form a right-angled triangle, in which $f - f^*$ is one of the sides, and $f - g$ is the hypotenuse.

The proof of the theorem is guided by the geometric interpretation. We first need a generalization of the Pythagorean law.

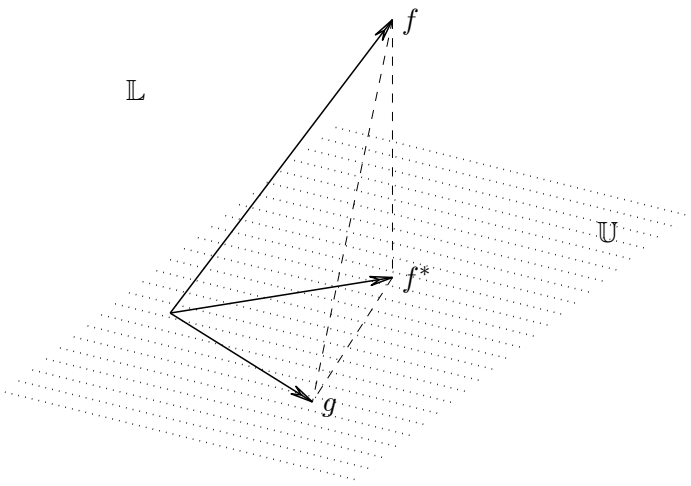


Figure 9.2. *Geometric interpretation of the least squares problem.*

Theorem 9.3.2. Generalized Pythagorean law. If f and h are orthogonal, then

$$\|f+h\|_2^2 = \|f\|_2^2 + \|h\|_2^2 \ .$$

Proof. $\|f+h\|_2^2 = (f+h, f+h)$

$$= (f,f) + (f,h) + (h,f) + (h,h) = \|f\|_2^2 + \|h\|_2^2 \ ,$$

since $(f,h) = (h,f) = 0$ because of the orthogonality. □

Now, we are ready to prove the theorem on page 270.

Proof. (Theorem 9.3.1). We first prove that the linear system of equations (9.3.2) has a unique solution, by showing that the matrix of the system is nonsingular. The proof is by contradiction: If the matrix is singular, then the homogeneous system has a nontrivial solution, ie there are $c_0, c_1, \ldots, c_n$, not all equal to zero, such that

$$\sum_{j=0}^{n} c_j(\varphi_j, \varphi_k) = 0, \quad k = 0, 1, \ldots, n \ .$$

But this means that $\varphi_0, \varphi_1, \ldots, \varphi_n$ are linearly dependent, since

$$\|\sum_{j=0}^{n} c_j\varphi_j\|_2^2 = (\sum_{k=0}^{n} c_k\varphi_k, \sum_{j=0}^{n} c_j\varphi_j)$$
$$= \sum_{k=0}^{n} c_k\big(\sum_{j=0}^{n} c_j(\varphi_j,\varphi_k)\big) = \sum_{k=0}^{n} c_k\cdot 0 = 0\ .$$

This contradicts the assumptions of the theorem.
Next, consider f^* and g in $\mathbb{U}$,

$$f^* = \sum_{j=0}^{n} c_j^*\varphi_j\ , \qquad g = \sum_{j=0}^{n} c_j\varphi_j\ .$$

We shall show that, if any $c_j \neq c_j^*$, then $\|f-g\|_2$ is larger than $\|f-f^*\|_2$, cf Figure 9.2. Since

$$f-g = f-f^*+f^*-g = f-f^* + \sum_{j=0}^{n}(c_j^*-c_j)\varphi_j\ ,$$

and $(\varphi_j, f-f^*)=0,\ j=0,\ldots,n$, we see that $f-f^*$ and f^*-g are orthogonal, and the Pythagorean law gives

$$\|f-g\|_2^2 = \|f-f^*\|_2^2 + \|\sum_{j=0}^{n}(c_j^*-c_j)\varphi_j\|_2^2 \geq \|f-f^*\|_2^2\ .$$

Since $\varphi_0,\varphi_1,\ldots,\varphi_n$ are linearly independent, equality is obtained only when $c_j^*-c_j=0,\ j=0,\ldots,n$. □

Example. Determine the straight line $f^*(x) = c_0 + c_1x$, which is the best approximation, in the least squares sense, to $f(x) = \sin(\frac{\pi}{2}x)$ on the interval $[0,1]$. We have $\varphi_0(x)=1$, $\varphi_1(x)=x$, and

$$(\varphi_0,\varphi_0) = \textstyle\int_0^1 1\,dx = 1\ , \qquad (\varphi_1,\varphi_1) = \int_0^1 x^2\,dx = 1/3\ ,$$
$$(\varphi_0,\varphi_1) = (\varphi_1,\varphi_0) = \textstyle\int_0^1 x\,dx = 1/2$$
$$(f,\varphi_0) = \textstyle\int_0^1 \sin(\frac{\pi}{2}x)\,dx = 2/\pi\ , \qquad (f,\varphi_1) = \int_0^1 x\,\sin(\frac{\pi}{2}x)\,dx = 4/\pi^2\ .$$

The normal equations are

$$\begin{pmatrix} 1 & 1/2 \\ 1/2 & 1/3 \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 2/\pi \\ 4/\pi^2 \end{pmatrix}\ ,$$

and this linear system of equations has the solution

$$c_0 = \frac{8\pi-24}{\pi^2} \simeq 0.1148\ , \qquad c_1 = \frac{48-12\pi}{\pi^2} \simeq 1.0437\ .$$

The approximation $f^*(x) = 0.1148 + 1.0437x$ is shown with a dotted line in Figure 9.1 on page 262. ■

Now, let us study the discrete case in more detail, in order to see the relation to the results in Section 8.14: For the sake of simplicity we let $w_i = 1$. The problem of minimizing

$$\sum_{i=1}^{m} \big(f(x_i) - f^*(x_i)\big)^2$$

is equivalent to determining the vector f_{G}^*, which is closest (in the least squares sense) to the vector f_{G}. We can write f_{G}^* in the form

$$f_{\mathrm{G}}^* = c_0\varphi_{0\mathrm{G}} + c_1\varphi_{1\mathrm{G}} + \cdots + c_n\varphi_{n\mathrm{G}} = Ac \ ,$$

where

$$A = \begin{pmatrix} \varphi_0(x_1) & \varphi_1(x_1) & \cdots & \varphi_n(x_1) \\ \varphi_0(x_2) & \varphi_1(x_2) & \cdots & \varphi_n(x_2) \\ \vdots & \vdots & & \vdots \\ \varphi_0(x_m) & \varphi_1(x_m) & \cdots & \varphi_n(x_m) \end{pmatrix}, \qquad c = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} \ .$$

Thus, in the notation of Section 8.14, the coefficients should be determined as the least squares solution of the overdetermined system[1)]

$$Ac \simeq f_{\mathrm{G}} \ .$$

The associated normal equations, as defined in Theorem 8.14.2, are

$$A^T A\, c^* = A^T f_{\mathrm{G}} \ .$$

We shall show that this linear system of equations is identical to (9.3.2): In the kth equation ($k = 0, 1, \ldots, n$) the coefficient of c_j is

$$(A^T A)_{k+1,j+1} = \varphi_{k\mathrm{G}}^T \varphi_{j\mathrm{G}} = (\varphi_k, \varphi_j) \ ,$$

and the right hand side is

$$(A^T f_{\mathrm{G}})_{k+1} = \varphi_{k\mathrm{G}}^T f_{\mathrm{G}} = (\varphi_k, f) \ .$$

Example. The MATLAB function `polyfit` can be used to determine a least squares polynomial fit in the discrete case. Let $\{(x_i, y_i)\}_{i=1}^m$ be the given data points, represented by the MATLAB vectors `x` and `y`. Then

```
c = polyfit(x,y,n)
```

returns an $(n+1)$-vector `c` with the coefficients of the least squares polynomial fit of degree n. The polynomial is represented in the form

$$c_1 x^n + c_2 x^{n-1} + \cdots + c_n x + c_{n+1} \ .$$

[1)] A is an $m \times (n+1)$ matrix and we assume that $m > n+1$.

The coefficients are computed by solving the overdetermined system $Ac \simeq y$ via QR factorization of A, see Section 8.15. ∎

Finally, we shall give an alternative derivation of the normal equations. Given f and $\{\varphi_j\}_{j=0}^n$. We want to find the coefficients $c_0^*, c_1^*, \ldots, c_n^*$ such that the function

$$\psi(c_0, c_1, \ldots, c_n) = \sum_{i=1}^m w_i \left(f(x_i) - f^*(x_i)\right)^2$$

is minimized. ψ is a continuously differentiable function of $c_0, c_1, \ldots, c_n$ and at a minimal point of ψ we must have

$$\frac{\partial \psi}{\partial c_k} = 0, \quad k = 0, 1, \ldots, n \ .$$

We shall show that this is equivalent to the normal equations.

For every k we have

$$\frac{\partial \psi}{\partial c_k} = -2 \sum_{i=1}^m w_i \left(f(x_i) - f^*(x_i)\right) \frac{\partial f^*(x_i)}{\partial c_k} \ = \ 0 \ ,$$

or

$$\sum_{i=1}^m w_i \left(f(x_i) - f^*(x_i)\right) \varphi_k(x_i) \ = \ 0 \ ,$$

or

$$c_0 \sum_{i=1}^m w_i \varphi_k(x_i)\varphi_0(x_i) + \cdots + c_n \sum_{i=1}^m w_i \varphi_k(x_i)\varphi_n(x_i) = \sum_{i=1}^m w_i \varphi_k(x_i) f(x_i) \ .$$

We rewrite this in terms of scalar products,

$$c_0(\varphi_k, \varphi_0) + \cdots + c_n(\varphi_k, \varphi_n) = (\varphi_k, f), \quad k = 0, 1, \ldots, n \ .$$

This is recognized as the normal equations (9.3.2) on page 270.

9.4. Orthogonal Functions

According to Theorem 9.3.1 the least squares approximation of a given function (or a given vector of function values) can be determined by solving the normal equations. In practice this system of equations may be ill-conditioned and numerical difficulties occur.

Example. Assume that a function f shall be approximated on the interval $[0,1]$ by a polynomial f^*. If we represent the polynomial as a linear combination of $\varphi_j(x) = x^j$, $j = 0, 1, \ldots, n$, then we get

$$(\varphi_i, \varphi_j) = \int_0^1 x^{i+j}\, dx = \frac{1}{i+j+1} \ ,$$

and the coefficient matrix in the normal equations is a so-called *Hilbert matrix*. For $n = 4$ it has the form

$$H_4 \ = \ \begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 \\ 1/2 & 1/3 & 1/4 & 1/5 \\ 1/3 & 1/4 & 1/5 & 1/6 \\ 1/4 & 1/5 & 1/6 & 1/7 \end{pmatrix} \ .$$

Hilbert matrices are known to be ill-conditioned; the Hilbert matrix of order n has condition number $\kappa_2(H_n) \simeq 10^{1.5(n-1)}$. Therefore, the simple polynomials $\varphi_j(x) = x^j$ are not recommended as basis functions. ∎

Instead, we can choose an orthogonal basis of the subspace $\mathbb{U}$. We shall show, that then this kind of numerical difficulties is avoided. If $(\varphi_j, \varphi_k) = 0$ for $j \neq k$, then the normal equations (9.3.2) simplify to

$$(\varphi_k, \varphi_k) c_k^* = (f, \varphi_k), \quad k = 0, 1, \ldots, n \ ,$$

and the coefficients for the best approximation are determined simply as

$$c_k^* = \frac{(f, \varphi_k)}{(\varphi_k, \varphi_k)} \ , \quad k = 0, 1, \ldots, n \ .$$

These coefficients are called *orthogonal coefficients* or *Fourier coefficients*.

Thus, the use of orthogonal basis functions gives a diagonal matrix in the normal equations. There is a further advantage of orthogonal basis functions: Assume that we have determined f_n^*, the best approximation of f in the subspace spanned by $\varphi_0, \ldots, \varphi_n$. In order to determine f_{n+1}^*, the best approximation of f in the subspace spanned by $\varphi_0, \ldots, \varphi_n, \varphi_{n+1}$, we only need to know φ_{n+1} and compute

$$c_{n+1}^* = \frac{(f, \varphi_{n+1})}{(\varphi_{n+1}, \varphi_{n+1})} \ .$$

Then $f_{n+1}^* = f_n^* + c_{n+1}^* \varphi_{n+1}$.

9.5. Orthogonal Polynomials

We now show how one can construct polynomials that are orthogonal with respect to a given scalar product, and we start with an example.

Example. Construct polynomials P_0, P_1, P_2 with degree $P_k = k$ and orthogonal with respect to the scalar product

$$(f, g) = \int_{-1}^{1} f(x)g(x)\,dx \ .$$

We let

$$P_0(x) = 1, \quad P_1(x) = x + a_{11}, \quad P_2(x) = x^2 + a_{21}x + a_{22} \ ,$$

and determine the constants so that the orthogonality conditions are satisfied. The condition $(P_1, P_0) = 0$ gives

$$0 \ = \ \textstyle\int_{-1}^{1}(x + a_{11})\,dx = 2a_{11} \ .$$

Hence, $a_{11} = 0$. Next, $(P_2, P_0) = 0$ and $(P_2, P_1) = 0$ lead to

$$0 \ = \ \textstyle\int_{-1}^{1}(x^2 + a_{21}x + a_{22})\,dx = \frac{2}{3} + 2a_{22} \ ,$$
$$0 \ = \ \textstyle\int_{-1}^{1}(x^2 + a_{21}x + a_{22})x\,dx = \frac{2}{3}a_{21} \ .$$

The solution is $a_{21} = 0$, $a_{22} = -\frac{1}{3}$, and the desired polynomials are the ones discussed in the example on page 267,

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = x^2 - \tfrac{1}{3} \ .$$

Note that P_1 has one zero, $x^* = 0$, and P_2 has two distinct zeros, $x^* = \pm\sqrt{\frac{1}{3}} \simeq \pm 0.5774$. All of these zeros lie in the open interval $]-1, 1[$. ∎

In general, let $P_0, P_1, P_2, \ldots$ with degree $P_k = k$ be orthogonal on $[a, b]$ with respect to some weight function $w(x)$. It can be shown that P_k has k distinct zeros in $]a, b[$.

The construction of orthogonal polynomials can often be simplified by the following observation

Lemma 9.5.1. Let $\{P_i\}_{i=0}^{n}$ be polynomials with degree $P_i = i$. The polynomials form an orthogonal system if and only if

$$(P_k, x^j) = 0, \quad j = 0, 1, \ldots, k-1$$

for $k = 1, \ldots, n$.

Proof. Assume that the polynomials $P_0, P_1, \ldots, P_n$ are an orthogonal system. Then they are linearly independent (Exercise E3). Therefore, x^j can be expressed as a linear combination of them:

$$x^j = \sum_{r=0}^{j} \alpha_{rj} P_r(x) \ .$$

Hence, $\quad (P_k, x^j) = \sum_{r=0}^{j} \alpha_{rj}(P_k, P_r) = 0 \quad \text{for} \ \ j < k \ .$

The converse follows similarly, since an arbitrary polynomial P_i can be written as a linear combination of $1, x, \ldots, x^i$. □

An alternative way of constructing orthogonal polynomials is to use the following theorem.

Theorem 9.5.2. For any scalar product there exists an essentially unique sequence of orthogonal polynomials $P_0, P_1, P_2, \ldots$ with degree $P_i = i$. The coefficients of the highest power can be chosen as arbitrary nonzero numbers. When these coefficients are fixed, the orthogonal system is uniquely determined. The polynomials satisfy the three-term recurrence

$$\begin{aligned} P_0(x) &= A_0 \\ P_1(x) &= (\alpha_0 x - \beta_0) P_0(x) \\ P_{k+1}(x) &= (\alpha_k x - \beta_k) P_k(x) - \gamma_k P_{k-1}(x), \quad k = 1, 2, \ldots \ , \end{aligned}$$

where

$$\begin{aligned} \beta_k &= \frac{\alpha_k (x P_k, P_k)}{(P_k, P_k)} \ , \qquad k = 0, 1, 2, \ldots \ , \\ \gamma_k &= \frac{\alpha_k (P_k, P_k)}{\alpha_{k-1} (P_{k-1}, P_{k-1})} \ , \quad k = 1, 2, \ldots \ . \end{aligned}$$

In the discrete case, with the grid $x_1, x_2, \ldots, x_m$, the last polynomial in the sequence is P_{m-1}.

Proof. In the proof the polynomials are constructed by the same principle as in the previous example. Put

$$\begin{aligned} P_0(x) &= A_0 \\ P_1(x) &= A_1 x + b = (\alpha_0 x - \beta_0) A_0 \ , \end{aligned}$$

with $\alpha_0 = A_1 / A_0$. The condition $(P_1, P_0) = 0$ gives

$$\big((\alpha_0 x - \beta_0) A_0, A_0\big) = \alpha_0 (x P_0, P_0) - \beta_0 (P_0, P_0) = 0 \ ,$$

which is satisfied by the β_0 given in the theorem.

Now, assume that we have constructed orthogonal polynomials $P_0, P_1,$

$\dots, P_k$, with $P_j(x) = A_j x^j + \cdots$. Then $P_{k+1}(x) = A_{k+1}x^{k+1} + \cdots$ shall be determined so that

$$(P_{k+1}, P_j) = 0, \quad j = 0, 1, \dots, k \ . \tag{9.5.1}$$

With $A_{k+1} = \alpha_k A_k$ the difference $P_{k+1}(x) - \alpha_k x P_k(x)$ is a polynomial of degree at most k, so that we can write

$$P_{k+1}(x) = \alpha_k x P_k(x) - \sum_{i=0}^{k} a_{ki} P_i(x) \ . \tag{9.5.2}$$

Because of the orthogonality of the polynomials $\{P_i\}_{i=0}^k$ we get

$$(P_{k+1}, P_j) = \alpha_k (xP_k, P_j) - a_{kj}(P_j, P_j), \quad j = 0, 1, \dots, k \ . \tag{9.5.3}$$

Since $(xP_k, P_j) = (P_k, xP_j)$ and xP_j is a polynomial of degree $j+1$, it follows by repeated use of Lemma 9.5.1 that $(xP_k, P_j) = 0$ for $j \le k-2$ and hence,

$$a_{kj} = 0, \quad j = 0, 1, \dots, k-2 \ .$$

Therefore, if we put $a_{kk} = \beta_k$ as given in the theorem, then the condition (9.5.1) is also satisfied for $j = k$. For $j = k-1$ condition (9.5.1) is satisfied for

$$a_{k,k-1} = \frac{\alpha_k (xP_k, P_{k-1})}{(P_{k-1}, P_{k-1})} \ .$$

This can be simplified. To do that, we note that $(xP_k, P_{k-1}) = (P_k, xP_{k-1})$, and according to the assumption P_k is given by the recurrence. Therefore

$$\alpha_{k-1} x P_{k-1}(x) = P_k(x) + \beta_{k-1} P_{k-1}(x) + \gamma_{k-1} P_{k-2}(x) \ ,$$

and

$$(xP_k, P_{k-1}) = \frac{(P_k, P_k) + \beta_{k-1}(P_k, P_{k-1}) + \gamma_{k-1}(P_k, P_{k-2})}{\alpha_{k-1}} = \frac{(P_k, P_k)}{\alpha_{k-1}} \ ,$$

so that

$$(P_{k+1}, P_{k-1}) = \frac{\alpha_k}{\alpha_{k-1}} (P_k, P_k) - a_{k,k-1}(P_{k-1}, P_{k-1}) \ .$$

This is zero when we choose $a_{k,k-1} = \gamma_k$ as given in the theorem.

It only remains to show that the sequence ends with P_{m-1} in the discrete case: The degree m polynomial

$$Q_m(x) = (x - x_1)(x - x_2) \cdots (x - x_m)$$

is orthogonal to $P_0, \ldots, P_{m-1}$, since for arbitrary $j \le m-1$ we have

$$(Q_m, P_j) = \sum_{i=1}^{m} w_i Q_m(x_i) P_j(x_i) = 0 \ .$$

The polynomial P_m is essentially unique, and therefore it must be a multiple of Q_m, say $P_m = cQ_m$, and

$$\|P_m\|_2 = |c| \cdot \|Q_m\|_2 = 0 \ .$$

This means that P_m cannot be a member of the orthogonal system. □

Example. Construct polynomials P_0, P_1, P_2 with leading coefficient 1 and orthogonal with respect to the scalar product

$$(f, g) = \sum_{i=1}^{3} f(x_i) g(x_i), \quad x_1 = -\tfrac{\sqrt{3}}{2},\ x_2 = 0,\ x_3 = \tfrac{\sqrt{3}}{2} \ .$$

(The reason for choosing these points will be apparent in Section 9.7).

We use the formulas in Theorem 9.5.2 with $A_0 = 1$ and $\alpha_k = 1$. This gives $P_0(x) = 1$ and $\quad (P_0, P_0) = 3, \quad (xP_0, P_0) = x_1 + x_2 + x_3 = 0$.

Therefore, $\beta_0 = 0$, and $\quad P_1(x) = xP_0(x) = x$.

For the next step we compute

$$(P_1, P_1) = x_1^2 + x_2^2 + x_3^2 = \tfrac{3}{2}, \quad (xP_1, P_1) = x_1^3 + x_2^3 + x_3^3 = 0 \ ,$$

$\beta_1 = 0,\ \gamma_1 = \frac{3/2}{3} = \frac{1}{2}$, and $P_2(x) = x^2 - \frac{1}{2}$. ■

When the coefficients $\{c_j\}$ and the parameters $\{\beta_j\}$ and $\{\gamma_j\}$ are known, then we can use the recurrence in Theorem 9.5.2 to evaluate the approximating polynomial

$$f^*(x) = c_0 P_0(x) + c_1 P_1(x) + \cdots + c_n P_n(x)$$

for any x. This is illustrated in the following example.

Example. Given the points $(x_1, y_1), \ldots, (x_m, y_m)$ and associated weights $w_1, \ldots, w_m$. The following MATLAB function computes the least squares polynomial fit of degree n to these data points. The fit has the form

$$f^*(x) = c_0 P_0(x) + c_1 P_1(x) + \cdots + c_n P_n(x)$$

where the $\{P_j\}$ have leading coefficient 1 and are orthogonal with respect to the scalar product

$$(f, g) = \sum_{i=1}^{m} w_i f(x_i) g(x_i) \ .$$

The function returns the coefficients $c_0, c_1, \ldots, c_n$ in the $(n+1)$-array `c`, and

the recurrence parameters $\{\beta_k\}$ and $\{\gamma_k\}$ are returned in arrays `b` and `g`, respectively.

```
function  [b,g,c] = orthpolfit(x,y,w,n)
% Weighted least squares fit with degree n polynomial
x = x(:);  y = y(:);  w = w(:);                 % column vectors
m = length(x);
b = zeros(1,max(1,n));  g = b;          % for beta_k and gamma_k
P = [zeros(m,1) ones(m,1)];    % values of two most recent pol.s
s = [0 sum(w)];                       % two most recent (P_j,P_j)
c = [sum(w.*y)/s(2)  zeros(1,n)];         % for c_j, j = 0,...,n
for  k = 1 : n
  b(k) = sum(w .* x .* P(:,2).^2)/s(2);             % beta_(k-1)
  if  k == 1
    g(k) = 0;
  else
    g(k) = s(2)/s(1);                                % gamma_(k-1)
  end
  P = [P(:,2) (x - b(k)).*P(:,2) - g(k)*P(:,1)]; % P_k in P(:,2)
  s = [s(2)  sum(w .* P(:,2).^2)];           % (P_k,P_k) in s(2)
  c(k+1) = sum(w .* P(:,2) .* y)/s(2);
end
```

With the elastic spring data from the example on page 244 and weights $w_i = 1$ we get

```
>> x = 0.8*(1:5);   y = [7.97 10.2 14.2 16 21.2];
>> [b,g,c] = orthpolfit(x,y,ones(1,5),1)
b = 2.4000
g = 0
c = 13.9140    4.0325
```

This shows that the least squares fit with a first degree polynomial is

$$f^*(x) = 13.9140 + 4.0325(x - 2.40) = 4.2360 + 4.0325x \ .$$

The result naturally agrees with the result found in the example on page 248.

A polynomial found by means of `orthpolfit` can be evaluated for arbitrary x by means of the following MATLAB function. The function takes x as a vector, so that we simultaneously compute f^* at several sites. This is useful, eg, if we want to plot f^*.

```
function  f = orthpolval(b,g,c,x)
% Value of orthogonal polynomial expansion
n = length(c)-1;
m = length(x);            % number of simultaneous arguments
P = ones(m,2);   % for values of two consecutive polynomials
f = c(1) * P(:,2);                              % initialize
for  j = 1 : n
```

```
  P = [P(:,2)  (x(:) - b(j)).*P(:,2) - g(j)*P(:,1)];
  f = f + c(j+1)*P(:,2);
end
```

As an example consider the third degree polynomial least squares approximation to $f(x) = \sin(\frac{\pi}{2}x)$ on the grid $x_i = \cos\frac{i\pi}{10},\ i = 0, 1, \ldots, 10$. The following MATLAB code plots the error $\sin(\frac{\pi}{2}x) - f^*(x)$.

```
>> x = cos((0:10)*pi/10);  y = sin(0.5*pi*x);
>> [b g c] = orthpolfit(x,y,ones(11,1),3);
>> t = linspace(0,1,201)';
>> plot(t, sin(0.5*pi*t)-orthpolval(b,g,c,t))
```

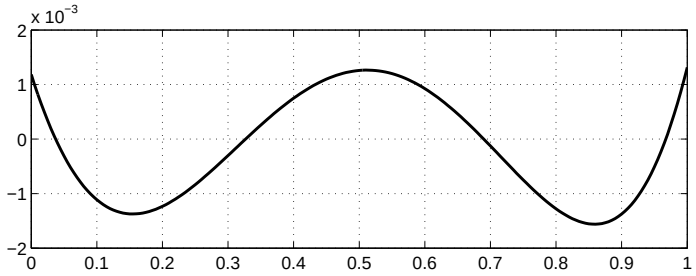


Figure 9.3. *Error* $\sin(\frac{\pi}{2}x) - f^*(x)$*, where* f^* *is a degree 3 polynomial, found by least squares.*

■

In the next two sections we look at special types of orthogonal polynomials that are frequently used in applications. Both of them are defined on $[-1, 1]$. An arbitrary, finite interval $a \le x \le b$ can be transformed to $-1 \le t \le 1$ by the simple transformation

$$t = \frac{2x - (b + a)}{b - a}\ . \tag{9.5.4}$$

9.6. Legendre Polynomials

The Legendre polynomials P_n are defined for $n \ge 0$ and $-1 \le x \le 1$ by

$$P_n(x) = \begin{cases} 1\ , & n = 0\ , \\ \dfrac{1}{2^n \cdot n!}\ \dfrac{d^n}{dx^n}\ (x^2 - 1)^n\ , & n \ge 1\ . \end{cases}$$

Since $(x^2 - 1)^n$ is a polynomial of degree $2n$, it is seen that P_n is a polynomial of degree n. The first five Legendre polynomials are

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = \frac{1}{2}(3x^2 - 1) \ ,$$
$$P_3(x) = \frac{1}{2}(5x^2 - 3x), \quad P_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3) \ .$$

They are shown in Figure 9.4.

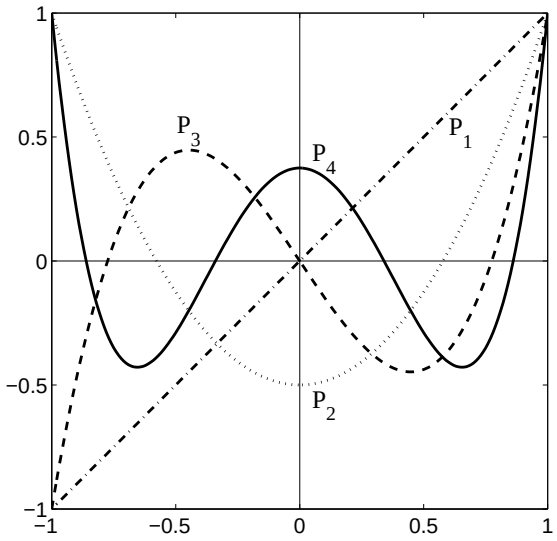


Figure 9.4. *Legendre polynomials P_j.*

In general it holds that $|P_n(x)| \leq 1$ for $x \in [-1, 1]$. Further, P_n has n distinct zeros in $]-1, 1[$, and between these zeros, P_n has maxima and minima, whose magnitude decrease towards the midpoint of the interval. It can be shown that the Legendre polynomials are orthogonal on the interval $[-1, 1]$ with respect to the weight function $w(x) = 1$, ie

$$\int_{-1}^{1} P_i(x)P_j(x)\,dx = 0 \quad \text{for} \quad i \neq j \ .$$

Further,

$$\int_{-1}^{1} P_n^2(x)\,dx = \frac{2}{2n+1} \ .$$

If we compare the Legendre polynomials with the orthogonal polynomials constructed in the example on page 276, we see that they are identical except for a factor. The Legendre polynomials are normalized so that $P_n(1) = 1$ for all n. They can be generated by the three-term recurrence

$$P_0(x) = 1 \ , \quad P_1(x) = x \ ,$$
$$P_{n+1}(x) = \frac{2n+1}{n+1}\, xP_n(x) - \frac{n}{n+1}\, P_{n-1}(x), \quad n = 1, 2, \dots \ .$$

It follows that

$$P_n(-x) = (-1)^n P_n(x) \ .$$

Example. We illustrate the use of orthogonal polynomials by determining the second degree polynomial f^* that minimizes

$$\int_{-1}^{1} \left(x^3 - f^*(x)\right)^2 \, dx \ .$$

In other words, we shall approximate x^3 by a lower degree polynomial, using the least squares method. We put

$$f^*(x) = c_0 P_0(x) + c_1 P_1(x) + c_2 P_2(x) \ ,$$

where $\{P_j\}$ are Legendre polynomials. We shall determine the Fourier coefficients

$$c_j = \frac{(x^3, P_j)}{(P_j, P_j)}, \quad j = 0, 1, 2 \ ,$$

and get

$$(x^3, P_0) = \textstyle\int_{-1}^{1} x^3 \, dx = 0, \quad c_0 = 0 \ ,$$
$$(x^3, P_1) = \tfrac{2}{5}, \quad (P_1, P_1) = \tfrac{2}{3}, \quad c_1 = \tfrac{3}{5} \ ,$$
$$(x^3, P_2) = 0, \quad c_2 = 0 \ .$$

Thus, $f^*(x) = \frac{3}{5}x$. Note that $x^3 - f^*(x) = \frac{2}{5}P_3(x)$. ∎

The result could have been obtained directly by use of the following theorem.

Theorem 9.6.1. Given the scalar product

$$(f, g) = \int_a^b w(x) f(x) g(x) \, dx$$

and the corresponding orthogonal polynomials $p_0, p_1, \dots, p_n$, all with leading coefficient equal to 1. Let q_n be an arbitrary degree n polynomial with leading coefficient equal to 1. Then

$$\int_a^b w(x) q_n^2(x) \, dx$$

is minimized when $q_n = p_n$.

Proof. Express q_n as a linear combination of the orthogonal polynomials $p_0, p_1, \ldots, p_n$:

$$q_n = p_n + \sum_{k=0}^{n-1} a_k p_k \ .$$

Then

$$\int_a^b w(x) q_n^2(x)\, dx = (q_n, q_n) = (p_n, p_n) + \sum_{k=0}^{n-1} a_k^2 (p_k, p_k) \ .$$

This is minimized when $a_0 = a_1 = \cdots = a_{n-1} = 0$. □

9.7. Chebyshev Polynomials

The Chebyshev polynomials T_n are defined for $n \geq 0$ and $-1 \leq x \leq 1$ by

$$T_n(x) = \cos(n \arccos x) \ . \tag{9.7.1}$$

In order to derive a three-term recursion for Chebyshev polynomials, we introduce $u = \arccos x$ and make use of a well-known formula for the sum of two cosines:

$$\begin{aligned} T_{n+1}(x) + T_{n-1}(x) &= \cos\bigl((n+1)u\bigr) + \cos\bigl((n-1)u\bigr) \\ &= 2\cos u \cdot \cos nu = 2x \cdot T_n(x) \ . \end{aligned}$$

This can be used for $n \geq 1$, and we see that the Chebyshev polynomials can be generated by the recurrence

$$\begin{aligned} &T_0(x) = 1 \ , \quad T_1(x) = x \ , \\ &T_{n+1}(x) = 2x\, T_n(x) - T_{n-1}(x), \quad n = 1, 2, \ldots \ . \end{aligned}$$

This formula also shows that the functions defined by (9.7.1) are indeed polynomials. The first five of them are

$$\begin{aligned} &T_0(x) = 1, \quad T_1(x) = x, \quad T_2(x) = 2x^2 - 1 \ , \\ &T_3(x) = 4x^3 - 3x, \quad T_4(x) = 8x^4 - 8x^2 + 1 \ . \end{aligned}$$

They are shown in Figure 9.5.

From the recurrence formula it follows that $T_n(x) = 2^{n-1}x^n + \ldots$ for $n \geq 1$, and that

$$T_n(-x) = (-1)^n T_n(x) \ .$$

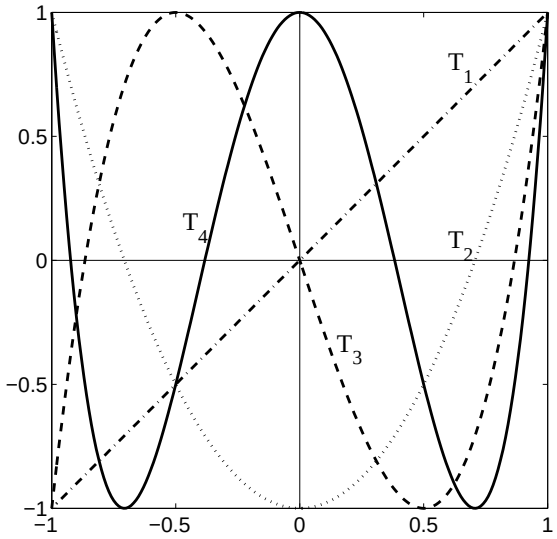


Figure 9.5. *Chebyshev polynomials T_j.*

In Section 9.9 we shall make use of some properties of Chebyshev polynomials. The zeros $x_1, \ldots, x_n$ of T_n (sometimes called *Chebyshev nodes*) can be found from (9.7.1):

$$T_n(x_i) = \cos(n \arccos x_i) = 0 \ .$$

This is satisfied by $n \arccos x_i = (2i-1)\frac{\pi}{2}$, or

$$x_i = \cos\big(\frac{2i-1}{2n}\,\pi\big), \quad i = 1, 2, \ldots, n \ . \tag{9.7.2}$$

As n grows, the zeros concentrate more and more close to the endpoints -1 and $+1$.

The Chebyshev polynomial T_n assumes the extreme values -1 and $+1$ at the $n+1$ points $\widetilde{x}_0, \widetilde{x}_1, \ldots, \widetilde{x}_n$ given by

$$T_n(\widetilde{x}_k) = \cos(n \arccos \widetilde{x}_k) = (-1)^k \ .$$

This leads to

$$\widetilde{x}_k = \cos\big(\frac{k}{n}\,\pi\big), \quad k = 0, 1, \ldots, n \ . \tag{9.7.3}$$

Note that the two endpoints $\widetilde{x}_0 = 1$ and $\widetilde{x}_n = -1$ are included in this point set.

We shall now show that Chebyshev polynomials are orthogonal on $[-1,1]$ with respect to the weight function $w(x) = 1/\sqrt{1-x^2}$:

$$\begin{aligned}(T_j, T_k) &= \int_{-1}^{1} \frac{T_j(x)T_k(x)}{\sqrt{1-x^2}}\,dx = \int_0^{\pi} T_j(\cos u)T_k(\cos u)\,du \\ &= \int_0^{\pi} \cos ju\,\cos ku\,du \\ &= \tfrac{1}{2}\int_0^{\pi} \big(\cos(j{+}k)u + \cos(j{-}k)u\big)du = \begin{cases} 0\,, & j\neq k\,, \\ \frac{1}{2}\pi\,, & j=k\neq 0\,, \\ \pi\,, & j=k=0\,. \end{cases}\end{aligned}$$

It can also be shown that $T_0, T_1, \ldots, T_m$ are orthogonal with respect to the scalar product

$$(f,g) = \sum_{i=1}^{m+1} f(x_i)g(x_i)\,,$$

where the x_i are the zeros of T_{m+1}. Note that $w_i = 1$ in the discrete case.

Example. The zeros of T_3 are

$$x_1 = \cos(\tfrac{1}{6}\pi) = \tfrac{\sqrt{3}}{2}\,, \quad x_2 = \cos(\tfrac{3}{6}\pi) = 0\,, \quad x_3 = \cos(\tfrac{5}{6}\pi) = -\tfrac{\sqrt{3}}{2}\,.$$

These were the points used to define the scalar product in the first example on page 279. The polynomials found there have leading coefficient 1, and they are $P_0(x) = T_0(x)$, $P_1(x) = T_1(x)$ and $P_2(x) = \frac{1}{2}T_2(x)$. ∎

The following theorem expresses the important "*minimax property*" of Chebyshev polynomials.

Theorem 9.7.1. Among all degree n polynomials with 1 as the leading coefficient, the polynomial $2^{-(n-1)}T_n$ has the smallest maximum norm on the interval $[-1,1]$.

Proof. By contradiction. Assume that there exists a polynomial $p_n \neq 2^{-(n-1)}T_n$, such that

$$p_n(x) = x^n + \sum_{r=0}^{n-1} a_r x^r$$

and $\qquad |p_n(x)| < 2^{-(n-1)} \quad$ for all $\; x \in [-1,1]$.

T_n has extrema in the points $\widetilde{x}_i$ given by (9.7.3). The values of p_n in these points satisfy

$$
\begin{aligned}
p_n(\widetilde{x}_0) &< 2^{-(n-1)}T_n(\widetilde{x}_0) = 2^{-(n-1)} , \\
p_n(\widetilde{x}_1) &> 2^{-(n-1)}T_n(\widetilde{x}_1) = -2^{-(n-1)} , \\
p_n(\widetilde{x}_2) &< 2^{-(n-1)}T_n(\widetilde{x}_2) = 2^{-(n-1)} , \\
&\text{etc}
\end{aligned}
$$

This can be reformulated to

$$
\begin{aligned}
p_n(\widetilde{x}_0) - 2^{-(n-1)}T_n(\widetilde{x}_0) &< 0 , \\
p_n(\widetilde{x}_1) - 2^{-(n-1)}T_n(\widetilde{x}_1) &> 0 , \\
p_n(\widetilde{x}_2) - 2^{-(n-1)}T_n(\widetilde{x}_2) &< 0 , \\
&\text{etc}
\end{aligned}
$$

The polynomial $p_n - 2^{-(n-1)}T_n$ has degree $\leq n-1$ and a zero in each of the intervals $]\widetilde{x}_i, \widetilde{x}_{i-1}[$, $i = 1, 2, \ldots, n$. Hence, the polynomial has (at least) n zeros. This is not possible, unless the polynomial is identically equal to zero. □

The theorem is illustrated in Figure 9.6 in the case $n = 4$. The solid line shows $\frac{1}{8}T_4$, and the dashed line shows another degree 4 polynomial with leading coefficient 1. The scaled Chebyshev polynomial stays within the band ± 0.125, while the other polynomial has values outside this band.

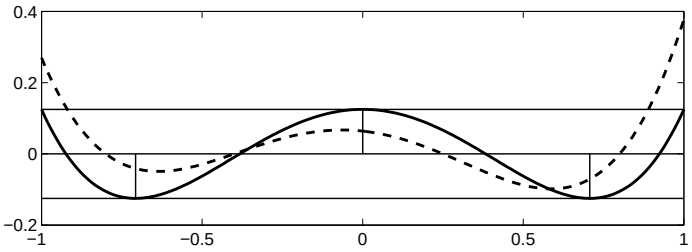


Figure 9.6. *Minimax property of Chebyshev polynomial.*

The following example shows how the minimax property of Chebyshev polynomials can be used when we want to minimize the maximum norm of the error function.

Example. Determine the polynomial f^* of degree ≤ 3, which minimizes

$$\max_{-1 \leq x \leq 1} |x^4 - f^*(x)| .$$

We use the fact that $\frac{1}{8}T_4(x) = x^4 - x^2 + \frac{1}{8}$ is the degree 4 polynomial with 1

as leading coefficient, that has the smallest maximum norm on $[-1,1]$, and choose f^* so that $x^4 - f^*(x) = \frac{1}{8}T_4(x)$, ie

$$f^*(x) = x^2 - \tfrac{1}{8} \ .$$

Note that the error function oscillates between its extrema $\pm\frac{1}{8}$ in five points. In Theorem 9.9.2 we shall see that this behaviour is characteristic of the error function corresponding to the best – in the maximum norm sense – polynomial approximation to a continuous function. ∎

9.8. Discrete Cosine Transform (DCT)

In this section we shall give a short introduction to the *discrete cosine transform* (*DCT*), which is widely used in signal processing and image analysis.

We first consider the one-dimensional version of DCT. A discrete *signal* of length m is a vector in $\mathbb{R}^m$. We can think of the signal as a vector f_{G} of values of some function f, corresponding to equidistant arguments, and we choose to let these arguments be the grid $G = \{x_l\}_{l=1}^m$ defined by

$$x_l = \frac{(2l-1)\pi}{2m} \ , \quad l = 1, 2 \ldots, m \ . \tag{9.8.1}$$

The grid corresponds to values of the independent variable x in the interval $[0, \pi]$. In order to find an approximation of f in this interval we choose the basis functions

$$\varphi_k(x) = \alpha_k \cos kx, \quad \alpha_k = \begin{cases} \sqrt{1/m} \ , & k = 0 \\ \sqrt{2/m} \ , & k > 0 \ . \end{cases} \tag{9.8.2}$$

Theorem 9.8.1. The functions $\varphi_0, \varphi_1, \ldots, \varphi_{m-1}$, defined by (9.8.2) form an orthonormal system with respect to the scalar product

$$(u, v) = \sum_{l=1}^{m} u(x_l) \cdot v(x_l) \ ,$$

where the arguments x_l are given by (9.8.1).

Proof. Let i denote the imaginary unit and use *Euler's formula*

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

This is equivalent to

$$\cos x = \tfrac{1}{2} \left(e^{ix} + e^{-ix} \right) \ ,$$

and we see that

$$\cos k x_l = \tfrac{1}{2} (e^{ik(2l-1)\frac{\pi}{2m}} + e^{-ik(2l-1)\frac{\pi}{2m}}) = \tfrac{1}{2} (\zeta_k^{2l-1} + \overline{\zeta}_k^{2l-1}) \ ,$$

where $\zeta_k = e^{ik\frac{\pi}{2m}}$ and $\overline{\zeta}_k$ is the complex conjugate, $\overline{\zeta}_k = \zeta_k^{-1} = e^{-ik\frac{\pi}{2m}}$. We insert this expression in the scalar product:

$$\begin{aligned} (\varphi_j, \varphi_k) &= \tfrac{1}{4} \alpha_j \alpha_k \sum_{l=1}^{m} (\zeta_j^{2l-1} + \overline{\zeta}_j^{2l-1})(\zeta_k^{2l-1} + \overline{\zeta}_k^{2l-1}) \\ &= \tfrac{1}{4} \alpha_j \alpha_k \sum_{l=1}^{m} (\beta_{jk}^{2l-1} + \overline{\beta}_{jk}^{2l-1} + \gamma_{jk}^{2l-1} + \overline{\gamma}_{jk}^{2l-1}) \ , \end{aligned} \tag{9.8.3}$$

where $\beta_{jk} = \zeta_j \zeta_k = e^{i(j+k)\frac{\pi}{2m}}$, $\gamma_{jk} = \zeta_j \overline{\zeta}_k = e^{i(j-k)\frac{\pi}{2m}}$.

We first show orthogonality, ie $(\varphi_j, \varphi_k) = 0$ for $j \neq k$:
Since $0 < j+k < 2m$, we see that $\beta_{jk} \neq 1$, and

$$\sum_{l=1}^{m} \beta_{jk}^{2l-1} = \beta_{jk} \left(1 + \beta_{jk}^2 + \cdots \beta_{jk}^{2(m-1)} \right) = \beta_{jk} \, \frac{1 - \beta_{jk}^{2m}}{1 - \beta_{jk}^2} \ .$$

In the reformulation we used the formula for the sum of a geometric series. Similar expressions are found for the other contributions to the sum in (9.8.3), and using the fact that $\overline{\beta}_{jk} = 1/\beta_{jk}$ we get

$$\begin{aligned} \sum_{l=1}^{m} (\beta_{jk}^{2l-1} + \overline{\beta}_{jk}^{2l-1}) \ &= \ \beta_{jk} \, \frac{1 - \beta_{jk}^{2m}}{1 - \beta_{jk}^2} + \overline{\beta}_{jk} \, \frac{1 - \overline{\beta}_{jk}^{2m}}{1 - \overline{\beta}_{jk}^2} \\ &= \ \frac{1 - \beta_{jk}^{2m} - (1 - \overline{\beta}_{jk}^{2m})}{\overline{\beta}_{jk} - \beta_{jk}} \ = \ \frac{\overline{\beta}_{jk}^{2m} (1 - \beta_{jk}^{4m})}{\overline{\beta}_{jk} - \beta_{jk}} \\ &= \ \frac{\overline{\beta}_{jk}^{2m} (1 - e^{i(j+k)\cdot 2\pi})}{\overline{\beta}_{jk} - \beta_{jk}} \ = \ 0 \ . \end{aligned}$$

Similarly, we see that

$$\sum_{l=1}^{m} (\gamma_{jk}^{2l-1} + \overline{\gamma}_{jk}^{2l-1}) = 0 \ ,$$

and we have proved the orthogonality.

To show that $(\varphi_k, \varphi_k) = 1$, we first note that $\beta_{00} = \gamma_{00} = 1$, and (9.8.3) takes the form

$$(\varphi_0, \varphi_0) = \tfrac{1}{4}\alpha_0^2 \sum_{l=1}^{m} 4 = \frac{1}{4m} \cdot 4m = 1 \ .$$

Next, for $k = 1, 2, \ldots, m-1$ we see that $\beta_{kk} \neq 1$, and proceeding as above we see that the terms in (9.8.3) with β_{kk} and $\overline{\beta}_{kk}$ sum to zero, while $\gamma_{kk} = 1$, and

$$(\varphi_k, \varphi_k) = \tfrac{1}{4}\alpha_k^2 \sum_{l=1}^{m} 2 = \frac{2}{4m} \cdot 2m = 1, \quad k = 1, 2, \ldots, m-1 \ .$$

This finishes the proof. □

The functions φ_j, $j = 1, 2, 3, 4$ are shown in Figure 9.7. Their amplitude is $\sqrt{2/m}$, but their shapes are independent of m.

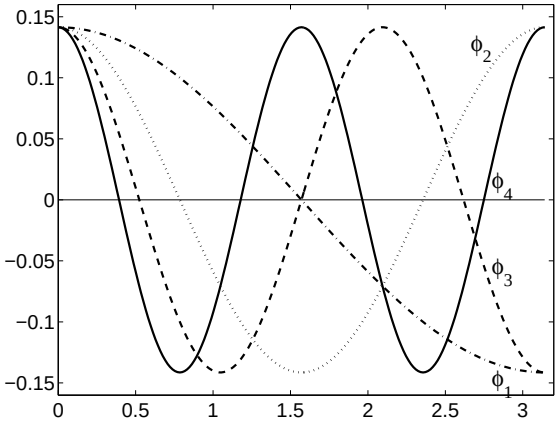


Figure 9.7. *Basis functions for the cosine transform.* $m = 100$.

Now we introduce the vectors $\varphi_{j\mathrm{G}} \in \mathbb{R}^m$, $j = 0, 1, \ldots, m-1$ with $(\varphi_{j\mathrm{G}})_l = \varphi_j(x_l)$, $l = 1, 2, \ldots, m$. According to the proof of Theorem 9.8.1 these vectors are orthogonal and have norm 1. This means that we can use them as an orthonormal basis in the space $\mathbb{R}^m$. In other words, we can write

$$f_{\mathrm{G}} = \sum_{j=0}^{m-1} c_j \varphi_{j\mathrm{G}}, \qquad c_j = \varphi_{j\mathrm{G}}^T f_{\mathrm{G}} \ . \tag{9.8.4}$$

The expression for the coefficient c_j follows from the discussion in Section 9.4.

The *discrete cosine transform* (*DCT*) of the signal $f_{\mathrm{G}} \in \mathbb{R}^m$ is the coefficient vector

$$c = (c_0, c_1, \ldots, c_{m-1})^T \ ,$$

and the relation (9.8.4) is known as the *inverse discrete cosine transform* (*IDCT*); it can be used to compute the signal from its DCT.

Example. The following MATLAB functions compute the DCT and the IDCT.[2)]

```
function  c = dct(f)
% Discrete cosine transform of f.
% Columnwise if f is a matrix
[m n] = size(f);  alpha = sqrt(2/m);
x = ([1:m] - 0.5)*(pi/m);  % grid as row vector
c = [sum(f)/sqrt(m)        % constant term
     zeros(m-1,n)];        % placeholder for c_1,...,c_(m-1)
for  k = 1 : m-1
  c(k+1,:) = alpha * (cos(k*x) * f);
end

function  y = idct(c)
% Inverse DCT.   Columnwise if c is a matrix
[m n] = size(c);  alpha = sqrt(2/m);
x = ([1:m]' - 0.5)*(pi/m);       % grid as column vector
y = repmat(c(1,:)/sqrt(m),m,1);  % initialize with constant term
for  k = 2 : m
  if  any(c(k,:) ~= 0)
    y = y + cos((k-1)*x) * (alpha*c(k,:));
  end
end
```

∎

Returning to (9.8.4), we see that the vector $\varphi_{k\mathrm{G}}$ corresponds to an oscillation with frequency $\frac{1}{2}k$, and the contribution $c_j\varphi_{j\mathrm{G}}$ has amplitude $|c_j|\alpha_j$. Typically, a signal contains "*noise*", ie each component in f_{G} has an error. The information that we are interested in is often contained in the small frequencies, while the noise has components with all frequencies. The vector

$$f_{\mathrm{G}}^* = \sum_{j=0}^{n} c_j \varphi_{j\mathrm{G}}, \qquad c_j = \varphi_{j\mathrm{G}}^T f_{\mathrm{G}} \ ,$$

with $n < m-1$ is the least squares fit to f_{G} by a linear combination of

[2)] It should be mentioned that these functions use $O(m^2)$ flops to compute a signal. A more efficient implementation, that only needs $O(n \log n)$ flops, makes use of the so-called *Fast Fourier Transform*, FFT, which, however, is outside the scope of this book.

$\{\varphi_{j\mathrm{G}}\}_{j=0}^{n}$. (Show that!) The vector is obtained by cutting off the high frequency components of the noise (we talk of a "*low pass filter*").

Example. The upper part of Figure 9.8 shows a signal with $m = 100$ points, and the signal contaminated with noise.

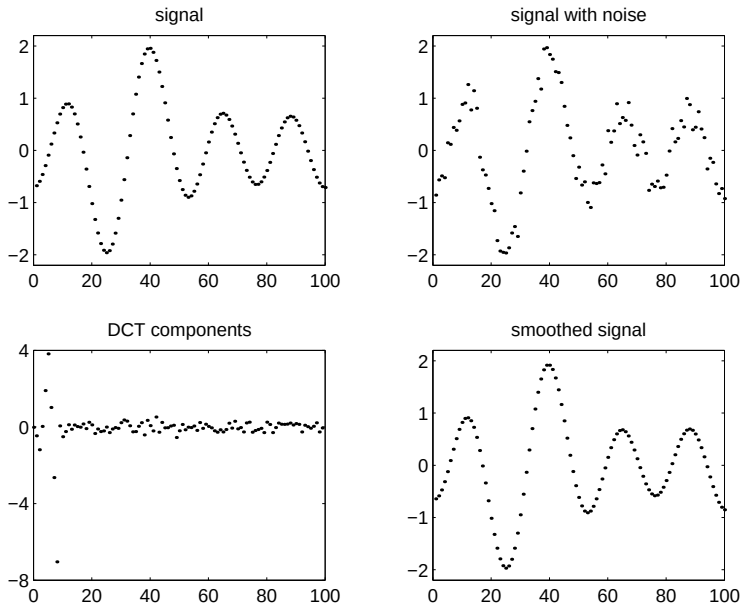


Figure 9.8. *Noise reduction by DCT.*

The signal with noise is given in a vector `fG`, and the lower left part of the figure shows the DCT of the signal, computed by the command.

```
>> c = dct(fG);
```

For $j > 8$ the coefficients c_j are dominated by noise, and the commands[3)]

```
>> c(10:end) = 0;  y = idct(c);
```

gives the smoothed signal, shown in the lower right subplot. We see a good agreement between this plot and the original noise-free signal. ∎

Example. The DCT is closely connected with *Fourier analysis*: It can be shown that if f is periodic with period 2π, and has a continuous first derivative, then the *Fourier series*

$$f(x) = \tfrac{1}{2}a_0 + \sum_{k=1}^{\infty}(a_k \cos kx + b_k \sin kx)$$

[3)] Remember that the coefficient c_j is found in `c(j+1)`.

converges uniformly. Therefore, it is reasonable to approximate f by a "trigonometric polynomial"

$$f^*(x) = \tfrac{1}{2}a_0 + \sum_{k=1}^{n}(a_k \cos kx + b_k \sin kx) \ .$$

We use Euler's formula and define $a_{-k} = a_k$, $b_{-k} = b_k$, $b_0 = 0$ and $c_k = \frac{1}{2}(a_k - ib_k)$. Then, we can express f^* in the form

$$f^*(x) = \tfrac{1}{2}\sum_{k=-n}^{n}(a_k - ib_k)e^{ikx} = \sum_{k=-n}^{n} c_k e^{ikx} \ . \tag{9.8.5}$$

Similar to the proof of Theorem 9.8.1 we can show that the functions

$$\psi_j(x) = e^{ijx}, \quad -m/2 < j < m/2$$

are orthogonal on the grid

$$G = \{x_1, x_2, \ldots, x_m\}, \quad x_r = (r-1)\,\frac{2\pi}{m} \ ,$$

with respect to the scalar product

$$(u, v) = \sum_{l=1}^{m} \overline{u}(x_l)v(x_l) \ .$$

The *discrete Fourier transform* (*DFT*) of the signal f_{G} is the complex valued vector c defined by

$$c = (c_0, c_1, \ldots, c_{m-1})^T, \quad c_k = \sum_{r=1}^{m} f_r e^{-ikx_r} \ ,$$

while the inverse DFT is the vector[4)]

$$y = (y_1, y_2, \ldots, y_m)^T, \quad y_r = \frac{1}{m}\sum_{j=0}^{m-1} c_j e^{ijx_r} \ .$$

The computation is performed by means of FFT in the MATLAB functions `fft` and `ifft`. As with the DCT we can smooth a noisy signal by putting large frequency components to zero.

It is outside the scope of this book to discuss the relative merits of DCT and Fourier transformation when applied to signal and image processing. ∎

Now, we generalize DCT to two dimensions: An *image* is an $m_1 \times m_2$ matrix F of pixel values. We can think of the image as values of a function $f(x, y)$, where we associate x with rows in F and y with columns. The DCT of F is the set of coefficients c_{kj} in the relation

[4)] Without giving details, this can be shown to be equivalent to the formulation in (9.8.5). The last half of the vector c corresponds to coefficients with negative index.

$$f(x_r, y_s) = \sum_{k=0}^{m_1-1} \sum_{j=0}^{m_2-1} c_{kj} \alpha_k^x \alpha_j^y \cos kx_r \cos jy_s \ ,$$

where the grids $\{x_r\}$ and $\{y_s\}$ are defined as in (9.8.1), with m replaced by m_1 and m_2, respectively. Similarly the normalization factors α are given by (9.8.2). The coefficients are conveniently stored in $m_1 \times m_2$ matrix C (with c_{kj} found in position $(k+1, j+1)$).

The transformation is computed in two steps. First, each column in F is a signal with m_1 elements, and the command

```
>> B = dct(F);
```

gives an $m_1 \times m_2$ matrix B, where $b_{k+1,s}$ is the factor in the contribution $b_k(y_s)\alpha_k^x \cos(kx)$ to the approximation of $f(x, y_s)$. Thus, the $(k+1)$st row in B is a signal consisting of m_2 discrete values of this function of y. The DCT of this signal is given by the vector

```
>> d = dct(B(k+1.:)');
```

The transposed of this vector is the $(k+1)$st row in C. The command

```
>> CT = dct(B');
```

returns the transposed of the matrix C. Similarly, the IDCT splits in two steps, that have to be made in reverse order. The two steps can be combined into one MATLAB command:

```
>> Y = idct( (idct(CT)' );
```

As in the one-dimensional case we can smooth the image by putting $c_{kj} = 0$ for large frequencies. We shall demonstrate another use of DCT, viz *data compression*. In practice m_1 and m_2 may be large, and the storage of the $m_1 * m_2$ pixel values needs a large chunk of memory. Also, electronic transfer of a large image may be very time consuming. Therefore, there has been intensive research in different ways of reducing the amount of data without loosing too much of the information.

Example. The simplest idea is simply to remove all components whose absolute value is smaller than a certain threshold `thr`, and compute the IDCT.

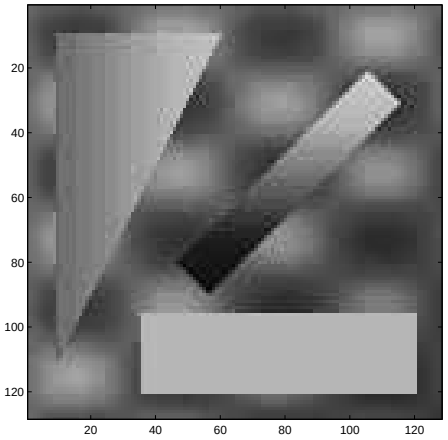
```
>> i = find(abs(CT) < thr);  CT(i) = 0;
>> Y = idct( (idct(CT))' );
```

■

This simple way of compressing an image is not very good (we return to it in a computer exercise). A reason for this is that if a certain basis function corresponding to $\cos kx \cdot \cos jy$ gives a significant contribution in one part of the image, then it is used in the entire image. As discussed in the chapters on interpolation and integration, we often get better results

by applying a "*divide-and-conquer*" strategy. One of the more successful algorithms is JPEG[5)]. Assume that both m_1 and m_2 are multiples of 8, $m_r = 8q_r$, $r = 1, 2$. Divide the image into $q_1 \times q_2$ blocks, each block consisting of 8×8 neighbouring pixels. Compute the two-dimensional DCT for each block, and keep only components with absolute values above a certain threshold. Thus, the contribution of $\cos kx \cdot \cos jy$ is kept only in those parts of the picture, where it is needed.

Example. Figure 6.6 on page 158 shows an image with $m_1 = m_2 = 128$. In Figure 9.9 we show a JPEG version of it, computed with the threshold value 2. For comparison, the largest of all the $16 * 16 * 8^2$ DCT components is 495.



4	3	3	4	4	3	3	4	4	2	3	4	4	2	3	3
3	52	9	11	11	9	9	45	3	4	3	2	3	3	4	2
3	10	2	2	2	2	38	3	3	3	3	3	43	47	3	3
3	8	2	2	2	2	40	3	3	4	3	45	41	42	50	2
4	8	2	2	2	41	3	4	4	2	44	41	2	43	49	3
4	10	2	2	2	36	3	4	4	49	45	2	45	46	4	3
2	8	2	2	38	3	3	2	53	49	2	47	47	2	3	2
4	10	2	2	38	2	3	50	51	2	50	48	4	2	4	4
4	8	2	39	4	3	54	52	2	48	49	3	4	2	3	3
4	8	2	40	3	55	56	2	51	50	4	3	3	4	3	2
3	10	40	3	3	54	56	54	53	2	3	3	3	3	3	2
3	8	43	2	39	8	58	56	9	9	9	10	9	8	9	2
4	44	5	4	8	1	1	1	1	1	1	1	1	1	1	4
4	40	3	4	8	1	1	1	1	1	1	1	1	1	1	4
2	4	4	2	8	1	1	1	1	1	1	1	1	1	1	2
3	2	3	4	4	2	3	3	3	2	3	4	4	2	3	2

Figure 9.9. *Left: JPEG version of the image on page 158. Right: Number of DCT component kept in each block.*

The right part of Figure 9.9 shows the number of DCT components kept in each of the 16×16 blocks. The rectangle in the lower right part of the image has constant pixel values, and we only need the DCT component c_{00}. The other extreme is a sharp edge, which is not parallel to the x-axis or the y-axis. There we may have to keep almost all the 64 DCT components of the block. The total picture needs 3082 components, so in this case we save about 81% of the $128^2 = 16384$ pixel values, and there is no discernible loss of detail. For larger pictures without too many sharp edges the saving may be even larger.

It is outside the scope of this book to discuss how to avoid storing the zeros in the DCT, and how to exploit the zeros during the inverse transform. ∎

[5)] Acronym for Joint Photographic Experts Group.

9.9. Minimax Approximation – Chebyshev Approximation

In this section we shall consider polynomial approximation of a function $f \in C[a,b]$ (the continuous case) when the norm is defined as

$$\|g\|_\infty = \max_{a \le x \le b} |g(x)| \ .$$

It can be shown that also with respect to this norm, there exists a uniquely defined polynomial p_n^* of degree $\le n$, such that

$$E_n(f) = \|f - p_n^*\|_\infty \le \|f - p_n\|_\infty$$

for all polynomials p_n of degree $\le n$.

According to a theorem of Weierstrass, any function $f \in C[a,b]$ can be approximated arbitrarily well by a polynomial, ie $E_n(f) \to 0$ as $n \to \infty$. There is also a known relation between the regularity properties of f and the behaviour of $E_n(f)$ for large n. If, eg, f is k times continuously differentiable, then $E_n(f) = O(1/n^k)$ as $n \to \infty$.

It is considerably more difficult to determine p_n^* than to determine the least squares approximation. Therefore, we start by describing some simple methods for *approximate* determination of p_n^*. For simplicity, we assume that the interval has been transformed to $[-1,1]$, cf (9.5.4).

First, p_n^* can be approximated by an interpolating polynomial. The simplest choice is to take *equidistant* interpolation points, but Runge's phenomenon (Figure 5.5 on page 118) illustrates that this is a bad choice when we want the error to be small in all parts of the interval. The figure shows that there are large errors close to the endpoints of the interval. This suggests that we should put more interpolation points close to $x = -1$ and $x = 1$, in order to force the interpolating polynomial better to follow the given curve there. Such a distribution of points is obtained if we use the Chebyshev nodes (9.7.2) defined as the roots of T_{n+1}. More specific, use the $n+1$ interpolation points

$$x_i = \cos\Big(\frac{2i+1}{2(n+1)}\,\pi\Big), \quad i = 0, 1, \ldots, n \ .$$

This so-called *Chebyshev interpolation* can be expected to give a good approximation to p_n^*. Another motivation for this choice of interpolation points is provided by Theorem 5.2.2: If f is $n+1$ times continuously differentiable, then the error is

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}\,(x-x_0)(x-x_1)\cdots(x-x_n)\ .$$

In order to minimize $\|f-p_n\|_\infty$, the interpolation points should be chosen so that

$$\max_{-1\le x\le 1} |(x-x_0)(x-x_1)\cdots(x-x_n)|$$

is minimized, and according to Theorem 9.7.1 this is the case when

$$(x-x_0)(x-x_1)\cdots(x-x_n) = 2^{-n}T_{n+1}(x)\ ,$$

ie, when $x_0, x_1, \ldots, x_n$ are the roots of T_{n+1}.

For $f \in C[-1,1]$ it can be shown that the magnitude of the error in Chebyshev interpolation is at most $4E_n(f)$ if $n \le 20$, and at most $5E_n(f)$ if $n \le 100$. This means that the method gives a good approximation to the minimax approximation p_n^*.

Example. Compute an approximation to the polynomial $c_0^*+c_1^*x$ that minimizes

$$\psi(c_0,c_1) = \max_{0\le x\le 1} \big|\sin\big(\frac{\pi}{2}x\big) - (c_0+c_1x)\big|\ .$$

First, we make a transformation of variables

$$x = \tfrac{1}{2}(t+1)$$

to get the standard interval $[-1,1]$. This gives

$$\psi(c_0,c_1) = \max_{-1\le t\le 1} \big|\sin\big(\frac{\pi}{4}(t+1)\big) - (c_0+\tfrac{1}{2}c_1(t+1))\big|\ .$$

Put $a_0 = c_0 + \frac{1}{2}c_1$, $a_1 = \frac{1}{2}c_1$ and $f(t) = \sin(\frac{\pi}{4}(t+1))$. Then the problem is to compute a_0, a_1 so as to minimize

$$\max_{-1\le t\le 1} |f(t) - (a_0+a_1t)|\ .$$

Chebyshev interpolation means that a_0 and a_1 are determined so that

$$a_0 + a_1t_i = f(t_i)\ , \quad i = 1, 2\ ,$$

where t_1 and t_2 are the roots of $T_2(t) = 2t^2-1$, ie $t_1 = -1/\sqrt{2}$, $t_2 = 1/\sqrt{2}$. We therefore get the linear system of equations

$$\begin{pmatrix} 1 & -1/\sqrt{2} \\ 1 & 1/\sqrt{2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} \sin(\frac{\pi}{4}(1-1/\sqrt{2})) \\ \sin(\frac{\pi}{4}(1+1/\sqrt{2})) \end{pmatrix}\ ,$$

which has the solution $a_0 \simeq 0.60084$, $a_1 \simeq 0.52725$, so that we get

$$c_0^+ \simeq 0.07359, \quad c_1^+ \simeq 1.0545\ .$$

The corresponding error function has largest absolute value for $x=1$, $f(1) - f^+(1) \simeq -0.1281$.

In the example on page 300 we shall see that

$$c_0^* \simeq 0.1053, \quad c_1^* = 1, \quad \|f - f^*\|_\infty \simeq 0.1053 \ .$$

Thus, in this case the approximate minimax approximation gives a maximum error that is about $0.1281/0.1053 \simeq 1.22$ times larger than the maximum error in the true minimax approximation. ∎

In this example, with a low degree polynomial, it was easy to compute the coefficients a_0 and a_1 as the solution to the linear system of equations. For higher degrees of the approximating polynomial it is advantageous to interpret the interpolation problem as a least squares problem, and use orthogonal polynomials to solve that. We illustrate this technique by the same problem.

Example. We seek a_0, a_1 so that

$$\sum_{i=1}^{2} \left(f(t_i) - (a_0 + a_1 t_i)\right)^2$$

is minimized. The Chebyshev polynomials T_0 and T_1 are orthogonal with respect to the scalar product

$$(g, h) = \sum_{i=1}^{2} g(t_i)h(t_i) \ .$$

Therefore, we express the approximating polynomial in terms of Chebyshev polynomials,

$$a_0 + a_1 t = a_0 T_0(t) + a_1 T_1(t) \ .$$

The coefficients are now obtained directly as orthogonal coefficients

$$a_k = \frac{(f, T_k)}{(T_k, T_k)} = \frac{\sum_{i=1}^{2} f(t_i)T_k(t_i)}{\sum_{i=1}^{2} T_k^2(t_i)} \ , \quad k = 0, 1 \ .$$

With $f(t) = \sin(\frac{\pi}{4}(t+1))$, $t_1 = -1/\sqrt{2}$, $t_2 = 1/\sqrt{2}$ we get

$$a_0 = \left(\sin(\tfrac{\pi}{4}(1 + 1/\sqrt{2})) + \sin(\tfrac{\pi}{4}(1 - 1/\sqrt{2}))\right)/2 \simeq 0.60084 \ ,$$
$$a_1 = \left(\sin(\tfrac{\pi}{4}(1 + 1/\sqrt{2})) - \sin(\tfrac{\pi}{4}(1 - 1/\sqrt{2}))\right)/\sqrt{2} \simeq 0.52725 \ .$$

We get the same values as before, of course. ∎

An alternative way to compute an approximation to the true minimax polynomial approximation of f on $[-1, 1]$ is to find the best approximation of f in the Euclidean norm with weight function $w(x) = 1/\sqrt{1-x^2}$. Intuitively we see that the weighted Euclidean norm of the error function can be small only if the error function itself is specially small close to the endpoints -1 and 1. To study this technique more closely we first define

the notion of an orthogonal expansion.

Definition 9.9.1. Let $\varphi_0, \varphi_1, \varphi_2, \ldots$ be an orthogonal system of polynomials with respect to the scalar product

$$(g,h) = \int_a^b w(x)g(x)h(x)\,dx\ .$$

The expansion

$$\sum_{j=0}^{\infty} c_j\varphi_j(x) \qquad \text{with} \quad c_j = (f,\varphi_j)/(\varphi_j,\varphi_j)$$

is called an *orthogonal expansion* of the function f.

From the discussion earlier in this chapter we know that, among all polynomials of degree $\le n$, the partial sum $p_n = \sum_{j=0}^{n} c_j\varphi_j$ is the best approximation to f, with respect to the weighted Euclidean norm $\|\cdot\|_{2,w}$.

As before, let p_n^* be the best (with respect to the Chebyshev norm) approximation to f among all polynomials of degree $\le n$. Then

$$\begin{aligned}\|f-p_n\|_{2,w}^2 \le \|f-p_n^*\|_{2,w}^2 &= \int_a^b w(x)\big(f(x)-p_n^*(x)\big)^2\,dx\\ &\le \max_{a\le x\le b}|f(x)-p_n^*(x)|^2\int_a^b w(x)\,dx\\ &= E_n^2(f)\int_a^b w(x)\,dx\ .\end{aligned}$$

Since $E_n(f)\to 0$ as $n\to\infty$ (see page 296), we also get

$$\|f-p_n\|_{2,w}\to 0 \qquad \text{as} \quad n\to\infty\ .$$

Only in certain special cases, however, does p_n converge uniformly to f. This is the case, eg, if f is twice continuously differentiable in $[-1,1]$, and the φ_j are chosen as Chebyshev polynomials. A truncated orthogonal expansion $\sum_{j=0}^{n} c_jT_j$ should therefore give a good approximation. For $f\in C[-1,1]$ it can be shown that

$$\big\|f-\sum_{j=0}^{n} c_jT_j\|_\infty < (4+\log n)E_n(f)\ .$$

If $n\le 400$, then the factor of $E_n(f)$ is at most 10. This shows that there is an acceptable loss of accuracy associated with using $\sum_{j=0}^{n} c_jT_j$ instead of p_n^*.

Example. Use the least squares method with weight function $1/\sqrt{1-t^2}$ to approximate $f(t) = \sin(\frac{\pi}{4}(t+1))$ on the interval $-1 \le t \le 1$ by a straight line $a_0 + a_1 t$.

$$a_0 = \frac{(f,T_0)}{(T_0,T_0)} = \frac{1}{\pi}\int_{-1}^{1} \frac{\sin\frac{\pi}{4}(t+1)}{\sqrt{1-t^2}}\,dt \simeq 0.60219\ ,$$

$$a_1 = \frac{(f,T_1)}{(T_1,T_1)} = \frac{2}{\pi}\int_{-1}^{1} \frac{t\sin\frac{\pi}{4}(t+1)}{\sqrt{1-t^2}}\,dt \simeq 0.51363\ .$$

The integrals were evaluated via substitution of variables, $t = \cos u$ and a standard program for numerical integration (we used the MATLAB function `quad`). The corresponding approximation to $\sin(\frac{\pi}{2}x)$ on $[0,1]$ is $c_0 + c_1 x$ with

$$c_0 = a_0 - a_1 \simeq 0.08857, \qquad c_1 = 2a_1 \simeq 1.0273\ .$$

The maximum error is $|f(1) - (c_0 + c_1)| \simeq 0.1158 \simeq 1.1 \cdot \|f - p_1^*\|_\infty$, where p_1^* is the degree 1 polynomial that is the true minimax approximation; see the next example. ∎

Now, we turn our attention to the computation of the true minimax polynomial approximation, p_n^*. This is actually quite easy if we want to approximate a *monotone* function by a *straight line*. The line shall be chosen so that the error function assumes extreme values with alternating signs at three points: the two endpoints of the interval and an interior point, cf Figure 9.10.

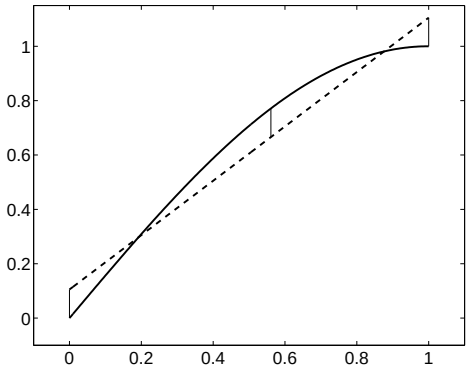


Figure 9.10. *Minimax approximation with a straight line*

It is obvious that the maximum error cannot be made smaller by moving this line.

Example. Compute c_0, c_1, that minimize

$$\max_{0\le x\le 1} |\sin(\frac{\pi}{2}x) - (c_0 + c_1 x)|$$

The error function $r(x) = \sin(\frac{\pi}{2}x) - (c_0 + c_1 x)$ shall assume the extreme value d with alternating signs at the points $0, \xi, 1$, where $r'(\xi) = 0$. This gives four equations in the four unknowns c_0, c_1, ξ and d:

$$\begin{aligned} -c_0 &= \ d \ , \\ \sin(\tfrac{\pi}{2}\xi) - c_0 - c_1\xi &= -d \ , \\ 1 - c_0 - c_1 &= \ d \ , \\ \tfrac{\pi}{2}\cos(\tfrac{\pi}{2}\xi) - c_1 &= \ 0 \ . \end{aligned}$$

The solution (shown in Figure 9.10) is

$$\begin{aligned} c_1 &= 1 \ , \\ \xi &= \tfrac{2}{\pi}\arccos(\tfrac{2}{\pi}) \simeq 0.5607 \ , \\ c_0 &= \tfrac{1}{2}(\sin(\tfrac{\pi}{2}\xi) - \xi) \simeq 0.1053 \ , \\ d &= -c_0 \simeq -0.1053 \ . \end{aligned}$$

■

The error function for the Chebyshev approximation by a straight line (a polynomial of degree 1) oscillates between the extreme values in three points, and in the example on page 287 we saw that the error function for the Chebyshev approximation of x^4 by a degree $n=3$ polynomial oscillates between the extreme values in $n+2 = 5$ points. I can be shown that this generalizes:

Theorem 9.9.2. Assume that $f \in C[a, b]$. Among all polynomials of degree $\leq n$, the polynomial p_n^* is the best maximum norm approximation of f if and only if there are points $a \leq \xi_1 < \xi_2 < \cdots < \xi_{n+2} \leq b$ such that

$$|f(\xi_k) - p_n^*(\xi_k)| = \|f - p_n^*\|_\infty, \quad k = 1, 2, \ldots, n+2$$

and

$$f(\xi_{k+1}) - p_n^*(\xi_{k+1}) = -\left(f(\xi_k) - p_n^*(\xi_k)\right), \quad k = 1, 2, \ldots, n+1 \ .$$

Thus, the error function $f - p_n^*$ alternates between $\pm\|f - p_n^*\|_\infty$ in at least $n+2$ points. This so-called *alternation property* is used in algorithms for constructing p_n^*. We did so for $n=1$ in the previous example, but it is harder for $n > 1$.

Example. Assume that $f(x) = \sin(\frac{\pi}{2}x)$ shall be approximated by a second degree polynomial $p_2(x) = c_0 + c_1 x + c_2 x^2$ on the interval $0 \leq x \leq 1$. Let the error function $r(x) = f(x) - p_2^*(x)$ have the extrema $\xi_1, \xi_2, \xi_3, \xi_4$. It is easy

to see that $\xi_1 = 0$ and $\xi_4 = 1$. The alternation property gives the conditions

$$\begin{aligned} -c_0 &= \quad d \ , \\ \sin(\tfrac{\pi}{2}\xi_2) - (c_0 + c_1\xi_2 + c_2\xi_2^2) &= -d \ , \\ \sin(\tfrac{\pi}{2}\xi_3) - (c_0 + c_1\xi_3 + c_2\xi_3^2) &= \quad d \ , \\ 1 - (c_0 + c_1 + c_2) &= -d \ . \end{aligned} \tag{9.9.1a}$$

The conditions $r'(\xi_2) = 0$ and $r'(\xi_3) = 0$ give

$$\begin{aligned} \tfrac{\pi}{2}\cos(\tfrac{\pi}{2}\xi_2) - (c_1 + 2c_2\xi_2) &= 0 \ , \\ \tfrac{\pi}{2}\cos(\tfrac{\pi}{2}\xi_3) - (c_1 + 2c_2\xi_3) &= 0 \ . \end{aligned} \tag{9.9.1b}$$

Thus, we have a system of 6 nonlinear equations in the 6 unknowns ξ_2, ξ_3, c_0, c_1, c_2 and d, and we can use one of the solution methods discussed in Section 4.8. A popular, special purpose iteration method is the *Remez algorithm* (also called the *exchange algorithm*):

0. Initialize: Chose $\xi_2^{[0]}$ and $\xi_3^{[0]}$. $k := 0$
1. Use $\xi_2^{[k]}$ and $\xi_3^{[k]}$ for ξ_2 and ξ_3 in (9.9.1a) and solve this linear system of equations in c_0, c_1, c_2 and d.
2. Insert these values in (9.9.1b), and use Newton-Raphson's method to find corrected values for the interior extremum points, $\xi_2^{[k+1]}$ and $\xi_3^{[k+1]}$.
3. If the error function alternates according to Theorem 9.9.2, then stop. Otherwise, $k := k{+}1$ and repeat from 1.

The starting values $\xi_2^{[0]}$ and $\xi_3^{[0]}$ may be obtained via one of the approximate methods described earlier in this chapter, eg Chebyshev interpolation.

The resulting error curve is shown in Figure 9.11. It corresponds to the

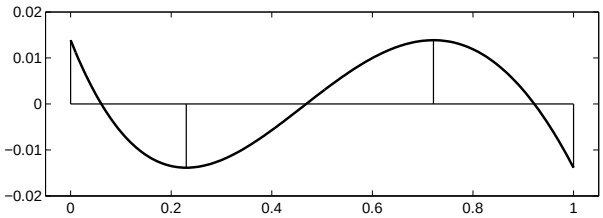


Figure 9.11. *Error curve for minimax approximation of* $\sin(\frac{\pi}{2}x)$ *by a degree 2 polynomial*

solution

$$\begin{array}{ll} \xi_2 = 0.2299 & c_0 = -0.0139 \\ \xi_3 = 0.7215 & c2 = \quad 1.8455 \qquad d = 0.0139 \ . \\ & c3 = -0.8178 \end{array}$$

■

Exercises

E1. Let p_1 and p_2 be polynomials of degree n, determined as least squares approximations to f_1 and f_2, respectively. Show that $\alpha_1 p_1 + \alpha_2 p_2$ is the least squares approximation to $\alpha_1 f_1 + \alpha_2 f_2$.

E2. Let the following measurements be given

x	-2	-1	1	3	4
$f(x)$	1.4	1.7	1.8	1.6	1.2

.

We wish to find a second degree polynomial

$$p(x) = c_0\varphi_0(x) + c_1\varphi_1(x) + x_2\varphi_2(x)$$

as a least squares fit (with $w_i = 1$) to these data.

Discuss the merits of each of the following three choices of $\{\varphi_j\}$:

$\varphi_0(x)$	$\varphi_1(x)$	$\varphi_2(x)$
1	x	x^2
1	$x-1$	$(x-1)^2$
$x-1$	x^2-2x-3	$x^2-2x-14$

.

E3. Let $\varphi_0, \varphi_1, \ldots, \varphi_n$ be an orthogonal system.

(a) Prove the following generalization of the Pythagorean law:

$$\Big\|\sum_{j=0}^{n} c_j\varphi_j\Big\|_2^2 = \sum_{j=0}^{n} c_j^2 \|\varphi_j\|_2^2 \ .$$

(b) Use this result to show that the functions in an orthogonal system are linearly independent.

E4. Let $f^* = \sum_{j=0}^{n} c_j^*\varphi_j$ be the least squares approximation to a given function f. Use the Pythagorean law to show that

$$\|f - f^*\|_2^2 = \|f\|_2^2 - \|f^*\|_2^2 \ .$$

E5. Let $P(x) = \sum_{k=0}^{n} a_k T_k(x)$, where the T_k are Chebyshev polynomials. For given x, the polynomial can be evaluated by the recurrence

$$\begin{aligned} b_{n+2} &= b_{n+1} = 0 \ , \\ b_k &= a_k + 2xb_{k+1} - b_{k+2} \ , \quad k = n, n-1, \ldots, 0 \ . \end{aligned}$$

Show that $P(x) = b_0 - xb_1$.

E6. Let $x = \cos v$ and define

$$U_n(x) = \frac{\sin(n+1)v}{\sin v} \ , \quad n = 0, 1, \ldots \ .$$

(a) The functions $U_n(x)$ satisfy a recursion formula of the type

$$U_{k+1}(x) = (\alpha_k x - \beta_k)U_k(x) - \gamma_k U_{k-1}(x) \ .$$

Show this, and determine the coefficients.

(b) Show that $U_n(x)$ is a polynomial in x of degree n. (The $U_n(x)$ are called *Chebyshev polynomials of the second kind*).

(c) Show that the functions $U_0, U_1, \ldots$ build an orthogonal system on the interval $[-1,1]$, with respect to the weight function $w(x) = \sqrt{1-x^2}$.

E7. For the function f it holds that $|f^{(n+1)}(x)| \le M$ for all $x \in [-1,1]$. Show that there is a polynomial P_n of degree n such that

$$|f(x) - P_n(x)| \le \frac{M}{2^n \cdot (n+1)!} \quad \text{for all} \ \ x \in [-1,1] \ .$$

E8. Use the minimax property of Chebyshev polynomials to show that for each polynomial $p_n(x) = \sum_{k=0}^{n} c_k x^k$, $c_n \neq 0$, there exists a point $\xi \in [-1,1]$ such that $|p_n(\xi)| \ge |c_n| \cdot 2^{1-n}$.

E9. Approximate $f(x) = \sqrt[3]{x}$ by a straight line in the interval $[0,1]$

(a) in the least squares sense with the weight function $w(x) = 1$,

(b) in the maximum norm.

In both cases give the norm of the error function for the best approximation.

E10. (a) Determine polynomials φ_k, $k = 0,1,2$, with leading coefficient 1, which are orthogonal on $[-1,1]$ with respect to the weight function $w(x) = x^2$.

(b) Compute the second degree polynomial $p_2^*(x)$, which minimizes

$$\int_{-1}^{1} x^2 \big(\sin(\frac{\pi}{2}\, x) - p_2(x)\big)^2 \, dx \ .$$

(c) Use the results from Exercises E3 and E4 to compute

$$\| \sin(\frac{\pi}{2}\, x) - p_2^*(x)\|_2^2 = \int_{-1}^{1} x^2 \big(\sin(\frac{\pi}{2}\, x) - p_2^*(x)\big)^2 \, dx \ .$$

E11. Given the data

x_i	-2	-1	0	1	2
$f(x_i)$	29	7	5	5	13

(a) Use the recurrence

$$\varphi_0(x) = 1 \ , \qquad \varphi_1(x) = x - \beta_0 \ ,$$
$$\varphi_{k+1}(x) = (x - \beta_k)\varphi_k(x) - \gamma_k \varphi_{k-1}(x) \ , \quad k = 1, 2$$

to determine polynomials φ_k, $k = 0, 1, 2, 3, 4$, that form an orthogonal system on $\{x_i\}$ with respect to $w_i = 1$.

(b) Is it possible to augment the orthogonal system with $\varphi_5(x) = x^5 + \ldots$?

(c) Use the polynomials from (a) to determine the polynomial p_n^* of degree $n = 3$, which minimizes

$$\sum_{i=1}^{5} \big(f(x_i) - p_n(x_i)\big)^2 \ .$$

(d) As (c), except that now $n = 4$.

E12. Let f_{G} be a signal with m elements and let Φ be the $m \times m$ matrix, whose $(j+1)$st column is $\varphi_{j\mathrm{G}}$, $j = 0, 1, \ldots, m-1$, where φ_j is defined by (9.8.2).

(a) Show that Φ is an orthogonal matrix and that the DCT of f_{G} and the IDCT can be formulated as

$$c = \Phi^T f_{\mathrm{G}} \ , \qquad f_{\mathrm{G}} = \Phi \, c \ .$$

(b) Let F be an $m \times m$ image. Show that the two-dimensional DCT and IDCT can be formulated as

$$C = \Phi^T \, F \, \Phi \ , \qquad F = \Phi \, C \, \Phi^T \ .$$

Computer Exercises

C1. In this exercise we shall simulate the data fitting problem outlined on page 262. Let $g(x) = 0.75x - 2x^2 + x^3$, and compute the data

```
x = linspace(0,1,11)
y = g(x) + u*randn(size(x))
```

Choose $u = 10^{-6}$ and $u = 10^{-2}$. In both cases use `polyfit` to compute the least squares polynomial fit of degree 3, and plot the data and the fit. Show the error functions in a separate plot.

C2. Consider the function

$$f(x) = \frac{x}{0.25 + x^2}$$

in the points $x_i = -2 + 0.5i$, $i = 0, 1, \ldots, 8$. Approximate the points $\big(x_i, f(x_i)\big)$ by polynomials of degree $3, 5, 7$, determined by the least squares method (use `polyfit`). Compare these results with the results of using piecewise polynomials and cubic splines, see Exercise C3 in Chapter 5.

C3. We know that $f(x) = \dfrac{1}{ax^2 + b}$, and are given the measurement data

x_i	1	2	3	4
$f(x_i)$	1.2	0.55	0.29	0.17

Use the least squares method to compute a and b. Choose weights $w_i = 1/x_i^2$ in order to get good agreement for x-values close to 1.

C4. Compute the radius r and centre (a, b) of the circle, that best possible (in the least squares sense) approximates the points

x_i	4	3.5	2.5	1.5	1	1.5	2.5	3.5
$f(x_i)$	2	3	3.5	3	2	1	0.5	1

The equation of the circle is $(x-a)^2 + (y-b)^2 = r^2$, or

$$2ax + 2by + (r^2 - a^2 - b^2) = x^2 + y^2 \ .$$

Put $c = r^2 - a^2 - b^2$, and determine a, b, c such as to minimize

$$\sum_{i=1}^{8} \big(x_i^2 + y_i^2 - 2x_i a - 2y_i b - c\big)^2 \ .$$

Hint: Start by formulating an overdetermined system of equations $Ad \simeq z$, where d is the vector of unknown parameters, $d = (a\ b\ c)^T$.
(The exercise is based on an idea of Walter Gander).

C5. The file exc9_5.mat in incbox contains the two vectors `fo` and `fG`, which are shown in the upper left and right part of Figure 9.8, respectively.

(a) Plot the DCT of the two signals in the same figure.

(b) The noise is `N = fG - fo` . Plot `N` and its DCT in separate figures.

(c) Let c be the DCT of `fG`, and let $f^*_{n\mathrm{G}}$ denote the approximate signal obtained by putting $c_j = 0$ for $j > n$. Plot the error in $f^*_{n\mathrm{G}}$ for $n = 4, 5, 8, 10, 20$.

(d) Experiment with compression of the signal given in `fo`.

C6. The file exc6_4.mat in incbox contains a matrix `A` that represents the image shown in Figures 6.6 and 9.9. Use the compression algorithm outlined in the first example on page 294 with the threshold given by `thr` $= \rho \cdot \max_{k,j} |c_{kj}|$, where $0 < \rho < 1$.

Investigate how the choice of ρ influences the compression and the quality of the compressed image.

C7. Consider the function $f(x) = \sqrt{x}$ on the interval $[1, 4]$, cf Section 4.7. We want to approximate f in the minimax sense, by polynomials p_n^* of degrees $n = 1, 2, 3$.

(a) Use Chebyshev interpolation to find p_n that approximate p_n^*.

(b) Use the Remez algorithm to find p_n^*.

In each case, what is the maximum error of the approximation ?

References

The presentation of the least squares method is inspired by the corresponding sections in

G. Dahlquist, Å. Björck, *Numerical methods*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

For a quite extensive treatment of different approximation methods, including a thorough description of trigonometric approximation and the fast Fourier transform (FFT) we recommend

D. Kincaid, W. Cheney, *Numerical analysis*, Brooks/Cole Publishing Company, Pacific Grove, CA, 1991.

The first book to give useful approximations for the computation of elementary functions on computers was

G. Hastings, *Approximations for digital computers*, Princeton University Press, Princeton, N.J., 1955.

The choice of approximating functions was made using intuition and great artistic skill.

Now there are systematic methods for approximation of a function by polynomials or rational functions. Examples of such approximations are given in

J.F. Hart et al., *Computer approximations*, John Wiley and Sons, New York, 1968.

For those who, in addition, want an extensive description of interpolation and approximation with spline functions, we recommend

M.J.D. Powell, *Approximation theory and methods*, Cambridge University Press, 1981.

A good and extensive treatment of the DCT and its use in image analysis can be found in

W.B. Pennebaker, J.L. Mitchell, *JPEG still image data compression standard*, Van Norstrand Reinhold, 1993.

Chapter 10

Ordinary Differential Equations

10.1. Introduction

Example. Consider a box resting on a firm foundation, and connected to a wall by a spring, see the figure.

Introduce a coordinate axis with the origin at the point where the box is when the spring exerts no force. If we move the box from the origin and then release it, then the spring will pull the box back towards the origin, and the friction between the box and the foundation will slow down the movement.

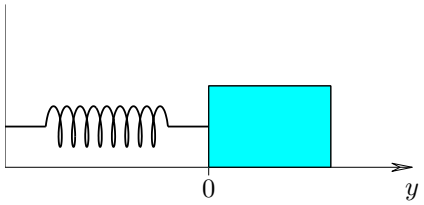


Figure 10.1. *Box on a foundation with friction.*

Let $y(t)$ denote the position of the box at time t. Hooke's law states that the spring force is proportional to y. The friction force is proportional to the velocity y'. Newton's law of motion gives

$$m\,y'' = -Ky - Qy' \ ,$$

where m is the mass of the box and K and Q are proportionality constants. In order to be able to find a unique solution of this differential equation we need to know the initial position and velocity. Thus, the initial conditions are

$$y(0) = \alpha, \qquad y'(0) = \beta \ ,$$

where α and β are given constants. ∎

A *differential equation* is an equation involving an unknown function and one or more of its derivatives. In this chapter we consider *ordinary differential equations of first order*, which can be written

$$y' = f(x, y) \ .$$

Here, $f(x, y)$ is a given real valued function of two variables, and $y = y(x)$ is the solution. It should be noted, that $y' = f(x, y)$ is shorthand notation for

$$y'(x) = f\big(x, y(x)\big) \ .$$

Example. Let the given function be $f(x, y) = y$. The differential equation

$$y' = y$$

is satisfied by $y(x) = Ce^x$ for any choice of C, see Figure 10.2.

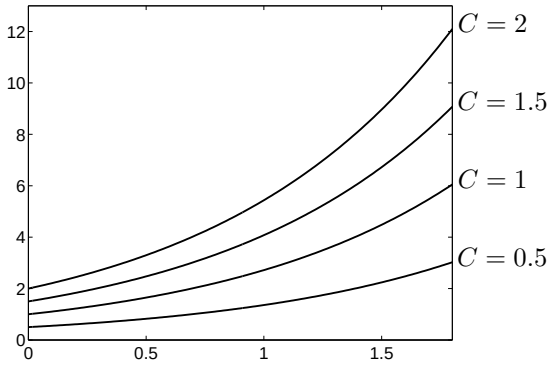


Figure 10.2.
Solutions of $y' = y$.

This example illustrates that in general an equation $y' = f(x, y)$ has an infinity of solutions. In order to fix the desired solution you have to supply a condition of the form $y(a) = \alpha$. This is called an *initial condition*, and the problem of solving

$$y' = f(x, y), \qquad y(a) = \alpha \ ,$$

is called an *initial value problem*.

Example. The initial value problem

$$y' = y, \qquad y(0) = 1 \ ,$$

has the unique solution $y(x) = e^x$.

In applied sciences the modelling of a dynamic process leads to an initial value problem[1)] . This was illustrated in the first example, where we derived an *ordinary differential equations of second order* (the equation involves the second derivative). In order to solve such an equation, two initial conditions are needed.

By introducing new unknown functions, any higher order differential equation can be written as a *system of first order differential equations.*

Example. In the first example put $y_1 = y$ and y_2 equal to the velocity, $y_2 = y'$. Then we get the differential equation

$$my_2' = -Ky_1 - Qy_2 \ ,$$

and this must be supplied with $y_1' = y_2$. Thus, the initial value problem is

$$\begin{aligned} y_1' &= y_2 \ , & y_1(0) &= \alpha \ , \\ y_2' &= -(K/m)y_1 - (Q/m)y_2 \ , & y_2(0) &= \beta \ . \end{aligned}$$

The problem is of the form

$$y' = f(t, y), \qquad y(0) = c \ ,$$

where y, c and $f(t, y)$ are vectors with two components,

$$y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad c = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad f(t, y) = \begin{pmatrix} y_2 \\ -(K/m)y_1 - (Q/m)y_2 \end{pmatrix} \ .$$

■

In most of this chapter the equation $y' = f(x, y)$ is assumed to be scalar, ie y is a function from $\mathbb{R}$ to $\mathbb{R}$. However, most of the material presented only needs small modifications to be applicable to systems of differential equations.

For completeness we state Theorem 10.1.1 below. It shows when an initial value problem has a unique solution.

An alternative formulation of the statement in the theorem is: Through every point $(\overline{x}, \overline{y})$ with $\overline{x}$ in the interval $[a, b]$, there passes exactly one curve that is a solution of the equation $y' = f(x, y)$.

Example. Consider the initial value problem

$$y' = f(x, y) = xy, \qquad y(0) = 1 \ ,$$

and assume that we want to solve the equation in the interval $[0, 1]$. We immediately get

$$|xy - x\widetilde{y}| \leq |y - \widetilde{y}| \ ,$$

[1)] The independent variable is often denoted t (for *time*) instead of x. In most of this chapter, however, we shall use x.

Theorem 10.1.1. Let the function $f(x, y)$ be defined and continuous for all points (x, y) in the strip $a \le x \le b$, $-\infty < y < \infty$, where a and b are finite. If f satisfies a *Lipschitz condition*, ie if there exists a constant L such that

$$|f(x, y) - f(x, \widetilde{y})| \le L|y - \widetilde{y}| \quad \text{for all} \quad x \in [a, b] \text{ and all } y, \widetilde{y}$$

then for any initial value α there exists a unique solution of the initial value problem

$$y' = f(x, y), \qquad y(a) = \alpha \ .$$

L is called the *Lipschitz constant*.

so the Lipschitz constant is $L = 1$. More generally, it is easy to show that if

$$\left|\frac{\partial f}{\partial y}\right| \le L \quad \text{for } a \le x \le b \ ,$$

then L can be used as the Lipschitz constant. ∎

Not all problems involving differential equations are initial value problems. In Sections 10.8 through 10.11 we study numerical methods for a *two-point boundary value problem for an ordinary differential equation*. More specific, we look at the problem

$$y'' = f(x, y, y'), \qquad y(a) = \alpha, \quad y(b) = \beta \ ,$$

where $[a, b]$ is an interval.

10.2. Solution of Initial Value Problems

Most differential equations that arise in applications cannot be solved by analytical methods. Therefore, it is necessary to make numerical approximations. We will discuss methods that are based on the following idea: Since we cannot determine the function $y(x)$ for all x in an interval $[a, b]$, we will have to be satisfied with computing *approximations* y_n of $y(x_n)$ for some points $\{x_n\}_{n=0}^{N}$ in the interval. We assume that the points are equidistant:

$$x_n = a + nh, \quad n = 0, 1, \ldots, N \ ,$$

where the *step length* h is defined as

$$h = \frac{b-a}{N} \ ,$$

for some integer N; see Figure 10.3.

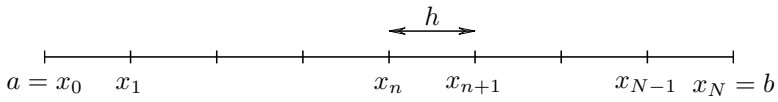


Figure 10.3. *The interval* $[a, b]$ *divided into* N *subintervals.*

The initial value gives us $y_0 = y(x_0) = \alpha$ for $x = x_0$. Now, we will first compute y_1 as an approximation of $y(x_1)$. We do that by *discretization* of the differential equation. To make the derivation general, assume that we know y_n (the approximation of $y(x_n)$) and want to compute y_{n+1}. At $x = x_n$ the differential equation is

$$y'(x_n) = f\big(x_n, y(x_n)\big) \ .$$

We first replace the derivative by a difference quotient,

$$y'(x_n) \simeq \frac{y(x_{n+1}) - y(x_n)}{h} \ ,$$

and the differential equation becomes

$$\frac{y(x_{n+1}) - y(x_n)}{h} \simeq f\big(x_n, y(x_n)\big) \ .$$

Next, we replace $y(x_n)$ by y_n; $y(x_{n+1})$ by y_{n+1}; and "$\simeq$" by "$=$", and after reordering we get

$$y_{n+1} = y_n + hf(x_n, y_n) \ .$$

This is a classical method for the numerical solution of initial value problems:

Definition 10.2.1. Euler's method:

$$y_0 = \alpha \ ,$$
$$y_{n+1} = y_n + hf(x_n, y_n), \quad n = 0, 1, \ldots, N-1 \ .$$

Euler's method is hardly ever used in practice, because there are more accurate and more efficient (but more complicated) methods. However, Euler's method is simple, and that is why we use it for introducing basic concepts in the numerical solution of initial value problems.

Example. In this chapter we shall illustrate different methods by numerically solving the initial value problem

$$y' = f(x, y) = xy, \qquad y(0) = 1 \ .$$

The problem has the analytical solution $y(x) = e^{0.5x^2}$, and we will use this to check the accuracy of the numerical solution.

Assume that we want to compute an approximation of $y(0.4)$. Euler's method applied to this problem is

$$\begin{aligned} y_0 &= 1, \\ y_{n+1} &= y_n + hx_n y_n, \quad n = 0, 1, \ldots, N-1 \ , \end{aligned}$$

where $Nh = 0.4$. We first let $h = 0.2$, corresponding to $N = 2$. Then $x_1 = 0.2$, $x_2 = 0.4$, and y_2 is the approximation of $y(0.4)$. We get

$$\begin{aligned} y_1 &= y_0 + hx_0 y_0 = 1 + 0.2 \cdot 0 \cdot 1 = 1 \ , \\ y_2 &= y_1 + hx_1 y_1 = 1 + 0.2 \cdot 0.2 \cdot 1 = 1.04 \ . \end{aligned}$$

We summarize the computation in a table, where we also give the analytical solution and the error $|y(x_n) - y_n|$.

n	x_n	y_n	$y(x_n)$	$\lvert y(x_n) - y_n \rvert$
0	0	1	1	0
1	0.2	1	1.0202	0.0202
2	0.4	1.04	1.0833	0.0433

Next, we halve the step length to $h = 0.1$ and compute a new approximation of $y(0.4)$. Note that this approximation is y_4.

n	x_n	y_n	$y(x_n)$	$\lvert y(x_n) - y_n \rvert$
0	0	1	1	0
1	0.1	1	1.0050	0.0050
2	0.2	1.0100	1.0202	0.0102
3	0.3	1.0302	1.0460	0.0158
4	0.4	1.0611	1.0833	0.0222

By comparing the two tables we see that the error was halved when the step length was halved. This indicates that the global truncation error (see the next section) is proportional to h. ∎

Example. The following MATLAB function implements Euler's method for a scalar differential equation. MATLAB does not allow index 0. Therefore x_n and y_n are stored in `x(n+1)` and `y(n+1)`, respectively.

```
function  [x, y] = eulers(f, ab, y0, N)
% Compute approximation of the solution of the initial value
```

```
% problem  y' = f(x,y), y(a) = y0  on the interval ab = [a,b].
% Euler's method with step length  h = (b-a)/N .
a = ab(1);  b = ab(2); h = (b - a)/N;
x = linspace(a,b,N+1);              % grid
y = zeros(size(x));  y(1) = y0;     % initialize  y
for  n = 1 : N
  y(n+1) = y(n) + h*feval(f, x(n),y(n));
end
```

The function $f(x, y) = xy$ is defined by

```
function  f = odef1(x,y)
f = x*y;
```

To compute the approximate solutions for $0 \leq x \leq 1$ with step lengths $h_1 = 1/5$ and $h_2 = 1/10$ we use the commands

```
>> [x1,y1] = eulers(@odef1, [0 1], 1, 5);
>> [x2,y2] = eulers(@odef1, [0 1], 1, 10);
```

The results are shown in Figure 10.4 together with the analytical solution.

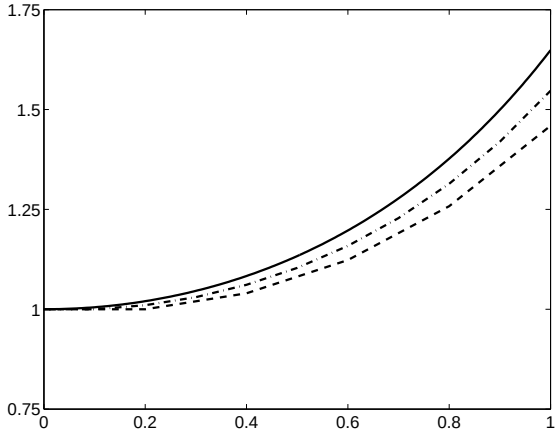


Figure 10.4. *Euler's method with step length* $h = 0.2$ *(dashed line),* $h = 0.1$ *(dash-dotted line), and analytical solution (solid line).*

■

Euler's method has a geometric interpretation: We start at the point (x_0, y_0) and approximate the solution curve by the tangent at this point. The slope of the tangent is computed from the differential equation, $y'(x_0) = f(x_0, y_0)$. We follow the tangent until we reach $x = x_1 = x_0 + h$; the corresponding y-value is y_1. Through the point (x_1, y_1) there passes a solution curve (which, however, does not correspond to the given initial

value). Similarly, we approximate this curve by its tangent at (x_1, y_1), and follow the tangent until we reach $x = x_2 = x_1 + h$, etc. See Figure 10.5.

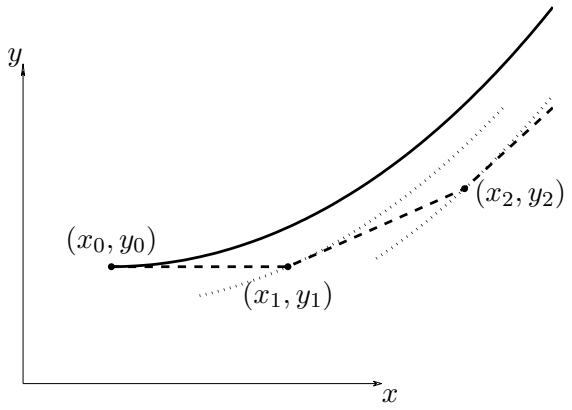


Figure 10.5. *Euler's method corresponds to a polygonal curve.*

10.3. Local and Global Error

The error sources in the numerical solution of differential equations are truncation error and rounding error. First, we ignore the rounding errors, and introduce the concepts of local and global truncation error.

Definition 10.3.1. The *local truncation error* at x_{n+1} is the difference between the computed value y_{n+1} and the value at x_{n+1} on the solution curve that passes through the point (x_n, y_n).

Figure 10.5 shows that the local truncation error in Euler's method is the deviation after each step between a solution curve and its tangent. We will show that if the solution to the differential equation is twice continuously differentiable, then the local truncation error is[2)] $O(h^2)$.

Let $\hat{y}(x)$ denote the solution curve that passes through the point (x_n, y_n). It is the solution of the initial value problem

$$\hat{y}' = f(x, \hat{y}), \qquad \hat{y}(x_n) = y_n \ .$$

[2)] The "big O" concept is discussed at the end of Section 1.1.

The Taylor expansion around $x = x_n$ is

$$\begin{aligned}\hat{y}(x_n + h) &= \hat{y}(x_n) + h\hat{y}'(x_n) + \tfrac{1}{2}h^2\hat{y}''(\xi) \\ &= y_n + hf(x_n, y_n) + \tfrac{1}{2}h^2\hat{y}''(\xi) \ ,\end{aligned}$$

where $x_n < \xi < x_n + h$. Thus, the local truncation error is

$$\hat{y}(x_n + h) - y_{n+1} = \tfrac{1}{2}h^2\hat{y}''(\xi) = O(h^2) \ .$$

Definition 10.3.2. The *global truncation error* at x_{n+1} is the difference

$$y(x_{n+1}) - y_{n+1} \ ,$$

where $y(x)$ is the solution of the given initial value problem.

In Figure 10.5 the global truncation error is the distance between a corner on the polygonal curve and the corresponding point on the solid curve.

Proposition 10.3.3. The global truncation error in Euler's method is $O(h)$.

Proof. The example on page 314 indicated this behaviour. We shall prove that it is generally true. We let $\epsilon_n = y(x_n) - y_n$ denote the global truncation error at x_n. A Taylor expansion around $x = x_n$ gives

$$\begin{aligned}y(x_{n+1}) = y(x_n + h) &= y(x_n) + hy'(x_n) + \tfrac{1}{2}h^2y''(\xi) \\ &= y(x_n) + hf\big(x_n, y(x_n)\big) + \tfrac{1}{2}h^2y''(\xi) \ .\end{aligned}$$

From this equation we subtract $y_{n+1} = y_n + hf(x_n, y_n)$ and get

$$\epsilon_{n+1} = \epsilon_n + h\big(f(x_n, y(x_n)) - f(x_n, y_n)\big) + \tfrac{1}{2}h^2y''(\xi) \ .$$

We assume that f satisfies a Lipschitz condition with Lipschitz constant L, and that there exists a constant $M < \infty$ such that $|y''(x)| \le M$ for all x in the interval, where we want to determine the solution. Then we can make the estimate

$$|\epsilon_{n+1}| \le (1 + hL)|\epsilon_n| + \tfrac{1}{2}h^2M \ . \tag{10.3.1}$$

A simple induction shows that

$$|\epsilon_n| \le (1+hL)^n|\epsilon_0| + \tfrac{1}{2}h^2M\sum_{k=0}^{n-1}(1+hL)^k \ .$$

Now, $\epsilon_0 = y(x_0) - y_0 = 0$, and by applying the formula for the sum of a geometrical progression, and the relation $1+x \le e^x$, we get

$$\begin{aligned} |\epsilon_n| &\le \tfrac{1}{2}h^2M\,\frac{(1+hL)^n - 1}{hL} \\ &\le \frac{hM}{2L}\left(e^{nhL} - 1\right) = \frac{hM}{2L}\left(e^{L(x_n - x_0)} - 1\right) \ . \end{aligned} \tag{10.3.2}$$

This shows that the global truncation error is $O(h)$. □

In general, if the local truncation error of a numerical method is $O(h^{p+1})$, then the global error is $O(h^p)$.

The estimate (10.3.2) of the global truncation error cannot be used for practical error estimation, since in most cases it is much too pessimistic. Also, in practice, one does not know L and M. The estimate has a certain theoretical interest, however. For instance, it shows that if one keeps $x_n = x_0 + nh$ fixed and lets h tend to zero, then

$$\lim_{h\to 0} \epsilon_n = 0 \ .$$

This means that Euler's method is convergent in the sense that the error in the approximate solution for a fixed x-value tends to zero as the step length tends to zero.

The discussion so far has neglected rounding errors. The *computed solution* will be affected by these, however, and instead of the approximations $\{y_n\}_{n=0}^N$ we get $\{\overline{y}_n\}_{n=0}^N$, defined by

$$\begin{aligned} \overline{y}_0 &= fl[\alpha] \ , \\ \overline{y}_{n+1} &= fl[\overline{y}_n + hf(x_n, \overline{y}_n)], \quad n = 0, 1, \ldots, N-1 \ . \end{aligned}$$

We can write

$$fl[\overline{y}_n + hf(x_n, \overline{y}_n)] = \overline{y}_n + hf(x_n, \overline{y}_n) + \mu\rho_n \ ,$$

where μ is the unit roundoff, and the local rounding error $\mu\rho_n$ has contributions from the evaluation of $f(x_n, \overline{y}_n)$, from the multiplication by h, and from the addition with $\overline{y}_n$.

Let $\delta_n = \overline{y}_n - y(x_n)$ denote the global error at x_n. Similar to (10.3.1) we can derive the estimate

$$|\delta_{n+1}| \le (1+hL)|\delta_n| + \tfrac{1}{2}h^2M + \mu R \ ,$$

where R is an upper bound on $|\rho_n|$. If the initial value α can be represented without rounding errors, then $\delta_0 = 0$, and similar to (10.3.2) we get the following estimate of the total global error,

$$|y(x_n) - \overline{y}_n| \le \left(\frac{hM}{2L} + \frac{\mu R}{hL}\right)\left(e^{L(x_n-x_0)} - 1\right) \ . \tag{10.3.3}$$

In practice, the number μR is small, and for large values of the step length h we can neglect the error contribution from rounding errors. As h tends to zero, however, this term gets increasing influence. We shall return to this aspect in the last example of the next section.

10.4. Runge-Kutta Methods

Since the global truncation error in Euler's method is $O(h)$, it is often necessary to use a very small step length (and therefore a very large number of steps) to get the desired accuracy in the approximate solution. There are several ways to derive more accurate methods. One can eg use a central difference to approximate the derivative,

$$y'(x) \simeq \frac{y(x+h) - y(x-h)}{2h} \ .$$

If we use this approximation in the equation $y' = f(x,y)$ we get the *midpoint method*

$$y_{n+1} = y_{n-1} + 2hf(x_n, y_n) \ , \tag{10.4.1}$$

which can be shown to have global error $O(h^2)$. We return to this method in Section 10.6.

In this section we shall use the geometrical interpretation of Euler's method to derive a more accurate method. In Figure 10.5 we see that the error in Euler's method is large because we go along the direction of the tangent at the point (x_n, y_n) for a whole step, while the solution curve deviates from this direction by a considerable amount during the step. Let $(x_{n+1}, \widetilde{y}_{n+1})$ denote the point reached by Euler's method. If we use the average of the tangent directions at the points (x_n, y_n) and $(x_{n+1}, \widetilde{y}_{n+1})$, we should be able to make a correction for the bending of the curve, see Figure 10.6.

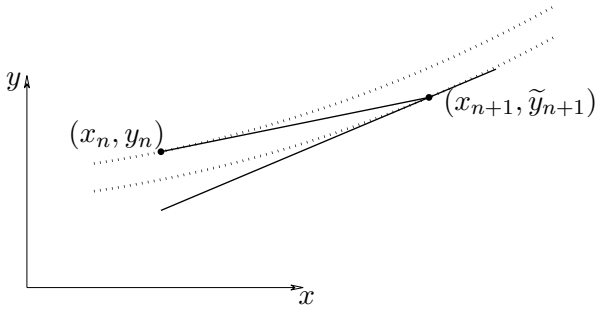


Figure 10.6. *Two tangent directions.*

The value $\widetilde{y}_{n+1}$ is given by $\widetilde{y}_{n+1} = y_n + hf(x_n, y_n)$, and the slopes of the two tangents are $f(x_n, y_n)$ and $f(x_{n+1}, \widetilde{y}_{n+1})$. Therefore, the method can be expressed in the form

$$\begin{aligned} k_1 &= f(x_n, y_n) \ , \\ k_2 &= f(x_n + h, y_n + hk_1) \ , \\ y_{n+1} &= y_n + \frac{h}{2}(k_1 + k_2) \ . \end{aligned} \tag{10.4.2}$$

This method is called *Heun's method*. It belongs to a large class of methods called *Runge-Kutta methods*.

Example. We shall use Heun's method with step length $h = 0.2$ to find an approximation of $y(0.4)$, where $y(x)$ is the solution of the initial value problem $y' = xy$, $y(0) = 1$. We first get

$$\begin{aligned} k_1 &= x_0 y_0 = 0 \cdot 1 = 0 \ , \\ k_2 &= (x_0 + h)(y_0 + hk_1) = 0.2 \cdot 1 = 0.2 \ , \\ y_1 &= y_0 + \tfrac{1}{2}h(k_1 + k_2) = 1 + 0.1 \cdot (0 + 0.2) = 1.02 \ . \end{aligned}$$

The second step is

$$\begin{aligned} k_1 &= x_1 y_1 = 0.2 \cdot 1.02 = 0.204 \ , \\ k_2 &= (x_1 + h)(y_1 + hk_1) = 0.4 \cdot 1.0608 = 0.42432 \ , \\ y_2 &= y_1 + \tfrac{1}{2}h(k_1 + k_2) = 1.02 + 0.1 \cdot (0.204 + 0.42432) = 1.082832 \ . \end{aligned}$$

y_2 is an approximation of $y(0.4)$, and the truncation error is about $4.55 \cdot 10^{-4}$. In the example on page 314 we had the truncation error $2.2 \cdot 10^{-2}$ when we used Euler's method with step length $h = 0.1$, ie when we used the same number of evaluations of the function $f(x, y) = xy$. ∎

The global truncation error in Heun's method can be shown to be $O(h^2)$, and the price we pay for this higher accuracy is that $f(x, y)$ has to be evaluated twice per step. If we allow four evaluations of $f(x, y)$ per step, we can use the above geometric approach to derive a method with global truncation error $O(h^4)$.

The classical Runge-Kutta method:

$$\begin{aligned} k_1 &= f(x_n, y_n) \ , \\ k_2 &= f(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_1) \ , \\ k_3 &= f(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_2) \ , \\ k_4 &= f(x_n + h, y_n + hk_3) \ , \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \ . \end{aligned} \tag{10.4.3}$$

The method has global truncation error $O(h^4)$.

Example. The following MATLAB implementation of the classical Runge-Kutta method can be used both for scalar differential equations and for systems of differential equations.

The command `y = repmat(y0(:),1,N+1)` gives a matrix `y` with $N+1$ columns, each containing the initial vector `y0`. During computation the approximation of $y(x_n)$ overwrites the $(n+1)^{\text{st}}$ column. The function returns the transpose of the array `y` in order to conform with the standard MATLAB functions for solving initial value problems.

```
function  [x, y] = rk4(f, ab, y0, N)
% Compute approximation of the solution of the initial value
% problem  y' = f(x,y), y(a) = y0  on the interval ab = [a,b].
% Classical Runge-Kutta method with step length  h = (b-a)/N .
a = ab(1);  b = ab(2);
h = (b - a)/N;  hh = h/2;  % full and half step length
x = linspace(a,b,N+1);     % grid
y = repmat(y0(:),1,N+1);   % initialize  y
for  n = 1 : N
  k1 = feval(f, x(n), y(:,n));
  k2 = feval(f, x(n)+hh, y(:,n)+hh*k1);
  k3 = feval(f, x(n)+hh, y(:,n)+hh*k2);
  k4 = feval(f, x(n+1), y(:,n)+h*k3);
  y(:,n+1) = y(:,n) + h/6*(k1 + 2*k2 + 2*k3 + k4);
end
x = x';  y = y';           % return in standard format
```

With the function `odef1` defined in the example on page 314 the call

```
>> [x,y] = rk4(@odef1, [0 0.4], 1, 1)
```

gives $y(0.4) \simeq 1.0832853$. This result has an error approximately equal to $1.7 \cdot 10^{-6}$. It was obtained with one step, involving four evaluations of the function $f(x, y) = xy$. ∎

Example. We have solved the initial value problem $y' = xy$, $y(0) = 1$ by means of Euler's method, Heun's method and the classical Runge-Kutta method with step lengths $h = 0.4, 0.2, \ldots, 0.4/2^{20}$. The computation was made with unit roundoff $\mu = 2^{-53} \simeq 1.11 \cdot 10^{-16}$. Figure 10.7 shows the errors $\delta_N = |y(0.4) - y_N|$, where $N = 0.4/h$. As in Figure 6.4 we use double

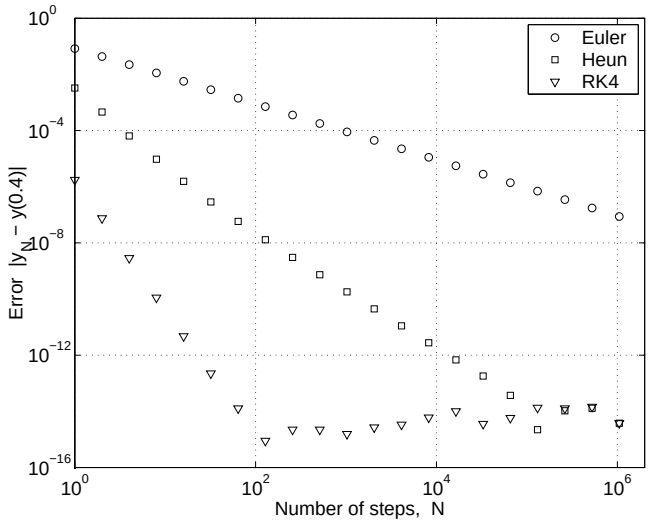


Figure 10.7. *Global error in the methods of Euler, Heun and Runge-Kutta.*

logarithmic scale, because we expect that

$$\delta_N \simeq A_1 \cdot h^p = A \cdot N^{-p} \ ,$$

where the positive constant A and the exponent p depend on the method. Then

$$\log \delta_N \simeq \log A - p \log N \ . \tag{10.4.4}$$

In other words: the points $(\log N, \log \delta_N)$ can be expected to lie close to a straight line with slope $-p$.

This is satisfied by the results in Figure 10.7. For the classical Runge-Kutta method the truncation error dominates for small values of N (large values

of h), and (10.4.4) is satisfied with $p = 4$. For large N-values the accumulated effect of rounding errors dominate, and the error increases corresponding to $p = -1$. This is in agreement with the analysis of rounding errors at the end of Section 10.3, (10.3.3). The smallest error with the Runge-Kutta method is obtained for $N = 128$, $|y(0.4) - y_{128}| \simeq 8.88 \cdot 10^{-16} = 8\mu$.

For Heun's method the estimate (10.4.4) is satisfied for "small" N-values with $p = 2$. The smallest error is obtained for $N \simeq 10^5$. Note that for larger values of N the errors in Heun's method and and the Runge-Kutta method are almost equal. This indicates that the accumulated effect of rounding errors is almost independent of the method.

The global truncation error in Euler's method decreases slower, corresponding to $p = 1$, and the total global error will reach a minimum for $N \simeq 10^8$, where the error is approximately 10^{-8}. ∎

10.5. An Implicit Method

Example. The system

$$y' = \begin{pmatrix} 0 & 1 \\ -M & -(M+1) \end{pmatrix} y$$

has the general solution

$$y_1(x) = a_1 e^{-x} + a_2 e^{-Mx} \ ,$$

where a_1 and a_2 are arbitrary constants. Both e^{-x} and e^{-Mx} tend to zero for $x \to \infty$, and if $M \gg 1$, then e^{-Mx} tends to zero much faster than e^{-x}. Such a system is said to be *stiff*. ∎

More generally, a problem is said to be *stiff*, if the solution contains both slow processes and very fast processes. The latter decay very fast as $x - x_0$ increases, but you need to use a very short step length to keep them from "blowing up" in the numerical solution. This is discussed further in the next section.

The methods discussed so far are very inefficient for a stiff problem. We will now derive a method that can be used for differential equations in general, and is particularly well suited for stiff problems.

If we integrate the left hand side of the differential equation

$$y' = f(x, y)$$

over the interval $[x_n, x_{n+1}]$, we get

$$\int_{x_n}^{x_{n+1}} y' \, dx = y(x_{n+1}) - y(x_n) \ .$$

The integral of the right hand side cannot be computed directly, since $y(x)$ is unknown. Using the trapezoidal rule from Section 7.2, we get

$$\int_{x_n}^{x_{n+1}} f(x,y) \, dx \simeq \tfrac{1}{2}h\big(f(x_n, y(x_n)) + f(x_{n+1}, y(x_{n+1}))\big) \ .$$

Now, we replace $y(x_n)$ and $y(x_{n+1})$ by the approximations y_n and y_{n+1} in both the left hand and the right hand side, and obtain the so-called trapezoidal method:

Definition 10.5.1. The trapezoidal method:

$$y_{n+1} = y_n + \tfrac{1}{2}h\big(f(x_n, y_n) + f(x_{n+1}, y_{n+1})\big) \ .$$

The global truncation error is $O(h^2)$.

The methods discussed earlier are *explicit*: the right hand side in the expression for y_{n+1} depends only on known quantities. The trapezoidal method is *implicit*: the unknown quantity y_{n+1} appears also on the right hand side. If f is a nonlinear function of y, then we must solve a nonlinear equation to get y_{n+1}.

Example. The equation $y' = xy$ is linear in y. The trapezoidal method takes the form

$$y_{n+1} = y_n + \tfrac{1}{2}h(x_n y_n + x_{n+1} y_{n+1}) \ ,$$

and (provided that $1 - \frac{1}{2}hx_{n+1}$ is nonzero) we can rewrite this relation to

$$y_{n+1} = \frac{1 + \frac{1}{2}hx_n}{1 - \frac{1}{2}hx_{n+1}} \, y_n \ .$$

The equation $y' = xy^2$ is nonlinear in y, and the trapezoidal method gives

$$y_{n+1} - \tfrac{1}{2}h\, x_{n+1}\, y_{n+1}^2 = y_n + \tfrac{1}{2}h\, x_n\, y_n^2 \ .$$

This is a quadratic equation in the unknown y_{n+1}. ∎

The value y_{n+1} can be computed by means of the fixed point iteration

$$\begin{aligned} y_{n+1}^{[0]} &= y_n + hf(x_n, y_n) \ , \\ y_{n+1}^{[k+1]} &= y_n + \frac{h}{2}\big(f(x_n, y_n) + f(x_{n+1}, y_{n+1}^{[k]})\big), \quad k = 0, 1, \ldots \end{aligned} \tag{10.5.1}$$

This way of using an implicit method is called a *predictor-corrector* procedure. Here, Euler's method is used as predictor: it gives the starting value for the iteration. Then one iterates a couple of times with the trapezoidal method, the corrector. The condition for the iteration to converge is that

$$\left| \frac{h}{2} \frac{\partial f}{\partial y} \right| < 1$$

in a neighbourhood of the point (x_{n+1}, y_{n+1}), cf Section 4.4.

Example. Assume that we use (10.5.1) to solve the initial value problem $y' = e^{-y}$, $y(0) = 1$. Does the corrector iteration converge if $h = 0.1$?

We see that the solution of the initial value problem always has positive derivative, ie it is increasing. Therefore, we cannot get negative y-values, and

$$\left| \frac{h}{2} \frac{\partial f}{\partial y} \right| = 0.05 e^{-y} < 1 \ .$$

Thus, with the chosen h-value the corrector iteration is guaranteed to converge for this problem. ∎

If $|\partial f / \partial y|$ is large, then one must use a very small step length for the corrector iteration to converge. In such cases it is much more efficient to use Newton-Raphson's method to solve the equation

$$y_{n+1} - \tfrac{1}{2} h f(x_{n+1}, y_{n+1}) - A_n = 0 \ ,$$

where $A_n = y_n + \frac{1}{2} h f(x_n, y_n)$ is a known quantity. The derivative that has to be computed in every step of the iteration, is (in the scalar case)

$$1 - \tfrac{1}{2}\, h \, \frac{\partial f}{\partial y} \ .$$

10.6. Stability

When a numerical method is used to find a sequence of approximations to the the solution of an initial value problem for an ordinary differential equation, it is necessary to analyze

a) how well the difference equation approximates the differential equation (*truncation error*),

b) what happens when the step length h tends to zero (*convergence*),

c) how sensitive the difference equation is to perturbations in the data (*stability*).

We already discussed the first two items, and in this section we study the stability of some numerical methods. This is done by investigating their performance when applied to the *test problem*

$$y' = \lambda y, \qquad y(0) = 1 \ ,$$

where λ is complex with negative real part. To simplify the discussion we first assume that λ is real (and negative). At the end of this section we shall discuss the general, complex case.

The exact solution to the test problem is $y(x) = e^{\lambda x}$, and since $\lambda < 0$, this is a decreasing function of x. It is reasonable to require that the numerical method gives a decreasing sequence. In that case we say that the method is *stable*.

Example. In (10.4.1) we presented the midpoint method

$$y_{n+1} = y_{n-1} + 2hf(x_n, y_n) \ .$$

We need both y_0 and y_1 before we can use this formula. If we apply this method to the initial value problem $y' = -2y$, $y(0) = 1$, using Euler's method to compute y_1, we get

$$\begin{aligned} y_0 &= 1 \ , \\ y_1 &= 1 - 2h \ , \\ y_{n+1} &= y_{n-1} - 4hy_n, \quad n = 1, 2, \dots \ . \end{aligned}$$

Figure 10.8 shows the results obtained with $h = 0.1$. We see that the numerical solution oscillates with increasing amplitude. For $x > 0.7$ it is no longer monotonically decreasing. This is an example of *instability*. Obviously, the midpoint method cannot be used to solve this problem. ∎

If we apply *Euler's method* to the test equation, we get

$$y_{n+1} = y_n + h\lambda y_n = (1 + h\lambda)y_n \ .$$

The solution is decreasing if the condition

$$|1 + h\lambda| < 1$$

is satisfied. This is equivalent to the condition

$$-2 < h\lambda < 0 \ ,$$

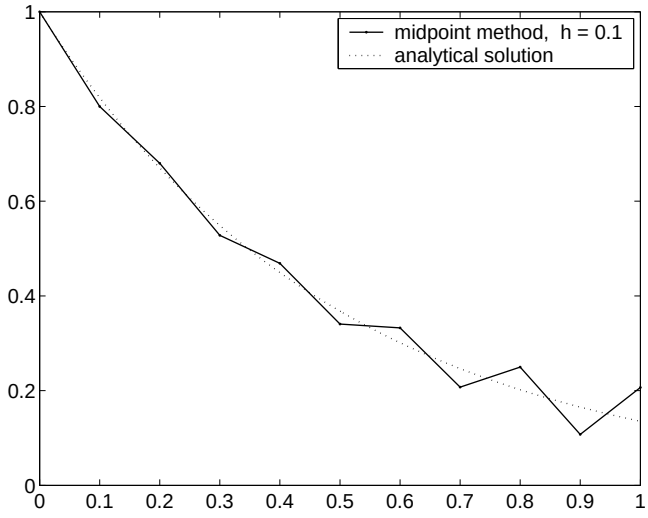


Figure 10.8. *Midpoint method applied to* $y' = -2y$, $y(0) = 1$.

and since $\lambda < 0$, we must choose the step length h so small, that

$$h < \frac{2}{|\lambda|} \ .$$

Thus, if $|\lambda|$ is very large, we must choose a very small step length h in Euler's method to get a decreasing solution.

Heun's method applied to the test equation gives

$$\begin{aligned} k_1 &= \lambda y_n \ , \\ k_2 &= \lambda(y_n + hk_1) = \lambda(1 + h\lambda)y_n \ , \\ y_{n+1} &= y_n + \tfrac{1}{2}h(k_1 + k_2) = (1 + h\lambda + \tfrac{1}{2}h^2\lambda^2)y_n \ . \end{aligned}$$

The solution is decreasing if

$$|1 + h\lambda + \tfrac{1}{2}h^2\lambda^2| < 1 \ ,$$

and we get the same condition as in Euler's method: $h < 2/|\lambda|$. A similar analysis shows that the classical Runge-Kutta method is stable if $h < 2.785/|\lambda|$.

The *trapezoidal method* applied to the test equation gives

$$y_{n+1} = y_n + \tfrac{1}{2}h\lambda(y_n + y_{n+1}) \ ,$$

or, equivalently,

$$y_{n+1} = \frac{1 + \frac{1}{2}h\lambda}{1 - \frac{1}{2}h\lambda}\, y_n \ .$$

It is easy to see that

$$\left| \frac{1 + \frac{1}{2}h\lambda}{1 - \frac{1}{2}h\lambda} \right| < 1$$

for $\lambda < 0$ and any $h > 0$. Thus, the numerical solution is decreasing for any step length. In other words, the trapezoidal method is unconditionally stable.

It is this stability property that make the trapezoidal method so useful for *stiff* differential equations (or stiff systems of differential equations). Such equations are characterized by the existence of some solution components that decrease very fast, and other components that decrease quite slowly. The former correspond to negative λ-values of large magnitude, and the latter correspond to negative λ-values of relatively small magnitude.

Consider the initial value problem

$$u'' + 101u' + 100u = 0, \qquad u(0) = 1.1, \quad u'(0) = -11 \ .$$

This can be written as a system

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -100 & -101 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \quad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 1.1 \\ -11 \end{pmatrix} \ . \qquad (10.6.1)$$

The matrix of this system has the eigenvalues -100 and -1, which correspond to to $\lambda = -100$ and $\lambda = -1$ in the test equation. The solution is

$$u(x) = y_1(x) = 0.1e^{-100x} + e^{-x} \ .$$

The first term in the solution decays very fast: at $x = 0.1$ it is about $5 \cdot 10^{-6}$, while the second term is about 0.9. When you solve such a system numerically over a large interval, you would like to take large steps as soon as the rapidly decaying components have disappeared. Because of their stability properties this cannot be done with Euler's method or other explicit methods like Heun's method or the Runge-Kutta method, but with the trapezoidal method it is possible to do so.

Example. In (10.6.1) we have a system of differential equations $y' = Ay$, where A is a square matrix. The equation is linear in y, and the trapezoidal method takes the form

$$(I - \tfrac{1}{2}hA)y_{n+1} = (I + \tfrac{1}{2}hA)y_n \ .$$

In each step we have to solve a linear system of equations with the matrix $I - \frac{1}{2}hA$. If the step length h is constant, then the matrix is the same in every step, and the most efficient algorithm is the use the LU factorization of the matrix, cf Section 8.6. This is implemented in the following code.

```
function  [x, y] = trapmeth(A, ab, y0, N)
% Compute approximation of the solution of the initial value
% problem  y' = Ay, y(a) = y0  on the interval ab = [a,b].
% A is a square matrix.
% Trapezoidal method with step length  h = (b-a)/N .
a = ab(1);  b = ab(2); h = (b - a)/N;
x = linspace(a,b,N+1);     % grid
y = repmat(y0(:), 1,N+1);  % initialize  y
I = eye(size(A));          % Unit matrix
[L,U] = lu(I - h/2*A);     % LU factorization
B = I + h/2*A;             % Constant matrix on right hand side
for  n = 1 : N
  y(:,n+1) = U \ (L \ (B*y(:,n)));
end
x = x';  y = y';          % return in standard format
```

■

Example. We solve the stiff system (10.6.1) by Euler's method and the trapezoidal method. In the interval $[0, 0.1]$ we use the step length $h = 0.005$. At this point the fast component has almost vanished, and we take $h = 0.1$ in the trapezoidal method on the interval $[0.1, 1]$.

```
>> A = [0 1;-100 -101];
>> [x1,y1] = trapmeth(A, [0 0.1], [1.1; -11], 20);
>> [x2,y2] = trapmeth(A, [0.1 1], y1(end,:), 9);
```

In Euler's method we still have to use a step length that satisfies $h < 2/|\lambda| = 0.02$ in order to get a decreasing solution. To demonstrate that Euler's method is not stable for longer steps, we use $h = 0.025$ on the interval $[0.1, 1]$. The results for $u(x) = y_1(x)$ are shown in Figure 10.9. ■

The *midpoint method* (10.4.1)

$$y_{n+1} = y_{n-1} + 2hf(x_n, y_n) \ .$$

is an example from a large class of methods called *multistep methods* because they involve values at more than two consecutive points. Applied to the test problem the midpoint method takes the form

$$y_{n+1} = y_{n-1} + 2h\lambda y_n, \quad n = 1, 2, \dots \ . \tag{10.6.2}$$

The simple analysis that we used with the one-step methods above is not applicable to a multistep method, but it can be shown that the values

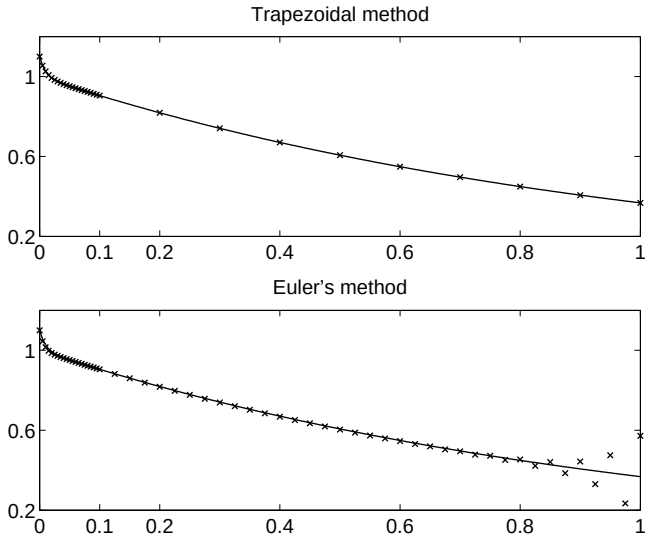


Figure 10.9. *Trapezoidal method (top) and Euler's method applied to the stiff system* (10.6.1)*. Computed approximations are marked by* $\times$*.*

computed by (10.6.2) can be expressed as

$$y_n = Ar_1^n + Br_2^n \ , \tag{10.6.3}$$

where r_1 and r_2 are the roots of the quadratic equation

$$r^2 - 2h\lambda r - 1 = 0 \ ,$$

and A and B are determined by the values of y_0 and y_1. The roots are

$$r_1 = h\lambda + \sqrt{h^2\lambda^2 + 1}, \qquad r_2 = -1/r_2 \ .$$

If $h\lambda < 0$, then $0 < r_1 < 1$ and $r_2 < -1$. This means that $|Ar_1^n| \to 0$ for $n \to \infty$, and (provided that $B \neq 0$) the contribution Br_2^n oscillates and $|Br_2^n| \to \infty$ for $n \to \infty$. This explains the behaviour shown in Figure 10.8 on page 327. For $\lambda < 0$ this instability occurs for all positive values of h. We say that the midpoint method is unconditionally unstable.

There are other multistep methods with better stability properties. Often they are used in pairs, so that an explicit method is used as predictor and an implicit method as corrector in a scheme like (10.5.1). For a good introduction to multistep methods see the book by Lambert or Shampine given in the references at the end of the chapter.

We finally consider the test problem

$$y' = \lambda y, \qquad y(0) = 1 \ ,$$

in the general case, where λ is complex with negative real part. The exact solution is $y(x) = e^{\lambda x}$, and if $\lambda = -\kappa + i\nu$ (i denotes the imaginary unit, and $\kappa > 0$), then

$$e^{\lambda x} = e^{-\kappa x}\big(\cos(\nu x) + i\sin(\nu x)\big) \ .$$

Both the real and the imaginary part of the solution are damped oscillations, and

$$|e^{\lambda x}| = e^{-\kappa x}$$

is a decaying exponential. The requirement that the numerical method gives a decreasing sequence leads to the following conditions,

$$\begin{aligned}
\text{Euler's method:} \quad & |1 + h\lambda| < 1 \ , \\
\text{Heun's method:} \quad & |1 + h\lambda + \tfrac{1}{2}h^2\lambda^2| < 1 \ , \\
\text{Classical R-K:} \quad & |1 + h\lambda + \tfrac{1}{2}h^2\lambda^2 + \tfrac{1}{6}h^3\lambda^3 + \tfrac{1}{24}h^4\lambda^4| < 1 \ , \\
\text{Trapezoidal method:} \quad & \left|\frac{1 + \frac{1}{2}h\lambda}{1 - \frac{1}{2}h\lambda}\right| < 1 \ .
\end{aligned}$$

For Euler's method the stability condition is satisfied when the number $h\lambda$ is inside the circle in the complex plane with centre $(-1, 0)$ and radius 1. This is shown in Figure 10.10 together with the stability regions for the other three methods.

Example. Consider the initial value problem

$$u'' + u' + 4.25u = 0, \qquad u(0) = 1, \quad u'(0) = -0.3 \ .$$

This can be written as a system

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -4.25 & -1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, \qquad \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} 1 \\ -0.5 \end{pmatrix} \ . \tag{10.6.4}$$

The matrix of this system has the eigenvalues $-0.5 \pm 2i$ which correspond to to λ in the test equation. The solution is

$$u(x) = y_1(x) = e^{-0.5x}\cos(2x) \ .$$

We have used Euler's method with step length $h = 1/3$ and $h = 0.025$ to approximate the solution for $0 \le x \le 5$. The result is shown in Figure 10.11. The exact solution is a damped oscillation, and so is the numerical solution found with $h = 0.025$. For $h = 1/3$ we get an oscillating numerical solution

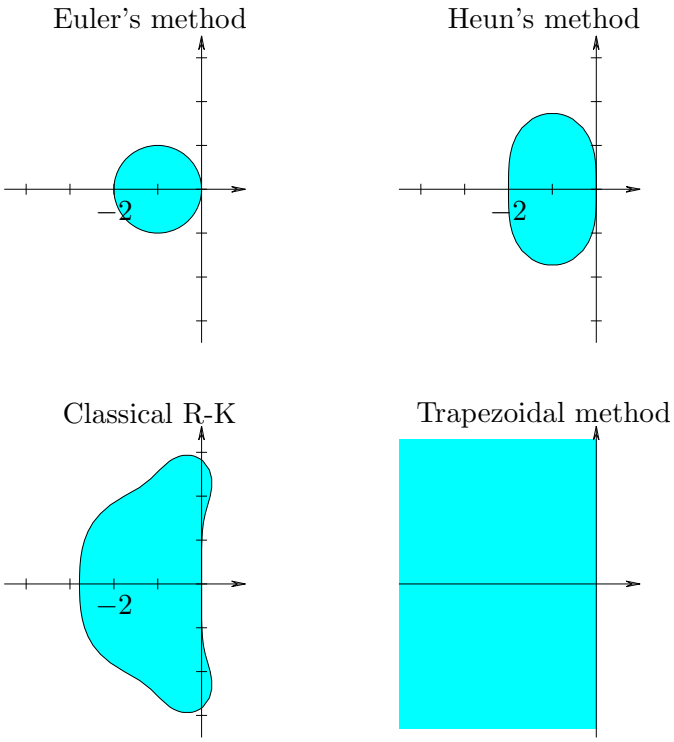


Figure 10.10. *Stability regions. The method is stable if $h\lambda$ is in the shaded part of the complex plane.*

with increasing amplitude. This is because the stability condition is not satisfied:

$$|1 + h\lambda| = |1 + \tfrac{1}{3}(-0.5 + 2i)| = 1.0672 > 1 \ .$$

For $h = 0.025$ we get $|1 + h\lambda| = 0.9888 < 1$. ∎

Example. The MATLAB function `ode45` is an implementation of an explicit Runge-Kutta method, and `ode15s` is an implicit method. Both functions use adaptive step length control (see the next section) ie they attempt to use as large steps as is possible with respect to stability and accuracy. We use the two functions (with the same, default accuracy parameters) to solve the stiff system

$$u'' + 10\,001u' + 10\,000u = 0, \quad u(0) = 1.001, \quad u'(0) = -11 \ . \tag{10.6.5}$$

(What are the eigenvalues of the corresponding system, and what is the

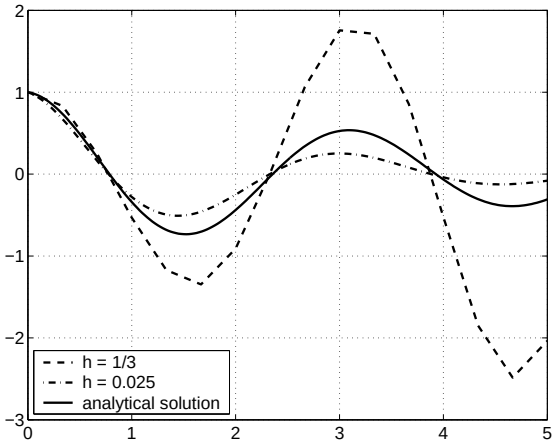


Figure 10.11. *Euler's method applied to* (10.6.4).

analytic solution?). The explicit method `ode45` requires 19 129 function evaluations, while the implicit method `ode15s` only needs 99. The results are shown in Figure 10.12.

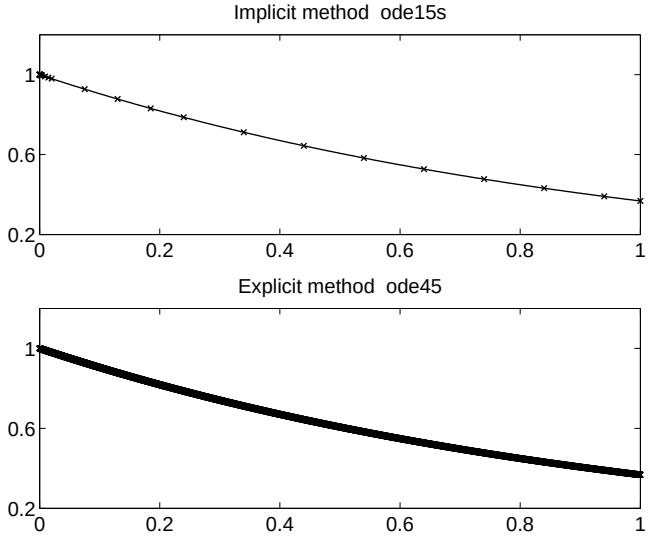


Figure 10.12. *Solution of the stiff system* (10.6.5) *by* `ode15s` *(top) and* `ode45` *(bottom; the markings are so close that they cannot be distinguished).*

■

10.7. Adaptive Step Length Control

When one solves an initial value problem for an ordinary differential equation, it is desirable to be able to vary the step length depending on how fast the solution changes. Since the solution is not known beforehand, the change of step length must be based on an estimate of the local truncation error. We shall describe one way of doing this.

Suppose that we use a Runge-Kutta method with local truncation error $O(h^5)$. Then an approximation y_{n+1} satisfies

$$y_{n+1} = \widehat{y}(x_{n+1}) + \psi(x_{n+1})h^5 + O(h^6) \ ,$$

where $\widehat{y}(x)$ is the solution curve through the point (x_n, y_n), and the dominating term $\psi(x_{n+1})$ is unknown. Suppose that we also compute another approximation $\widetilde{y}_{n+1}$ with local truncation error $O(h^6)$,

$$\widetilde{y}_{n+1} = \widehat{y}(x_{n+1}) + O(h^6) \ .$$

Then the local truncation error can be estimated by the difference

$$d_{n+1} = y_{n+1} - \widetilde{y}_{n+1} = \psi(x_{n+1})h^5 + O(h^6) \ .$$

Assume that we are willing to accept a local truncation error τ. If the estimated error is larger than this tolerance, we have to recompute the step with a smaller step length. Otherwise, we accept the step, and if $|d_{n+1}|$ is considerably smaller than τ, then it indicates that we can increase the step length.

More precisely, let h denote the current value of the step length. Compute y_{n+1}, $\widetilde{y}_{n+1}$ and d_{n+1}. We have

$$|d_{n+1}| \simeq |\psi(x_{n+1})|h^5 \ ,$$

and the new step length h_{new} should satisfy

$$|\psi(x_{n+1})|h_{\text{new}}^5 \simeq \tau \ .$$

This leads to $h_{\text{new}} = \gamma h$, where

$$\gamma = \min\left\{ 0.8 \left(\frac{\tau}{|d_{n+1}|} \right)^{1/5} , \ 5 \right\} \ .$$

The factor 0.8 and the upper bound 5 are used for reasons of caution.

If $|d_{n+1}| \leq \tau$, then y_{n+1} is sufficiently accurate, and since $\widetilde{y}_{n+1}$ is even better, we accept $(x_n+h, \widetilde{y}_{n+1})$ as the next approximate solution point and h is changed to γh. Otherwise, the computation from (x_n, y_n) is repeated with the reduced step length γh.

In order to be efficient, the two methods must be constructed so that one uses the same evaluations of the function $f(x, y)$ to compute both y_{n+1} and $\widetilde{y}_{n+1}$. This is satisfied by the following *Runge-Kutta-Fehlberg* method.

$$
\begin{aligned}
k_1 &= f(x_n, y_n) \ , \\
k_r &= f\big(x_n{+}\alpha_r h, y_n + h\sum_{j=1}^{r-1} \beta_{rj} k_j\big), \quad r = 2,3,4,5,6 \ , \\
y_{n+1} &= y_n + h\sum_{j=1}^{6} w_r k_r \ , \\
\widetilde{y}_{n+1} &= y_n + h\sum_{j=1}^{6} \widetilde{w}_r k_r \ , \\
d_{n+1} &= h\sum_{j=1}^{6} (w_r - \widetilde{w}_r) k_r \ .
\end{aligned}
$$

The values of the $\{\alpha_r\}$, $\{\beta_{rj}\}$, $\{w_r\}$ and $\{\widetilde{w}_r\}$ can be found in the next example.

Example. The Runge-Kutta-Fehlberg method can be implemented as follows.

```
function  [x,y,neval] = adaptrk45(f, ab, y0, h0, tol)
% Runge-Kutta with adaptive step length control
% Input
% f     handle to function that defines the rhs in  y' = f(x,y)
% ab    range of integration is [a, b],  a=ab(1), b=ab(2)
% y0    vector of initial values
% h0    initial step length
% tol   tolerance for step length control
% Output
% x     vector of x-values, where the solution is computed
% y     matrix with approximations of the solution
% neval number of function evaluations

% Runge-Kutta-Fehlberg45 constants.  Alpha in A, beta in B
A = [0 1/4 3/8 12/13 1 1/2];
B = [0 0 0 0 0 0
     1/4 0 0 0 0 0
     3/32 9/32 0 0 0 0
     1932/2197 -7200/2197 7296/2197 0 0 0
     439/216 -8 3680/513 -845/4104 0 0
     -8/27 2 -3544/2565 1859/4104 -11/40 0]';
```

```
% Weights
w4 = [25/216 0 1408/2565 2197/4104 -1/5 0]';
w5 = [16/135 0 6656/12825 28561/56430 -9/50 2/55]';
dw = w4 - w5;
% Initialize
a = ab(1);  b = ab(2); m = length(y0);
x = a;  y = y0(:);  neval = 0;              % initialize output
n = 0;  xn = a;  yn = y0;  h = min(h0, b-a);
K = zeros(m,6);                          % for storing k1,...,k6
while  xn < b
  xn1 = min(xn + h, b);            % possible adjustment at end
  h = xn1 - xn;
  for  j = 1 : 6
    K(:,j) = feval(f, xn+A(j)*h, yn+h*K*B(:,j));
  end
  neval = neval + 6;
  y5 = yn + h*K*w5;                        % RKF5 approximation
  d = h*norm(K*dw);                           % estimated error
  if  d <= tol                                % accept the step
    n = n+1;  xn = xn1;  yn = y5;
    x(n+1) = xn;   y(:,n+1) = yn;              % save in output
  end
  h = h * min(0.8*(tol/d)^0.2, 5);         % Update step length
end
x = x';  y = y';                    % return in standard format
```

■

Adaptive step length control is especially useful when the solution varies fast in parts of the interval $[a, b]$, and more slowly in other parts. This is the case with the so-called *Brusselator equation*, which models a certain chemical reaction.

$$y_1' = 1 + y_1^2 y_2 - 4y_1 \ ,$$
$$y_2' = 3y_1 - y_1^2 y_2 \ ,$$

with initial values $y_1(0) = 1.5$, $y_2(0) = 3$.

Example. The Brusselator equation is implemented in

```
function  f = odef2(x,y)
f = [1 + y(1)*(y(1)*y(2) - 4)
     y(1)*(3 - y(1)*y(2))];
```

Figure 10.13 shows the results obtained by the call

```
>> [x,y,neval] = ark45(@odef2, [0 20], [1.5;3], 0.1, 1e-4);
```

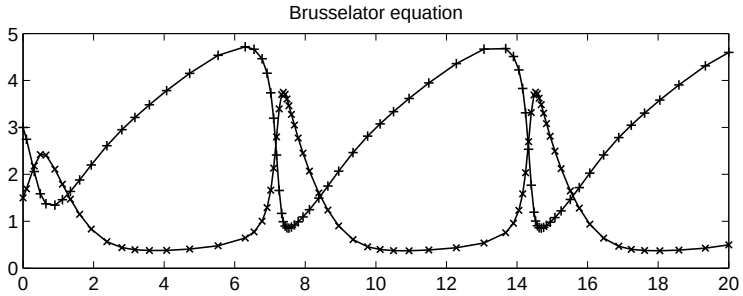


Figure 10.13. *Runge-Kutta with adaptive step length control. Computed points are marked by* $\times$ *for* y_1 *and* $+$ *for* y_2.

The number of evaluations of $f(x, y)$ is `neval = 516`. The step length varies between 0.0466 and 0.8055.

The MATLAB function `ode45` is a more advanced implementation, based on a similar set of Runge-Kutta methods. The commands

```
>> opts = odeset('NormControl','on',...
        'AbsTol',1e-4, 'RelTol',1e-8, 'Stats','on');
>> [xx,yy] = ode45(@odef2, [0 20], [1.5;3], opts);
```

give a result similar to Figure 10.13, and uses 505 function evaluations. ∎

10.8. Boundary Value Problems

Example. Consider a thin rod of length 1 with variable heat conduction properties. Assume that the rod is insulated along its length, and that the endpoint temperatures are kept at different, constant temperatures, α and β. Let $y(x)$ denote the temperature at x. It satisfies the differential equation

$$\frac{d}{dx}\big(\kappa(x)\,\frac{dy}{dx}\big) = 0\ ,$$

where $\kappa(x)$ is the heat conduction coefficient of the rod. The given temperatures at the ends lead to the boundary conditions

$$y(0) = \alpha, \quad y(1) = \beta\ .$$

If, eg, the rod is made of two different materials,

$$\kappa(x) = \begin{cases} 1 & \text{for } 0 \le x < 0.4\ , \\ 2 & \text{for } 0.4 < x \le 1\ , \end{cases}$$

and the endpoint temperatures are $\alpha = 10$, $\beta = 20$, then the solution is the broken line shown in the figure.

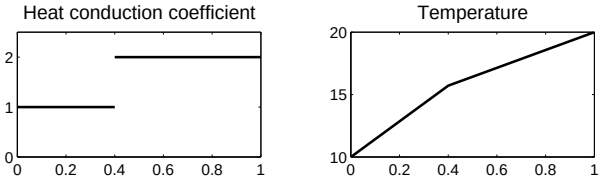


Figure 10.14. *Temperature in insulated rod.*

■

Many physical processes can be modelled by a second order differential equation with extra conditions given in two points, a and b. The solution is required in the interval $[a, b]$. Often, such a problem can be written in the form

$$y'' = g(x, y, y'), \qquad y(a) = \alpha, \quad y(b) = \beta \ ,$$

where g is a given function and α and β are given numbers. This is called a *boundary value problem*. Remember that two extra conditions are needed to define a solution of a second order differential equation. If they are given at the same point, then we have an initial value problem, and if they are given at two points, then we have a boundary value problem. The extra conditions are called *boundary values*.

Example. If the rod from the previous example contains a heat source, eg an electrical resistance or a radioactive isotope, then one gets an inhomogeneous equation

$$\frac{d}{dx}\big(\kappa(x)\,\frac{dy}{dx}\big) = f(x) \ ,$$

where $f(x)$ describes the heat production as a function of x. If κ and f are constant, then the boundary value problem can be solved analytically (eg if κ is constant and $f(x) = 0$, then the solution is a straight line). For problems with non-constant coefficients it is usually necessary to find an approximate solution via a discretization of the problem. ■

In the following sections we discuss three different methods for approximate solution of boundary value problems: a (finite) difference method, a finite element method, and the shooting method. For simplicity we shall present the theory using a special case, eg the linear equation $y'' - q(x)y = f(x)$, where q and f are given functions of x.

10.9. A Difference Method

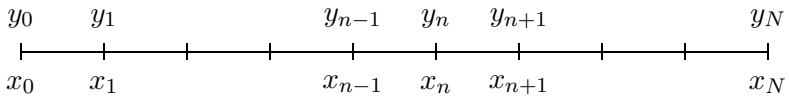
Difference methods are based on a division of the interval $[a, b]$ into subintervals, which we assume have equal length

$$h = \frac{b-a}{N} \ .$$

The points

$$x_n = a + nh, \quad n = 0, 1, 2, \ldots, N$$

are said to form a *grid*. We seek the values $\{y_n\}$ as approximations of $\{y(x_n)\}$.



Note that we already know y_0 and y_N, since they are given by the boundary values.

Consider the special case when the equation is

$$y'' - q(x)y = f(x), \qquad q(x) \geq 0 \ .$$

According to chapter 6 we can approximate the second derivative at the point $x = x_n$ as follows,

$$y''(x_n) \simeq \frac{y(x_{n-1}) - 2y(x_n) + y(x_{n+1})}{h^2} \ .$$

This can be done at the interior points, $n = 1, 2, \ldots, N-1$. We insert this approximation in the differential equation,

$$\frac{y(x_{n-1}) - 2y(x_n) + y(x_{n+1})}{h^2} - q(x_n)y(x_n) \simeq f(x_n), \quad n = 1, 2, \ldots, N-1 \ .$$

The discretized equation is obtained when we replace $y(x_k)$ by y_k and "$\simeq$" by equality:

$$\frac{y_{n-1} - 2y_n + y_{n+1}}{h^2} - q(x_n)y_n = f(x_n), \quad n = 1, 2, \ldots, N-1 \ ,$$

or, equivalently

$$y_{n-1} - (2 + h^2 q_n)y_n + y_{n+1} = h^2 f_n, \quad n = 1, 2, \ldots, N-1 \ .$$

Here we have introduced the notation $q_n = q(x_n)$, $f_n = f(x_n)$.

The known boundary value y_0 is included in the first equation,

$$y_0 - (2 + h^2 q_1)y_1 + y_2 = h^2 f_1 \ .$$

We move the known value $y_0 = \alpha$ to the right hand side and get

$$-(2 + h^2 q_1)y_1 + y_2 = h^2 f_1 - \alpha \ .$$

Similarly, in the last equation we introduce the known boundary value $y_N = \beta$ and get

$$y_{N-2} - (2 + h^2 q_{N-1})y_{N-1} = h^2 f_{N-1} - \beta \ .$$

Thus, we have derived a linear system of equations in the unknowns $y_1, \ldots, y_{n-1}$. We can write the system in the form

$$Ay = b \ ,$$

where

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-2} \\ y_{N-1} \end{pmatrix}, \qquad b = \begin{pmatrix} h^2 f_1 - \alpha \\ h^2 f_2 \\ \vdots \\ h^2 f_{N-2} \\ h^2 f_{N-1} - \beta \end{pmatrix} \ ,$$

and A is the tridiagonal matrix

$$A = \begin{pmatrix} -(2+h^2 q_1) & 1 & & & \\ 1 & -(2+h^2 q_2) & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -(2+h^2 q_{N-2}) & 1 \\ & & & 1 & -(2+h^2 q_{N-1}) \end{pmatrix} .$$

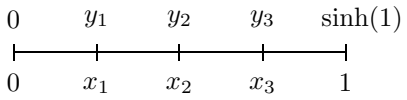
This method of discretizing the boundary value problem is called a *difference method*, because derivatives are replaced by differences.

We have made the assumption that $q(x) \geq 0$. This ensures that the matrix A is diagonally dominant, cf Section 8.4. This implies that the matrix is nonsingular, and the linear system of equations has a unique solution for any right hand side b. The system can be solved by Gaussian elimination without pivoting, and the operation count is $O(N)$, cf the algorithm on page 220.

Example. The boundary value problem

$$y'' - y = 0, \qquad y(0) = 0, \quad y(1) = \sinh(1)$$

has the analytic solution $y(x) = \sinh(x)$. We discretize the problem using the difference method with $n = 4$, $h = 0.25$.



The boundary values give $y_0 = 0$, $y_4 = \sinh(1)$, and the unknowns y_1, y_2, y_3 satisfy

$$y_{n-1} - (2 + h^2)y_n + y_{n+1} = 0, \quad n = 1, 2, 3 \ ,$$

or, in matrix form

$$\begin{pmatrix} -(2+h^2) & 1 & 0 \\ 1 & -(2+h^2) & 1 \\ 0 & 1 & -(2+h^2) \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -\sinh(1) \end{pmatrix} \ .$$

The solution of this linear system is given in the table.

x_n	y_n	error
0.250	0.252803	$1.9 \cdot 10^{-4}$
0.500	0.521406	$3.1 \cdot 10^{-4}$
0.750	0.822598	$2.8 \cdot 10^{-4}$

If we halve the step length, we get

x_n	y_n	error
0.125	0.125351	$2.5 \cdot 10^{-5}$
0.250	0.252660	$4.8 \cdot 10^{-5}$
0.375	0.383918	$6.6 \cdot 10^{-5}$
0.500	0.521174	$7.8 \cdot 10^{-5}$
0.625	0.666573	$8.1 \cdot 10^{-5}$
0.750	0.822387	$7.1 \cdot 10^{-5}$
0.875	0.991052	$4.5 \cdot 10^{-5}$

Note that at the points that are common in the two tables, $x_n = 0.25$, 0.50, 0.75, the error is reduced by a factor of approximately four when the step length is halved. This indicates that the truncation error is $O(h^2)$. ∎

The following theorem, which we give without proof, shows that the truncation error of the method is indeed $O(h^2)$.

Theorem 10.9.1. Consider the boundary value problem

$$y'' - q(x)y = f(x), \qquad y(a) = \alpha, \quad y(b) = \beta \ ,$$

with $q(x) \geq 0$. Let $y(x)$ denote the solution, and let y_n be the approximation of $y(x_n)$ obtained by the difference method.

If $|y^{(4)}| \leq M$ for all $x \in [a, b]$, then

$$|y_n - y(x_n)| \leq \frac{M\, h^2}{24}\,(x_n - a)(b - x_n), \quad n = 1, 2, \ldots N{-}1 \ .$$

Example. The following MATLAB function uses the difference method to compute an approximate solution of the boundary value problem defined in Theorem 10.9.1.

```
function  [x,y] = bvp1(q,f,bv,ab,N)
% Solve the boundary value problem  y'' - q(x)y = f(x) ,
% y(a) = bv(1), y(b) = bv(2);   a=ab(1), b=ab(2).
% Grid with step length  h = (b-a)/N.
% q(x)  is assumed to be nonnegative.
a = ab(1);  b = ab(2); h = (b - a)/N;
x = linspace(a,b,N+1)';        % grid.  Column vector
% Function values at interior points
qv = feval(q,x(2:N));  fv = feval(f,x(2:N));
% Tridiagonal matrix in sparse format
A = spdiags([ones(N-1,1) -(2+h^2*qv) ones(N-1,1)], ...
  -1:1, N-1,N-1);
% Right hand side
b = h^2*fv; b(1) = b(1)-bv(1);  b(end) = b(end)-bv(2);
% Solve the system and append the boundary values
y = [bv(1); A\b; bv(2)];
```

We use this function to solve the boundary value problem

$$y'' - y = x^2 - 2, \qquad y(0) = 1, \quad y(1) = \cosh(1) - 1 \ ,$$

with step lengths $h = 0.02$ and $h = 0.01$. The analytic solution is $y(x) = \cosh(x) - x^2$.

```
>> q = inline('ones(size(x))');
>> f = inline('x.^2 - 2');
>> [x1,y1] = bvp1(q,f,[1 cosh(1)-1],[0 1],50);
>> [x2,y2] = bvp1(q,f,[1 cosh(1)-1],[0 1],100);
```

The solution and the errors are shown in Figure 10.15.

The errors behave as stated in the theorem: halving the step length reduces the error by a factor approximately four, and for fixed h the error is largest close to the middle of the interval. ∎

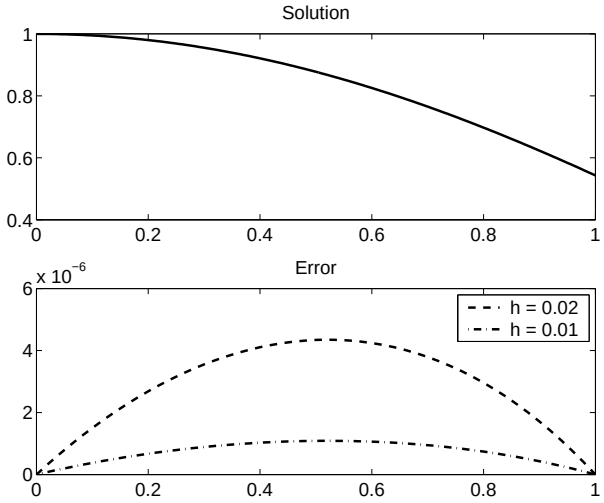


Figure 10.15. *Top: solution to the boundary value problem. Bottom: error with* $h = 0.02$ *and* $h = 0.01$.

We have discussed the difference method applied to the special case of a linear differential equation without a term with the first derivative. It is easy to generalize to problems involving y', but in this case the matrix is not diagonally dominant if the step length is chosen too large, see Exercise E5.

If the difference method is applied to a problem

$$y'' = f(x, y), \qquad y(a) = \alpha, \quad y(b) = \beta \ ,$$

where $f(x, y)$ is nonlinear in y, then one gets a nonlinear system of equations

$$Ay = F(y) \ ,$$

where $y = (y_1, \ldots, y_{N-1})^T$, and

$$A = \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix}, \quad F(y) = \begin{pmatrix} h^2 f(x_1, y_1) - \alpha \\ h^2 f(x_2, y_2) \\ \vdots \\ h^2 f(x_{N-2}, y_{N-2}) \\ h^2 f(x_{N-1}, y_{N-1}) - \beta \end{pmatrix} .$$

The nonlinear system can eg be solved by means of a fixed point iteration, cf Section 4.8,

$$Ay^{[k+1]} = F(y^{[k]}) \ .$$

In each iteration you have to solve a tridiagonal system with the matrix A.

Under quite mild restrictions on the function f it can be shown that also in the nonlinear case, the truncation error is $O(h^2)$.

10.10. A Finite Element Method

Finite element methods are often used for solving eg problems in structural mechanics. They are based on the principle of subdividing the construction under study into small parts, "*finite elements*". One can formulate equations for the influence on each element when a load is put on the construction. Each element is also influenced by its neighbours, and there results a system of equations, that describes the total effect on the construction.

Structural mechanics computations usually involve *partial differential equations*, and often the *Rayleigh-Ritz method* is used to solve them. This method is closely related to the physical background: it is based on the principle of minimizing the potential energy of the system. We shall study a similar method, *Galerkin's method*, applied to a boundary value problem for an ordinary differential equation. When this method is used as described below, it gives the same results as the Rayleigh-Ritz method.

We aim at introducing some basic mathematical and numerical ideas, not at giving a full account of finite element methods. We deliberately avoid certain details, and therefore the presentation is not completely stringent.

To formulate Galerkin's method, we define the *scalar product* of two functions g and h defined on the interval $[a, b]$, cf Chapter 9,

$$(g, h) = \int_a^b g(x)h(x)\, dx \ .$$

The functions g and h are said to be *orthogonal* if $(g, h) = 0$.

Consider the boundary value problem

$$Ly = -y'' + qy = f, \qquad y(a) = y(b) = 0 \ ,$$

where we assume that $q \geq 0$ is a constant, and that $f(x)$ is not identically zero. We have introduced the notation L for the differential operator that maps a twice differentiable function y onto $-y'' + qy$.

Let $\mathbb{V}$ be a class of *test functions*, that satisfy the boundary conditions

$$\mathbb{V} = \big\{ v \mid v' \text{ is piecewise continuous and bounded on } [a,b], \\ \text{and } v(a) = v(b) = 0 \ \big\} .$$

If y satisfies the differential equation $Ly = f$, then, trivially,

$$(v, Ly - f) = 0 \quad \text{for all } v \in \mathbb{V} .$$

Conversely, it can be shown that if y is such that

$$(v, Ly) = (v, f) \quad \text{for all } v \in \mathbb{V} ,$$

then y also satisfies $Ly = f$. The above condition is called the weak form of the differential equation.

Definition 10.10.1. The weak form:

$$(v, Ly) = (v, f) \quad \text{for all } v \in \mathbb{V} .$$

The weak form is similar to the normal equations for an overdetermined linear system of equations (cf Section 8.14): There, we define a residual vector and require it to be orthogonal to all vectors in a certain linear space. In the weak form, the residual function $r = Ly - f$ is required to be orthogonal to all functions in $\mathbb{V}$.

We will now reformulate the left hand side in the weak form. The definitions of the operator L and the scalar product give

$$(v, Ly) = (v, -y'' + qy) = \int_a^b -v(x)y''(x)\,dx + q\int_a^b v(x)y(x)\,dx .$$

By partial integration of the first integral we get

$$\int_a^b -v(x)y''(x)\,dx = \big[-v(x)y'(x)\big]_a^b + \int_a^b v'(x)y'(x)\,dx .$$

The boundary term vanishes because $v(a) = v(b) = 0$, and we have shown that the weak form can be written

The weak form **partially integrated:**

$$(v, Ly) = (v', y') + q(v, y) = (v, f) \quad \text{for all } v \in \mathbb{V} .$$

By a partial integration we have moved one derivative from y to v.

Let y^h denote an approximate solution of the boundary value problem. We write y^h in the form

$$y^h = \sum_{j=1}^{N-1} c_j \varphi_j \ ,$$

where $\{\varphi_j\}_{j=1}^{N-1}$ are given functions. The notation y^h suggests that the φ_j are related to a discretization with step length h, but for a while we refrain from specifying them. We merely assume that they are linearly independent (cf Chapter 9), which means that they constitute a *basis* in an $(N-1)$-dimensional function space, that we call $\mathbb{V}^h$. Further, we assume that all the basis functions φ_j satisfy the boundary conditions on y:

$$\varphi_j(a) = \varphi_j(b) = 0, \quad j = 1, 2, \ldots, N-1 \ .$$

We now require that the approximation y^h satisfies the weak form,

$$(v, Ly^h) = (v, f) \ .$$

Further, we "discretize" the test functions v by requiring them to belong to $\mathbb{V}^h$. We get

$$(v^h, Ly^h) = (v^h, f) \quad \text{for all } v^h \in \mathbb{V}^h \ .$$

This is a special case of Galerkin's method.

> Let $\mathbb{V}^h$ be a finite-dimensional class of functions with basis $\{\varphi_j\}_{j=1}^{N-1}$. *Galerkin's method* applied to the equation $Ly = f$ amounts to determining the function $y^h \in \mathbb{V}^h$ that satisfies
>
> $$(v^h, Ly^h) = (v^h, f) \quad \text{for all } v^h \in \mathbb{V}^h \ .$$

Since all the basis functions φ_j are assumed to satisfy the boundary conditions, it follows that y^h will do so.

The requirement that $(v^h, Ly^h) = (v^h, f)$ for all $v^h \in \mathbb{V}^h$ is equivalent to the same requirement with v^h replaced by any of the basis functions:

$$(\varphi_i, Ly^h) = (\varphi_i, f), \quad i = 1, 2, \ldots, N-1 \ .$$

We will now show that this constitutes a linear system of equations for the coefficients in the expression $y^h = \sum_{j=1}^{N-1} c_j \varphi_j$. Inserting this in the partially integrated form, we get

$$(\varphi_i, Ly^h) = \sum_{j=1}^{N-1} c_j (\varphi_i', \varphi_j') + q \sum_{j=1}^{N-1} c_j (\varphi_i, \varphi_j) = (\varphi_i, f), \quad i = 1, \ldots, N-1 \ .$$

This is a linear system of equations for the coefficients c_j:

$$Kc = F \ .$$

The matrix K is the sum of two matrices,

$$K = K_1 + K_0 \ .$$

The so-called *stiffness matrix* K_1 is given by

$$K_1 = \begin{pmatrix} (\varphi_1', \varphi_1') & (\varphi_1', \varphi_2') & \cdots & (\varphi_1', \varphi_{N-1}') \\ (\varphi_2', \varphi_1') & (\varphi_2', \varphi_2') & \cdots & (\varphi_2', \varphi_{N-1}') \\ \vdots & \vdots & & \vdots \\ (\varphi_{N-1}', \varphi_1') & (\varphi_{N-1}', \varphi_2') & \cdots & (\varphi_{N-1}', \varphi_{N-1}') \end{pmatrix} ,$$

the so-called *mass matrix* K_0 is

$$K_0 = q \begin{pmatrix} (\varphi_1, \varphi_1) & (\varphi_1, \varphi_2) & \cdots & (\varphi_1, \varphi_{N-1}) \\ (\varphi_2, \varphi_1) & (\varphi_2, \varphi_2) & \cdots & (\varphi_2, \varphi_{N-1}) \\ \vdots & \vdots & & \vdots \\ (\varphi_{N-1}, \varphi_1) & (\varphi_{N-1}, \varphi_2) & \cdots & (\varphi_{N-1}, \varphi_{N-1}) \end{pmatrix} ,$$

and the right hand side is given by

$$F_i = (\varphi_i, f), \quad i = 1, 2, \ldots, N-1 \ .$$

Note that both K_1 and K_0, and therefore K are *symmetric*. Further, under the given assumptions on the operator L, one can show that K is *positive definite*.

So far, the presentation has been independent of the choice of basis functions φ_j. Now, we will choose *linear elements*. First, we divide the interval $[a, b]$ into N subintervals, $[x_{j-1}, x_j]$, $j = 1, \ldots, n$.[3)] For the sake of simplicity we use equidistant $\{x_j\}$,

$$x_j = a + jh, \quad h = \frac{b-a}{N}, \quad j = 0, 1, \ldots, N \ .$$

The linear element φ_j is a piecewise linear function, defined by

$$\varphi_j(x) = \begin{cases} 0 \ , & x_0 \le x \le x_{j-1} \ , \\ (x - x_{j-1})/h \ , & x_{j-1} \le x \le x_j \ , \\ (x_{j+1} - x)/h \ , & x_j \le x \le x_{j+1} \ , \\ 0 \ , & x_{j+1} \le x \le x_N \ . \end{cases}$$

Note that $\varphi_j(x_j) = 1$. Two of the functions are shown in Figure 10.16.

[3)] More precisely: the subintervals are *elements*, and in each element the function y^h varies linearly between its endpoint values.

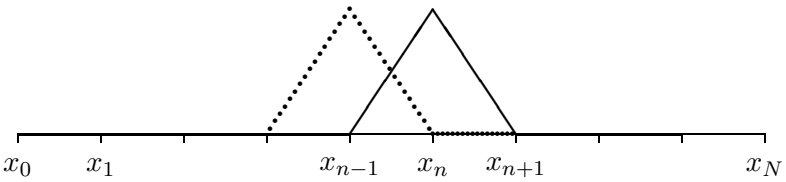


Figure 10.16. *Linear elements φ_{n-1} (dotted) and φ_n (solid).*

The functions φ_j were introduced in Section 5.10 under the name "linear B-splines". The space $\mathbb{V}^h$ that is spanned by these basis functions, is therefore a space of linear splines, ie piecewise straight lines.

The scalar products (φ_i', φ_j') and (φ_i, φ_j) in the stiffness and mass matrices are zero unless φ_i and φ_j share an interval, where both of them are nonzero. It is readily seen that

$$(\varphi_i', \varphi_j') = (\varphi_i, \varphi_j) = 0 \quad \text{if} \quad |j-i| \geq 2 \ .$$

In other words: both K_1 and K_0 (and therefore K) are tridiagonal matrices. The nonzero elements are

$$(\varphi_j', \varphi_j') = \int_{x_{j-1}}^{x_j} \left(\frac{1}{h}\right)^2 dx + \int_{x_j}^{x_{j+1}} \left(\frac{-1}{h}\right)^2 dx = \frac{2}{h} \ ,$$

$$(\varphi_j', \varphi_{j+1}') = \int_{x_j}^{x_{j+1}} \left(\frac{-1}{h}\right)\left(\frac{1}{h}\right) dx = \frac{-1}{h} \ ,$$

$$(\varphi_j, \varphi_j) = \int_{x_{j-1}}^{x_j} \left(\frac{x - x_{j-1}}{h}\right)^2 dx + \int_{x_j}^{x_{j+1}} \left(\frac{x_{j+1} - x}{h}\right)^2 dx = \frac{2h}{3} \ ,$$

$$(\varphi_j, \varphi_{j+1}) = \int_{x_j}^{x_{j+1}} \frac{x_{j+1} - x}{h} \, \frac{x - x_j}{h} \, dx = \frac{h}{6} \ .$$

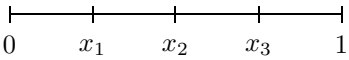
(Check the details!). Thus, the two matrices are

$$K_1 = \frac{1}{h} \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}, \quad K_0 = \frac{qh}{6} \begin{pmatrix} 4 & 1 & & & \\ 1 & 4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 4 & 1 \\ & & & 1 & 4 \end{pmatrix} . \tag{10.10.1}$$

Example. We discretize the boundary value problem

$$-y'' + y = 1, \qquad y(0) = y(1) = 0 \ ,$$

using linear elements with step length $h = 0.25$.



The stiffness and mass matrices are

$$K_1 = \frac{1}{0.25}\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}, \quad K_0 = \frac{0.25}{6}\begin{pmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{pmatrix} \ ,$$

and the components of the right hand side are

$$F_i = (\varphi_i, f) = \textstyle\int_0^1 \varphi_i(x)\, dx = h = 0.25, \quad n = 1, 2, 3 \ .$$

After multiplying by h we get the finite element equation

$$\begin{pmatrix} 2 + 1/24 & -1 + 1/96 & 0 \\ -1 + 1/96 & 2 + 1/24 & -1 + 1/96 \\ 0 & -1 + 1/96 & 2 + 1/24 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1/16 \\ 1/16 \\ 1/16 \end{pmatrix} \ .$$

The solution is $c_1 \simeq 0.0857$, $c_2 \simeq 0.1137$, $c_3 \simeq 0.0857$, and the corresponding $y^h = c_1\varphi_1 + c_2\varphi_2 + c_3\varphi_3$ is shown in Figure 10.17 together with the error $y(x) - y^h(x)$. The exact solution is

$$y(x) = 1 - \cosh(x) + \frac{\cosh(1) - 1}{\sinh(1)} \sinh(x) \ .$$

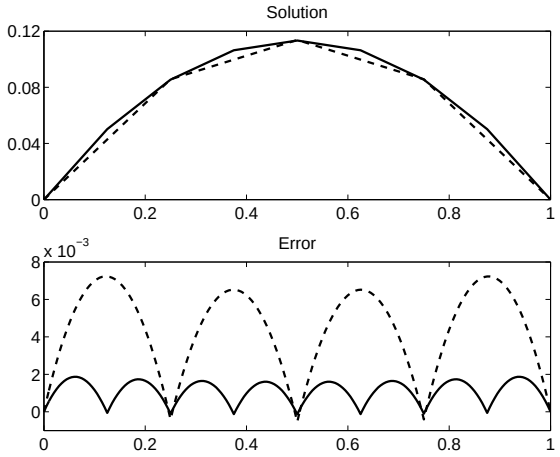


Figure 10.17. *Finite element solution y^h and error $y - y^h$ for $h = 1/4$ (dashed) and $h = 1/8$ (solid).*

■

We summarize:

The finite element equation for the boundary value problem

$$-y'' + qy = f, \qquad y(a) = y(b) = 0 \ ,$$

discretized using linear elements on an equidistant grid, is given by

$$Kc = F, \qquad K = K_1 + K_0 \ ,$$

where the stiffness and mass matrices K_1 and K_0 are given by (10.10.1), and the right hand side is given by

$$F_i = (\varphi_i, f), \quad i = 1, 2, \ldots, N-1 \ .$$

The approximate solution y^h is a piecewise linear function

$$y^h = \sum_{j=1}^{N-1} c_j \varphi_j \ .$$

The following can be shown.

Theorem 10.10.2. The piecewise linear finite element approximation y^h, derived in this section, satisfies

$$\|y^h - y\| \leq C\, h^2 \, \|f\| \ ,$$

for some constant C. (The norm is defined as $\|u\| = \sqrt{(u,u)}$).

In the same way as we introduced cubic splines in Section 5.11, we can define finite element methods for cubic elements. This leads to finite element equations with bandwidth 5, and the truncation error is (usually) $O(h^4)$. This is as expected, when you compare with the error estimates in Chapter 5.

Example. The MATLAB function `bvp4c` is based on cubic elements and *collocation*. This is a Galerkin method, where the solution is approximated by a cubic spline, and the test function v_n evaluates in discrete points: for a given function g one gets $(v_i, g) = g(x_i)$.

The boundary value problem

$$u'' = 1 + uu', \qquad u(0) = 1, \quad u(0.6) = 2 \ ,$$

is reformulated to a system of first order equations

$$\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} y_2 \\ 1 + y_1 y_2 \end{pmatrix}, \quad y_1(0) = 1, \quad y_1(0.6) = 2 \ .$$

The differential equation and the boundary conditions are implemented in the functions

```
function  f = odef3(x,y)
f = [y(2); 1+y(1)*y(2)];

function  r = bvres(ya,yb)
r = [ya(1)-1;  yb(1)-2];
```

Note that `bvres` returns a vector of zeros if the boundary conditions are satisfied.

`bvp4c` can solve nonlinear problems (our equation is nonlinear!), and therefore it needs an initial guess of the solution. We use the straight line between the two endpoints.

```
xi = linspace(0,0.6,5);
yi = [1 + (1/0.6)*xi; (1/0.6)*ones(size(xi))];
solinit.x = xi;  solinit.y = yi;
sol = bvp4c(@odef3, @bvres, solinit);
plot(sol.x,sol.y(1,:),'o', sol.x,sol.y(2,:),'v')
x = linspace(0,0.6);  y = bvpval(sol,x);
hold on,  plot(x,y(1,:),'-', x,y(2,:),'--')
```

The resulting plot is shown in Figure 10.18.

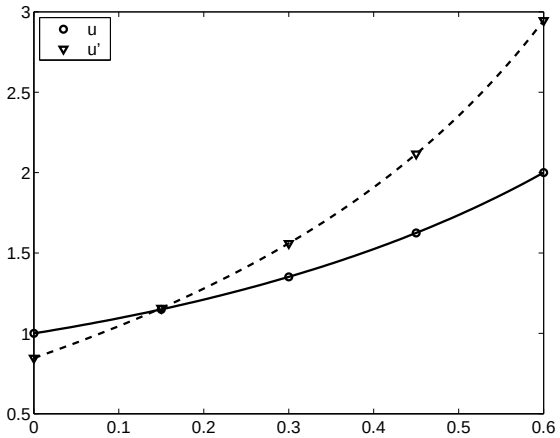


Figure 10.18. *Solution from `bvp4c` and interpolation by `bvpval`.*

The output from `bvp4c` is a *struct* with fields `sol.x`, `sol.y` and `sol.yp`, containing respectively values x_i of the independent variable, and the corresponding values of the approximations $y(x_i)$ and $y'(x_i)$. Intermediate values of the approximate solution can be obtained by the function `bvpval`, which is based on Hermite interpolation, cf Section 5.7. ∎

In Section 10.7 we saw that it may be advantageous to use varying step length in the solution of initial value problems. This is also the case with boundary value problems, and it is easier to do that with a finite element method than with a difference method. The advantage of finite element methods is even more pronounced in the solution of partial differential equations, especially because of the ease with which it is possible to treat complicated geometries and different boundary conditions. Further, there is a well-developed mathematical theory, that allows one to prove convergence and derive error estimates, even for complicated problems.

10.11. The Shooting Method

Finally, we study a third method for solving a boundary value problem

$$y'' = f(x, y, y'), \qquad y(a) = \alpha, \quad y(b) = \beta \ .$$

We assume that the problem has a unique solution. Further, we have to assume that also the initial value problem for the same equation has a unique solution.

Suppose that we knew the derivative at the left hand endpoint of the interval, ie $y'(a) = \gamma$, for some γ. Then, we would have an initial value problem,

$$y'' = f(x, y, y'), \qquad y(a) = \alpha, \quad y'(a) = \gamma \ .$$

Putting $v = y'$, we can write this as a system of first order differential equations,

$$\begin{pmatrix} y' \\ v' \end{pmatrix} = \begin{pmatrix} v \\ f(x, y, v) \end{pmatrix}, \qquad \begin{pmatrix} y(a) \\ v(a) \end{pmatrix} = \begin{pmatrix} \alpha \\ \gamma \end{pmatrix} \ . \tag{10.11.1}$$

This problem can be solved by one of the methods discussed earlier in this chapter, eg we could use a Runge-Kutta method.

Let $y(x, \gamma)$ denote the solution to the initial value problem (10.11.1). The idea behind the shooting method is to determine γ such that $y(b, \gamma) = \beta$, the given boundary value at the right hand end. In other words, we seek $\gamma = \gamma^*$ as a root of the equation

$$g(\gamma) = y(b, \gamma) - \beta = 0 \ .$$

A number of methods for solving this problem were discussed in Chapter 4. We cannot give an explicit expression for the function g, but we can get an approximation of $g(\gamma)$ by solving (10.11.1) numerically. Further, it would be rather complicated to compute the derivative $g'(\gamma)$. This leaves out Newton-Raphson's method, but we can use the *secant method* defined on page 73,

$$\gamma_{k+1} = \gamma_k - g(\gamma_k)\,\frac{\gamma_k - \gamma_{k-1}}{g(\gamma_k) - g(\gamma_{k-1})}\,, \quad k = 1, 2, \ldots$$

The values γ_0 and γ_1 must be chosen in another way.

Example. Given the boundary value problem

$$y'' = -y, \qquad y(0) = 0, \quad y(\pi/2) = 1\,,$$

(which has the analytic solution $y(x) = \sin x$). From the formulation we know that the solution passes through the two points $(0, 0)$ and $(\pi/2, 1)$, so a first guess on γ is the slope of the straight line between these two points,

$$\gamma_0 = \frac{1-0}{\pi/2 - 0} \simeq 0.6366\,.$$

We use the classical Runge-Kutta method with step length $h = \pi/100$ (ie we take $n = 50$ steps). The result is shown in the left part of Figure 10.19.

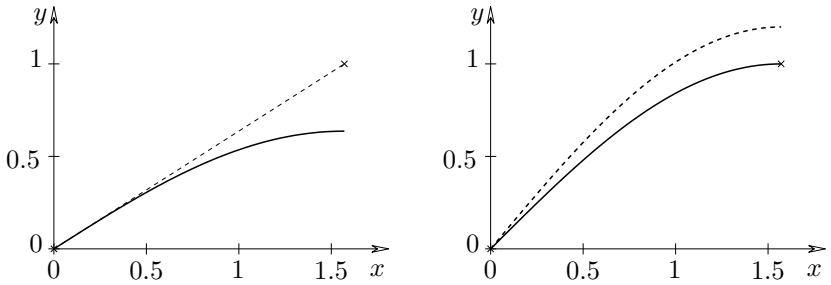


Figure 10.19. *Left: Boundary values and solution with* $\gamma_0 = 0.6366$. *Right: Solution with* $\gamma_1 = 1.2$ *(dotted) and* $\gamma_2 = 1$ *(solid).*

The resulting $y(b, \gamma_0)$ is too small, so we try a larger initial slope, $\gamma_1 = 1.2$. The result is shown with dotted line in the right part of Figure 10.19. The two results for the function g are

k	γ_k	$g(\gamma_k)$
0	0.6366	−0.3634
1	1.2000	0.2000

Now, we can use the secant method, and get

$$\gamma_2 = 1.2 - 0.2\,\frac{1.2 - 0.6366}{0.2 + 0.3634} = 1\ .$$

The solution with this initial slope is shown with solid line in the right part of Figure 10.19; $y(b, 1) = 1$, so we are finished. ∎

The procedure is reminiscent of the way that soldiers used to adjust the artillery in the old days; thus, the name "shooting method".

It was no coincidence in the example that already in the third try we had the correct $y'(0)$. One can show that if the function $f(x, y, y')$ in the differential equation is linear in y and y', then the function g is a first degree polynomial in γ, and the secant method is exact. In the more general case, when f is nonlinear in y and/or y', then also g is nonlinear, and one must perform several iterations.

Example. The boundary value problem from the example on page 350,

$$y'' = 1 + yy', \qquad y(0) = 1, \quad y(0.6) = 2\ ,$$

is nonlinear in y and y', so $g(\gamma) = y(0.6, \gamma) - 2$ is nonlinear in γ. Figure 10.18 on page 351 indicates that the solution is smooth, so if we use the classical Runge-Kutta method with $n = 10$ steps ($h = 0.06$), we should get a reasonably small truncation error.

With γ_0 chosen as the slope of the line between the two endpoints, $(0, 1)$ and $(0.6, 2)$, and $\gamma_1 = \frac{1}{2}\,\gamma_0$ we get the following results

n	γ_k	$y_n(\gamma_k)$	$g(\gamma_k)$
10	1.6666667	2.8754957	8.75e-01
10	0.8333333	1.9872039	-1.28e-02
10	0.8453378	1.9990253	-9.75e-04
10	0.8463275	2.0000012	1.16e-06
10	0.8463263	2.0000000	-1.06e-10

We see fast convergence to $\gamma^* \simeq 0.8463$. In order to reduce the truncation error, we halve the step length to $h = 0.03$, and repeat the secant iteration, starting with the last two γ_k-values:

n	γ_k	$y_n(\gamma_k)$	$g(\gamma_k)$
20	0.8463275	2.0000046	4.58e-06
20	0.8463263	2.0000034	3.42e-06
20	0.8463229	2.0000000	1.46e-12

When we compare the results obtained with the two step lengths, we see that the function values $g(\gamma_k)$ are changed by approximately $3.42 \cdot 10^{-6}$. This change is dominated by the global truncation error associated with the larger

of the two step lengths, $h = 0.06$. The computed value of γ^* is changed by $3.47 \cdot 10^{-6}$, so in the terms of the discussion in Section 4.5 we have $M \simeq 1$, so the root γ^* is well conditioned.

Since the global truncation error with the Runge-Kutta method is $O(h^4)$, a further halving of h should reduce the error by a factor of approximately 16. This is borne out by computation: with $h = 0.015$ we get $\gamma^* = 0.8463227$, and with 6 digits accuracy the solution is $\gamma^* = 0.846323$. ∎

Exercises

E1. Write the differential equation

$$3y''' + 4xy'' + \sin y = f(x)$$

as a system of first order differential equations.

E2. An *autonomous* system of differential equations is a system of the form $y' = f(y)$, where the independent variable (x or t, usually) does not enter explicitly into the right hand side. Introduce a new variable in a system $y' = f(x, y)$, and rewrite it in autonomous form.

E3. Given the initial value problem

$$y' = y, \qquad y(0) = 1 \ .$$

(a) Use Euler's method to compute an approximation of $y(x)$, ie take a step with Euler's method, using the step length $h = x$.

(b) Do the same with Heun's and Runge-Kutta's methods. Compare to the Maclaurin expansion of the solution.

E4. Consider the midpoint method applied to the test problem $y' = \lambda y$, $y(0) = 1$, with $y_0 = 1$.

(a) Show that the factors A and B in (10.6.3) can be computed by

$$B = \frac{y_1 - r_1}{r_2 - r_1} \ , \qquad A = 1 - B \ .$$

(b) Compute A and B when $\lambda = -100$, $h = 5 \cdot 10^{-3}$, and Euler's method is used to compute y_1.

(c) Show that r_1 is an $O(h^3)$ approximation to $y(h)$, and discuss the effect of taking $y_1 = r_1$.

Also see Computer Exercise C1.

E5. Given the boundary value problem

$$y'' + p(x)y' - q(x)y = 0, \qquad y(a) = \alpha, \quad y(b) = \beta \ ,$$

where $q(x) \geq 0$ and $|p(x)| \leq P$ for $x \in [a, b]$. Discretize the derivatives using central differences, and write down the corresponding system of equations. Show that the matrix is diagonally dominant if $h \leq 2/P$.

Computer Exercises

C1. Write a MATLAB script that solves the initial value problem $y' = -100y$, $y(0) = 1$ by means of the midpoint method and plots the numerical solution for $0 \leq x \leq 0.5$. Use the step length $h = 5 \cdot 10^{-3}$ and the value of y_1 computed by

(a) Euler's formula,

(b) $y_1 = r_1$, cf Exercise E4.

Discuss the results.

C2. Modify the model for the rocket in Chapter 1, so that it is valid also after all the fuel has been used. Use `ode45` to solve the initial value problem, and estimate the maximum height of the rocket in the two cases when air resistance is included in the model, or ignored, respectively.

C3. Given a long, straight conductor that carries a current $I = 1$ in positive direction along the y-axis. Let $(x(t), y(t))$ denote the position at time t of an electron, that was shot in vacuum at $t = 0$ from the point $(x(0), y(0))$ with velocity $(x'(0), y'(0))$. One can show that (ignoring effects of gravity and the magnetic field of Earth) the motion of the electron satisfies the equation

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} -Cy'/x \\ Cx'/x \end{pmatrix} \ ,$$

where $C = -3.5176 \cdot 10^4$. Use `ode45` to solve this initial value problem numerically in the interval $0 \leq t \leq t_{\max}$, and plot the orbit in the xy-plane. Try the values

$x(0)$	$x'(0)$	$y(0)$	$y'(0)$	$t_{\max}$
1	-10^{-5}	0	0	$6 \cdot 10^{-4}$
1	0	0	$-5 \cdot 10^2$	$9 \cdot 10^{-4}$

Use `ode45` both with default accuracy parameters and with `options` given by `odeset('Abstol',1e-12, 'Reltol',1e-4)` (cf the example on page 336).
(The assignment was proposed by Michael Hörnquist).

C4. Consider the initial value problem

$$\begin{pmatrix} u' \\ v' \\ w' \end{pmatrix} = \begin{pmatrix} -0.04u + 10^4 vw \\ 0.04u - 10^4 vw - 3{\cdot}10^7 v^2 \\ 3{\cdot}10^7 v^2 \end{pmatrix}, \quad \begin{pmatrix} u(0) \\ v(0) \\ w(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} .$$

The system is stiff. Use `ode45` and `ode15s` to solve the problem, and illustrate the results. See the example on page 336 or try `help ode45` to get information about how to set options for the `ode` solvers.

C5. Solve the boundary value problem

$$y'' = 1 + yy', \qquad y(0) = 1, \quad y'(0.8) = 4 .$$

(a) Using the MATLAB function `bvp4c`.

(b) Using the shooting method.

References

Numerical methods for the solution of ordinary differential equations have a long history. Leonhard Euler (1707 – 1783) made basic contributions around 1760. C.D.T. Runge (1856 – 1927) and M.W. Kutta (1867 – 1944) developed the classical Runge-Kutta method at the end of the nineteenth century. During the 1960s and 1970s active research lead to efficient methods for stiff systems. Good introductions to the solution of initial value problems are given in

J.D. Lambert, *Numerical Methods for Ordinary Differential Systems*, John Wiley and Sons, Chichester, 1991.

L.F. Shampine, *Numerical Solution of Ordinary Differential Equations*, Chapman and Hall, New York, 1994.

A more extensive treatment can be found in

E. Hairer, S.P. Nørsett and G. Wanner, *Solving Ordinary Differential Equations I*, 2nd edition, Springer, Berlin, 1993.

Boundary value problems are treated in

H.B. Keller, *Numerical Methods for two-point Boundary Value Problems*, Blaisdell, Waltham, Mass., 1968.

U. Ascher, R. Mattheij and R. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, SIAM, Philadelphia, PA, 1995.

Finite element methods have many important applications, and there is a vast literature in the area. The following two books are mathematically-numerically oriented, and mostly treat the solution of partial differential equations.

C. Johnson, *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Studentlitteratur, Lund, 1987.

G. Strang and G.J. Fix, *An Introduction to the Finite Element Method*, McGraw-Hill, New York, 1984.

Short Biographies

Below we give short biographies of some of the people whose names appear in this book, and mention the fields in numerical analysis, to which they have made important contributions. Further information can be found, eg, in the book

> H.H. Goldstine, *A History of Numerical Analysis from the 16th through the 19th Century*, Springer Verlag, New York, 1977.

and at the URL http://www-gap.dcs.st-and.ac.uk/∼history/

Pafnuty Lvovich Chebyshev (1821–1894). Russia. Professor in St Petersburg 1850. Contributions in number theory, integration, statistics and approximation theory. Applications in map construction, calculation of geometric volumes, and the construction of calculating machines in the 1870s. Introduced the general concept of orthogonal polynomials. First used Chebyshev polynomials in 1854.

Andre–Louis Cholesky (1875–1918). French artillery officer. Worked with land surveying in France, Crete and North Africa. Developed his factorization method around 1906, in order to get faster and more accurate solutions of the normal equations. This was done in the field by hand.

Roger Cotes (1682–1716). England. Professor in astronomy and experimental philosophy in Cambridge, 1710. Contributions to interpolation and table construction.

Euclid of Alexandria (about 325 BC–265 BC). Egypt. Collected and systematized (almost) all that was known at that time about geometry and number theory. His "Elements" was used as a mathematics textbook for more than 2000 years.

Leonhard Euler (1707–1783). Switzerland and Russia. Professor in physics in St Petersburg 1830. Fathered 13 children and "made some of his greatest mathematical discoveries while holding a baby in his arms with other children playing around his feet". Founder of mathematical modelling of physical phenomena (1736). Major contributions in summation of series, calculus of variations and differential equations.

Jean Baptiste Joseph Fourier (1768–1830). France. Professor at École Polytechnique 1797. Scientific adviser to Napoleon's army at the invasion of Egypt 1798. Prefect in Grenoble 1801–15. In this period he first used trigonometric expansions in the study of heat flow in solid bodies.

Johann Carl Friedrich Gauss (1777–1855). Germany. Director of Göttingen observatory 1807–37. Major contributions in almost all fields of numerical analysis. Gave the statistical motivation for the use of the least squares method to predict the orbit of Ceres from a few observations (1809). Discussed rounding errors and numerical stability (1809), and discovered the fast Fourier transform (FFT) in 1805.

James Wallace Givens (1910–1993). USA. Director of the applied mathematics division at Argonne National Laboratory. Pioneered the use of plane rotations in matrix computations.

Charles Hermite (1822–1901). France. Professor of analysis in Paris 1869. Contributions in interpolation and in linear algebra (Hermitean matrices).

William George Horner (1786–1837). English school headmaster. Is only known for his algorithm for evaluating a polynomial.

Alston Scott Householder (1904–1993). USA. Director of Oak Ridge National Laboratory 1948, professor of mathematics at the University of Tennessee 1969. One of the founders of modern numerical analysis, with focus on algorithm development. Formalized the decompositional approach to matrix computation 1951.

Carl Gustav Jacob Jacobi (1804–1851). Germany. Professor in mathematics in Königsberg 1827, Berlin 1844. Major contributions in number theory, elliptic functions and partial differential equations, with applications in dynamics. In 1841 he published a treatise on the matrix that bears his name, but was first presented in 1815 by Cauchy.

Martin Wilhelm Kutta (1867–1944). Germany. Professor in Stuttgart 1911. Contributions in aerodynamics and differential equations. Presented the classical Runge-Kutta method in 1901.

Joseph-Louis Lagrange (1736–1813). Italy, Germany and France. Director of mathematics at the Academy of Sciences in Berlin 1766, in Paris 1787. Major contributions in mathematical modelling, calculus of variations and differential equations. Applications in mechanics.

Adrien–Marie Legendre (1752–1833). France. Worked at the Academy of Sciences in Paris from 1785. Introduced his polynomials in 1784 in the study of the movements of planets. Took part in the project of determining the length of the meter 1791, involving advanced space trigonometry. Introduced the least squares method to determine orbits of comets, 1806.

Rudolf Otto Sigismund Lipschitz (1832–1903). Germany. Professor in Bonn 1864. Contributions in special functions, differential equations and analytical mechanics.

Colin Maclaurin (1698–1746). Scotland. Professor in mathematics in Aberdeen 1717. First used "Maclaurin expansions" in 1742 and introduced the integral test for the convergence of infinite series.

Cleve Moler (born 1939). USA. Professor in computer science for almost 20 years at Michigan, Stanford and New Mexico universities. After that 5 years work with computer hardware companies and almost 15 years as senior scientist at MathWorks. Important contributions in numerical linear algebra. He was one of the authors of the first versions of the LINPACK and EISPACK libraries. Best known for being the father of MATLAB, he developed the first version of it around 1980.

Eric Harold Neville (1889-1961). England. Fellow of Trinity College in Cambridge 1911–1917, professor of mathematics at the University of Reading 1919–1954. His method was first presented in the paper *Iterative Interpolation*, J. Indian Mathematical Society, 1933.

Isaac Newton (1642–1727). England. Fellow of Trinity College, Cambridge 1667, master of the Royal Mint 1699. Most famous for his discovery of the laws of motion and (in competition with Leibniz) for the invention of differential and integral calculus. Important contributions to numerical analysis in fields of interpolation, integration and roots of functions.

Pythagoras of Samos (about 569 BC–475 BC). Grece and Italy. Founded a philosophical and religious school that stressed secrecy, so there is no direct documentation of his achievements. It is known, however, that he contributed to number theory (with applications in astronomy and music) and geometry. His theorem about a right-angled triangle was known to the Babylonians 1000 years earlier, but he seems to have been the first to prove it.

Joseph Raphson (1648–1715). England. Seems to have invented the "Newton-Raphson's method" independently from Newton, in the special case of roots of polynomials.

Evgeny Yakovlevich Remez (1896–1975). Ukraine. Professor in mathematics in Kiev 1935. Contributions in approximation theory. Developed his algorithm for minimax approximation in the mid 1930s.

Lewis Fry Richardson (1881–1953). England and Scotland. Introduced the method of finite differences in weather forecasting, 1922.

Michel Rolle (1652–1719). Worked in diophantine analysis (integer solutions of equations), algebra and geometry. Presented his theorem in 1691.

Werner Romberg (1909–2003). Germany and Norway. Professor in applied mathematics in Trondheim 1960, Heidelberg 1968. Presented the method known as Romberg integration in 1955.

Carl David Tolmé Runge (1856–1927). Germany. Professor in applied mathematics in Göttingen 1904–1925. Initiated Runge-Kutta methods in 1895.

Isaac Jacob Schoenberg (1903–1990). Romania and USA. Institute for Advanced Studies, Princeston 1933, Professor at University of Pennsylvania 1941, Army's Ballistic Research Laboratory 1943–45, Mathematics Research Center, Wisconsin 1966. Important contributions to the theory of splines.

Thomas Simpson (1710–1761). England. Taught in the London coffee houses. Contributions in interpolation and numerical methods for integration.

Brook Taylor (1685–1731). England. His book "Methodus incrementorum directa et inversa" from 1715 introduced what is now known as Taylor expansion, calculus of finite differences, and integration by parts.

Karl Theodor Wilhelm Weierstrass (1815–1897). Germany. Professor in Berlin 1856. Is considered as the father of modern analysis.

Anne Whitney (Born 1921). Ph.D. in 1949: "On the positivity of translation determinants for Polya frequency functions with an application to the interpolation problem by spline curves". Married in 1952 and took the name Calloway.

James Hardy Wilkinson (1919–1986). England. Worked at the National Physical Laboratory from 1946. Major contributions in numerical linear algebra. Developed backward error analysis in the 1950s. Coedited and contributed to the collection of high quality software in "Handbook series in linear algebra", published in Numerische Mathematik in the 1960s. This collection can be said to be the backbone of present day's LAPACK.

Answers to Exercises

Chapter 2

E1. $|\overline{\pi} - \pi| \le 1.7 \cdot 10^{-4}$.

E3. (a) $\Delta y \simeq \dfrac{\Delta x}{x}$. (b) $\dfrac{\Delta f}{f} \simeq \alpha_1 \dfrac{\Delta x_1}{x_1} + \alpha_2 \dfrac{\Delta x_2}{x_2} + \alpha_3 \dfrac{\Delta x_3}{x_3}$.

E5. (a) $2(x + x^3/3! + x^5/5! + \cdots)$. (b) $-\cos 2x/(\sin x + \cos x)$. (c) $2\sin^2 \frac{x}{2}$. (d) $(\sqrt{1+x^2} + \sqrt{1-x^2})/(2x^2)$.

E8. (a) $fl[1+\mu] = 1$ because of rounding to even. (c) $\epsilon = 2\mu$.

E9. **if** $|x_1| > |x_2|$ **then** $u := x_2/x_1$; $s := |x_1|\sqrt{1+u^2}$;
else $u := x_1/x_2$; $s := |x_2|\sqrt{1+u^2}$;

E10. $\delta_r \le (1+\mu)^r - 1$ and $\delta_r \ge (1-\mu)^r - 1 \ge -((1+\mu)^r - 1)$ imply that $|\delta_r| \le (1+\mu)^r - 1$.
Let $\tau = r\mu$, then $(1+\mu)^r - 1 \le e^\tau - 1 = \tau + \frac{\tau^2}{2!} + \frac{\tau^3}{3!} + \cdots \le \tau(1 + \tau/2 + (\tau/2)^2 + \cdots) = \tau/(1-\tau/2) \le \tau/(1-0.1/2) < 1.06\tau$.

E12. $S \simeq 1.6$ has the exponent $e = 0$, and if $1/n^2 < \frac{1}{2}2^{-23}$, then $fl[S + 1/n^2] = S$. This is equivalent to $n^2 > 2^{24}$, or $n > 2^{12} = 4096$.

E13. $\hat{S}_n = \sum_{i=1}^n x_i(1+\eta_i)$, $|\eta_i| \le 1.06k\mu$.

C2. (b) $\dfrac{|S_2(x) - \sinh x|}{|\sinh x|} \le \dfrac{|x^7/7! + x^9/9! + \cdots|}{|x|} \le \dfrac{x^6}{7!}\big(1 + \dfrac{x^2}{72} + \dfrac{x^4}{72^2} + \cdots\big) \le \dfrac{x^6}{7!(1 - x^2/72)} \le 2 \cdot 10^{-10} < 2^{-24}$.

Chapter 3

E1. (a) Both π_0 and n have 8 bits and their product $n\pi_0 = 998.71875$ has 16 bits and is computed without error. Further, x and $n\pi_0$ have the same exponent ($e = 9$), so $fl[x - n\pi_0] = x - n\pi_0$.

(b) $\overline{u} = ((x - n\pi_0) - nR(1+\epsilon_0)(1+\epsilon_1))(1+\epsilon_2)$, $|\epsilon_k| \le \mu$.
$|\Delta u/u| \le 2.28\mu < 1.4 \cdot 10^{-7}$.

E2. For $|x - n\pi| \le 1.76 \cdot 10^{-8}$ ($\Delta u| \le 3n\pi \cdot 2^{-53}$).

E4. $\tau = 0.60725293500888$.

E5. For $2^{-3i}/3 < 2^{-46}$, ie $i \geq 15$.

Chapter 4

E2. $x_{k+1} = x_k(2 - ax_k)$. Can assume $a \in [1, 2[$.
$\epsilon_k = x_k - 1/a$, $|\epsilon_k| \leq \frac{1}{2}(2\epsilon_0)^{2^k}$.
Table with n bits: size $N = 2^{n-1}$, $|\epsilon_0| \leq 2^{-n}$. $2^k(n-1) \geq -\log_2(2\mu)$.

E3. $x^* = 0.523596 \pm 0.3 \cdot 10^{-5}$.

E4. $-1 \pm \sqrt{\delta/3}$, $2 - \delta/9$.

C3. $N = 128$.

C4. Each estimate in E4 is based on the dominating term in Taylor expansions around the unperturbed root. The effect of this truncation error grows with the size of $|\delta|$.

C5. (b) 18 (if we take $\overline{x}$ as the midpoint of the resulting bracket).
(c) $x_1 = 0.64884894512009$, $x_2 = 0.65887458885001$,
$x_3 = 0.65882715616353$, $x_4 = 0.65882715493945$.
(d) $d_{k+1} \simeq -0.5 \cdot d_k^2$, $k = 2, 3$.

C6. Solutions: $\begin{pmatrix} 2.9984 \\ 0.1484 \end{pmatrix}$, $\begin{pmatrix} 1.3364 \\ 1.7542 \end{pmatrix}$, $\begin{pmatrix} -3.0016 \\ 0.1481 \end{pmatrix}$, $\begin{pmatrix} -0.9013 \\ -2.0866 \end{pmatrix}$.

Chapter 5

E1. (a) $h = 2^{-10} \leq \sqrt{2} \cdot 2^{-10}$, Table size $N = 3075$. (b) $N \geq 1536$.

E2. 3.

E3. $2x(x+1)(x-1)(x-2)$.

E4. Use Theorem 5.4.3.

E5. Alternative expressions for the unique polynomial of degree ≤ 2, that interpolates $(x_i, f(x_i))$, $i = 0, 1, 2, 3$.

E6. (a) 684.895. (b) 666.666. (c) sin and cos have errors $\leq 0.5 \cdot 10^{-6}$ and cot has error ≤ 0.23. (d) Interpolation near a singularity for cot.

E7. $\displaystyle\int_a^b f(x)\,dx \simeq \frac{b-a}{n}\left(\tfrac{1}{2}f_0 + f_1 + \cdots + f_{n-1} + \tfrac{1}{2}f_n\right)$.

E8. Similar to the proof of Theorem 5.11.3. Simpler, since s' is piecewise constant.

E9. Use (5.12.1).

C1. At the kth stage of the loop the ith element ($k \le i \le m$) in `f` contains $f[x_{i-k}, \ldots, x_{i-1}]$ in `intpolc` and $f[x_0, \ldots, x_{k-2}, x_{i-1}]$ in `intpolc1`. Only the "diagonal" elements ($i = k$) are used, and they are identical.

C2. `p(i,:)  = [p(i,2) 2*p(i,3) 3*p(i,4) 0]/(x(i+1) - x(i))`

Chapter 6

E1. $$\frac{1}{2h^3}\big(f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)\big)$$
$$= f^{(3)}(x) + \frac{2^5-2}{5!}\,h^2 f^{(5)}(x) + \frac{2^7-2}{7!}\,h^4 f^{(7)}(x) + \cdots \quad .$$

E2. $$F(h/3) + \frac{F(h/3) - F(h)}{3^3 - 1} \quad .$$

E3. (a) $D(h) = f'(x) - \frac{24}{5!}\,h^4 f^{(5)}(x) - \frac{120}{7!}\,h^6 f^{(7)}(x) - \cdots$.

(b) For the approximation $D(h)$ we use the estimate $|R_{\mathrm{T}}| \le |D(h) - D(2h)|$ and

$$|R_{\mathrm{XF}}| \le \frac{0.5{\cdot}10^{-6}(1+8+8+1)}{12h} = \frac{0.75{\cdot}10^{-6}}{h} \quad .$$

$D(0.2) = 0.1224625,$
$D(0.1) = 0.1224217, \quad |R_{\mathrm{T}}| \le 4.1{\cdot}10^{-5}, \quad |R_{\mathrm{XF}}| \le 7.5{\cdot}10^{-7}$,
$D(0.05) = 0.1224267, \quad |R_{\mathrm{T}}| \le 5.0{\cdot}10^{-6}, \quad |R_{\mathrm{XF}}| \le 1.5{\cdot}10^{-5}$.

A smaller value for h or the use of Richardson extrapolation will increase the effect of R_{XF}. $f'(x) = 0.12243 \pm 0.00002$.

C2. The most accurate results are obtained with $q = 2$, where there are most points in the region with a clear dominating term in the error expansion.

Chapter 7

C1. (a) $I \simeq 0.7468$. (b) $I \simeq 1.9101$. (c) $x = t^2$ gives $I = \int_0^1 2\cos t^2\, dt \simeq 1.8090$. (d) $f(0) = 0$. $I \simeq 6.4319$.

(e) $I \simeq 1.5396$. `romberg` obtains this with $n = 128$, ie 129 function evaluations, and the trapezoidal rule gets the best result. `quad` gets the result after 41 function evaluations.

(f) `romberg` stops after one halving, because $T(1) = \frac{1}{2}(\cos(8\pi a) + 1 + \cos(8\pi(a+1)) + 1) = \cos(8\pi a) + 1$. $T(\frac{1}{2}) = \frac{1}{2}T(1) + \frac{1}{2}(\cos(8\pi(a+0.5)) + 1) = T(1)$. This result is therefore believed to be the correct result. In both cases `quad` gets the correct result after 49 function evaluations.

C2. $I \simeq 0.33199$. If the interval is divided into $[0, 0.375]$, $[0.375, 1.5]$ and $[1.5, 4]$, then `romberg` uses $n = 16$ function evaluations in each subinterval, ie a total of 51. `quad` uses 33 function evaluations.

Chapter 8

E1. 2° $\|x\|_\infty = \max_i |x_i| \le |x_1| + \cdots + |x_n| = \|x\|_1$
$\le n \cdot \max_i |x_i| = n \cdot \|x\|_\infty$

3° $\|x\|_\infty^2 = \big(\max_i |x_i|\big)^2 \le x_1^2 + \cdots + x_n^2 = \|x\|_2^2$
$\le n\big(\max_i |x_i|\big)^2 = n\|x\|_\infty^2$

1° $\|x\|_2^2 = x_1^2 + \cdots + x_n^2 \le (|x_1| + \cdots + |x_n|)^2 = \|x\|_1^2$
combined with 3° and 2° : $\|x\|_2^2 \le n\|x\|_\infty^2 \le n\|x\|_1^2$

E3. Since $\|Qx\|_2 = \|x\|_2$, Definition 8.10.2 shows that $\|Q\|_2 = 1$. Similarly, $\|Q^{-1}\|_2 = \|Q^T\|_2 = 1$, and Definition 8.11.1 gives $\kappa_2(Q) = 1$.

Chapter 9

E1. Use the normal equations.

E2. Choice 1: Conceptually simplest.
2: The matrix of the normal equations has zeros in positions $(1, 2)$, $(2, 1)$, $(2, 3)$ and $(3, 2)$.
3: The matrix of the normal equations is diagonal.

E3. (a) $\big\|\sum_{j=0}^n c_j\varphi_j\big\|_2^2 = \big(\sum_{j=0}^n c_j\varphi_j, \sum_{k=0}^n c_k\varphi_k\big) = \sum_{j=0}^n\sum_{k=0}^n c_j c_k(\varphi_j, \varphi_k)$
$= \sum_{j=0}^n c_j^2(\varphi_j, \varphi_j) = \sum_{j=0}^n c_j^2\|\varphi_j\|_2^2$.

(b) Use (a) and Definition 9.2.7.

E4. f^* and $f - f^*$ are orthogonal, and the relation is just a reordering of the terms in Theorem 9.3.2.

E5. $P(x) = \sum_{k=0}^n a_k T_k(x) = \sum_{k=0}^n (b_k - 2xb_{k+1} + b_{k+2})T_k(x)$
$= b_0 + b_2 - 2xb_1x + \sum_{k=1}^n ((b_k + b_{k+2})T_k(x) - b_{k+1}(T_{k+1}(x) + T_{k-1}(x)))$
$= \cdots = b_0 - xb_1$.

E6. (a) $U_0(x) = 1$, $U_1(x) = 2x$, $U_{k+1}(x) = 2xU_k(x) - U_{k-1}(x)$, $k = 1, 2, \ldots$.

(b) Follows from the recurrence in (a)

(c) Use the variable transformation $x = \cos v$

$$\int_{-1}^{1} w(x)U_k(x)U_j(x)\,dx$$
$$= \tfrac{1}{2}\int_0^{\pi} (\cos\frac{k-j}{2}\,v - \cos\frac{k+j}{2}\,v)\,dv \;=\; \begin{cases} \pi/2\,, & k=j\,, \\ 0\,, & k\neq j\,. \end{cases}$$

E7. Choose P_n as the polynomial that interpolates f at the zeros of $T_{n+1}(x)$.

E8. $p_n(x) = c_n\big(2^{1-n}T_n(x)+b_{k-1}x^{k-1}+\cdots+b_0\big)$ combined with Theorem 9.7.1 shows that $\|p_n\|_\infty \geq |c_n|\cdot\|2_{1-n}T_n\|_\infty = |c_n|\cdot 2_{1-n}$.

E9. (a) $f^*(x) = \dfrac{3}{14}\,(2+3x), \quad \|f^*-f\|_2 = \sqrt{\dfrac{3}{980}} \simeq 0.055\,.$

(b) $f^*(x) = \dfrac{1}{3\sqrt{3}} + x, \quad \|f^*-f\|_\infty = \dfrac{1}{3\sqrt{3}} \simeq 0.192.$

E10. (a) $\varphi_0(x) = 1, \quad \varphi_1(x) = x, \quad \varphi_2(x) = x^2 - \frac{3}{5}.$

(b) $p_2^* = c_0\varphi_0 + c_1\varphi_1 + c_2\varphi_2, \quad c_0 = c_2 = 0, \quad c_1 = \dfrac{60(\pi^2-8)}{\pi^4} \simeq 1.1516\,.$

(c) $\|f-p_2^*\|_2^2 = \|f\|_2^2 - c_1^2\|\varphi_1\|_2^2 = \dfrac{\pi^2+6}{3\pi^2} - c_1^2\cdot\dfrac{2}{5} \simeq 0.0753\,.$

E11. (a) $\varphi_0(x) = 1, \quad \varphi_1(x) = x, \quad \varphi_2(x) = x^2-2, \quad \varphi_3(x) = x^3 - 3.4x,$
$\varphi_4(x) = x^4 - \frac{31}{7}x^2 + \frac{72}{35}\,.$

(b) No. $\varphi_5(x) = x^5 - 5x^3 + 4x$ is orthogonal to $\varphi_0,\ldots,\varphi_4$, but $\|\varphi_5\| = 0\,.$

(c) $p_3^* = \frac{59}{5}\,\varphi_0 - \frac{17}{5}\,\varphi_1 + \frac{31}{7}\,\varphi_2 - \varphi_3 = -x^3 + \frac{31}{7}\,x^2 + \frac{103}{35}\,.$

(d) $p_4^* = p_3^* + \varphi_4 = x^4 - x^3 + 5\,.$

C3. $\sqrt{w_i}(ax_i^2 + b - 1/y_i) \simeq 0,\ i=1,2,3,4$ gives $a = 0.3334$, $b = 0.4954$.

C4. $a = 2.5, \quad b = 2, \quad r = 1.4577\,.$

C5.

n	$\|f-p_n\|_\infty$	$\|f-p_n^*\|_\infty$
1	$5.74\cdot10^{-2}$	$4.17\cdot10^{-2}$
2	$1.03\cdot10^{-2}$	$7.01\cdot10^{-3}$
3	$2.25\cdot10^{-3}$	$1.47\cdot10^{-3}$

Chapter 10

E1. $$\begin{pmatrix} y_1' \\ y_2' \\ y_3' \end{pmatrix} \;=\; \begin{pmatrix} y_2 \\ y_3 \\ (f(x) - 4xy_3 - \sin y_1)/3 \end{pmatrix}\,.$$

E2. $\begin{pmatrix} y_1' \\ y_2' \end{pmatrix} = \begin{pmatrix} 1 \\ f(y_1, y_2) \end{pmatrix}$.

E3. Euler: $y_1 = 1 + x$, Heun: $y_1 = 1 + x + \frac{1}{2}\,x^2$,
Runge-Kutta: $y_1 = 1 + x + \frac{1}{2}\,x^2 + \frac{1}{6}\,x^3 + \frac{1}{24}\,x^4$.
Maclaurin: $y(x) = 1 + x + \frac{1}{2!}\,x^2 + \frac{1}{3!}\,x^3 + \frac{1}{4!}\,x^4 + \cdots$.

E4. (a) Solve the system $Ar_1^0 + Br_2^0 = y_0$, $Ar_1^1 + Br_2^1 = y_1$.
(b) $A = 0.947214$, $B = 0.0527864$.
(c) Use Maclaurin series for $r_1 = h\lambda + \sqrt{h^2\lambda^2 + 1}$ and $y(h) = e^{\lambda h}$. $B = 0$, so the "spurious term" Br_2^n disappears.

E5. $(1 - \frac{1}{2}\,hp_n)y_{n-1} - (2 + h^2 q_n)y_n + (1 + \frac{1}{2}\,hp_n)y_{n+1} = 0$.

C1. With $y_1 = 1 - 100h$ the "spurious term" Br_2^n dominates almost from the start. With $y_1 = r_1$ we have $B = 0$, cf Exercise E4, but rounding errors will result in nonzero contributions. For $x \gtrsim 0.35$ this term explodes.

C2. With air resistance: $h_{\max} = 2.80 \cdot 10^3$ $(2.86 \cdot 10^3)$. Without air resistance: $h_{\max} = 1.25 \cdot 10^4$ $(1.34 \cdot 10^4)$. The results in parenthesis are obtained with default setting of the tolerance parameters.

Index