# Contents

Chapter 1

# Principles of Numerical Calculations

## 1.1 Introduction

Although mathematics has been used for centuries in one form or another within many areas of science and industry, modern scientific computing using electronic computers has its origin in research and developments during the second world war. In the late forties and early fifties the foundation of numerical analysis was laid as a separate discipline of mathematics. The new capabilities of performing millions of operations led to new classes of algorithms, which needed a careful analysis to ensure their accuracy and stability.

Recent modern development has increased enormously the scope for using numerical methods. Not only has this been caused by the continuing advent of faster computers with larger memories. Gain in problem solving capabilities through better mathematical algorithms have in many cases played an equally important role! This has meant that today one can treat much more complex and less simplified problems through massive amounts of numerical calculations. This development has caused the always close interaction between mathematics on the one hand and science and technology on the other to increase tremendously during the last decades. Advanced mathematical models and methods are now used more and more also in areas like medicine, economics and social sciences. It is fair to say that today experiment and theory, the two classical elements of scientific method, in many fields of science and engineering are supplemented in many areas by computations as an equally important component.

As a rule, applications lead to mathematical problems which in their complete form cannot be conveniently solved with exact formulas unless one restricts oneself to special cases or simplified models which can be exactly analyzed. In many cases, one thereby reduces the problem to a linear problem—for example, a linear system of equations or a linear differential equation. Such an approach can quite often lead to concepts and points of view which can, at least qualitatively, be used even in the unreduced problems.

In most numerical methods one applies a small number of general and rela-

tively simple ideas. These are then combined in an inventive way with one another and with such knowledge of the given problem as one can obtain in other ways—for example, with the methods of mathematical analysis. Some knowledge of the background of the problem is also of value; among other things, one should take into account the order of magnitude of certain numerical data of the problem.

In this chapter we shall illustrate the use of some general ideas behind numerical methods on some simple problems which may occur as subproblems or computational details of larger problems, though as a rule they occur in a less pure form and on a larger scale than they do here. When we present and analyze numerical methods, we use to some degree the same approach which was described first above: we study in detail special cases and simplified situations, with the aim of uncovering more generally applicable concepts and points of view which can guide in more difficult problems.

It is important to have in mind that the success of the methods presented depends on the smoothness properties of the functions involved. In this first survey we shall tacitly assume that the functions have as many well-behaved derivatives as is needed.

### 1.1.1 Fixed-Point Iteration

One of the most frequently recurring ideas in many contexts is **iteration** (from the Latin *iteratio*, "repetition") or **successive approximation**. Taken generally, iteration means the repetition of a pattern of action or process. Iteration in this sense occurs, for example, in the repeated application of a numerical process—perhaps very complicated and itself containing many instances of the use of iteration in the somewhat narrower sense to be described below—in order to improve previous results. To illustrate a more specific use of the idea of iteration, we consider the problem of solving a nonlinear equation of the form

$$x = F(x), \tag{1.1.1}$$

where $F$ is assumed to be a differentiable function whose value can be computed for any given value of a real variable $x$, within a certain interval. Using the method of iteration, one starts with an initial approximation $x_0$, and computes the sequence

$$x_1 = F(x_0), \qquad x_2 = F(x_1), \qquad x_3 = F(x_2), \ldots \tag{1.1.2}$$

Each computation of the type $x_{n+1} = F(x_n)$ is called an iteration. If the sequence $\{x_n\}$ converges to a limiting value $\alpha$ then we have

$$\alpha = \lim_{n\to\infty} x_{n+1} = \lim_{n\to\infty} F(x_n) = F(\alpha),$$

so $x = \alpha$ satisfies the equation $x = F(x)$. As $n$ grows, we would like the numbers $x_n$ to be better and better estimates of the desired root. One then stops the iterations when sufficient accuracy has been attained.

A geometric interpretation is shown in Fig. 1.1.1. A root of Equation (1.1.1) is given by the abscissa (and ordinate) of an intersecting point of the curve $y = F(x)$

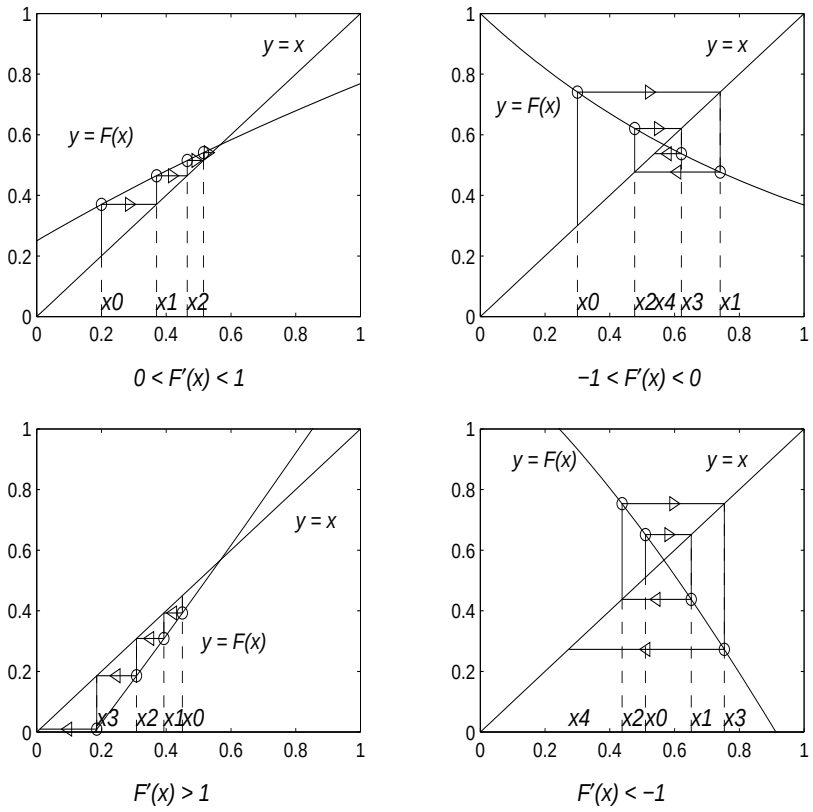



**Figure 1.1.1.** (a)–(d) *Geometric interpretation of iteration* $x_{n+1} = F(x_n)$.

and the line $y = x$. Using iteration and starting from $x_0$ we have $x_1 = F(x_0)$. The point $x_1$ on the $x$-axis is obtained by first drawing a horizontal line from the point $(x_0, F(x_0)) = (x_0, x_1)$ until it intersects the line $y = x$ in the point $(x_1, x_1)$ and from there drawing a vertical line to $(x_1, F(x_1)) = (x_1, x_2)$ and so on in a "staircase" pattern. In Fig. 1.1.1a it is obvious that $\{x_n\}$ converges monotonically to $\alpha$. Fig. 1.1.1b shows a case where $F$ is a decreasing function. There we also have convergence but not monotone convergence; the successive iterates $x_n$ are alternately to the right and to the left of the root $\alpha$.

There are also divergent cases, exemplified by Figs. 1.1.1c and 1.1.1d. One can see geometrically that the quantity, which determines the rate of convergence (or divergence), is the slope of the curve $y = F(x)$ in the neighborhood of the root. Indeed, from the mean value theorem we have

$$\frac{x_{n+1} - \alpha}{x_n - \alpha} = \frac{F(x_n) - F(\alpha)}{x_n - \alpha} = F'(\xi_n),$$

where $\xi_n$ lies between $x_n$ and $\alpha$. We see that, if $x_0$ is chosen sufficiently close to the root, (yet $x_0 \neq \alpha$), the iteration will converge if $|F'(\alpha)| < 1$. In this case $\alpha$ is

called a **point of attraction**. The convergence is faster the smaller $|F'(\alpha)|$ is. If $|F'(\alpha)| > 1$ then $\alpha$ is a **point of repulsion** and the iteration diverges.

**Example 1.1.1.**

A classical fast method for calculating square roots:

The equation $x^2 = c$ $(c > 0)$ can be written in the form $x = F(x)$, where $F(x) = \frac{1}{2}(x + c/x)$. If we set

$$x_0 > 0, \quad x_{n+1} = \tfrac{1}{2}(x_n + c/x_n),$$

then the $\alpha = \lim_{n\to\infty} x_n = \sqrt{c}$ (see Fig. 1.1.2)

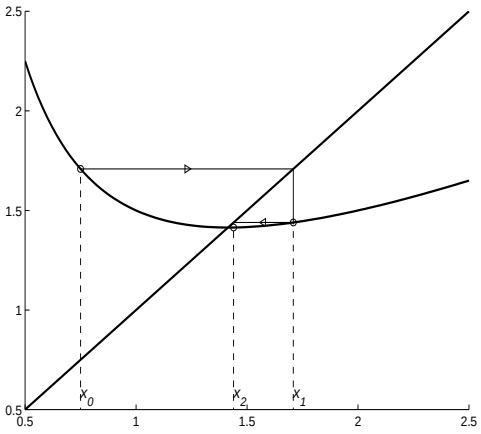



**Figure 1.1.2.** *The fix-point iteration* $x_n = (x_n + c/x_n)/2$, $c = 2$, $x_0 = 0.75$.

For $c = 2$, and $x_0 = 1.5$, we get $x_1 = \frac{1}{2}(1.5 + 2/1.5) = 1\frac{5}{12} = 1.4166666\ldots$, and

$$x_2 = 1.414215\,686274, \qquad x_3 = 1.414213\,562375,$$

which can be compared with $\sqrt{2} = 1.414213\,562373\ldots$ (correct to digits shown). As can be seen from Fig. 1.1.2 a rough value for $x_0$ suffices. The rapid convergence is due to the fact that for $\alpha = \sqrt{c}$ we have

$$F'(\alpha) = (1 - c/\alpha^2)/2 = 0.$$

One can in fact show that if $x_n$ has $t$ correct digits, then $x_{n+1}$ will have at least $2t - 1$ correct digits; see Example 6.3.3 and the following exercise. The above iteration method is used quite generally on both pocket calculators and computers for calculating square roots. The computation converges for any $x_0 > 0$.

Iteration is one of the most important aids for the practical as well as theoretical treatment of both linear and nonlinear problems. One very common application of iteration is to the solution of *systems of equations*. In this case $\{x_n\}$ is a sequence

of vectors, and $F$ is a vector-valued function. When iteration is applied to *differential equations* $\{x_n\}$ means a sequence of functions, and $F(x)$ means an expression in which integration or other operations on functions may be involved. A number of other variations on the very general idea of iteration will be given in later chapters.

The form of equation (1.1.1) is frequently called the **fixed point form**, since the root $\alpha$ is a fixed point of the mapping $F$. An equation may not be given originally in this form. One has a certain amount of choice in the rewriting of equation $f(x) = 0$ in fixed point form, and the rate of convergence depends very much on this choice. The equation $x^2 = c$ can also be written, for example, as $x = c/x$. The iteration formula $x_{n+1} = c/x_n$, however, gives a sequence which alternates between $x_0$ (for even $n$) and $c/x_0$ (for odd $n$)—the sequence does not even converge!

Let an equation be given in the form $f(x) = 0$, and for any $k \neq 0$, set

$$F(x) = x + kf(x).$$

Then the equation $x = F(x)$ is equivalent to the equation $f(x) = 0$. Since $F'(\alpha) = 1 + kf'(\alpha)$, we obtain the fastest convergence for $k = -1/f'(\alpha)$. Because $\alpha$ is not known, this cannot be applied literally. However, if we use $x_n$ as an approximation this leads to the choice $F(x) = x - f(x)/f'(x)$, or the iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{1.1.3}$$

This is the celebrated **Newton's method**.[1] (Occasionally this method is referred to as the Newton–Raphson method.) We shall derive it in another way below.

**Example 1.1.2.**

The equation $x^2 = c$ can be written in the form $f(x) = x^2 - c = 0$. Newton's method for this equation becomes

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n} = \frac{1}{2}\Big(x_n + \frac{c}{x_n}\Big),$$

which is the fast method in Example 1.1.1.

### 1.1.2 Linearization and Extrapolation

Another often recurring idea is that of **linearization**. This means that one *locally*, i.e. in a small neighborhood of a point, *approximates a more complicated function with a linear function.* We shall first illustrate the use of this idea in the solution of the equation $f(x) = 0$. Geometrically, this means that we are seeking the intersection point between the $x$-axis and the curve $y = f(x)$; see Fig. 1.1.3. Assume that

[1] Isaac Newton (1642–1727), English mathematician, astronomer and physicist, invented, independently of the German mathematician and philosopher Gottfried W. von Leibniz (1646–1716), the infinitesimal calculus. Newton, the Greek mathematician Archimedes (287–212 B.C.) and the German mathematician Carl Friedrich Gauss (1777–1883) gave pioneering contributions to numerical mathematics and to other sciences.

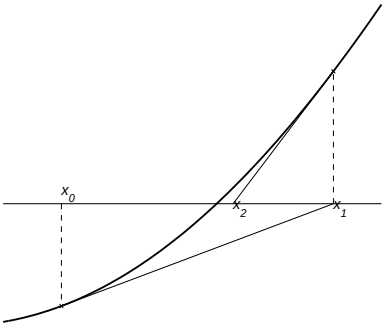



**Figure 1.1.3.** *Newton's method.*

we have an approximating value $x_0$ to the root. We then approximate the curve with its *tangent* at the point $(x_0, f(x_0))$. Let $x_1$ be the abscissa of the point of intersection between the $x$-axis and the tangent. Since the equation for the tangent reads

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

we obtain by setting $y = 0$, the approximation

$$x_1 = x_0 - f(x_0)/f'(x_0).$$

In many cases $x_1$ will have about twice as many correct digits as $x_0$. However, if $x_0$ is a poor approximation and $f(x)$ far from linear, then it is possible that $x_1$ will be a worse approximation than $x_0$.

If we combine the ideas of iteration and linearization, that is, we substitute $x_n$ for $x_0$ and $x_{n+1}$ for $x_1$, we rediscover Newton's method mentioned earlier. If $x_0$ is close enough to $\alpha$ the iterations will converge rapidly; see Fig. 1.1.3, but there are also cases of divergence.

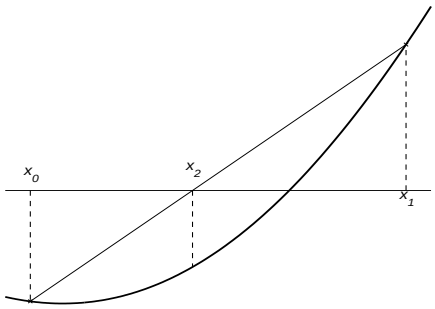



**Figure 1.1.4.** *The secant method.*

Another way, instead of drawing the tangent, to approximate a curve locally with a linear function is to choose two neighboring points on the curve and to approximate the curve with the *secant* which joins the two points; see Fig. 1.1.4. The

**secant method** for the solution of nonlinear equations is based on this approximation. This method, which preceded Newton's method, is discussed more closely in Sec. 6.4.1.

Newton's method can easily be generalized to solve a *system of nonlinear equations*

$$f_i(x_1, x_2, \ldots, x_n) = 0, \quad i = 1 : n.$$

or $f(x) = 0$, where $f$ and $x$ now are vectors in $\mathbf{R}^n$. Then $x_{n+1}$ is determined by the *system of linear equations*

$$f'(x_n)(x_{n+1} - x_n) = f(x_n), \tag{1.1.4}$$

where

$$f'(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \in \mathbf{R}^{n \times n}, \tag{1.1.5}$$

is the matrix of partial derivatives of $f$ with respect to $x$. This matrix is called the **Jacobian** of $f$ and often denoted by $J(x)$. System of nonlinear equations arise in many different contexts in scientific computing, e.g., in the solution of differential equations and optimization problems. We shall several times, in later chapters, return to this fundamental concept.

The secant approximation is useful in many other contexts. It is, for instance, generally used when one "reads between the lines" or interpolates in a table of numerical values. In this case the secant approximation is called **linear interpolation**. When the secant approximation is used in **numerical integration**, that is in the approximate calculation of a definite integral,

$$I = \int_a^b y(x)\, dx, \tag{1.1.6}$$

(see Fig. 1.1.5) it is called the **trapezoidal rule**. With this method, the area between the curve $y = y(x)$ and the $x$-axis is approximated with the sum $T(h)$ of the areas of a series of parallel trapezoids.

Using the notation of Fig. 1.1.5, we have

$$T(h) = h\frac{1}{2}\sum_{i=0}^{n-1}(y_i + y_{i+1}), \quad h = \frac{b-a}{n}. \tag{1.1.7}$$

(In the figure, $n = 4$.) We shall show in a later chapter that the error is very nearly proportional to $h^2$ when $h$ is small. One can then, in principle, attain arbitrary high accuracy by choosing $h$ sufficiently small. However, the computational work involved is roughly proportional to the number of points where $y(x)$ must be computed, and thus inversely proportional to $h$. Thus the computational work grows rapidly as one demands higher accuracy (smaller $h$).

Numerical integration is a fairly common problem because in fact it is quite seldom that the "primitive" function can be analytically calculated in a finite expression containing only elementary functions. It is not possible, for example, for

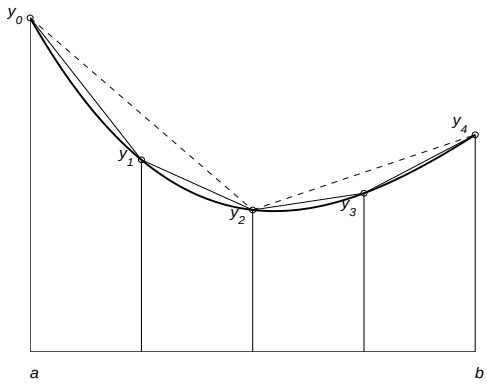



**Figure 1.1.5.** *Numerical integration by the trapezoidal rule ($n = 4$).*

such simple functions as $e^{x^2}$ or $(\sin x)/x$. In order to obtain *higher accuracy* with significant less work than the trapezoidal rule requires, one can use one of the following two important ideas:

(a) **Local approximation** of the integrand with a polynomial of higher degree, or with a function of some other class, for which one knows the primitive function.

(b) Computation with the trapezoidal rule for several values of $h$ and then extrapolation to $h = 0$, so-called **Richardson extrapolation**[2] or **the deferred approach to the limit**, with the use of general results concerning the dependence of the error on $h$.

The technical details for the various ways of approximating a function with a polynomial, among others Taylor expansions, interpolation, and the method of least squares, are treated in later chapters.

The extrapolation to the limit can easily be applied to numerical integration with the trapezoidal rule. As was mentioned previously, the trapezoidal approximation (1.1.7) to the integral has an error approximately proportional to the square of the step size. Thus, using two step sizes, $h$ and $2h$, one has:

$$T(h) - I \approx kh^2, \qquad T(2h) - I \approx k(2h)^2,$$

and hence $4(T(h) - I) \approx T(2h) - I$, from which it follows that

$$I \approx \tfrac{1}{3}(4T(h) - T(2h)) = T(h) + \tfrac{1}{3}(T(h) - T(2h)).$$

Thus, by adding the corrective term $\frac{1}{3}(T(h) - T(2h))$ to $T(h)$, one should get an estimate of $I$ which typically is far more accurate than $T(h)$. In Sec. 3.6 we shall see

[2]Lewis Fry Richardson (1881–1953) studied mathematics, physics, chemistry, botany and zoology. He graduated from King's College, Cambridge 1903. He was the first (1922) to attempt to apply the method of finite differences to weather prediction, long before the computer age!

that the improvements is in most cases quite striking. The result of the Richardson extrapolation is in this case equivalent to the classical **Simpson's rule**[3] for numerical integration, which we shall encounter many times in this volume. It can be derived in several different ways. Sec. 3.6 also contains application of extrapolation to other problems than numerical integration, as well as a further development of the extrapolation idea, namely **repeated Richardson extrapolation**. In numerical integration this is also known as **Romberg's method**.

Knowledge of the behavior of the error can, together with the idea of extrapolation, lead to a powerful method for improving results. Such a line of reasoning is useful not only for the common problem of numerical integration, but also in many other types of problems.

**Example 1.1.3.**

The integral $\int_{10}^{12} f(x)\,dx$ is computed for $f(x) = x^3$ by the trapezoidal method. With $h = 1$ we obtain

$$T(h) = 2,695, \qquad T(2h) = 2,728,$$

and extrapolation gives $T = 2.684$, equal to the exact result. Similarly, for $f(x) = x^4$ we obtain

$$T(h) = 30,009, \qquad T(2h) = 30,736,$$

and with extrapolation $T = 29,766.7$ (exact $29,766.4$).

### 1.1.3 Finite Difference Approximations

The local approximation of a complicated function by a linear function leads to another frequently encountered idea in the construction of numerical methods, namely the approximation of a derivative by a difference quotient. Fig. 1.1.6 shows the graph of a function $y(x)$ in the interval $[x_{n-1}, x_{n+1}]$ where $x_{n+1} - x_n = x_n - x_{n-1} = h$; $h$ is called the step size. If we set $y_i = y(x_i)$, $i = n-1, n, n+1$, then the derivative at $x_n$ can be approximated by a **forward difference quotient**,

$$y'(x_n) \approx \frac{y_{n+1} - y_n}{h}, \tag{1.1.8}$$

or a similar backward difference quotient involving $y_n$ and $y_{n-1}$. The error in the approximation is called a **discretization error**.

However, it is conceivable that the **centered difference approximation**

$$y'(x_n) \approx \frac{y_{n+1} - y_{n-1}}{2h} \tag{1.1.9}$$

will usually be more accurate. It is in fact easy to motivate this. By Taylor's formula,

$$y(x+h) - y(x) = y'(x)h + y''(x)h^2/2 + y'''(x)h^3/6 + \dots \tag{1.1.10}$$

$$-y(x-h) + y(x) = y'(x)h - y''(x)h^2/2 + y'''(x)h^3/6 - \dots \tag{1.1.11}$$

[3] Thomas Simpson (1710–1761), English mathematician best remembered for his work on interpolation and numerical methods of integration. He taught mathematics privately in the London coffee–houses and from 1737 began to write texts on mathematics.

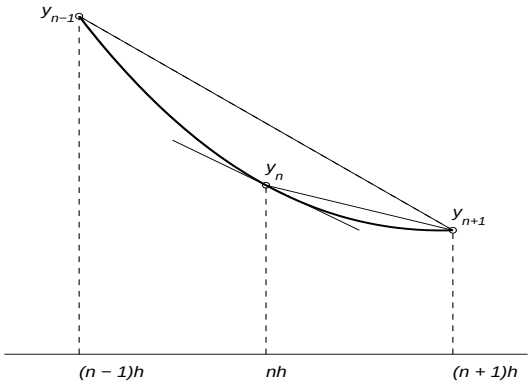



**Figure 1.1.6.** *Finite difference quotients.*

Set $x = x_n$. Then, by the first of these equations,

$$y'(x_n) = \frac{y_{n+1} - y_n}{h} - \frac{h}{2}y''(x_n) + \dots$$

Next, add the two Taylor expansions and divide by $2h$. Then the first error term cancels and we have

$$y'(x_n) = \frac{y_{n+1} - y_{n-1}}{2h} + \frac{h^2}{6}y'''(x_n) + \dots$$

We shall in the sequel call a formula (or a method), where a step size parameter $h$ is involved, **accurate of order** $p$, if its error is approximately proportional to $h^p$. Since $y''(x)$ vanishes for all $x$ if and only if $y$ is a linear function of $x$, and similarly, $y'''(x)$ vanishes for all $x$ if and only if $y$ is a quadratic function, we have established the following important result:

**Lemma 1.1.1.** *The forward difference approximation* (1.1.8) *is exact only for a linear function, and it is only first order accurate in the general case. The centered difference approximation* (1.1.9) *is exact also for a quadratic function, and is second order accurate in the general case.*

For the above reason the approximation (1.1.9) is, in most situations, preferable to (1.1.8). However, there are situations when these formulas are applied to the approximate solution of differential equations where the forward difference approximation suffices, but where the centered difference quotient is entirely unusable, for reasons which have to do with how errors are propagated to later stages in the calculation. We shall not discuss it more closely here, but mention it only to intimate some of the surprising and fascinating mathematical questions which can arise in the study of numerical methods.

Higher derivatives are approximated with **higher differences**, that is, differ-

ences of differences, another central concept in numerical calculations. We define:

$$\begin{aligned}(\Delta y)_n &= y_{n+1} - y_n;\\ (\Delta^2 y)_n = (\Delta(\Delta y))_n &= (y_{n+2} - y_{n+1}) - (y_{n+1} - y_n)\\ &= y_{n+2} - 2y_{n+1} + y_n;\\ (\Delta^3 y)_n = (\Delta(\Delta^2 y))_n &= y_{n+3} - 3y_{n+2} + 3y_{n+1} - y_n;\end{aligned}$$

etc. For simplicity one often omits the parentheses and writes, for example, $\Delta^2 y_5$ instead of $(\Delta^2 y)_5$. The coefficients that appear here in the expressions for the higher differences are, by the way, the binomial coefficients. In addition, if we denote the step length by $\Delta x$ instead of by $h$, we get the following formulas, which are easily remembered:

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x}, \qquad \frac{d^2y}{dx^2} \approx \frac{\Delta^2 y}{(\Delta x)^2}, \tag{1.1.12}$$

etc. Each of these approximations is second order accurate for the value of the derivative at an $x$ which equals the *mean value* of the largest and smallest $x$ for which the corresponding value of $y$ is used in the computation of the difference. (The formulas are only first order accurate when regarded as approximations to derivatives at other points between these bounds.) These statements can be established by arguments similar to the motivation for the formulas (1.1.8) and (1.1.9).

Taking the difference of the Taylor expansions (1.1.10)–(1.1.11) with one more term in each, and dividing by $h^2$ we obtain the following important formula

$$y''(x_n) = \frac{y_{n+1} - 2y_n + y_{n-1}}{h^2} - \frac{h^2}{12}y^{iv}(x_n) + \cdots,$$

Introducing the **central difference operator**

$$\delta y_n = \left(x_n + \tfrac{1}{2}h\right) - y\left(x_n - \tfrac{1}{2}h\right), \tag{1.1.13}$$

and neglecting higher order terms we get

$$y''(x_n) \approx \frac{1}{h^2}\delta^2 y_n - \frac{h^2}{12}y^{iv}(x_n). \tag{1.1.14}$$

The approximation of equation (1.1.9) can be interpreted as an application of (1.1.12) with $\Delta x = 2h$, or else as the mean of the estimates which one gets according to equation (1.1.12) for $y'((n + \frac{1}{2})h)$ and $y'((n - \frac{1}{2})h)$.

When the values of the function have errors, for example, when they are rounded numbers, the difference quotients become more and more uncertain the less $h$ is. Thus if one wishes to compute the derivatives of a function given by a table, one should as a rule use a step length which is greater than the table step.

**Example 1.1.4.**

For $y = \cos x$ one has, using function values correct to six decimal digits:

This arrangement of the numbers is called a **difference scheme**. Note that the differences are expressed in units of $10^{-6}$. Using (1.1.9) and (1.1.12) one gets

$$\begin{aligned}y'(0.60) &\approx (0.819648 - 0.830941)/0.02 = -0.56465,\\ y''(0.60) &\approx -83 \cdot 10^{-6}/(0.01)^2 = -0.83.\end{aligned}$$

| $x$ | $y$ | $\Delta y$ | $\Delta^2 y$ |
|---|---|---|---|
| 0.59 | 0.830941 | | |
| | | -5605 | |
| 0.60 | 0.825336 | | -83 |
| | | -5688 | |
| 0.61 | 0.819648 | | |

The correct results are, with six decimals,

$$y'(0.60) = -0.564642, \quad y''(0.60) = -0.825336.$$

In $y''$ we only got two correct decimal digits. This is due to **cancellation**, which is an important cause of loss of accuracy; see further Sec. 2.2.3. Better accuracy can be achieved by *increasing* the step $h$; see Problem 5 at the end of this section.

Finite difference approximations are useful for partial derivatives too. Suppose that the values $u_{i,j} = u(x_i, y_j)$ of a function $u(x, y)$ are given on a square grid with grid size $h$, i.e. $x_i = x_0 + ih$, $y_j = y_0 + jh$, $0 \le i \le M$, $0 \le j \le N$ that covers a rectangle. A very important equation of Mathematical Physics is **Poisson's equation**:[4]

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \tag{1.1.15}$$

where $f(x, y)$ is a given function. Under certain conditions, gravitational, electric, magnetic, and velocity potentials satisfy **Laplace equation**[5], which is (1.1.15) with $f(x, y) = 0$. By (1.1.14), a second order accurate approximation of Poisson's equation is given by

$$\begin{aligned} &\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} \\ &\quad = \frac{1}{h^2}\big(u_{i,j+1} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} - 4u_{i,j}\big) = f_{i,j}. \end{aligned}$$

This corresponds to the "computational molecule"

$$\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}$$

## Review Questions

**1.** Make lists of the concepts and ideas which have been introduced. Review their use in the various types of problems mentioned.

[4] Siméon Denis Poisson (1781–1840).
[5] Pierre Simon, Marquis de Laplace (1749–1827).

**2.** Discuss the convergence condition and the rate of convergence of the method of iteration for solving $x = F(x)$.

**3.** What is the trapezoidal rule? What is said about the dependence of its error on the step length?

## Problems and Computer Exercises

**1.** Calculate $\sqrt{10}$ to seven decimal places using the method in Example 1.1.1. Begin with $x_0 = 2$.

**2.** Consider $f(x) = x^3 - 2x - 5$. The cubic equation $f(x) = 0$ has been a standard test problem, since Newton used it in 1669 to demonstrate his method. By computing (say) $f(x)$ for $x = 1, 2, 3$, we see that $x = 2$ probably is a rather good initial guess. Iterate then by Newton's method until you trust that the result is correct to six decimal places.

**3.** The equation $x^3 - x = 0$ has three roots, $-1, 0, 1$. We shall study the behaviour of Newton's method on this equation, with the notations used in §1.1.2 and Fig. 1.1.3.

(a) What happens if $x_0 = 1/\sqrt{3}$ ? Show that $x_n$ converges to 1 for any $x_0 > 1/\sqrt{3}$. What is the analogous result for convergence to $-1$?

(b) What happens if $x_0 = 1/\sqrt{5}$? Show that $x_n$ converges to 0 for any $x_0 \in (-1/\sqrt{5},\ 1/\sqrt{5})$.

*Hint:* Show first that if $x_0 \in (0,\ 1/\sqrt{5})$ then $x_1 \in (-x_0, 0)$. What can then be said about $x_2$?

(c) Find, by a drawing (with paper and pencil), $\lim x_n$ if $x_0$ is a little less than $1/\sqrt{3}$. Find by computation $\lim x_n$ if $x_0 = 0.46$.

*(d) A complete discussion of the question in (c) is rather complicated, but there is an implicit recurrence relation that produces a decreasing sequence $\{a_1 = 1/\sqrt{3},\ a_2,\ a_3, \ldots\}$, by means of which you can easily find $\lim_{n\to\infty} x_n$ for any $x_0 \in (1/\sqrt{5},\ 1/\sqrt{3})$. Try to find this recurrence.

Answer: $a_i - f(a_i)/f'(a_i) = -a_{i-1}$; $\lim_{n\to\infty} x_n = (-1)^i$ if $x_0 \in (a_i, a_{i+1})$; $a_1 = 0.577$, $a_2 = 0.462$, $a_3 = 0.450$, $a_4 \approx \lim_{i\to\infty} a_i = 1/\sqrt{5} = 0.447$.

**4.** Calculate $\int_0^{1/2} e^x\, dx$

(a) to six decimals using the primitive function.

(b) with the trapezoidal rule, using step length $h = 1/4$.

(c) using Richardson extrapolation to $h = 0$ on the results using step length $h = 1/2$, and $h = 1/4$.

(d) Compute the ratio between the error in the result in (c) to that of (b).

**5.** In Example 1.1.4 we computed $y''(0.6)$ for $y = \cos(x)$, with step length $h = 0.01$. Make similar calculations using $h = 0.1$, $h = 0.05$ and $h = 0.001$. Which value of $h$ gives the best result, using values of $y$ to six decimal places? Discuss qualitatively the influences of both the rounding errors in the function values

and the error in the approximation of a derivative with a difference quotient on the result for various values of $h$.

## 1.2 Some Numerical Algorithms

For a given numerical problem one can consider many different algorithms. These can differ in efficiency and reliability and give approximate answers sometimes with widely varying accuracy. In the following we give a few examples of how algorithms can be developed to solve some typical numerical problems.

### 1.2.1 Recurrence Relations

One of the most important and interesting parts of the preparation of a problem for a computer is to *find a recursive description of the task*. Often an enormous amount of computation can be described by a small set of recurrence relations. Euler's method for the step-by-step solution of ordinary differential equations is an example. Other examples will be given in this section; see also problems at the end of this section.

A common computational task is the evaluation of a polynomial, at a given point $x$ where, say,

$$p(x) = a_0x^3 + a_1x^2 + a_2x + a_3 = ((a_0x + a_1)x + a_2)x + a_3.$$

We set $b_0 = a_0$, and compute

$$b_1 = b_0x + a_1, \quad b_2 = b_1x + a_2, \quad p(x) = b_3 = b_2x + a_3.$$

This illustrates, for $n = 3$, **Horner's rule** for evaluating a polynomial of degree $n$,

$$p(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n,$$

This algorithm can be described by the recurrence relation:

$$b_0 = a_0, \quad b_i = b_{i-1}x + a_i, \quad i = 1 : n, \tag{1.2.1}$$

where $p(x) = b_n$.

The quantities $b_i$ in (1.2.1) are of intrinsic interest because of the following result, often called **synthetic division**:

$$\frac{p(x) - p(z)}{x - z} = \sum_{i=0}^{n-1} b_i x^{n-1-i}, \tag{1.2.2}$$

where the $b_i$ are defined by (1.2.1). The proof of this result is left as an exercise. Synthetic division is used, for instance, in the solution of algebraic equations, when already computed roots are successively eliminated. After each elimination, one can deal with an equation of lower degree. This process is called **deflation**see Sec. 6.5.5. . (As shown in Sec. 6.6.4, some care is necessary in the numerical application of this idea.)

The proof of the following useful relation is left as an exercise to the reader:

**Lemma 1.2.1.**

Let the $b_i$ be defined by (1.2.1) and

$$c_0 = b_0, \qquad c_i = b_i + zc_{i-1}, \quad i = 1 : n-1. \tag{1.2.3}$$

Then $p'(z) = c_{n-1}$.

Recurrence relations are among the most valuable aids in numerical calculation. Very extensive calculations can be specified in relatively short computer programs with the help of such formulas. However, unless used in the right way errors can grow exponentially and completely ruin the results.

**Example 1.2.1.**

To compute the integrals $I_n = \int_0^1 \frac{x^n}{x+5}\,dx$, $i = 1 : N$ one can use the recurrence relation

$$I_n + 5I_{n-1} = 1/n, \tag{1.2.4}$$

which follows from

$$I_n + 5I_{n-1} = \int_0^1 \frac{x^n + 5x^{n-1}}{x+5}\,dx = \int_0^1 x^{n-1}\,dx = \frac{1}{n}.$$

Below we use this formula to compute $I_8$, using six decimals throughout. For $n = 0$ we have

$$I_0 = [\ln(x+5)]_0^1 = \ln 6 - \ln 5 = 0.182322.$$

Using the recurrence relation we get

$$\begin{aligned}
I_1 &= 1 - 5I_0 = 1 - 0.911610 = 0.088390,\\
I_2 &= 1/2 - 5I_1 = 0.500000 - 0.441950 = 0.058050,\\
I_3 &= 1/3 - 5I_2 = 0.333333 - 0.290250 = 0.043083,\\
I_4 &= 1/4 - 5I_3 = 0.250000 - 0.215415 = 0.034585,\\
I_5 &= 1/5 - 5I_4 = 0.200000 - 0.172925 = 0.027075,\\
I_6 &= 1/6 - 5I_5 = 0.166667 - 0.135375 = 0.031292,\\
I_7 &= 1/7 - 5I_6 = 0.142857 - 0.156460 = -0.013603.
\end{aligned}$$

It is strange that $I_6 > I_5$, and obviously absurd that $I_7 < 0$! The reason for the absurd result is that the round-off error $\epsilon$ in $I_0 = 0.18232156\ldots$, whose magnitude is about $0.44 \cdot 10^{-6}$ is *multiplied* by $(-5)$ in the calculation of $I_1$, which then has an error of $-5\epsilon$. That error produces an error in $I_2$ of $5^2\epsilon$, etc. Thus the magnitude of the error in $I_7$ is $5^7\epsilon = 0.0391$, which is larger than the true value of $I_7$. On top of this comes the round-off errors committed in the various steps of the calculation. These can be shown in this case to be relatively unimportant.

If one uses higher precision, the absurd result will show up at a later stage. For example, a computer that works with a precision corresponding to about 16

decimal places, gave a negative value to $I_{22}$ although $I_0$ had full accuracy. The above algorithm is an example of a disagreeable phenomenon, called **numerical instability**.

We now show how, in this case, one can avoid numerical instability by choosing a more suitable algorithm.

**Example 1.2.2.**

We shall here use the recurrence relation in the other direction,

$$I_{n-1} = (1/n - I_n)/5. \tag{1.2.5}$$

Now the errors will be *divided* by $-5$ in each step. But we need a starting value. We can directly see from the definition that $I_n$ decreases as $n$ increases. One can also surmise that $I_n$ decreases slowly when $n$ is large (the reader is recommended to motivate this). Thus we try setting $I_{12} = I_{11}$. It then follows that

$$I_{11} + 5I_{11} \approx 1/12, \qquad I_{11} \approx 1/72 \approx 0.013889.$$

(show that $0 < I_{12} < 1/72 < I_{11}$). Using the recurrence relation we get

$$I_{10} = (1/11 - 0.013889)/5 = 0.015404, \qquad I_9 = (1/10 - 0.015404)/5 = 0.016919,$$

and further

$$\begin{aligned} &I_8 = 0.018838, \quad I_7 = 0.021232, \quad I_6 = 0.024325, \quad I_5 = 0.028468, \\ &I_4 = 0.034306, \quad I_3 = 0.043139, \quad I_2 = 0.058039, \quad I_1 = 0.088392, \end{aligned}$$

and finally $I_0 = 0.182322$. Correct!

If we instead simply take as starting value $I_{12} = 0$, one gets $I_{11} = 0.016667$, $I_{10} = 0.018889$, $I_9 = 0,016222$, $I_8 = 0.018978$, $I_7 = 0.021204$, $I_6 = 0.024331$, and $I_5, \ldots, I_0$ have the same values as above. The difference in the values for $I_{11}$ is 0.002778. The subsequent values of $I_{10}, I_9, \ldots, I_0$ are quite close *because the error is divided by -5 in each step.* The results for $I_n$ obtained above have errors which are less than $10^{-3}$ for $n \leq 8$.

The reader is warned, however, not to draw erroneous conclusions from the above example. The use of a recurrence relation "backwards" is not a universal recipe as will be seen later on! Compare also Problems 6 and 7 at the end of this section.

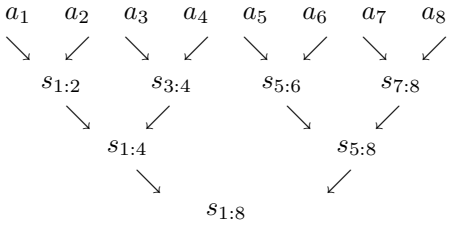
### 1.2.2 Divide and Conquer Strategy

A powerful strategy for solving large scale problems is the **divide and conquer** strategy. The idea is to split a high dimensional problem into problems of lower dimension. Each of these are then again split into smaller subproblems, etc., until a number of sufficiently small problems are obtained. The solution of the initial problem is then obtained by combining the solution of the subproblems working backwards in the hierarchy.

We illustrate the idea on the computation of the sum $s = \sum_{i=1}^{n} a_i$. The usual way to proceed is to use the recursion

$$s_0 = 0, \qquad s_i = s_{i-1} + a_i, \quad i = 1 : n.$$

Another order of summation is as illustrated below for $n = 2^3 = 8$:




where $s_{i,j} = a_i + \cdots + a_j$. In this table each new entry is obtained by adding its two neighbors in the row above. Clearly this can be generalized to compute an arbitrary sum of $n = 2^k$ terms in $k$ steps. In the first step we perform $n/2$ sums of two terms, then $n/4$ partial sums each of four terms, etc., until in the $k$th step we compute the final sum.

This summation algorithm uses the same number of additions as the first one. However, it has the advantage that it splits the task in *several subtasks that can be performed in parallel*. For large values of $n$ this summation order can also be much more accurate than the conventional order (see Problem 2.3.5, Chapter 2). Espelid [9] gives an interesting discussion of such summation algorithms.

The algorithm can also be described in another way. Consider the following definition of a summation algorithm for computing the $s(i, j) = a_i + \cdots + a_j$, $j > i$:

$sum = s(i, j)$;
**if** $j = i + 1$ **then** $sum = a_i + a_j$;
  **else** $k = \lfloor (i + j)/2 \rfloor$; $sum = s(i, k) + s(k + 1, j)$;
**end**

(Here and in the following $\lfloor x \rfloor$ denotes the **floor** of $x$, i.e. the largest integer $\leq x$. Similarly, $\lceil x \rceil$ denotes the **ceiling** of $x$, i.e. the smallest integer $\geq x$.) This function defines $s(i, j)$ in a recursive way; if the sum consists of only two terms then we add them and return with the answer. Otherwise we split the sum in two and use the function again to evaluate the corresponding two partial sums. This approach is aptly called the divide and conquer strategy. The function above is an example of a **recursive algorithm**—it calls itself. Many computer languages (e.g., MATLAB ) allow the definition of such recursive algorithms. The divide and conquer is a **top down** description of the algorithm in contrast to the **bottom up** description we gave first.

There are many other less trivial examples of the power of the divide and conquer approach. It underlies the Fast Fourier Transform and leads to efficient implementations of, for example, matrix multiplication, Cholesky factorization, and other matrix factorizations. Interest in such implementations have increased lately since it has been realized that they achieve very efficient automatic parallelization of many tasks.

### 1.2.3 Approximation of Functions

Many important function in applied mathematics cannot be expressed in finite terms of elementary functions, and must be approximated by numerical methods. Examples from statistics are the normal probability function, the chi-square distribution function, the exponential integral, and the Poisson distribution. These can, by simple transformations, be brought to particular cases of the **incomplete gamma function**

$$\gamma(a,z) = \int_0^z e^{-t}t^{a-1}dt, \quad \Re a > 0, \tag{1.2.6}$$

A collection of formulas that can be used to evaluate this function is found in Abramowitz and Stegun [1, Sec. 6.5]. Codes and some theoretical background are given in Numerical Recipes [31, Sec. 6.2–6.3].

**Example 1.2.3.**

As a simple example we consider evaluating the **error function** defined by

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt, \tag{1.2.7}$$

for $x \in [-1, 1]$. This function is encountered in computing the distribution function of a normal deviate. It takes the values $\operatorname{erf}(0) = 0$, $\operatorname{erf}(\infty) = 1$.

Suppose one wishes to compute $\operatorname{erf}(x)$ for $x \in [-1, 1]$ with a relative error less than $10^{-8}$. One can then approximate the function by a power series. Setting $z = -t^2$ in the well known Maclaurin series for $e^z$, truncating after $n + 1$ terms, and integrating term by term we obtain

$$erf(x) \approx \frac{2}{\sqrt{\pi}} \int_0^x \sum_{j=0}^{n} (-1)^j \frac{t^{2j}}{j!}\, dt = \frac{2}{\sqrt{\pi}} \sum_{j=0}^{n} a_j x^{2j+1}, \tag{1.2.8}$$

where

$$a_0 = 1, \quad a_j = \frac{(-1)^j}{j!(2j+1)}.$$

(Note that $\operatorname{erf}(x)$ is a odd function of $x$.) This series converges for all $x$, but is suitable for numerical computations only for values of $x$ which are not too large. To evaluate the series we note that the coefficients $a_j$ satisfies the recurrence relation

$$a_j = -a_{j-1}\frac{(2j-1)}{j(2j+1)}.$$

This recursion shows that for $x \in [0, 1]$ the absolute values of the terms $t_j = a_j x^{2j+1}$ decrease monotonically. This implies that the absolute error in a partial sum is bounded by the absolute value of the first neglected term. (Why? For an answer see Theorem 3.1.5 in Chapter 3.)

A possible algorithm for evaluating the sum in (1.2.8) is then:

Set $s_0 = t_0 = x$; for $j = 1, 2, \ldots$ compute

$$t_j = -t_{j-1}\frac{(2j-1)}{j(2j+1)}x^2, \quad s_j = s_{j-1} + t_j, \quad \text{until} \quad |t_j| \leq \cdot 10^{-8} s_j.$$

Here we have estimated the error by the last term added in the series. Since we have to compute this term for the error estimate we might as well use it! Note also that in this case, where the number of terms is fixed in advance, Horner's scheme is not suitable for the evaluation. Fig. 1.2.1 shows the graph of the relative error

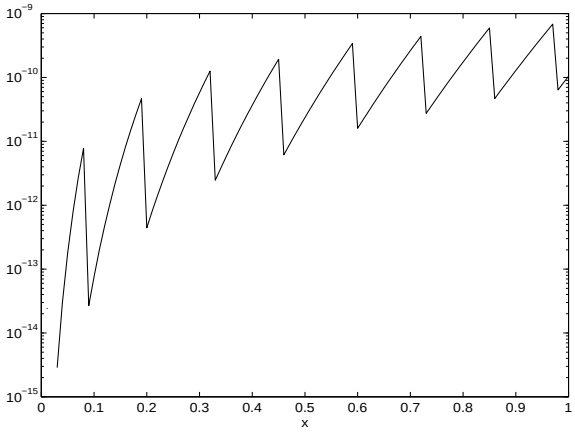



**Figure 1.2.1.** *Relative error* $e(x) = |p_{2n+1}(x) - \text{erf}(x)|/\text{erf}(x)$.

in the computed approximation $p_{2n+1}(x)$. At most twelve terms in the series were needed.

Clearly the "model" of approximating the error function with a polynomial is not exact, since the function demonstrably is not a polynomial. The error from truncating the series is called **truncation error**. In general the error due to replacing an infinite process by a finite is referred to as a truncation error. In the above example this error can be made as small as one wants by choosing the degree of the polynomial sufficiently large by taking more terms in the Maclaurin series.

The use of power series and rational approximations will be studied in depth in Chapter 3, where also other more efficient methods than the Maclaurin series for approximation by polynomials will be treated.

A different approximation problem, which occurs in many variants, is to approximate a function $f$ by a member $f^*$ of a class of functions which is easy to work with mathematically (e.g., polynomials, rational functions, or trigonometric polynomials), where each particular function in the class is specified by the numerical values of a number of parameters.

In computer aided design (CAD) curves and surfaces have to be represented mathematically, so that they can be manipulated and visualized easily. Important applications occur in aircraft and automotive industries. For this purpose **spline**

**functions** are now used extensively. The name **spline** comes from a very old technique in drawing smooth curves, in which a thin strip of wood, called a draftsman's spline, is bent so that it passes trough a given set of points. The points of interpolation are called **knots** and the spline is secured at the knots by means of lead weights called **ducks**. Before the computer age splines were used in ship building and other engineering designs.

Bézier curves, which can also be used for these purposes, were developed in 1962 by Bézier and de Casteljau, when working for the French car companies Renault and Citroën,

### 1.2.4 The Principle of Least Squares

In many applications a linear mathematical model is to be fitted to given observations. For example, consider a model described by a scalar function $y(t) = f(x,t)$, where $x \in \mathbf{R}^n$ is a parameter vector to be determined from measurements $(y_i, t_i)$, $i = 1 : m$. There are two types of shortcomings to take into account: errors in the input data, and shortcomings in the particular model (class of functions, form), which one intends to adopt to the input data. For ease in discussion. We shall call these **measurement errors** and **errors in the model**, respectively.

In order to reduce the influence of measurement errors in the observations one would like to use a greater number of measurements than the number of unknown parameters in the model. If $f(x,t)$ be *linear* in $x$ and of the form

$$f(x,t) = \sum_{j=1}^{n} x_j \phi_j(t).$$

Then the equations

$$y_i = \sum_{j=1}^{n} x_j \phi_j(t_i), \quad i = 1 : m,$$

form an overdetermined linear system $Ax = b$, where $a_{ij} = \phi_j(t_i)$ and $b_i = y_i$. The resulting problem is then to "solve" an **overdetermined** linear system of equations $Ax = b$. where $b \in \mathbf{R}^m$, $A \in \mathbf{R}^{m\times n}$ $(m > n)$. Thus we want to find a vector $x \in \mathbf{R}^n$ such that $Ax$ is the "best" approximation to $b$. We refer in the following to $r = b - Ax$ as the **residual vector**.

There are many possible ways of defining the "best" solution. A choice which can often be motivated for statistical reasons and which also leads to a simple computational problem is to take as solution a vector $x$, which minimizes the sum of the squared residuals, i.e.

$$\min_{x\in\mathbf{R}^n} \sum_{i=1}^{m} r_i^2, \tag{1.2.9}$$

The principle of least squares for solving an overdetermined linear system was first used by Gauss, who in 1801 used it to successively predicted the orbit of the asteroid Ceres. It can shown that the **least squares solution** satisfies the **normal equations**

$$A^T Ax = A^T b. \tag{1.2.10}$$

The matrix $A^T A$ is symmetric and can be shown to be nonsingular if $A$ has linearly independent columns, in which case $Ax = b$ has a *unique* least squares solution.

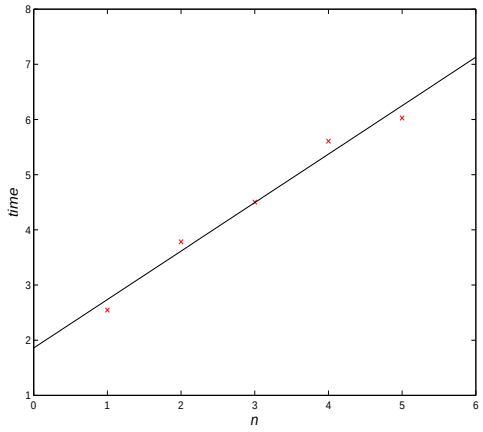



**Figure 1.2.2.** *Fitting a linear relation to observations.*

**Example 1.2.4.**

The points in Fig. 1.2.2 show for $n = 1 : 5$, the time $t_n$, for the $n$th passage of a swinging pendulum through its point of equilibrium. The condition of the experiment were such that a linear relation of the form $t = a + b\,n$ can be assumed to be valid. Random errors in measurement are the dominant cause of the deviation from linearity shown in Fig. 1.3.2. This deviation causes the values of the parameters $a$ and $b$ to be uncertain. The least squares fit to the model, shown by the straight line in Fig 1.2.2, minimizes the sum of squares of the deviations $\sum_{n=1}^{5}(a+b\,n-t_n)^2$.

**Example 1.2.5.**

The recently discovered comet 1968 Tentax is supposed to move within the solar system. The following observations of its position in a certain polar coordinate system have been made

| $r$ | 2.70 | 2.00 | 1.61 | 1.20 | 1.02 |
|---|---|---|---|---|---|
| $\phi$ | 48° | 67° | 83° | 108° | 126° |

By Kepler's first law the comet should move in a plane orbit of elliptic or hyperbolic form, if the perturbations from planets are neglected. Then the coordinates satisfy

$$r = p/(1 - e\cos\phi),$$

where $p$ is a parameter and $e$ the eccentricity. We want to estimate $p$ and $e$ by the method of least squares from the given observations.

We first note that if the relationship is rewritten as

$$1/p - (e/p)\cos\phi = 1/r,$$

it becomes linear in the parameters $x_1 = 1/p$ and $X_2 = e/p$. We then get the linear system $Ax = b$, where

$$A = \begin{pmatrix} 1.0000 & -0.6691 \\ 1.0000 & -0.3907 \\ 1.0000 & -0.1219 \\ 1.0000 & 0.3090 \\ 1.0000 & 0.5878 \end{pmatrix}, \qquad b = \begin{pmatrix} 0.3704 \\ 0.5000 \\ 0.6211 \\ 0.8333 \\ 0.9804 \end{pmatrix}.$$

The least squares solution is $x = (\,0.6886 \quad 0.4839\,)^T$ giving $p = 1/x_1 = 1.4522$ and finally $e = px_2 = 0.7027$.

In practice, both the measurements and the model are as a rule insufficient. One can also see approximation problems as analogous to the task of a communication engineer, to filter away *noise* from the *signal*. These questions are connected with both Mathematical Statistics and the mathematical discipline Approximation Theory.

## Review Questions

1. Describe Horner's rule and synthetic division.
2. Give a concise explanation why the algorithm in Example 1.2.1 did not work and why that in Example 1.2.2 did work.
3. Describe the idea behind the divide and conquer strategy. What is a main advantage of this strategy? How do you apply it to the task of summing $n$ numbers?
4. Describe the least squares principle for solving an overdetermined linear system.

## Problems and Computer Exercises

1. (a) Use Horner's scheme to compute $p(2)$ where

$$p(x) = x^4 + 2x^3 - 3x^2 + 2.$$

(b) Count the number of multiplications and additions required for the evaluation of a polynomial $p(z)$ of degree $n$ by Horner's rule. Compare with the work needed when the powers are calculated recursively by $x^i = x \cdot x^{i-1}$ and subsequently multiplied by $a_{n-i}$.

2. Show how repeated synthetic division can be used to move the origin of a polynomial, i.e., given $a_1,\ a_2, \ldots, a_n$ and $z$, find $c_1,\ c_2, \ldots, c_n$ so that

$$p_n(x) = \textstyle\sum_{j=1}^{n} a_j x^{j-1} \equiv \sum_{j=1}^{n} c_j (x - z)^{j-1}.$$

Write a program for synthetic division (with this ordering of the coefficients), and apply it to this algorithm.

*Hint:* Apply synthetic division to $p_n(x)$, $p_{n-1}(x) = (p_n(x) - p_n(z))/(x - z)$, etc.

**3.** (a) Show that the transformation made in Problem 2 can also be expressed by means of the matrix-vector equation,

$$c = \operatorname{diag}(z^{1-i})\, P \operatorname{diag}(z^{j-1})\, a,$$

where $a = [a_1, a_2, \ldots a_n]^T$, $c = [c_1, c_2, \ldots c_n]^T$, and $\operatorname{diag}(z^{j-1})$ is a diagonal matrix with the elements $z^{j-1}$, $j = 1 : n$. The matrix $P \in \mathbf{R}^{n\times n}$ has elements $p_{i,j} = \binom{j-1}{i-1}$, if $j \geq i$, else $p_{i,j} = 0$. By convention, $\binom{0}{0} = 1$ here.

(b) Note the relation of $P$ to the Pascal triangle, and show how $P$ can be generated by a simple recursion formula. Also show how each element of $P^{-1}$ can be expressed in terms of the corresponding element of $P$. How is the origin of the polynomial $p_n(x)$ moved, if you replace $P$ by $P^{-1}$ in the matrix-vector equation that defines $c$?

(c) If you reverse the order of the elements of the vectors $a$, $c$—this may sometimes be a more convenient ordering—how is the matrix $P$ changed?

*Comment:* With a terminology to be used much in this book (see Sec. 4.1.2), we can look upon $a$ and $c$ as different coordinate vectors for the same element in the $n$-dimensional linear space $\mathcal{P}_n$ of polynomials of degree *less than* $n$. The matrix $P$ gives the coordinate transformation.

**4.** Derive recurrence relations and write a program for computing the coefficients of the *product* $r$ of two polynomials $p$ and $q$,

$$r(x) = p(x)q(x) = \Big(\sum_{i=1}^{m} a_i x^{i-1}\Big)\Big(\sum_{j=1}^{n} b_j x^{j-1}\Big) = \sum_{k=1}^{m+n-1} c_k x^{k-1}.$$

**5.** Let $x, y$ be nonnegative integers, with $y \neq 0$. The division $x/y$ yields the quotient $q$ and the remainder $r$. Show that if $x$ and $y$ have a common factor, then that number is a divisor of $r$ as well. Use this remark to design an algorithm for the determination of the greatest common divisor of $x$ and $y$ (*Euclid's algorithm*).

**6.** Derive a forward and a backward recurrence relation for calculating the integrals

$$I_n = \int_0^1 \frac{x^n}{4x+1}\, dx.$$

Why is in this case the forward recurrence stable and the backward recurrence unstable?

**7.** (a) Solve Example 1.2.1 on a computer, with the following changes: Start the recursion (1.2.4) with $I_0 = \ln 1.2$, and compute and print the sequence $\{I_n\}$ until $I_n$ for the first time becomes negative.

(b) Start the recursion (1.2.5) first with the condition $I_{19} = I_{20}$, then with

$I_{29} = I_{30}$. Compare the results you obtain and assess their approximate accuracy. Compare also with the results of 7 (a).

***8.** (a) Write a program (or study some library program) for finding the quotient $Q(x)$ and the remainder $R(x)$ of two polynomials $A(x), B(x)$, i.e., $A(x) = Q(x)B(x) + R(x)$, $\deg R(x) < \deg B(x)$.

(b) Write a program (or study some library program) for finding the coefficients of a polynomial with given roots.

***9.** (a) Write a program (or study some library program) for finding the greatest common divisor of two *polynomials*. Test it on a number of polynomials of your own choice. Choose also some polynomials of a rather high degree, and do not only choose polynomials with small integer coefficients. Even if you have constructed the polynomials so that they should have a common divisor, rounding errors may disturb this, and some tolerance is needed in the decision whether a remainder is zero or not. One way of finding a suitable size of the tolerance is to make one or several runs where the coefficients are subject to some small random perturbations, and find out how much the results are changed.

(b) Apply the programs mentioned in the last two problems for finding and eliminating multiple zeros of a polynomial.

*Hint:* A multiple zero of a polynomial is a common zero of the polynomial and its derivative.

**10.** It is well known that $\text{erf}(x) \to 1$ as $x \to \infty$. If $x \gg 1$ the relative accuracy of the complement $1 - \text{erf}(x)$ is of interest. However, the series expansion used in Example 1.2.3 for $x \in [0, 1]$ is not suitable for large values of $x$. Why?

*Hint:* Derive an approximate expression for the largest term.

## 1.3 Matrix Computations

Matrix computations are ubiquitous in Scientific Computing. A survey of basic notations and concepts in matrix computations and linear vector spaces is given in Appendix A. This is needed for several topics treated in later chapters of this first volume. A fuller treatment of this topic will be given in Vol. II.

In this section we focus on some important developments since the 1950s in the solution of linear systems. One is the systematic use of matrix notations and the interpretation of Gaussian elimination as matrix factorization. This **decompositional approach** has several advantages, e.g, a computed factorization can often be used with great saving to solve new problems involving the original matrix. Another is the rapid developments of sophisticated iterative methods, which are becoming increasingly important as the size of systems increase.

### 1.3.1 Matrix Multiplication

A **matrix** $A$ is a collection of $m \times n$ numbers ordered in $m$ rows and $n$ columns

$$A = (a_{ij}) = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}.$$

We write $A \in \mathbf{R}^{m\times n}$, where $\mathbf{R}^{m\times n}$ denotes the set of all real $m \times n$ matrices. If $m = n$, then the matrix $A$ is said to be square and of order $n$. If $m \neq n$, then $A$ is said to be rectangular.

The **product** of two matrices $A$ and $B$ is defined if and only if the number of columns in $A$ equals the number of rows in $B$. If $A \in \mathbf{R}^{m\times p}$ and $B \in \mathbf{R}^{p\times n}$ then

$$C = AB \in \mathbf{R}^{m\times n}, \qquad c_{ij} = \sum_{k=1}^{p} a_{ik}b_{kj}, \quad 1 \le i, j \le m, \tag{1.3.1}$$

but $BA \in \mathbf{R}^{p\times p}$. Hence matrix multiplication is *not commutative*, and in general, $AB \neq BA$ even when $m = n = p$. If $AB = BA$ the matrices are said to **commute**.

Matrix multiplication satisfies the distributive rules

$$A(BC) = (AB)C, \qquad A(B + C) = AB + AC.$$

However, *the number of arithmetic operations required to compute, the left- and right-hand sides of these equations can be very different*!

**Example 1.3.1.** If $C \in \mathbf{R}^{p\times q}$ then computing the product $ABC$ as $(AB)C$ requires $mp(n + q)$ operations whereas $A(BC)$ requires $nq(m + p)$ operations. For example, if $A$ and $B$ are square $n \times n$ matrices and $x$ a column vector of length $n$ then computing the product $ABx$ as $(AB)x$ requires $n^3 + n^2$ operations whereas $A(Bx)$ only requires $2n^2$ operations. When $n \gg 1$ this makes a great difference!

It is often useful to think of a matrix as being built up of blocks of lower dimensions. The great convenience of this lies in the fact that the operations of addition and multiplication can be performed by treating the blocks as *non-commuting scalars* and applying the definition (1.3.1). Of course the dimensions of the blocks must correspond in such a way that the operations can be performed.

**Example 1.3.2.**

Assume that the two $n \times n$ matrices are partitioned into $2 \times 2$ block form

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

where $A_{11}$ and $B_{11}$ are square matrices of the same dimension. Then the product $C = AB$ equals

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}. \tag{1.3.2}$$

Be careful to note that since matrix multiplication is not commutative the *order of the factors in the products cannot be changed*! In the special case of block upper triangular matrices this reduces to

$$\begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix} \begin{pmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{pmatrix} = \begin{pmatrix} R_{11}S_{11} & R_{11}S_{12} + R_{12}S_{22} \\ 0 & R_{22}S_{22} \end{pmatrix}. \tag{1.3.3}$$

Note that the product is again block upper triangular and its block diagonal simply equals the products of the diagonal blocks of the factors.

It is important to know roughly how much work is required by different matrix algorithms. By inspection of (1.3.1) it is seen that computing the $mp$ elements $c_{ij}$ requires $mnp$ additions and multiplications.

In matrix computations the number of multiplicative operations $(\times, /)$ is usually about the same as the number of additive operations $(+, -)$. Therefore, in older literature, a **flop** was defined to mean roughly the amount of work associated with the computation

$$s := s + a_{ik}b_{kj},$$

i.e., one addition *and* one multiplication (or division). In more recent textbooks (e.g., Golub and Van Loan [12, 1996]) a flop is defined as one floating point operation doubling the older flop counts.[6] Hence, multiplication $C = AB$ of two two square matrices of order $n$ requires $2n^3$ flops. The matrix-vector multiplication $y = Ax$, where $x \in \mathbf{R}^{n\times 1}$ requires $2mn$ flops.

Operation counts are meant only as a rough appraisal of the work and one should not assign too much meaning to their precise value. On modern computer architectures the rate of transfer of data between different levels of memory often limits the actual performance. Also ignored here is the fact that on current computers division usually is 5–10 times slower than a multiply.

However, an operation count still provides useful information, and can serve as an initial basis of comparison of different algorithms. For example, it tells us that the running time for multiplying two square matrices on a computer roughly will increase cubically with the dimension $n$. Thus, doubling $n$ will approximately increase the work by a factor of eight; cf. (1.3.2).

An intriguing question is whether it is possible to multiply two matrices $A, B \in \mathbf{R}^{n\times n}$ (or solve a linear system of order $n$) in less than $n^3$ (scalar) multiplications. The answer is yes! Strassen [35] developed a fast algorithm for matrix multiplication, which, if used recursively to multiply two square matrices of dimension $n = 2^k$, reduces the number of multiplications from $n^3$ to $n^{\log_2 7} = n^{2.807\ldots}$. It is still an open question what is the minimum exponent $\omega$ such that matrix multiplication can be done in $O(n^\omega)$ operations. The current best upper bound is $\omega \leq 2.376$; see Higham [15, Ch. 23]. (Note that for many of the "fast" methods large constants are hidden in the $O$ notation.)

[6] Stewart [p. 96][33] uses **flam** (floating point addition and multiplication) to denote an "old" flop.

### 1.3.2 Solving Triangular Systems

The solution of **linear systems of equations** is one of the most frequently encountered problems in scientific computing. One important source of linear systems is discrete approximations of continuous differential and integral equations.

A linear system can be written in matrix-vector form as

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}, \tag{1.3.4}$$

where $a_{ij}$ and $b_i$, $1 \le i \le m$, $1 \le j \le n$ be the known input data and the task is to compute the unknown variables $x_j$, $1 \le j \le n$. More compactly $Ax = b$, where $A \in \mathbf{R}^{m\times n}$ is a matrix and $x \in \mathbf{R}^n$ and $b \in \mathbf{R}^m$ are column vectors. If $A$ is square and nonsingular there is an inverse matrix $A^{-1}$ such that $A^{-1}A = AA^{-1} = I$, the identity matrix. The solution to (1.3.4) can then be written as $x = A^{-1}b$, but *in almost all cases one should avoid computing the inverse* $A^{-1}$.

Linear systems which (possibly after a permutation of rows and columns) are of triangular form are particularly simple to solve. Consider a square **upper triangular** linear system ($m = n$)

$$\begin{pmatrix} u_{11} & \cdots & u_{1,n-1} & u_{1n} \\ & \ddots & \vdots & \vdots \\ & & u_{n-1,n-1} & u_{n-1,n} \\ & & & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}.$$

The matrix $U$ is nonsingular if and only if

$$\det(U) = u_{11} \cdots u_{n-1,n-1} u_{nn} \neq 0.$$

If this is the case the unknowns can be computed by the following recursion

$$x_n = b_n/u_{nn}, \qquad x_i = \Big(b_i - \sum_{k=i+1}^{n} u_{ik}x_k\Big)/u_{ii}, \quad i = n-1, \ldots, 1. \tag{1.3.5}$$

It follows that the solution of a triangular system of order $n$ can be computed in about $n^2$ flops. Note that this is the same amount of work as required for *multiplying* a vector by a triangular matrix.

Since the unknowns are solved for in *backward* order, this is called **back-substitution**. Similarly, a square linear system of **lower triangular** form $Lx = b$,

$$\begin{pmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \ldots & l_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

where $L$ is nonsingular, can be solved by **forward-substitution**

$$x_1 = b_1/l_{11}, \qquad x_i = \Big(b_i - \sum_{k=1}^{i-1} l_{ik}x_k\Big)/l_{ii}, \quad i = 2:n. \tag{1.3.6}$$

(Note that by reversing the order of the rows and columns an upper triangular system is transformed into a lower triangular and vice versa.)

When implementing a matrix algorithm on a computer, the *order of operations* in matrix algorithms may be important. One reason for this is the economizing of storage, since even matrices of moderate dimensions have a large number of elements. When the initial data is not needed for future use, computed quantities may overwrite data. To resolve such ambiguities in the description of matrix algorithms it is important to be able to describe computations like those in equations (1.3.5) in a more precise form. For this purpose we will use an informal programming language, which is sufficiently precise for our purpose but allows the suppression of cumbersome details. We illustrate these concepts on the back-substitution algorithm given above. In the following back-substitution algorithm the solution $x$ overwrites the data $b$.

**Algorithm 1.3.1** Back-substitution

Given a nonsingular upper triangular matrix $U \in \mathbf{R}^{n\times n}$ and a vector $b \in \mathbf{R}^n$, the following algorithm computes $x \in \mathbf{R}^n$ such that $Ux = b$:

**for** $i = n : (-1) : 1$
  $s := \sum_{j=i+1}^{n} u_{ik}b_k;$
  $b_i := (b_i - s)/u_{ii};$
**end**

Here $x := y$ means that the value of $y$ is evaluated and assigned to $x$. We use the convention that when the upper limit in a sum is smaller than the lower limit the sum is set to zero.

Another possible sequencing of the operations in Algorithm 1.3.1 is the following:

**for** $k = n : (-1) : 1$
  $b_k := b_k/u_{kk};$
  **for** $i = k-1 : (-1) : 1$
    $b_i := b_i - u_{ik}b_k;$
  **end**
**end**

Here the elements in $U$ are accessed column-wise instead of row-wise as in the previous algorithm. Such differences can influence the efficiency of the implementation depending on how the elements in the matrix $U$ are stored.

### 1.3.3 Gaussian Elimination

**Gaussian elimination**[7] is taught already in elementary courses in linear algebra. However, although the theory is deceptively simple the practical solution of large linear systems is far from trivial. In the beginning of the computer age in 1940s there was a mood of pessimism about the possibility of accurately solving systems even of modest order, say $n = 100$. Today there is a much deeper understanding of how Gaussian elimination performs in finite precision arithmetic and linear systems with hundred of thousands unknowns are routinely solved in scientific computing!

Clearly the following elementary operation can be performed on the system without changing the set of solutions:

- Interchanging two equations
- Multiplying an equation by a nonzero scalar $\alpha$.
- Adding a multiple $\alpha$ of the $i$th equation to the $j$th equation.

These operations correspond in an obvious way to row operations carried out on the augmented matrix $(A, b)$. By performing a sequence of such elementary operations one can always transform the system $Ax = b$ into a simpler system, which can be trivially solved.

In the most important direct method Gaussian elimination the unknowns are eliminated in a systematic way, so that at the end an equivalent triangular system is produced, which can be solved by substitution. Consider the system (1.3.4) with $m = n$ and assume that $a_{11} \neq 0$. Then we can eliminate $x_1$ from the last $(n-1)$ equations as follows. Subtracting from the $i$th equation the multiple

$$l_{i1} = a_{i1}/a_{11}, \quad i = 2 : n,$$

of the first equation, the last $(n-1)$ equations become

$$\begin{pmatrix} a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \vdots \\ a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix} \begin{pmatrix} x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_2^{(2)} \\ \vdots \\ b_n^{(2)} \end{pmatrix},$$

where the new elements are given by

$$a_{ij}^{(2)} = a_{ij} - l_{i1}a_{1j}, \qquad b_i^{(2)} = b_i - l_{i1}b_1, \quad i = 2 : n.$$

This is a system of $(n-1)$ equations in the $(n-1)$ unknowns $x_2, \ldots, x_n$. If $a_{22}^{(2)} \neq 0$, we can proceed and in the next step eliminate $x_2$ from the last $(n-2)$ of these equations. This gives a system of equations containing only the unknowns $x_3, \ldots, x_n$. We take

$$l_{i2} = a_{i2}^{(2)}/a_{22}^{(2)}, \quad i = 3 : n,$$

[7] Named after Carl Friedrich Gauss (1777–1855), but known already in China as early as in the first century BC.

and the elements of the new system are given by

$$a_{ij}^{(3)} = a_{ij}^{(2)} - l_{i2}a_{2j}^{(2)}, \qquad b_i^{(3)} = b_i^{(2)} - l_{i2}b_2^{(2)}, \quad i = 3:n.$$

The diagonal elements $a_{11}, a_{22}^{(2)}, a_{33}^{(3)}, \ldots$, which appear during the elimination are called **pivotal elements**. As long as these are nonzero, the elimination can be continued. After $(n-1)$ steps we get the single equation

$$a_{nn}^{(n)} x_n = b_n^{(n)}.$$

Collecting the first equation from each step we get

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots \\ & & & a_{nn}^{(n)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(n)} \end{pmatrix}, \tag{1.3.7}$$

where we have introduced the notations $a_{ij}^{(1)} = a_{ij}$, $b_i^{(1)} = b_i$ for the coefficients in the original system. Thus, we have reduced (1.3.4) to an equivalent nonsingular, upper triangular system (1.3.7), which can be solved by back-substitution. In passing we remark that the determinant of a matrix $A$, defined in (A.2.4), does not change under row operations we have from (1.3.7)

$$\det(A) = a_{11}^{(1)} a_{22}^{(2)} \cdots a_{nn}^{(n)} \tag{1.3.8}$$

*Gaussian elimination is indeed in general the most efficient method for* **computing determinants**!

**Algorithm 1.3.2** Gaussian Elimination (without row interchanges)

Given a matrix $A = A^{(1)} \in \mathbf{R}^{n\times n}$ and a vector $b = b^{(1)} \in \mathbf{R}^n$, the following algorithm computes the elements of the reduced system of upper triangular form (1.3.7). It is assumed that $a_{kk}^{(k)} \neq 0$, $k = 1:n$:

**for** $k = 1 : n-1$
  **for** $i = k+1 : n$
    $l_{ik} := a_{ik}^{(k)} / a_{kk}^{(k)}$; $\quad a_{ik}^{(k+1)} := 0$;
    **for** $j = k+1 : n$
      $a_{ij}^{(k+1)} := a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$;
    **end**
    $b_i^{(k+1)} := b_i^{(k)} - l_{ik} b_k^{(k)}$;
  **end**
**end**

We remark that no extra memory space is needed to store the multipliers. When $l_{ik} = a_{ik}^{(k)}/a_{kk}^{(k)}$ is computed the element $a_{ik}^{(k+1)}$ becomes equal to zero, so the multipliers can be stored in the lower triangular part of the matrix. Note also that if the multipliers $l_{ik}$ are saved, then the operations on the vector $b$ can be carried out at a later stage. This observation is important in that it shows that *when solving a sequence of linear systems*

$$Ax_i = b_i, \quad i = 1 : p,$$

*with the same matrix $A$ but different right hand sides the operations on $A$ only have to be carried out once.*

If we form the matrices

$$L = \begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \dots & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots \\ & & & a_{nn}^{(n)} \end{pmatrix} \tag{1.3.9}$$

then it can be shown that we have $A = LU$. Hence Gaussian elimination provides *a factorization of the matrix $A$ into a lower triangular matrix $L$ and an upper triangular matrix $U$*. This interpretation of Gaussian elimination has turned out to be extremely fruitful. For example, it immediately follows that the inverse of $A$ (if it exists) has the factorization

$$A^{-1} = (LU)^{-1} = U^{-1}L^{-1}.$$

This shows that the solution of linear system $Ax = b$,

$$x = A^{-1}b = U^{-1}(L^{-1}b),$$

can be computed by solving the two triangular systems $Ly = b$, $Ux = y$. Indeed it has been said (G. E. Forsythe and C. B. Moler [10]) that

"*almost anything you can do with $A^{-1}$ can be done without it*"

Several other important matrix factorizations will be studied at length in Volume II.

From Algorithm 1.3.2 it follows that $(n-k)$ divisions and $(n-k)^2$ multiplications and additions are used in step $k$ to transform the elements of $A$. A further $(n-k)$ multiplications and additions are used to transform the elements of $b$. Summing over $k$ and neglecting low order terms we find that the total number of flops required for the reduction of $Ax = b$ to a triangular system by Gaussian elimination is

$$\sum_{k=1}^{n-1} 2(n-k)^2 \approx 2n^3/3,$$

for the LU factorization of$A$ and

$$\sum_{k=1}^{n-1} 2(n-k) \approx n^2,$$

for each right hand side vector $b$. Comparing this with the $n^2$ flops needed to solve a triangular system we conclude that, except for very small values of $n$, *the LU factorization of A dominates the work in solving a linear system.* If several linear systems with the same matrix $A$ but different right-hand sides are to be solved, then the factorization needs to be performed only once!

**Example 1.3.3.** Many applications give rise to linear systems where the matrix $A$ only has a few nonzero elements close to the main diagonal. Such matrices are called **band matrices**. An important example is, banded matrices of the form

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ a_1 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & a_{n-1} & b_n \end{pmatrix}, \tag{1.3.10}$$

which are called **tridiagonal**. Tridiagonal systems of linear equations can be solved by Gaussian elimination with much less work than the general case. The following algorithm solves the tridiagonal system $Ax = g$ by Gaussian elimination without pivoting.

First compute the LU factorization $A = LU$, where

$$L = \begin{pmatrix} 1 & & & & \\ \gamma_1 & 1 & & & \\ & \gamma_2 & 1 & & \\ & & \ddots & \ddots & \\ & & & \gamma_{n-1} & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} \beta_1 & c_1 & & & \\ & \beta_2 & c_2 & & \\ & & \ddots & \ddots & \\ & & & \beta_{n-1} & c_{n-1} \\ & & & & \beta_n \end{pmatrix}.$$

The new elements in $L$ nd $U$ are obtained from the recursion: Set $\beta_1 = b_1$, and

$$\gamma_k = a_k/\beta_k, \qquad \beta_{k+1} = b_{k+1} - \gamma_k c_k, \quad k = 1 : n-1. \tag{1.3.11}$$

(Check this by computing the product $LU$!) The solution to $Ax = L(Ux) = g$ is then obtained in two steps. First a forward substitution to get $y = Ux$

$$y_1 = g_1, \qquad y_{k+1} = g_{k+1} - \gamma_k y_k, \quad k = 1 : n-1, \tag{1.3.12}$$

followed by a backward recursion for $x$

$$x_n = y_n/\beta_n, \qquad x_k = (y_k - c_k x_{k+1})/\beta_k, \quad k = n-1 : -1 : 1. \tag{1.3.13}$$

In this algorithm the LU factorization requires only about $n$ divisions and $n$ multiplications and additions. The solution of the two triangular systems require about twice as much work.

Consider the case when in step $k$ of Gaussian elimination a zero pivotal element is encountered, i.e. $a_{kk}^{(k)} = 0$. (The equations may have been reordered in previous

steps, but we assume that the notations have been changed accordingly.) If $A$ is nonsingular, then in particular its first $k$ columns are linearly independent. This must also be true for the first $k$ columns of the reduced matrix and hence some element $a_{ik}^{(k)}$, $i = k : n$ must be nonzero, say $a_{rk}^{(k)} \neq 0$. By interchanging rows $k$ and $r$ this element can be taken as pivot and it is possible to proceed with the elimination. The important conclusion is that *any nonsingular system of equations can be reduced to triangular form by Gaussian elimination, if appropriate row interchanges are used.*

Note that when rows are interchanged in $A$ the same interchanges must be made in the elements of the right-hand side $b$. Also the computed factors $L$ and $U$ will be the same as had the the row interchanges first been performed on $A$ and the Gaussian elimination been performed without interchanges.

To ensure the numerical stability in Gaussian elimination it will, except for special classes of linear systems, be necessary to perform row interchanges *not only when a pivotal element is exactly zero.* Usually it suffices to use **partial pivoting**, i.e. to choose the pivotal element in step $k$ as the element of largest magnitude in the unreduced part of the $k$th column.

**Example 1.3.4.**

The linear system

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

is nonsingular for any $\epsilon \neq 1$ and has the unique solution $x_1 = -x_2 = -1/(1-\epsilon)$. However, when $a_{11} = \epsilon = 0$ the first step in Gaussian elimination cannot be carried out. The remedy here is obviously to interchange the two equations, which directly gives an upper triangular system.

Suppose that in the system above $\epsilon = 10^{-4}$. Then the exact solution, rounded to four decimals equals $x = (-1.0001, 1.0001)^T$. However, if Gaussian elimination is carried through without interchanges we obtain $l_{21} = 10^4$ and the triangular system

$$\begin{aligned} 0.0001x_1 + x_2 &= 1 \\ (1 - 10^4)x_2 &= -10^4. \end{aligned}$$

Suppose that the computation is performed using arithmetic with three decimal digits. Then in the last equation the coefficient $a_{22}^{(2)}$ will be rounded to $-10^4$ and the solution computed by back-substitution is $\bar{x}_2 = 1.000$, $\bar{x}_1 = 0$, which is a catastrophic result!

If before performing Gaussian elimination we interchange the two equations then we get $l_{21} = 10^{-4}$ and the reduced system becomes

$$\begin{aligned} x_1 + x_2 &= 0 \\ (1 - 10^{-4})x_2 &= 1. \end{aligned}$$

The coefficient $a_{22}^{(2)}$ is now rounded to 1, and the computed solution becomes $\bar{x}_2 = 1.000$, $\bar{x}_1 = -1.000$, which is correct to the precision carried.

In this simple example it is easy to see what went wrong in the elimination without interchanges. The problem is that *the choice of a small pivotal element gives rise to large elements in the reduced matrix* and the coefficient $a_{22}$ in the original system is lost through rounding. Rounding errors which are small when compared to the large elements in the reduced matrix are unacceptable in terms of the original elements! When the equations are interchanged the multiplier is small and the elements of the reduced matrix of the same size as in the original matrix.

In general an algorithm is said to be **backward stable** if the computed solution $\overline{w}$ always equals *the exact solution* of a problem with "slightly perturbed data". It will be shown in Volume II, Sec. 7.5, that backward stability can almost always be ensured for Gaussian elimination with partial pivoting. The essential condition for stability is that no substantial growth occurs in the elements in $L$ and $U$. To formulate a basic result of the error analysis we need to introduce some new notations. In the following the absolute values $|A|$ and $|b|$ of a matrix $A$ and vector $b$ should be interpreted componentwise,

$$|A|_{ij} = (|a_{ij}|), \qquad |b|_i = (|b_i|).$$

Similarly the **partial ordering** "$\leq$" for the absolute values of matrices $|A|$, $|B|$ and vectors $|b|$, $|c|$, is to be interpreted component-wise.

**Theorem 1.3.1.**

*Let $\overline{L}$ and $\overline{U}$ denote the LU factors and $\overline{x}$ the solution of the system $Ax = b$, using LU factorization and substitution. Then $\overline{x}$ satisfies exactly the linear system*

$$(A + \Delta A)\overline{x} = b, \tag{1.3.14}$$

*where $\delta A$ is a matrix depending on both $A$ and $b$, such that*

$$|\Delta A| \lessapprox 3nu|\overline{L}|\,|\overline{U}|, \tag{1.3.15}$$

*where $u$ is a measure of the precision in the arithmetic.*

It is important to note that the result that the solution satisfies (1.3.14) with a small $|\Delta A|$ *does not mean that the solution has been computed with a small error.* If the matrix $A$ is ill-conditioned then the solution is very sensitive to perturbations in the data. This is the case, e.g., when the rows (columns) of $A$ are almost linearly dependent. However, this inaccuracy is intrinsic to the problem and cannot be avoided except by using higher precision in the calculations. Condition numbers for linear systems are discussed in Sec. 2.4.4.

### 1.3.4 Sparse Matrices and Iterative Methods

A matrix $A$ is called a **sparse** if it contains much fewer than the $n^2$ nonzero elements of a **full matrix** of size $n \times n$. Sparse matrices typically arise in many different applications. In Figure 1.3.1 we show a sparse matrix and its LU factors. In this case the original matrix is of order $n = 479$ and contains 1887 nonzero elements,

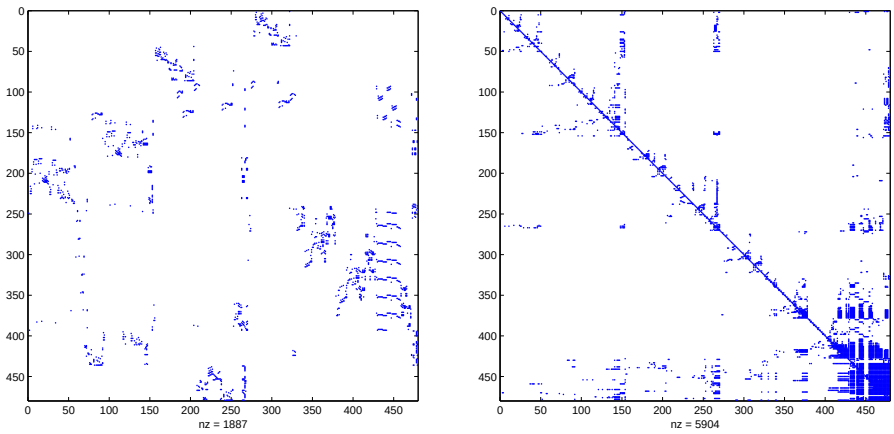



**Figure 1.3.1.** *Nonzero pattern of a sparse matrix and its LU factors.*

i.e., less than 0.9% of the elements are nonzero. The LU factors are also sparse and contain together 5904 nonzero elements or about 2.6%.

For many classes of sparse linear systems **iterative methods** when a are more efficient o use than **direct methods** such as Gaussian elimination. Typical examples are those arising when a differential equation in 2D or 3D is discretized. In iterative methods a sequence of approximate solutions is computed, which in the limit converges to the exact solution $x$. Basic iterative methods work directly with the original matrix $A$ and therefore has the added advantage of requiring only extra storage for a few vectors.

In a classical iterative method due to Richardson [32], a sequence of approximate solutions $x^{(k)}$ is defined by $x^{(0)} = 0$,

$$x^{(k+1)} = x^{(k)} + \omega(b - Ax^{(k)}), \quad k = 0, 1, 2, \ldots, \tag{1.3.16}$$

where $\omega > 0$ is a parameter to be chosen. It follows easily from (1.3.16) that the error in $x^{(k)}$ satisfies $x^{(k+1)} - x = (I - \omega A)(x^{(k)} - x)$, and hence

$$x^{(k)} - x = (I - \omega A)^k (x^{(0)} - x).$$

The convergence of Richardson's method will be studied in Sec. 10.1.4 in Volume II.

Iterative methods are used most often for the solution of very large linear systems, which typically arise in the solution of boundary value problems of partial differential equations by finite difference or finite element methods. The matrices involved can be huge, sometimes involving several million unknowns. The LU factors of matrices arising in such applications typically contain order of magnitudes more nonzero elements than $A$ itself. Hence, because of the storage and number of arithmetic operations required, Gaussian elimination may be far too costly to use.

**Example 1.3.5.**

In a typical problem for Poisson's equation (1.1.15) the function is to be determined in a plane domain $D$, when the values of $u$ are given on the boundary $\partial D$. Such **boundary value problems** occur in the study of steady states in most branches of Physics, such as electricity, elasticity, heat flow, fluid mechanics (including meteorology). Let $D$ be the a square grid with grid size $h$, i.e. $x_i = x_0 + ih$, $y_j = y_0 + jh$, $0 \le i \le N+1$, $0 \le j \le N+1$. Then the difference approximation yields

$$u_{i,j+1} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} - 4u_{i,j} = h^2 f(x_i, y_j),$$

$(1 \le i \le M,\ 1 \le j \le N)$. This is a huge system of linear algebraic equations; one equation for each interior gridpoint, altogether $N^2$ unknown and equations. (Note that $u_{i,0},\ u_{i,N+1},\ u_{0,j},\ u_{N+1,j}$ are known boundary values.) To write the equations in matrix-vector form we order the unknowns in a vector

$$u = (u_{1,1}, \ldots, u_{1,N}, u_{2,1}, \ldots, u_{2,N-1}, u_{N,1}, \ldots, u_{N,N}).$$

If the equations are ordered in the same order we get a system $Au = b$ where $A$ is symmetric with all nonzero elements located in five diagonals; see Figure 1.2.3 (left).

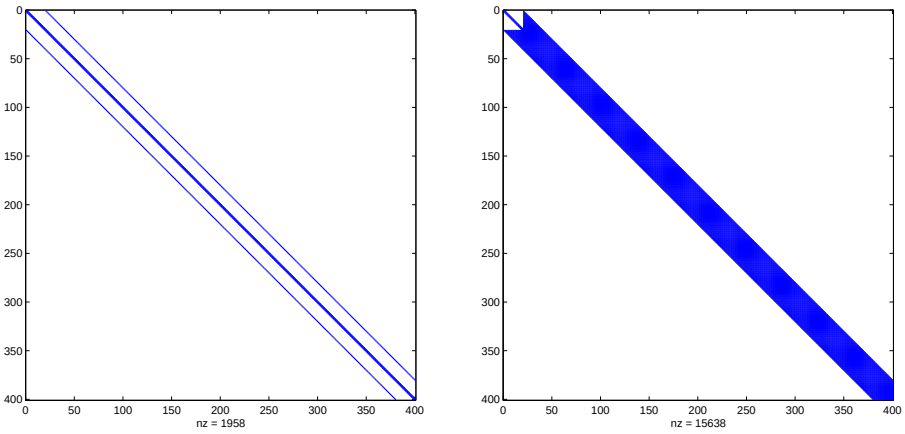



**Figure 1.3.2.** *Structure of $A$ (left) and $L + U$ (right) for the Poisson problem, $N = 20$. (Row-wise ordering of the unknowns)*

In principle Gaussian elimination can be used to solve such systems. However, even taking symmetry and the banded structure into account this would require $\frac{1}{2}N^4$ multiplications, since in the LU factors the zero elements inside the outer diagonals will fill-in during the elimination as shown in Figure 1.3.2 (right).

The linear system arising from Poisson's equation has several features common to boundary value problems for all linear partial differential equations. One of these is that there are at most 5 nonzero elements in each row of $A$, i.e. only a tiny fraction of the elements are nonzero. Therefore one iteration in Richardson's method requires only about $5{\cdot}N^2$ multiplications or equivalently five multiplications

per unknown. Using iterative methods which take advantage of the sparsity and other features does allow the efficient solution of such systems. This becomes even more essential for three-dimensional problems!

### 1.3.5 Software for Matrix Computations

In most computers in use today the key to high efficiency is to avoid as much as possible data transfers between memory, registers and functional units, since these can be more costly than arithmetic operations on the data. This means that the operations have to be carefully structured. One observation is that Gaussian elimination consists of three nested loops, which can be ordered in $3 \cdot 2 \cdot 1 = 6$ ways. Disregarding the right hand side vector $b$, each version does the operations

$$a_{ij}^{(k+1)} := a_{ij}^{(k)} - a_{kj}^{(k)} a_{ik}^{(k)} / a_{kk}^{(k)},$$

and only the ordering in which they are done differs. The version given above uses row operations and may be called the "$kij$" variant, where $k$ refers to step number, $i$ to row index, and $j$ to column index. This version is not suitable for programming languages like Fortran 77, in which matrix elements are stored sequentially by columns. In such a language the form "$kji$" should be preferred, which is the column oriented back-substitution rather than Algorithm 1.3.1 might be preferred.

An important tool for structuring linear algebra computations are the Basic Linear Algebra Subprograms (BLAS). These are now commonly used to formulate matrix algorithms and have become an aid to clarity, portability and modularity in modern software. The original set of BLAS identified frequently occurring vector operations in matrix computation such as scalar product, adding of a multiple of one vector to another. For example, the operation

$$y := \alpha x + y$$

in Single precision is named SAXPY. These BLAS were adopted in early Fortran programs and by carefully optimizing them for each specific computer the performance was enhanced without sacrificing portability.

For modern computers is is important to avoid excessive data movements between different parts of memory hierarchy. To achieve this so called level 3 BLAS have been introduced in the 1990s. These work on blocks of the full matrix and perform, e.g., the operations

$$C := \alpha AB + \beta C, \qquad C := \alpha A^T B + \beta C, \qquad C := \alpha AB^T + \beta C,$$

Since level 3 BLAS use $O(n^2)$ data but perform $O(n^3)$ arithmetic operations and gives a surface-to-volume effect for the ratio of data movement to operations. LAPACK [2], is a linear algebra package initially released in 1992, which forms the backbone of the interactive matrix computing system MATLAB . LAPACK achieves close to optimal performance on a large variety of computer architectures by expressing as much as possible of the algorithm as calls to level 3 BLAS.

**Example 1.3.6.**

In 1974 the authors wrote in [8, Sec. 8.5.3] that "a full $1,000 \times 1,000$ system of equations is near the limit at what can be solved at a reasonable cost". Today systemsof this size can easily be handled on a personal computer. The benchmark problem for the Japanese Earth Simulator, one of the worlds fastest computers in 2004, was the solution of a system of size $1,041,216$ on which a speed of $35.6 \times 10^{12}$ operations per second was measured. This is a striking illustration of the progress in high speed matrix computing that has occurred in these 30 years!

## Review Questions

**1.** How many operations are needed (approximately) for

(a) The multiplication of two square matrices?

(b) The LU factorization of a square matrix?

(b) The solution of $Ax = b$, when the triangular factorization of $A$ is known?

**2.** Show that if the $k$th diagonal entry of an upper triangular matrix is zero, then its first $k$ columns are linearly dependent.

**3.** What is the $LU$-decomposition of an $n$ by $n$ matrix $A$, and how is it related to Gaussian elimination? Does it always exist? If not, give sufficient conditions for its existence.

**4.** (a)For what type of linear systems are iterative methods to be preferred to Gaussian elimination?

(b) Describe Richardson's method for solving $Ax = b$. What can you say about the error in successive iterations?

**5.** What does the acronym BLAS stand for? What is meant by level 3 BLAS and why are they used in current linear algebra software??

## Problems and Computer Exercises

**1.** Let $A$ be a square matrix of order $n$ and $k$ a positive integer such that $2^p \leq k < 2^{p+1}$. Show how $A^k$ can be computed in at most $2p \cdot n^3$ multiplications.

*Hint*: Compute $A^2, A^4, A^8, \ldots$, by successive squaring and write $k$ in the binary number system.

**2.** (a) Let $A$ and $B$ be square upper triangular matrices of order $n$. Show that the product matrix $C = AB$ is also upper triangular. Determine how many multiplications are needed to compute $C$.

(b) Show that if $R$ is an upper triangular matrix with zero diagonal elements, then $R^n = 0$.

**3.** Show that there cannot exist a factorization

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{pmatrix}.$$

*Hint*: Equate the $(1,1)$-elements and deduce that either the first row or the first column in $LU$ must be zero.

**4.** (a) Consider the special upper triangular matrix of order $n$,

$$U_n(a) = \begin{pmatrix} 1 & a & a & \cdots & a \\ & 1 & a & \cdots & a \\ & & 1 & \cdots & a \\ & & & \ddots & \vdots \\ & & & & 1 \end{pmatrix},$$

Determine the solution $x$ to the triangular system $U_n(a)x = e_n$, where $e_n = (0, 0, \ldots, 0, 1)^T$ is the $n$th unit vector.

(b) Show that the inverse of an upper triangular matrix is also upper triangular. Determine for $n = 3$ the inverse of of $U_n(a)$. Try also to determine $U_n(a)^{-1}$ for an arbitrary $n$.

*Hint*: Use the property of the inverse that $UU^{-1} = U^{-1}U = I$, the identity matrix.

**5.** A matrix $H_n$ of order $n$ such that $h_{ij} = 0$ whenever $i > j + 1$ is called an upper **Hessenberg** matrix. For $n = 5$ it has the structure e.g.,

$$H_5 = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} & h_{15} \\ h_{21} & h_{22} & h_{23} & h_{24} & h_{25} \\ 0 & h_{32} & h_{33} & h_{34} & h_{35} \\ 0 & 0 & h_{43} & h_{44} & h_{45} \\ 0 & 0 & 0 & h_{54} & h_{55} \end{pmatrix}.$$

(a) Determine the approximate number of operations needed to compute the LU factorization of $H_n$ if no pivoting is needed.

(b) Determine the approximate number of operations needed to solve the system $H_n x = b$, when the factorization in (a) is given.

**6.** Compute the product $|L|\,|U|$ for the LU factors of the matrix in Example 1.3.4 with and without pivoting.

## 1.4 Numerical Solution of Differential Equations

### 1.4.1 Euler's Method

Approximate solution of *differential equations* is a very important task in scientific computing. Nearly all the areas of science and technology contain mathematical models which leads to systems of ordinary (or partial) differential equations. An **initial value problem** for an ordinary differential equation is to find $y(x)$ such that

$$\frac{dy}{dt} = f(t, y), \quad y(0) = c.$$

The differential equation gives, at each point $(t, y)$, the direction of the tangent to the solution curve which passes through the point in question. The direction of the tangent changes continuously from point to point, but the simplest approximation (which was proposed as early as the 18th century by Euler) is that one studies the solution for only certain values of $t = t_n = nh$, $n = 0, 1, 2, \ldots$ ($h$ is called the "step" or "step length") and assumes that $dy/dt$ is constant between the points. In this way the solution is approximated by a polygon segment (Fig. 1.4.1) which joins the points $(t_n, y_n)$, $0, 1, 2, \ldots$, where

$$y_0 = c, \quad \frac{y_{n+1} - y_n}{h} = f(t_n, y_n). \tag{1.4.1}$$

Thus we have a simple *recursion formula*, **Euler's method**:

$$y_0 = c, \quad y_{n+1} = y_n + hf(t_n, y_n), \quad n = 0, 1, 2 \ldots \tag{1.4.2}$$

During the computation, each $y_n$ occurs first on the left-hand side, then *recurs*

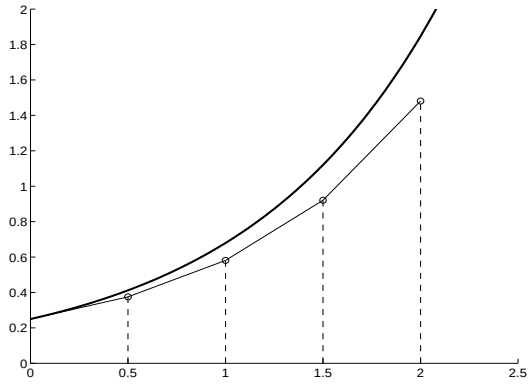



**Figure 1.4.1.** *Approximate solution of $dy/dx = y$, $y_0 = 0.25$, by Euler's method with $h = 0.5$.*

later on the right-hand side of an equation: hence the name **recursion formula**. (One could also call equation (1.4.2) an iteration formula, but one usually reserves the word "iteration" for the special case where a recursion formula is used solely as a means of calculating a limiting value.)

## 1.4.2 An Introductory Example

One of the most important techniques in computer applications to science and technology is the **step by step simulation** of a process or the time development of a system. A **mathematical model** is first set up, i.e., **state variables** which describe the essential features of the state of the system are set up. Then the laws are formulated, which govern *the rate of change of the state variables*, and other *mathematical relations* between these variables. Finally, these equations are programmed for a computer to calculate approximately, step by step, the development in time of the system.

The reliability of the results depends primarily on the goodness of the mathematical model and on the size of the time step. The choice of the time step is partly a question of economics. Small time steps may give you good accuracy, but also long computing time. More accurate numerical methods are often a good alternative to the use of small time steps. Such questions will be discussed in depth in Chapter 13 in Volume III.

The construction of a mathematical model is not trivial. Knowledge of numerical methods and programming helps also in that phase of the job, but more important is a good understanding of the fundamental processes in the system, and that is beyond the scope of this text. It is, however, important to realize that if the mathematical model is bad, no sophisticated numerical techniques or powerful computers can stop the results from being unreliable, or even harmful.

A mathematical model can be studied by analytic or computational techniques. Analytic methods do not belong to this text. We want, though, to emphasize that the comparison with results obtained by analytic methods, in the special cases when they can be applied, can be very useful when numerical methods and computer programs are tested. We shall now illustrate these general comments on a particular example.

**Example 1.4.1.**

Consider the motion of a ball (or a shot) under the influence of gravity and air resistance. It is well known that the trajectory is a parabola, when the air resistance is neglected and the force of gravity is assumed to be constant. We shall still neglect the variation of the force of gravity and the curvature and the rotation of the earth. This means that we forsake serious applications to satellites, etc. We shall, however, take the air resistance into account. We neglect the rotation of the shot around its own axis. Therefore we can treat the problem a a motion in a plane, but we have to forsake the application to, for example, table tennis or a rotating projectile. Now we have introduced a number of assumptions, which define our **model** of reality.

The state of the ball is described by its position $(x, y)$ and velocity $(u, v)$, each of which has two Cartesian coordinates in the plane of motion. The $x$-axis is horizontal, and the $y$-axis is directed upwards. Assume that the air resistance is a force $P$, such that the direction is opposite to the velocity, and the strength is proportional to the square of the speed and to the square of the radius $R$ of the shot. If we denote by $P_x$ and $P_y$ the components of $P$ along the $x$ and $y$ directions, respectively, we can then write,

$$P_x = -mzu, \quad P_y = -mzv, \quad z = \frac{cR^2}{m}\sqrt{u^2 + v^2}, \tag{1.4.3}$$

where $m$ is the mass of the ball.

For the sake of simplicity we assume that $c$ is a constant. It actually depends on the density and the viscosity of the air. Therefore, we have to forsake the application to cannon shots, where the variation of the density with height is important. If one has access to a good model of the atmosphere, the variation of $c$ would not make the numerical simulation much more difficult. This contrasts to analytic methods, where such a modification is likely to mean a considerable complication. In fact,

even with a constant $c$, a purely analytic treatment offers great difficulties.

Newton's law of motion tells us that,

$$mdu/dt = P_x, \qquad mdv/dt = -mg + P_y, \tag{1.4.4}$$

where the term $-mg$ is the force of gravity. Inserting (1.4.3) into (1.4.4) and dividing by $m$ we get

$$du/dt = -zu, \qquad dv/dt = -g - zv, \tag{1.4.5}$$

By the definition of velocity,

$$dx/dt = u, \qquad dy/dt = v, \tag{1.4.6}$$

Equations (1.4.5) and (1.4.6) constitute a system of four differential equations for the four variables $x, y, u, v$. The initial state $x_0, y_0$, and $u_0, v_0$ at time $t_0 = 0$ is assumed to be given. A fundamental proposition in the theory of differential equations tells that, if initial values of the state variables $u, v, x, y$ are given at some initial time $t = t_0$, then they will be uniquely determined for all $t > t_0$.

The simulation in Example 1.4.1 means that, at a sequence of times, $t_n$, $n = 0, 1, 2, \ldots$, we determine the approximate values, $u_n, v_n, x_n, y_n$. We first look at the simplest technique, using Euler's method with a constant time step $h$. Set therefore $t_n = nh$. We replace the derivative $du/dt$ by the forward difference quotient $(u_{n+1} - u_n)/h$, and similarly for the other variables. Hence after multiplication by $h$, the differential equations are replaced by the following system of **difference equations**:

$$\begin{aligned} u_{n+1} - u_n &= -hz_n u_n, \\ v_{n+1} - v_n &= -h(g + z_n v_n), \\ x_{n+1} - x_n &= hu_n, \quad y_{n+1} - y_n = hv_n, \end{aligned} \tag{1.4.7}$$

from which $u_{n+1}$, $v_{n+1}$, $x_{n+1}$, $y_{n+1}$, etc. are solved, step by step, for $n = 0, 1, 2, \ldots$, using the provided initial values $u_0, v_0, x_0, y_0$. Here $z_n$ is obtained by insertion of $u = u_n$, $v = v_n$ into (1.4.3).

We performed these computations until $y_{n+1}$ became negative for the first time, with $g = 9.81$, $\phi = 60^o$, and the initial values

$$x_0 = 0, \quad y_0 = 0, \quad u_0 = 100\cos\phi, \quad v_0 = 100\sin\phi.$$

In Fig. 1.4.2 are shown curves obtained for $h = 0.01$, and $cR^2/m = 0.25i \cdot 10^{-3}$, $i = 0, 1, 2, 3, 4$. There is, in this graphical representation, also an error due to the limited resolution of the plotting device.

In Euler's method the state variables are *locally approximated by linear functions* of time, one of the often recurrent ideas in numerical computation. We can use the same idea for computing the coordinate $x^*$ of the point, where the shot hits the ground. Suppose that $y_{n+1}$ becomes negative for the first time when $n = N$. For $x_N \le x \le x_{N+1}$ we then approximate $y$ by a linear function of $x$, represented by the secant through the points $(x_N, y_N)$ and$(x_{N+1}, y_{N+1})$ , i.e.,

$$y = y_N + (x - x_N)\frac{y_{N+1} - y_N}{x_{N+1} - x_N}.$$

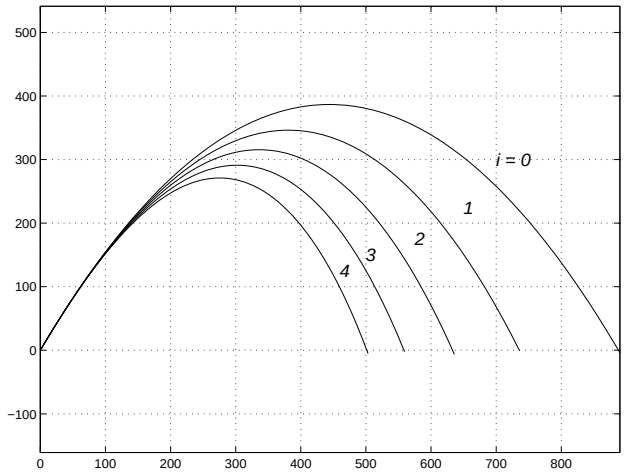



**Figure 1.4.2.** *Approximate trajectories computed with Euler's method for* $cR^2/m = 0.25i \cdot 10^{-3}$, $i = 0 : 4$, *and* $h = 0.01$.

By setting $y = 0$ we obtain

$$x^* = x_N - y_N \frac{x_{N+1} - x_N}{y_{N+1} - y_N}. \tag{1.4.8}$$

The error from the linear approximation in (1.4.8) used for the computation of $x^*$ is proportional to $h^2$. It is thus approximately equal to the error committed in *one single step* with Euler's method, and hence of less importance than the other error.

The case without air resistance ($i = 0$) can be solved exactly. In fact it can be shown that $x^* = 2u_0v_0/9.81 = 5000 \cdot \sqrt{3}/9.81 = 882.7986$. The computer produced $x^* = 883.2985$ for $h = 0.01$, and $x^* = 883.7984$ for $h = 0.02$. The error for $h = 0.01$ is therefore 0.4999, and for $h = 0.02$ it is 0.9998. The approximate proportionality to $h$ is thus verified, actually more strikingly than could be expected!

It can be shown that *the error in the results obtained with Euler's method is also proportional to* $h$ *(not* $h^2$*).* Hence a disadvantage of the above method is that the step length $h$ must be chosen quite short if reasonable accuracy is desired. In order to improve the method we can apply another idea mentioned in the previously, namely Richardson extrapolation. The application differs a little from the one you saw there, because now the error is approximately proportional to $h$, while for the trapezoidal rule it was approximately proportional to $h^2$. For $i = 4$, the computer produced $x^* = 500.2646$ and $x^* = 500.3845$ for, respectively, $h = 0.01$ and $h = 0.02$. Now let $x^*$ denote the *exact* coordinate of the landing point. Then

$$x^* - 500.2646 \approx 0.01k, \qquad x^* - 500.3845 \approx 0.02k.$$

By elimination of $k$ we obtain

$$x^* \approx 2 \cdot 500.2646 - 500.3845 = 500.1447,$$

which should be a more accurate estimate of the landing point. By a more accurate integration method we obtained 500.1440. So in this case, we gained more than two decimal digits by the use of Richardson extrapolation.

The simulations shown in Fig. 1.4.2 required about 1500 time steps for each curve. This may seem satisfactory, but we must not forget that this is a very small task, compared to most serious applications. So we would like to have a method that allows *much larger time steps* than Euler's method.

### 1.4.3 A Second Order Accurate Method

In step by step computations we have to distinguish between the **local error**, i.e., the error that is committed at a single step, and the **global error**, i.e., the error of the final results. Recall that we say that a method is accurate of order $p$, if its global error is approximately proportional to $h^p$. Euler's method is only first order accurate; we shall below present a method that is second order accurate. To achieve the same accuracy as with Euler's method the number of steps can then be reduced to about the square root of the number of steps in Euler's method, e.g., in the above ball problem to $\sqrt{1500} \approx 40$ steps. Since the amount of work is closely proportional to the number of steps this is an enormous saving!

Another question is how the step size $h$ is to be chosen. It can be shown that even for rather simple examples (see below) it is adequate to use very *different step size* in different parts of the computation. Hence the automatic control of the step size (also called *adaptive* control) is an important issue.

Both requests can be met by an improvement of the Euler method (due to Runge) obtained by the applying the Richardson extrapolation in every second step. This is different from our previous application of the Richardson idea. We first introduce a better notation by writing a **system of differential equations** and the initial conditions in vector form

$$d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(a) = \mathbf{c}, \tag{1.4.9}$$

where $\mathbf{y}$ is a column vector that contains all the state variables.[8] With this notation methods for large systems of differential equations can be described as easily as methods for a single equation. The change of a system with time can then be thought of as a motion of the state vector in a multidimensional space, where the differential equation defines the **velocity field**. This is our first example of the central role of vectors and matrices in modern computing. We temporarily use *superscripts* for the vector components, because we need subscripts for the same purpose as in the above description of Euler's method.

For the ball example, we have by (1.4.5) and (1.4.6)

$$\mathbf{y} = \begin{pmatrix} y^1 \\ y^2 \\ y^3 \\ y^4 \end{pmatrix} \equiv \begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix}, \quad \mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} y^3 \\ y^4 \\ -zy^3 \\ -g - zy^4 \end{pmatrix}, \quad \mathbf{c} = 10^2 \begin{pmatrix} 0 \\ 0 \\ \cos\phi \\ \sin\phi \end{pmatrix},$$

[8] The boldface notation is temporarily used for vectors *in this section*, not in the rest of the book.

where

$$z = \frac{cR^2}{m}\sqrt{(y^3)^2 + (y^4)^2}.$$

The computations in the step which leads from $t_n$ to $t_{n+1}$ are then as follows:

i. One Euler step of length $h$ yields the estimate:

$$\mathbf{y}^*_{n+1} = \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n).$$

ii. Two Euler steps of length $\frac{1}{2}h$ yield another estimate:

$$\mathbf{y}_{n+\frac{1}{2}} = \mathbf{y}_n + \frac{1}{2}h\mathbf{f}(t_n, y_n); \qquad \mathbf{y}^{**}_{n+1} = \mathbf{y}_{n+\frac{1}{2}} + \frac{1}{2}h\mathbf{f}(t_{n+1/2}, \mathbf{y}_{n+1/2}),$$

where $t_{n+1/2} = t_n + h/2$.

iii. Then $\mathbf{y}_{n+1}$ is obtained by Richardson extrapolation:

$$\mathbf{y}_{n+1} = \mathbf{y}^{**}_{n+1} + (\mathbf{y}^{**}_{n+1} - \mathbf{y}^*_{n+1}).$$

It is conceivable that this yields a 2nd order accurate method. It is left as an exercise (Problem 2) to verify that *this scheme is identical to the following somewhat simpler scheme* known as **Runge's 2nd order method**:

$$\begin{aligned} \mathbf{k}_1 &= h_n\mathbf{f}(t_n, \mathbf{y}_n); \\ \mathbf{k}_2 &= h_n\mathbf{f}(t_n + h_n/2, \mathbf{y}_n + \mathbf{k}_1/2); \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \mathbf{k}_2, \end{aligned} \tag{1.4.10}$$

where we have replaced $h$ by $h_n$ in order to include the use of variable step size. Another explanation of the 2nd order accuracy of this method is that the displacement $\mathbf{k}_2$ equals the product of the step size and a sufficiently accurate estimate of the velocity at the *midstep* of the time step. A more detailed analysis of this method comes in Sec. 13.3.2. Sometimes this method is called the improved Euler method or Heun's method, but these names are also used to denote other 2nd order accurate methods.

We shall now describe how the step size can be **adaptively** (or automatically) controlled by means of a tolerance TOL, by which the user tells the program how large error he tolerates in values of variables (relative to the values themselves).[9] Compute

$$\delta = \max_i |k_2^i - k_1^i|/|3y^i|,$$

where $\delta$ is related to the *relative* errors of the components of the vector $\mathbf{y}$; see below.

A step size is *accepted* if $\delta \le$ TOL, and the next step should be

$$h_{next} = h\min\{1.5, \sqrt{\text{TOL}/(1.2\delta)}\},$$

[9] With the terminology that will be introduced in the next chapter, TOL is, with the step size control described here, related to the *global relative errors* . At the time of writing, this contrasts to most codes for the solution of ordinary differential equations, in which the *local* errors per step are controlled by the tolerance.

where 1.2 is a safety factor, since the future is never exactly like the past .... The square root occurring here is due to the fact that this method is 2nd order accurate, i.e., the global error is almost proportional to the square of the step size and $\delta$ is approximately proportional to $h^2$.

A step is *rejected*, if $\delta > \text{TOL}$, and *recomputed* with the step size

$$h_{next} = h \max\{0.1, \sqrt{\text{TOL}/(1.2\delta)}\}.$$

The program needs a suggestion for the size of the first step. This can be be a very rough guess, because the step size control described above, will improve it automatically, so that an adequate step size is found after a few steps (or recomputations, if the suggested step was too big). In our experience, a program of this sort can efficiently handle guesses that are wrong by several powers of 10. If $y(a) \neq 0$ and $y'(a) = 0$, you may try the initial step size

$$h = \frac{1}{4} \sum_i |y^i| \Big/ \sum_i |dy^i/dt|$$

evaluated at the initial point $t = a$. When you encounter the cases $y(a) = 0$ or $y'(a) = 0$ for the first time, you are likely to have gained enough experience to suggest something that the program can handle. More professional programs take care of this detail automatically.

The request for a certain *relative* accuracy may cause trouble when some components of $y$ are close to zero. So, already in the first version of your program, you had better *replace* $y^i$ *in the above definition of* $\delta$ *by* $\bar{y}^i = \max\{|y^i|, 0.001\}$. A more detailed discussion of such matters follows in Sections 13.1 and 13.2 in Volume II (see in particular Computer Exercise 13.1.1). (You may sometimes have to replace the default value 0.001 by something else.)

It is a good habit to make a second run with a predetermined sequence of times (if your program allows this) instead of adaptive control. Suppose that the sequence of times used in the first run is $t_0$, $t_1$, $t_2$, .... Divide each subinterval $[t_n, t_{n+1}]$ into two steps of equal length. So, the second run still has variable step size and twice as many steps as the first run. The errors are therefore expected to be approximately $\frac{1}{4}$ of the errors of the first run. The first run can therefore use a tolerance that is 4 times as large than the error you can tolerate in the final result. Denote the results of the two runs by $y_I(t)$ and $y_{II}(t)$. You can plot $\frac{1}{3}(y_{II}(t) - y_I(t))$ versus $t$; this is an error curve for $y_{II}(t)$ Alternatively you can add $\frac{1}{3}(y_{II}(t) - y_I(t))$ to $y_{II}(t)$. This is another application of the Richardson extrapolation idea. The cost is only 50% more work than the plain result without an error curve.

If there are no singularities in the differential equation, $\frac{1}{3}(y_{II}(t) - y_I(t))$ *strongly overestimates the error of the extrapolated values*—typically by a factor like $\text{TOL}^{-1/2}$. It is, however, a non-trivial matter to find an error curve that strictly and realistically tells how good the extrapolated results are. There will be more comments about these matters in Sec. 3.3.4 in Volume II (see also Example 13.2.1 in Volume III). The reader is advised to test experimentally how this works on examples where the exact results are known.

An easier, though inferior, alternative is to run a problem with two different tolerances. One reason why it is inferior is that the two runs do not "keep in step". For example, Richardson extrapolation cannot be easily applied.

If you request very high accuracy in your results, or if you are going to simulate a system over a very long time, you will need a method with a higher order of accuracy than two. The reduction of computing time if you replace this method by a higher order method can be large, but the improvements are seldom as drastic as when you replace Euler's method by a second order accurate scheme like this. Runge's 2nd order method is, however, no universal recipe. There are special classes of problems, notably the problems which are called "stiff", which need special methods. These matters are treated in Chapter 13.

One advantage of a second order accurate scheme when requests for accuracy are modest, is that the quality of the computed results is normally not ruined by the use of *linear interpolation* at the graphical output, or at the post-processing of numerical results. (After you have used a more than second order accurate integration method, it may be necessary to use a more sophisticated interpolation at the graphical or numerical treatment of the results.)

**Example 1.4.2.**

The differential equation $y' = -\frac{1}{2}y^3$, with initial condition $y(1) = 1$, was treated by a program, essentially constructed as described above, with $\text{TOL} = 10^{-4}$ until $t = 10^4$.

In this example we can compare with the exact solution, $y(t) = t^{-1/2}$. It was found that the actual relative error stayed a little less than $1.5\,\text{TOL}$ all the time when $t > 10$. The step size increased almost linearly with $t$ from $h = 0.025$ to $h = 260$. The number of steps increased almost proportionally to $\log t$; the total number of steps was 374. Only one step had to be recomputed (except for the first step, where the program had to find an appropriate step size).

The computation was repeated with $\text{TOL} = 4 \cdot 10^{-4}$. The experience was the same, except that the steps were about twice as long all the time. This is what can be expected, since the step sizes should be approximately proportional to $\sqrt{\text{TOL}}$, for a second order accurate method. The total number of steps was 194.

**Example 1.4.3.**

The example of the motion of a ball was treated by Runge's 2nd order method with the constant step size $h = 0.9$. The coordinate of the landing point became $x^* = 500.194$, which is more than twice as accurate than the result obtained by Euler's method (without Richardson extrapolation) with $h = 0.01$, which uses about 90 times as many steps.

We have now seen a variety of ideas and concepts which can be used in the development of numerical methods. A small warning is perhaps warranted here: it is not certain that the methods will work as well in practice as one might expect. This is because approximations and the restriction of numbers to a certain number of digits introduce errors which are propagated to later stages of a calculation. The

manner in which errors are propagated is decisive for the practical usefulness of a numerical method. We shall examine such questions in Chapter 2. Later chapters will treat **propagation of errors** in connection with various typical problems.

The risk that error propagation may up-stage the desired result of a numerical process should, however, not dissuade one from the use of numerical methods. It is often wise, though, to experiment with a proposed method on a simplified problem before using it in a larger context. The development of hardware as well as software has created a far better environment for such work than we had a decade ago. In this area too, the famous phrase of the Belgian-American chemist Baekeland holds:

> "*Commit your blunders on a small scale and make your profits on a large scale.*"

## Review Questions

**1.** Explain the difference between the local and global error of a numerical method for solving a differential equation. What is meant by the order of accuracy for a method?

**2.** Describe how Richardson extrapolation can be used to increase the order of accuracy of Euler's method.

## Problems and Computer Exercises

**1.** Integrate numerically using Euler's method the differential equation $dy/dx = y$, with initial conditions $y(0) = 1$, to $x = 0.4$:

(a) with step length $h = 0.2$ and $h = 0.1$.

(b) Extrapolate to $h = 0$, using the fact that the error is approximately proportional to the step length. Compare the result with the exact solution of the differential equation and determine the ratio of the errors in the results in (a) and (b).

(c) How many steps would have been needed in order to attain, without using extrapolation, the same accuracy as was obtained in (b)?

**2.** (a) Write a program for the simulation of the motion of the ball, using Euler's method and the same initial values and parameter values as above. Print only $x, y$ at integer values of $t$ and at the last two points (i.e. for $n = N$ and $n = N + 1$) as well as the coordinate of the landing point. Take $h = 0.05$ and $h = 0.1$. As post-processing, improve the estimates of $x^*$ by Richardson extrapolation, and estimate the error by comparison with the results given in the text above.

(b) In Equation (1.4.8) replace in the equations for $x_{n+1}$ and $y_{n+1}$ the right hand sides $u_n$ and $v_n$ by, respectively, $u_{n+1}$ and $v_{n+1}$. Then proceed as in (a) and compare the accuracy obtained with that obtained in (a).

(c) Choose initial values which correspond to what you think is reasonable for shot put. Make experiments with several values of $u_0, v_0$ for $c = 0$. How much is $x^*$ influenced by the parameter $cR^2/m$?

**3.** Verify that Runge's 2nd order method, as described by equation (1.4.10), is equivalent to the scheme described a few lines earlier (with Euler steps and Richardson extrapolation).

**4.** Write a program for Runge's 2nd order method with automatic step size control that can be applied to a system of differential equations, *or* use the MATLAB program on the diskette. Store the results so that they can be processed afterwards, e.g., for making table of the results, and/or curves to be drawn showing $y(t)$ versus $t$, or (for a system) $y^2$ versus $y^1$, or some other interesting curves.
Apply the program to Examples 1.4.2 and 1.4.3, and to the circle test, i.e.

$$y_1' = -y_2, \qquad y_2' = y_1,$$

with initial conditions $y_1(0) = 1$, $y_2(0) = 0$. Verify that the exact solution is a uniform motion along the unit circle in the $(y_1, y_2)$-plane. Stop the computations after 10 revolutions ($t = 20\pi$). Make experiments with different tolerances, and determine how small the tolerance has to be in order that the circle on the screen should not become "thick".

## 1.5 Monte Carlo Methods

### 1.5.1 Origin of Monte Carlo Methods

In most of the applications of probability theory one makes a mathematical formulation of a stochastic problem (i.e., a problem where chance plays some part), and then solves the problem by using analytical or numerical methods. In the **Monte Carlo method**, one does the opposite; a mathematical or physical problem is given, and one constructs **numerical game of chance**, the mathematical analysis of which leads to the same equations as the given problem, e.g., for the probability of some event, or for the mean of some random variable in the game. One plays it $N$ times and estimates the relevant quantities by traditional statistical methods. Here $N$ is a large number, because the standard deviation of a statistical estimate typically *decreases only inversely proportional to* $\sqrt{N}$.

The idea behind the Monte Carlo method was used by the Italian physicist Enrico Fermi to study the neutron diffusion in the early 1930s. Fermi used a small mechanical adding machine for this purpose. With the development of computers larger problems could be tackled. At Los Alamos in the late 1940s the use of

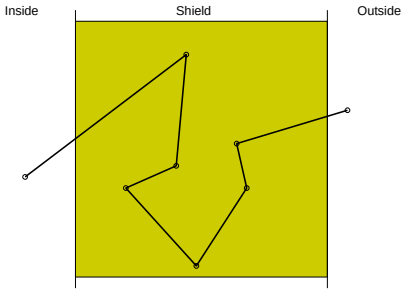



**Figure 1.5.1.** *Neutron scattering.*

the method was pioneered by von Neumann,[10] and Ulam[11] and others for many problems in mathematical physics including approximating complicated multidimensional integrals. The picturesque name of the method was coined by Nicholas Metropolis.

The Monte Carlo method is now so popular that the definition is too narrow. For instance, in many of the problems where the Monte Carlo method is successful, there is already an element of chance in the system or process which one wants to study. Thus such games of chance can be considered to be a numerical simulation of the most important aspects. In this wider sense the "Monte Carlo methods also include techniques used by statisticians since around 1900, under names like *experimental* or *artificial sampling*. For example, one used statistical experiments to check the adequacy of certain theoretical probability laws, which the eminent scientist W .S .Gosset, who used the pseudonym "Student" when he wrote on Probability, had derived mathematically.

Monte Carlo methods may be used, when the changes in the system are described with a much more complicated type of equation than a system of ordinary differential equations. Note that there are many ways to combine analytical methods and Monte Carlo methods. An important rule is that *if a part of a problem can be treated with analytical or traditional numerical methods, then one should use such methods.*

The following are some areas where the Monte Carlo method has been applied:

**(a)** Problems in reactor physics; for example, a neutron, because it collides with other particles, is forced to make a random journey. In infrequent but important cases the neutron can go through a layer of (say) shielding material (see

[10] John von Neumann was born János Neumann in Budapest 1903, and died in Washington D.C. 1957. He studied under Hilbert in Göttingen during 1926–27, was appointed professor at Princeton University in 1931, and in 1933 joined the newly founded Institute for Advanced Studies in Princeton. He built a framework for quantum mechanics, worked in game theory and was one of the pioneers of computer science.

[11] Stanislaw Marcin Ulam, born in Lemberg, Poland (now Lwow, Ukraine) 1909, died in Santa Fe, New Mexico, USA, 1984. Ulam obtained his Ph.D. in 1933 from the Polytechnic institute of Lwow, where he studied under Banach. He was invited to Harward University by G. D. Birkhoff in 1935, and left Poland permanently in 1939. In 1943 he was asked by von Neumann to come to Los Alamos, where he remained until 1965.

Fig. 1.5.1).

**(b)** Technical problems concerning traffic (telecommunication, railway networks, regulation of traffic lights and other problems concerning automobile traffic).

**(c)** Queuing problems.

**(d)** Models of conflict.

**(e)** Approximate computation of multiple integrals.

**(f)** Stochastic models in financial mathematics.

Monte Carlo methods are often used for the evaluation of high dimensional (10–100) integrals over complicated regions. Such integrals occur in such diverse areas as quantum physics and mathematical finance. The integrand is then evaluated at random points uniformly distributed in the region of integration. The arithmetic mean of these function values is then used to approximate the integral. Such randomization makes multivariate integration computationally feasible. Interestingly choosing the evaluation points uniformly distributed in the region of integration is not the optimal strategy. Instead one should use "quasi-random numbers" designed specifically for that purpose; see Sec. 5.??.

In a simulation, one can study the result of various actions more cheaply, more quickly, and with less risk of organizational problems than if one were to take the corresponding actions on the actual system. In particular, for problems in applied operations research, it is quite common to take a shortcut from the actual system to a computer program for the game of chance, without formulating any mathematical equations. The game is then a model of the system. In order for the term Monte Carlo method to be correctly applied, however, **random choices** should occur in the calculations. This is achieved by using so-called **random numbers**; the values of certain variables are determined by a process comparable to dice throwing. Simulation is so important that several special programming languages have been developed exclusively for its use.[12]

In the rest of this section we assume that the reader is familiar with some basic concepts, formulas and results from Probability and Statistics, and we make use of them without proofs (which may be found in most texts on these subjects). The terminology of Probability and Statistics is varied, in particular within areas of application. We shall use the following terms for probability distributions in $\mathbf{R}$:

The **distribution function** of a random variable $X$ is denoted by $F(x)$ and defined by

$$F(x) = Pr\{X \leq x\}.$$

Note that $F(x)$ is non-negative and non-decreasing, $F(-\infty) = 0$, $F(\infty) = 1$. If

[12] One notable example is the SIMULA programming language designed and built by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo 1962–1967, It was originally built as a language for discrete event simulation, but was influential also because it introduced object-oriented programming concepts.

$F(x)$ is differentiable, the (probability) **density function** [13] is $f(x) = F'(x)$. Note that

$$f(x) \geq 0, \quad \int_{\mathbf{R}} f(x)\,dx = 1,$$

and

$$Pr\{X \in [x, x+dx] = f(x)\,dx + o(dx)\}.$$

In the discrete case $X$ can only take on discrete values $x_i$, $i = 1:N$, and

$$Pr\{X = x_i\} = p_i, \quad i = 1:N.$$

where $p_i \geq 0$ and $\sum_i p_i = 1$.

The **mean** or the **expectation** of $X$ is

$$E(X) = \begin{cases} \displaystyle\int_{\mathbf{R}} x f(x)\,dx, & \text{continuous case,} \\ \displaystyle\sum_{i=1}^{N} p_i x_i, & \text{discrete case,} \end{cases}$$

The **variance** of $X$ equals

$$\text{var}(X) = E((X-m)^2),$$

where $m = E(X)$ and $\text{std}(X) = \sqrt{\text{var}(X)}$ is the **standard deviation**. If $X_j$ and $X_k$, $i \neq j$, are two random variables with mean values $m_j$ and $m_k$, then their **covariance** is

$$\text{covar}(X_j, X_k) = E((X_j - m_j)(X_k - m_k)).$$

If $\text{covar}(X_j, X_k) = 0$ then $X_j$ and $X_k$ are said to be **uncorrelated**.

Some formulas for the estimation of mean, standard deviation etc., from results of simulation experiments or other statistical data are given in the computer exercises of Sec. 2.3. See also the references to the Matlab Reference Guide in the problems and exercises of the present section.

### 1.5.2 Random and Pseudo-Random Numbers

In the beginning coins, dice and roulettes were used for creating the randomness, e.g., the sequence of twenty digits

$$11100 \;\; 01001 \;\; 10011 \;\; 01100$$

is a record of twenty tosses of a coin where "heads" are denoted by 1 and "tails" by 0. Such digits are sometimes called (binary) **random digits**, assuming that we have a perfect coin—i.e., that heads and tails have the same probability of occurring.

[13] In old literature a density function is often called a frequency function. The term cumulative distribution is also used as a synonym of distribution function. Unfortunately, distribution or probability distribution is sometimes used in the meaning of a density function.

We also assume that the tosses of the coin are made in a statistically independent way. (Of course, these assumptions cannot be obtained in practice!)

Similarly, decimal random digits could in principle be obtained by using a well-made icosahedral (twenty-sided) dice, and assigning each decimal digit to two of its sides. Such mechanical (or analogous electronical) devices have been used to produce **tables of random sampling digits**; the first one by Tippett was published in 1927 and was to be considered as a sequence of $40,000$ independent observations of a random variable that equals one of the integer values $0, 1, 2, \ldots, 9$, each with probability $1/10$. In the early 1950s the Rand Corporation constructed a million-digit table of random numbers using an electrical "roulette wheel" ([6, 1955]). The wheel had 32 slots, of which 12 were ignored; the others were numbered from 0 to 9 twice. To test the quality of the randomness several tests were applied. Every block of a thousand digits in the tables (and also the table as a whole) were tested.

**Example 1.5.1.**

A random number generator to be used for drawing of prizes of Swedish Premium Saving Bonds was developed in 1962 by Dahlquist [7]. For this application speed is not a major concern, since relatively few random decimal digits (about 50,000) are needed. Therefore an algorithm, which is easier to analyze, was chosen. This uses a primary series of less than 240 decimal random digits produced by some mechanical device, or taken from a table of random numbers. The length of the primary series is $n = p_1 + p_2 + \cdots + p_k$, where $p_i$ are prime numbers and $p_i \neq p_j$, $i \neq j$. For the analysis it is assumed that the primary series is perfectly random.

The primary series is used in a way that is best described by a mechanical analogy. Think of $k$ cog-wheels with $p_i$ cogs, $i = 1 : k$, and place the digits from the primary series on the cogs of these. The first digit in the secondary series is obtained by adding the $k$ digits (modulus 10) that are at the top position of each cog-wheel. Then each wheel is turned one cog clock-wise and the second digit is obtained in the same way as the first, etc. After $p_1 \cdot p_2 \cdots p_k$ steps we are back in the original position. This is the minimum period of the secondary series of random digits.

For the application mentioned above $k = 7$ prime numbers, in the range $13 \leq p_i \leq 53$, are randomly selected. This gives a varying minimum period approximately equal to $10^8$, which is much more than the number of digits used to produce the drawing list. Considering the public reaction the primary series is generated by a tombola drawing.

Random digits from a table can be packed together to give a sequence of equidistributed integers. For example, the sequence

$$55693 \quad 02945 \quad 81723 \quad 43588 \quad 81350 \quad 76302 \quad \ldots$$

can be considered as six five-digit random numbers, where each element in the sequence has probability of $10^{-5}$ of taking on the value, 0,1,2,...,99,999. From the

same digits one can also construct the sequence

$$0.556935, 0.029455, 0.817235, 0.435885, 0.813505, 0.763025, \ldots, \tag{1.5.1}$$

which can be considered a good approximation to a sequence of independent observations of a variable which is a sequence of uniform deviates in on the interval $[0, 1)$. The 5 in the sixth decimal place is added in order to get the correct mean (without this the mean would be 0.499995 instead of 0.5).

We shall return to this in the next subsection, together with the further development in the computer age, where **arithmetic methods** are used for producing the so-called **pseudo-random numbers** needed for the large-scale simulations that nowadays are demanded, e.g. in the areas applications mentioned below.[14]

In a computer it is usually not appropriate to store a large table of random numbers. One instead computes a sequence of uniform deviates $u_0$, $u_1$, $u_2, \ldots, \in [0, 1]$, by a **random number generator**, (RNG) i.e., some arithmetic algorithm. Sequences obtained in this way are uniquely determined by one or more starting values (**seeds**), to be given by the user (or some default values). The aim of a pseudo-random number generator is to imitate the abstract mathematical concept of mutually independent random variables uniformly distributed over the interval $[0, 1)$. They should be analyzed theoretically and be backed by practical evidence from extensive statistical testing. According to a much quoted statement by D. H. Lehmer[15]

> "*A random sequence is a vague notion ... in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians...*"

Because the set of floating point numbers in $[0, 1]$ is finite, although very large, there will eventually appear a number that has appeared before, (say) $u_{i+j} = u_i$ for some positive $i, j$. The sequence $\{u_n\}$ therefore repeats itself periodically for $n \geq i$; the length of the period is $j$. A truly random sequence is, of course, never periodic. A sequence generated like this is, for this and for other reasons, called **pseudo-random**. However, the ability to repeat exactly the same sequence of numbers, which is needed for program verification and variance reduction, is a major advantage over generation by physical devices.

There are two popular myths about the making of random number generators:

$$(1)\ \textit{it is impossible}; \qquad (2)\ \textit{it is trivial}\ldots.$$

We have seen that the first myth is correct, unless we add the prefix "pseudo".[16] The second myth, however, is completely false.

[14] Several physical devices for random number generation, using for instance electronic or radioactive noise, have been proposed but very few seem to have been inserted in an actual computer system.

[15] Some readers may think that Lehmer's definition is too vague. There have been many deep attempts for more precise formulation. See Knuth [17, pp. 149–179], who catches the flavor of the philosophical discussion of these matters and contributes to it himself.

[16] "Anyone who considers arithmetic methods of producing random numbers is, of course, in a state of sin ", John von Neumann (1951).

In a computer the fundamental concept is not a sequence of decimal random *digits*, but the **uniform random deviates**, i.e., a sequence of *mutually independent observations of a random variable* $U$ with a uniform distribution on $[0, 1)$; the density function of $U$ is thus (with a temporary notation)

$$f_1(u) = \begin{cases} 1, & \text{if } u \in (0,1); \\ 0, & \text{otherwise.} \end{cases}$$

Random deviates for other distributions, are generated by means of uniform deviates, e.g., the variable $X = a + (b-a)U$ is a *uniform deviate on* $(a, b)$.. Its density function is$f(x) = f_1((x-a)/(b-a))$. If $[a, b] = [0, 1]$ we usually write "uniform deviate" (without mentioning the interval). We often write "deviate" instead of "random deviate", when the meaning is evident from the context.

The most widely generators used for producing pseudo-random numbers are the **multiple recursive generator** based on linear recurrences of order $k$

$$x_i = a_1 x_{i-1} + \cdots + a_k x_{i-k} + c \mod m, \tag{1.5.2}$$

i.e., $x_i$ is the remainder obtained when the right hand side is divided by the modulus $m$. Here $m$ is a positive integer and the coefficients $a_1, \ldots, a_k$ belong to the set $\{0, 1, \ldots, m-1\}$. The state at step $i$ is $s_i = (x_{i-k+1}, \ldots, x_i)$ and the generator is started from a seed $s_{k-1} = (x_0, \ldots, x_{k-1})$. When $m$ is large the output can be taken as the number $u_i = x_i/m$. When $k = 1$, we obtain the classical **linear congruential generator**.

An important characteristic of a RNG is its **period**, which is the maximum length of the sequence before it begins to repeat. Note that if the algorithm for computing $x_i$ only depends on $x_{i-1}$, then the entire sequence repeats once the seed $x_0$ is duplicated.

A good RNG should have an extremely long period. If $m$ is a prime number and and if the coefficients $a_j$ satisfy certain conditions, then the generated sequence has the maximal period length $m^k - 1$; see Knuth [17].

The linear congruential generator defined by

$$x_i = 16807 x_{i-1} \mod (2^{31} - 1), \tag{1.5.3}$$

with period length $(2^{31} - 2)$, was proposed originally by Lewis, Goodman, and Miller (1969). It has been widely used in many software libraries for statistics, simulation and optimization. In the survey by Park and Miller [29] this generator was proposed as a "minimal standard" against which other generators should be judged. A similar generator but with the multiplier $7^7 = 823543$ was used in MATLAB 4.

Marsaglia [22] pointed out a theoretical weakness of all linear congruential generators. He showed that if $k$ successive random numbers $(x_{i+1}, \ldots, x_{i+k})$ at a time are generated and used to plot points in $k$-dimensional space, then they will lie on $(k-1)$-dimensional hyperplanes, and will not fill up the space. More precisely the values will lie on a set of, at most $(k!m)^{1/k} \sim (k/e)m^{1/k}$ equidistant parallel hyperplanes in the $k$-dimensional hypercube $(0, 1)^k$. When the number of hyperplanes is too small, this obviously is a strong limitation to the $k$-dimensional

uniformity. For example, for $m = 2^{31} - 1$ and $k = 3$, this is only about 1600 planes. This clearly may interfere with a simulation problem.

If the constants $m, a$ and $c$ are not very carefully chosen, there will be many fewer hyperplanes than the maximum possible. One such infamous example is the linear congruential generator with $a = 65539, c = 0$ and $m = 2^{31}$ used by IBM mainframe computers for many years.

Another weakness of linear congruential generators is that their low-order digits are much less random than their high-order digits. Therefore when only part of a generated random number is used one should pick the high-order digits.

One approach to better generators is to combine two RNGs. One possibility is to use a second RNG to shuffle the output of a linear congruential generator. In this way it is possible to get rid of some serial correlations in the output; see the generator ran1 described in Press et. al. [31, Chapter 7.1].

At the time of writing simplistic and unreliable RNGs still abound in some other commercial software products, despite the availability of much better alternatives. L'Ecuyer [19] reports on tests of RNGs used in some popular software products. Microsoft Excel uses the linear congruential generator

$$u_i = 9821.0u_{n-1} + 0.211327 \mod 1,$$

implemented directly for the $u_i$ in floating point arithmetic. Its period length depends on the precision of the arithmetic and it is not clear what it is. Microsoft Visual Basic uses a linear congruential generator with period $2^{24}$, defined by

$$x_i = 1140671485x_{i-1} + 12820163 \mod (2^{24}),$$

and takes $u_i = x_i/2^{24}$. The Unix standard library uses the recurrence

$$x_i = 25214903917x_{i-1} + 12820163 \mod (2^{48}),$$

with period length $2^{48}$ and sets $u_i = x_i/2^{48}$. The Java standard library uses the same recurrence but construct random deviates $u_i$ from $x_{2i}$ and $x_{2i+1}$.

One conclusion of recent tests is that when large sample sizes are needed all the above RNGs are unsafe to use and can fail decisively. It has been observed that to avoid misleading results the period length $\rho$ of the RNG needs to be such that generating $\rho^{1/3}$ numbers is not feasible. Thus a period length of $2^{24}$ or even $2^{48}$ may not be enough. Linear RNGs are also unsuitable for cryptographic applications, because the output is too predictable. For this reason, nonlinear generators have been developed, but these are in general much slower than the linear generators.

In MATLAB 5 and later versions the previous linear congruential generator has been replaced with a much better generator, based on ideas of G, Marsaglia. This generator has a 35 element state vector and can generate all the floating point numbers in the closed interval $[2^{-53}, 1 - 2^{-53}]$. Theoretically it can generate $2^{1492}$ values before repeating itself; see Moler [26]. If one generates one million random numbers a second it would take $10^{435}$ years before it repeats itself!

Some recently developed linear RNGs can generate huge samples of pseudo-random numbers very fast and reliably. The multiple recursive generator MRG32k3a

proposed by L'Ecuyer has a period near $2^{191}$. The **Mersenne twister** MT19937 by Matsumoto and Nishimura [25], the current "World Champion" among RNGs, has a period length of $2^{19937} - 1$!

### 1.5.3 Testing Pseudo-Random Number Generators

Many statistical tests have been adapted and extended for the examination of *arithmetic* methods of (pseudo-)random number generation, in use or proposed for digital computers. In these the observed frequencies (a histogram) for some random variable associated with the test, is compared with the theoretical frequencies on the hypothesis that the numbers are independent observations from a true sequence of random digits without bias. This is done by means of the famous $\chi^2$-test of K. Pearson [30][17], which we now describe.

Suppose that the space $S$ of the random variable is divided into a finite number $r$ of non-overlapping parts $S_1, \ldots, S_r$. These parts may be groups into which the sample values have been arranged for tabulation purposes. Let the corresponding group probabilities be $p_i = Pr(S_i)$, $i = 1, \ldots, r$, where $\sum_i p_i = 1$. We now form a measure of the deviation of the observed frequencies $\nu_1, \ldots, \nu_r$, $\sum_i \nu_i = n$, from the expected frequencies

$$\chi^2 = \sum_{i=1}^{r} \frac{(\nu_i - np_i)^2}{np_i} = \sum_{i=1}^{r} \frac{\nu_i^2}{np_i} - n. \tag{1.5.4}$$

It is known that as $n$ tends to infinity the distribution of $\chi^2$ tends to a limit independent of $P(S_i)$, which is the $\chi^2$-distribution with $r - 1$ degrees of freedom.

Now let $\chi_p^2$ be a value such that $Pr(\chi^2 > \chi_p^2) = p\%$. Here $p$ is chosen so small that we are practically certain that an event of probability $p\%$ will not occur in a single trial. The hypothesis is rejected if the observed value of $\chi^2$ is larger than $\chi_p^2$. Often a rejection level of 5% or 1% is used.

**Example 1.5.2.**

In $n = 4040$ throws with a coin, Buffon obtained $\nu = 2048$ heads and hence $n - \nu = 1992$ tails. Is this consistent with the hypothesis that there is a probability of $p = 1/2$ of throwing tails? Here we obtain

$$\chi^2 = \frac{(\nu_i - np)^2}{np} + \frac{(n - \nu - np)^2}{np_i} = 2\frac{(2048 - 2020)^2}{2020} = 0.776.$$

Using a rejection level of 5% we find from a table of the $\chi^2$-distribution with one degree of freedom that $\kappa_5^2 = 3.841$. Hence the hypothesis is accepted at this level.

Some test that have been used for testing RNGs are:

[17] This paper by the English mathematician Karl Pearson (1857–1936) is considered as one of the foundations of modern statistics. In it he gave several examples, e.g., he proved that some runs at roulette he had observed during a visit to Monte Carlo were so far from expectations that the odds against an honest wheel was about $10^{29}$ to one.

1. **Frequency test** This test is to find out if the generated numbers are equidistributed. One divides the possible outcomes in equal non-overlapping intervals and tallies the amount of numbers in each interval.

2. **Poker test** This test applies to generated digits, which are divided into non-overlapping groups of 5 digits. Within the groups we study some (unordered) combinations of interest in poker. These are given below together with their probabilities.

| | | |
|---|---|---|
| All different: | abcde | 0.3024 |
| One pair: | aabcd | 0.5040 |
| Two pairs: | aabbc | 0.1080 |
| Three of a kind: | aaabc | 0.0720 |
| Full house: | aaabb | 0.0090 |
| Four of a kind: | aaaab | 0.0045 |
| Five of a kind: | aaaaa | 0.0001 |

3. **Gap test** This test examines the length of "gaps" between occurrences of $U_j$ in a certain range. If $\alpha$ and $\beta$ are two numbers with $0 \le \alpha < \beta \le 1$, we consider the length of consecutive subsequences $U_j, U_{j+1}, \ldots, U_{j+r}$ in which $U_{j+r}$ lies between $\alpha$ and $\beta$ but $U_j, U_{j+1}, \ldots, U_{j+r-1}$ does not. This subsequence then represents a gap of length $r$.

   Working with single digits the gap equals the distance between two equal digits. The probability of a gap of length $r$ in this case equals

$$p_r = 0.1(1-0.1)^r = 0.1(0.9)^r, \quad r = 0, 1, 2, \ldots.$$

   Several other tests are described in Knuth [17, Sec. 3.3].

**Example 1.5.3.**

It is also important to test the serial correlation of the generated numbers. To test the two-dimensional behavior of a RNG we generated $10^6$ pseudo-random numbers $U_i$. We then placed the numbers each plot $(U_i, U_{i+1})$ in the unit square. A thin slice of the surface of the square 0.0001 wide by 1.0 high was the cut on its left side and stretched out horizontally. This corresponds to plotting only the pairs $(U_i, U_{i+1})$ such that $U_i < 0.0001$ (about 1000 points).

In Figure 1.6.2 we show the two plots from the generators in MATLAB 4 and MATLAB 5, respectively. The lattice structure is quite clear in the first plot. With the new generator no lattice structure is visible.

A good generator should have been analyzed theoretically and be supported by practical evidence from extensive statistical and other tests. Knuth [17, Chapter 3], ends his masterly chapter on Random Numbers with the following exercise: *Look at the subroutine library at your computer installation, and replace the random number generators by good ones. Try to avoid to be too shocked at what you find.* He has in the chapter pointed out both the important ideas, concepts and facts of the

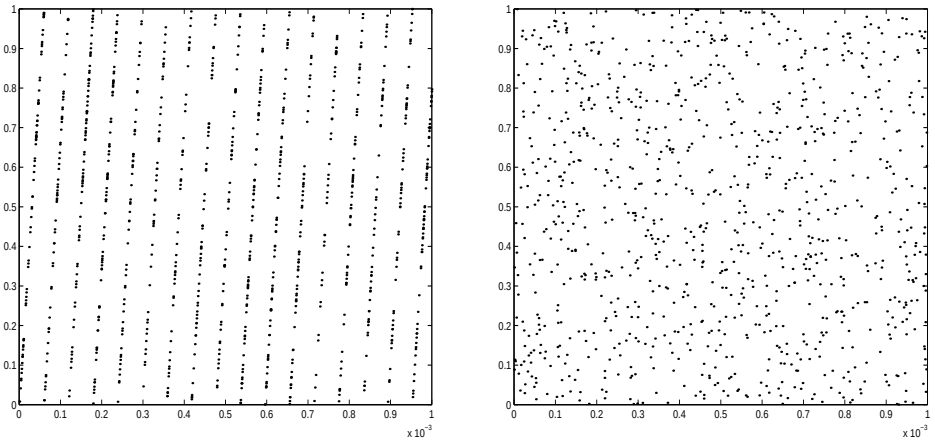


**Figure 1.5.2.** *Plots of pairs of* $10^6$ *random uniform deviates* $(U_i, U_{i+1})$ *such that* $U_i < 0.0001$. *Left:* MATLAB *4; Right:* MATLAB *5.*

topic, and also mentioned some scandalously poor random number generators that were in daily use for decades as standard tools in widely spread computer libraries. Although the generators in daily use have improved, *many are still not satisfactory.* L'Ecuyer [19] writes in 2001:

> *"Unfortunately, despite repeated warnings over the past years about certain classes of generators, and despite the availability of much better alternatives, simplistic and unsafe generators still abound in commercial software."*

## 1.5.4 Random Deviates for Other Distributions.

We have so far discussed how to generate sequences that behave as if they were random uniform deviates $U$ on $[0, 1)$. By arithmetic operations one can form random numbers with other distributions. A simple example is that $S = a + (b - a)U$ will be uniformly distributed on $[a, b)$. We can also easily generate a random integer between 1 and $k$; see Example 1.5.2.

Monte Carlo methods often call for other kinds of distributions, for example normal deviates. As we shall see, these can also be generated from a sequence of uniform deviates. Many of the tricks used to do this were originally suggested by John von Neumann in the early 1950s, but have since been improved and refined. We now exemplify, how to use uniform deviates to generate random deviates $X$ for some other distributions.

### Discrete Distributions

To make a random choice from a finite number $k$ *equally probable* possibilities is equivalent to generate a random integer $X$ between 1 and $k$. To do this we take a

random deviate $U$ uniformly distributed on $[0, 1)$ multiply by $k$ and take the integer part, and 1, i.e.

$$X = \lceil kU \rceil,$$

where $\lceil x \rceil$ denotes the smallest integer larger than or equal to $x$.There is a small error because the set of floating point numbers is finite, but this is usually negligible.

In a *more general situation*, we might want to give different probabilities to the values of a variable. Suppose we give the values $X = x_i$, $i = 1 : k$ the probabilities $p_i$, $i = 1 : k$; note that $\sum p_i = 1$. We can the generate a uniform number $U$ and let

$$X = \begin{cases} x_1, & \text{if } 0 \le U < p_1; \\ x_2, & \text{if } p_1 \le U < p_1 + p_2; \\ \vdots & \\ x_k, & \text{if } p_1 + p_2 + \cdots p_{k-1} \le U < 1. \end{cases}$$

If $k$ is large, and the sequence $\{p_i\}$ is irregular, may require some thought how to find $x$ quickly for a given $u$. See the analogous question to find a first guess to the root of Equation (1.5.5) below, and the discussion in Knuth [17, Sec. 3.4.1].

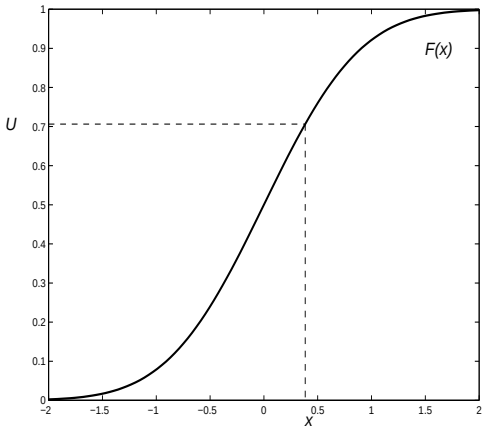



**Figure 1.5.3.** *Random number with distribution $F(x)$.*

### A General Transformation from $U$ to $X$

Suppose we want to generate numbers for a random variable $X$ with a given continuous or discrete distribution function $F(x)$. (In the discrete case, the graph of the distribution function becomes a staircase, see the formulas above.) A general method for this is to solve the equation

$$F(X) = U, \quad \text{or equivalently,} \quad X = F^{-1}(U), \tag{1.5.5}$$

see Figure 1.6.4. Because $F(x)$ is a nondecreasing function, and $Pr\{U \le u\} = u, \forall u \in [0, 1]$, equation (1) is proved by the line

$$Pr\{X \le x\} = Pr\{F(X) \le F(x)\} = Pr\{U \le F(x)\} = F(x).$$

How to solve (1.5.5) fast is often a problem with this method, and for some distributions we shall see better methods below.

**Exponential Deviates.**

As an example consider the exponential distribution with parameter $\lambda > 0$. This distribution occurs in queuing problems, e.g., in tele-communication, to model arrival and service times. The important property is that the intervals of time between two successive events are a sequence of exponential deviates. The exponential distribution with mean $1/\lambda$ has density function $f(t) = \lambda e^{-\lambda t}$, $t > 0$, and distribution function

$$F(x) = \int_0^x \lambda e^{-\lambda t}\, dt = 1 - e^{-\lambda x}. \tag{1.5.6}$$

Using the general rule given above, exponentially distributed random numbers $X$ can be generated as follows: Let $U$ be a uniformly distributed random number in $[0, 1]$. Solving the equation $1 - e^{-\lambda X} = U$, we obtain

$$X = -\lambda^{-1} \ln(1 - U).$$

A drawback of this method is that the evaluation of the logarithm is relatively slow.

One important use of exponentially distributed random numbers is in the generation of so-called **Poisson processes**. Such processes are often fundamental in models of telecommunications systems and other service systems. A Poisson process with frequency parameter $\lambda$ is a sequence of events characterized by the property that the probability of occurrence of an event in a short time interval $(t, t + \Delta t)$ is equal to $\lambda \cdot \Delta t + o(\Delta t)$, independent of the sequence of events previous to time $t$. In applications an "event" can mean a call on a telephone line, the arrival of a customer in a store, etc. For simulating a Poisson process one can use the important property that the intervals of time between two successive events are independent exponentially distributed random numbers.

**Normal Deviates.**

A normal deviate $N$ is a random variable with zero mean and unit standard deviation, and has the density function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}, \quad (m = 0,\ \sigma = 1).$$

A normal deviate with mean $m$ and standard deviation $\sigma$ is $m + \sigma N$; the density function is $(1/\sigma) f((x - m)/\sigma)$. The normal distribution function

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt = \frac{1}{2}\Big(1 + \operatorname{erf}\Big(\frac{x}{\sqrt{2}}\Big)\Big)$$

is related to the error function $\mathrm{erf}(x)$ introduced in Sec. 1.2.3 and is not an elementary function, In this case solving the equation (1.5.5) is time consuming.

Fortunately random normal deviates can be obtained in easier ways. In the **polar algorithm** a random point in the unit circle is generated as follows. Let $U_1, U_2$ be two independent, uniformly distributed random numbers on $[0, 1]$. Then the point $(V_1, V_2)$, where $V_i = 2U_i - 1$, $i = 1, 2$, is uniformly distributed in the square $[-1, 1] \times [-1, 1]$. We compute $S = V_1^2 + V_2^2$ and reject the point if it outside the unit circle, i.e. if $S > 1$. The remaining points are uniformly distributed on the unit circle.

For each accepted point we form

$$N_1 = V_1\sqrt{\frac{-2\ln S}{S}}, \qquad N_2 = V_2\sqrt{\frac{-2\ln S}{S}}. \tag{1.5.7}$$

It can be proved that $N_1$, $N_2$ are two independent, normally distributed random numbers with mean zero and standard deviation 1. We point out that $N_1, N_2$ can be considered to be rectangular coordinates of a point whose polar coordinates $(r, \phi)$ are determined by the equations

$$r^2 = N_1^2 + N_2^2 = -2\ln S, \quad \cos\phi = U_1/\sqrt{S}, \quad \sin\phi = U_2/\sqrt{S}.$$

Thus the problem is to show that the distribution function for a pair of independent, normally distributed random variables is rotationally symmetric (uniformly distributed angle) and that their sum of squares is exponentially distributed with mean 2; see Knuth [17, p. 123].

The polar algorithm, which was used for MATLAB 4, is moderately expensive. First, about $(1 - \pi/4) = 21.5\%$ of the uniform numbers are rejected because the generated point falls outside the unit circle. Further, the calculation of the logarithm contributes significantly to the cost. From MATLAB 5 on a more efficient table look-up algorithm developed by Marsaglia and Tsang [24] is used. This is called the "ziggurat" algorithm after the name of ancient Mesopotamian terraced temples mounds, that look like two-dimensional step functions. A popular description of the ziggurat algorithm is given by Moler [27]; see also [16].

### Chi-square Distribution

The chi-square distribution function $P(\chi^2, n)$ is related to the incomplete gamma function (see Sec. 3.?? by

$$P(\chi^2, n) = (n/2, \chi^2/2). \tag{1.5.8}$$

Its complement $Q(\chi^2, n) = 1 - P(\chi^2, n)$ is the probability that the observed chi-square will exceed the value $\chi^2$ even for a correct model. Subroutines for evaluating the $\chi^2$-distribution function as well as other important statistical distribution functions are given in [31, Sec. 6.2–6.3].

Numbers belonging to the **chi-square distribution** can also be obtained by using the definition of the distribution. If $N_1, N_2, \ldots, N_n$ are normal deviates with

mean 0 and variance 1, the number

$$Y_n = N_1^2 + N_2^2 + \cdots + N_n^2$$

is distributed as $\chi^2$ with $n$ degrees of freedom.

#### Other Distributions

Methods to generate random deviates with, e.g., Poisson, gamma and binomial distribution, are described in Knuth [17, Sec. 3.4]) and Press et al. [31, Chapter 7.3]. A general method, introduced by G. Marsaglia [21], is the **rectangle-wedge-tail method**. It been further developed and applied by Marsaglia and coauthors, see references in Knuth [17, Sec. 3.4]). The **rejection method** is based on ideas of von Neumann.Several authors, notably G. Marsaglia, have developed powerful combinations of rejection methods and the rectangle-wedge-tail method.

### 1.5.5 Reduction of Variance.

From statistics, we know that if one makes $n$ independent observations of a quantity whose standard deviation is $\sigma$, then the standard deviation of the mean is $\sigma/\sqrt{n}$. Hence to increase the accuracy by a factor of 10 (say) we have to increase the number of experiments $n$ by a factor 100.

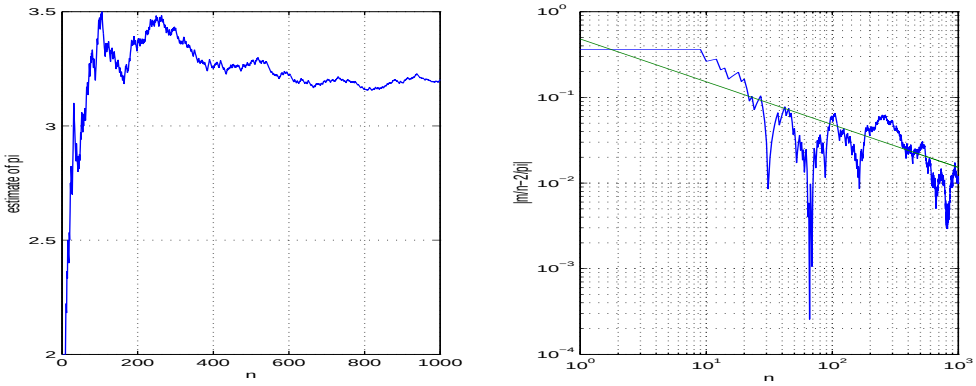



**Figure 1.5.4.** *The left part shows how the estimate of $\pi$ varies with the number of throws. The right part compares $|m/n - 2/\pi|$ with the standard deviation of $m/n$. The latter is inversely proportional to $n^{1/2}$, and is therefore a straight line in the figure.*

**Example 1.5.4.**

In 1777 Buffon[18] carried out a probability experiment by throwing sticks over his shoulder onto a tiled floor and counting the number of times the sticks fell across

[18] Compte de Buffon (1707–1788), French natural scientist that contributed to the understanding of probability. He also computed the probability that the sun would continue to rise after having been observed to rise on $n$ consecutive days.

the lines between the tiles. He stated that the favourable cases correspond "to the area of part of the cycloid whose generating circle has diameter equal to the length of the needle". To simulate Buffon's experiment we suppose a board is ruled with equidistant parallel lines and that a needle fine enough to be considered a segment of length $l$ not longer than the distance $d$ between consecutive lines is thrown on the board. The probability is then $2l/(\pi d)$ that it will hit one of the lines.

The Monte Carlo method and this game can be used to approximate the value of $\pi$. Take the distance $\delta$ between the center of the needle and the lines and the angle $\phi$ between the needle and the lines to be random numbers. By symmetry we can choose these to be rectangularly distributed on $[0, d/2]$ and $[0, \pi/2]$, respectively. Then the needle hits the line if $\delta < (l/2)\sin\phi$.

We took $l = d$. Let $m$ be the number of hits in the first $n$ throws in a Monte Carlo simulation with 1000 throws. The expected value of $m/n$ is therefore $2/\pi$, and so $2n/m$ is an estimate of $\pi$ after $n$ throws. In the left part of Fig. 1.5.3 we see, how $2n/m$ varies with $n$ in one simulation. The right part compares $|m/n - 2/\pi|$ with the standard deviation of $m/n$, which equals $\sqrt{2/\pi(1-2/\pi)/n}$ and is, in the log-log-diagram, represented by a straight line, the slope of which is $-1/2$. This can be taken as a test that the random number generator in MATLAB is behaving correctly! (The spikes, directed downwards in the figure, typically indicate where $m/n - 2/\pi$ changes sign.)

A more efficient way than increasing the number of samples, often is to instead try to decrease the value of $\sigma$ by redesigning the experiment in various ways. Assume that one has two ways (which require the same amount of work) of carrying out an experiment, and these experiments have standard deviations $\sigma_1$ and $\sigma_2$ associated with them. If one repeats the experiments $n_1$ and $n_2$ times (respectively), the same precision will be obtained if $\sigma_1/\sqrt{n_1} = \sigma_2/\sqrt{n_2}$, or

$$n_1/n_2 = \sigma_1^2/\sigma_2^2. \tag{1.5.9}$$

Thus if a variance reduction by a factor $k$ can be achieved, then the number of experiments needed is also reduced by the same factor $k$.

An important means of reducing the variance of estimates obtained from the Monte Carlo method is to use **antithetic sequences**. For example, if $U_i$ is a series of random uniform deviates on $[0, 1]$ then $U_i' = 1 - U_i$ are also uniformly distributed on $[0, 1]$. For example, from the sequence in (1.5.1) we get the sequence

$$0.443065, 0.970545, 0.182765, 0.564115, 0.186495, 0.236975, \ldots, \tag{1.5.10}$$

which is the antithetic sequence derived from (1.5.1). Antithetic sequences of normally distributed numbers are obtained simply by reversing the sign of the original sequence.

Roughly speaking, since the influence of chance has opposing effects in the two antithetic experiments, one can presume that the effect of chance on the *means* is much less than the effect of chance in the original experiments. In the following example we show how to make a quantitative estimate of the reduction of variance accomplished with the use of antithetic experiments.

**Example 1.5.5.**

Suppose the numbers $x_i$ are the results of statistically independent measurements of a quantity with expected value $m$, and standard deviation $\sigma$. Set

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

Then $\bar{x}$ is an estimate of $m$, and $s/\sqrt{n-1}$ is an estimate of $\sigma$.

In ten simulation and their antithetic experiments of a service system the following values were obtained for the treatment time:

$$685 \quad 1,045 \quad 718 \quad 615 \quad 1,021 \quad 735 \quad 675 \quad 635 \quad 616 \quad 889\,.$$

From this experiment the mean for the treatment time is estimated as $763,4$, and the standard deviation 51.5, which we write $763 \pm 52$. Using an antithetic series, one got the following values:

$$731 \quad 521 \quad 585 \quad 710 \quad 527 \quad 574 \quad 607 \quad 698 \quad 761 \quad 532\,.$$

The series means is thus

$$708 \quad 783 \quad 652 \quad 662 \quad 774 \quad 654 \quad 641 \quad 666 \quad 688 \quad 710\,,$$

from which one gets the estimate $694 \pm 16$.

When one instead supplemented the first sequence with ten values using independent random numbers, the estimate $704 \pm 36$ using all twenty values was obtained. These results indicate that, in this example, using antithetical sequence produces the desired accuracy with $(16/36)^2 \approx 1/5$ of the work required if completely independent random numbers are used. This rough estimate of the work saved is uncertain, but indicates that it is profitable to use the technique of antithetic series.

**Example 1.5.6.**

Monte Carlo methods have been successfully used to study queuing problems. A well known example is a study by Bailey [3] to determine how to give appointment times to patients at a polyclinic. The aim is to find a suitable balance between the mean waiting times of both patients and doctors. This problem was in fact solved analytically—much later—after Bailey already had gotten the results that he wanted; this situation is not uncommon when numerically methods (and especially Monte Carlo methods) have been used.

Suppose that $k$ patients have been booked at the time $t = 0$ (when the clinic opens), and that the rest of the patients (altogether 10) are booked at intervals of 50 time units thereafter. The time of treatment is assumed to be exponentially distributed with mean 50. (Bailey used a distribution function which was based on empirical data.) Three alternatives, $k = 1, 2, 3$, are to be simulated. *By using the same random numbers for each $k$ (hence the same treatment times) one gets a* **reduced variance** *in the estimate of the change in waiting times as $k$ varies.*

**Table 1.5.1.** *Simulation of patients at a polyclinic.*

| | $k=1$ | | | | | $k=2$ | |
|---|---|---|---|---|---|---|---|
| $P_{no}$ | $P_{arr}$ | $T_{beg}$ | $R$ | $T_{time}$ | $T_{end}$ | $P_{arr}$ | $T_{end}$ |
| 1 | 0* | 0 | 211 | 106 | 106 | 0* | 106 |
| 2 | 50 | 106 | 3 | 2 | 108 | 0 | 108 |
| 3 | 100 | 108 | 53 | 26 | 134 | 50 | 134 |
| 4 | 150* | 150 | 159 | 80 | 230 | 100 | 214 |
| 5 | 200 | 230 | 24 | 12 | 242 | 150 | 226 |
| 6 | 250* | 250 | 35 | 18 | 268 | 200 | 244 |
| 7 | 300* | 300 | 54 | 27 | 327 | 250* | 277 |
| 8 | 350* | 350 | 39 | 20 | 370 | 300* | 320 |
| 9 | 400* | 400 | 44 | 22 | 422 | 350* | 372 |
| 10 | 450* | 450 | 13 | 6 | 456 | 400* | 406 |
| $\Sigma$ | 2,250 | | | 319 | 2,663 | 1,800 | 2,407 |

The computations are shown in the Table 1.5.1. The following abbreviations are used: $P$ = patient, $D$ = doctor, $T$ = treatment. An asterisk indicates that the patient did not need to wait. In the table $P_{arr}$ follows from the rule for booking patients given previously. The treatment time $T_{time}$ equals $50R/100$ where $R$ are exponentially distributed numbers with mean 100 taken from a table. $T_{beg}$ equals the larger of the number $P_{arr}$ (on the same row) and $T_{end}$ (in the row just above), where $T_{end} = T_{beg} + T_{treat}$.

From the table we find that for $k = 1$ the doctor waited the time $D = 456 - 319 = 137$; the total waiting time for patients was $P = 2,663 - 2,250 - 319 = 94$. For $k = 2$ the corresponding waiting times were $D = 406 - 319 = 87$ and $P = 2,407 - 1,800 - 319 = 288$. Similar calculations for $k = 3$ gave $D = 28$ and $P = 553$ (see Fig. 1.5.5). For $k \geq 4$ the doctor never needs to wait.

One cannot, of course, draw any tenable conclusions from one experiment. More experiments should be made in order to put the conclusions on statistically solid ground. Even isolated experiments, however, can give valuable suggestions for the planning of subsequent experiments, or perhaps suggestions of appropriate approximations to be made in the analytic treatment of the problem. The large-scale use of Monte Carlo methods requires careful planning to avoid drowning in in enormous quantities of unintelligible results.

Two methods for **reduction of variance** have here been introduced: *antithetic sequence of random numbers* and the technique of using *the same random numbers in corresponding situations*. The latter technique is used when studying the changes in behavior of a system when a certain parameter is changed (e.g., the parameter $k$ in Exercise 4). (Note that we just have restart the RNG using the same seed.)

Many effective methods have been developed for reducing variance, e.g., **im-**

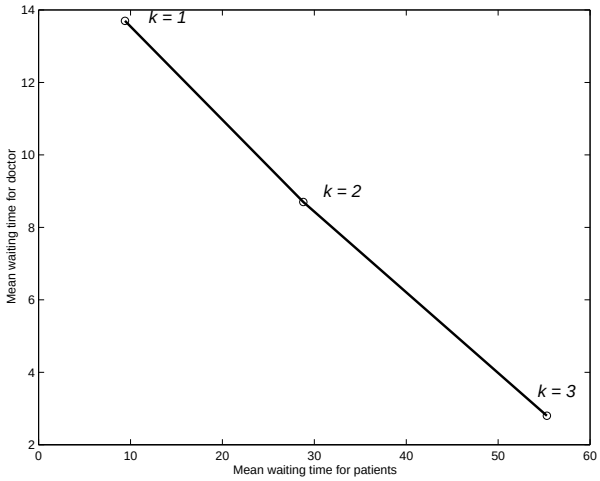



**Figure 1.5.5.** *Mean waiting times for doctor/patients at polyclinic.*

**portance sampling** and **splitting techniques** (see Hammersley and Handscomb [13]).

## Review Questions

1. What is a uniformly distributed random number?
2. Describe a general method for obtaining random numbers with a given discrete or continuous distribution. Give examples of their use.
3. What are the most important properties of a Poisson process? How can one generate a Poisson process with the help of random numbers?
4. What is the mixed congruential method for generating pseudo-random numbers? What important difference is there between the numbers generated by this method and "genuine" random numbers?
5. Give three methods for reduction of variance in estimates made with the Monte Carlo method, and explain what is meant by this term. Give a quantitative connection between reducing variance and decreasing the amount of computation needed in a given problem?

## Problems and Computer Exercises

1. (C. Moler) Consider the toy random number generator, $x_i = ax_i \bmod m$, with $a = 13$, $m = 31$ and start with $x_0 = 1$. Show that this generates a sequence consisting of a permutation of all integers from 1 to 30, and then repeats itself. Thus this generator has the period equal to $m-1 = 30$, equal to the maximum

possible.

**2.** Simulate (say) 360 throws with two usual dices. Denote the sum of the number of dots on the two dice in the $n$'th throw by $Y_n$, $2 \le Y_n \le 12$. Tabulate or draw a histogram, i.e., the (absolute) frequency of the occurrence of $j$ dots versus $j$, $j = 2:12$. Make a conjecture about the true value of $P(Y_n = j)$. Try to confirm it by repeating the experiment with fresh uniform random numbers. When you have found the right conjecture, it is not hard to prove it.

**3.** (a) Let $X, Y$ be independent uniform random numbers on the interval $[0, 1]$. Show that $P(X^2 + Y^2 \le 1) = \pi/4$, and estimate this probability by a Monte Carlo experiment with (say) 1000 pairs of random numbers. For example, make graphical output like in the Buffon needle problem.

(b) Make an antithetic experiment, and take the average of the two results. Is the average better than one can expect if the second experiment had been independent of the first one.

(c) Estimate similarly the volume of the four-dimensional unit ball. If you have enough time, use more random numbers. (The exact volume of the unit ball is $\pi^2/2$.)

**4.** A famous result by P. Diaconis asserts that it takes approximately $\frac{3}{2}\log_2 52 \approx 8.55$ riffle shuffles to randomize a deck of 52 cards, and that randomization occurs abruptly according to a "cutoff phenomenon". (For example, after six shuffles the deck is still far from random.)

The following definition can be used for simulating a riffle shuffle. The deck of cards is first cut roughly in half according to a binomial distribution, i.e. the probability that $\nu$ cards are cut is $\frac{n}{\nu}/2^n$. The two halves are then riffled together by dropping cards roughly alternately from each half onto a pile, with the probability of a card being dropped from each half being proportional to the number of cards in it.

Write a program that uses uniform random numbers (and perhaps uses the the formula $X = \lceil kR \rceil$ for several values of $k$ ) to simulate a random "shuffle" of a deck of 52 cards according to the above precise definition. This is for a *numerical* game; do not spend time on drawing beautiful hearts, clubs etc.

**5.** Brownian motion is the irregular motion of dust particles suspended in a fluid, being bombarded by molecules in a random way. Generate two sequences of random normal deviates $a_i$ and $b_i$, and use these to simulate Brownian motion by generating a path defined by the points $(x_i, y_j)$, where $x_0 = y_0 = 0$, $x_i = x_{i-1} + a_i$, $y_i = y_{i-1} + b_i$. Plot each point and connect the points with a straight line to visualize the path.

**6.** Repeat the simulation in the queuing problem in Example 1.5.6 for $k = 1$ and $k = 2$ using the sequence of exponentially distributed numbers $R$

$$13 \quad 365 \quad 88 \quad 23 \quad 154 \quad 122 \quad 87 \quad 112 \quad 104 \quad 213\,,$$

antithetic to that used in Example 1.5.6. Compute the mean of the waiting times for the doctor and for all patients for this and the previous experiment.

**7.** A target with depth $2b$ and very large width is to be shot at with a can-

non. (The assumption that the target is very wide makes the problem one-dimensional.) The distance to the center of the target is unknown, but estimated to be $D$. The difference between the actual distance and $D$ is assumed to be a normally distributed variable $X$ with zero mean and standard deviation $\sigma_1$.

One shoots at the target with a salvo of three shots, which are expected to travel a distance $D-a$, $D$ and $D+a$, respectively. The difference between the actual and the expected distance traveled is assumed to be a normally distributed random variable with zero mean and standard deviation $\sigma_2$; the resulting error component in the three shots is denoted by $Y_{-1}, Y_0, Y_1$. We further assume that these three variables are stochastically independent of each other and $X$.

One wants to know how the probability of at least one "hit" in a given salvo depends on $a$ and $b$. Use normally distributed pseudo-random numbers to shoot ten salvos and determine for each salvo, the least value of $b$ for which there is at least one "hit" in the salvo. Show that this is equal to

$$\min_k |X - (Y_k + ka)|, \quad k = -1, 0, 1.$$

Fire an "antithetic salvo" for each salvo.

Graph using $\sigma_1 = 3$, $\sigma_2 = 1$, for both $a = 1$ and $a = 2$ using the same random numbers curves, which give the probability of a hit as a function of the depth of the target.

## Notes and References

The development of Numerical Analysis during he period when the foundation was lade in the 16th through the 19th century is traced in Goldstine [11]. An acount of the developments in the 20th Century is found in [5]. An eloquent essay on the foundations of computational mathematics and its relation to other fields is given by Baxter and Iserles [4]. Many of the methods and problems introduced in this introductory chapter will be studied in more detail in later chapters and volumes. Numerical quadrature methods are studied in Chapter 5 and iterative methods for solving a single nonlinear equation in Chapter 6.

The later chapters in this book assumes a working knowledge in numerical linear algebra. In Appendix A notations and basic results on Linear Vector Spaces and Matrix Computations are given. For a more elementary introduction to Linear Algebra we refer to one of several good textbooks, e.g., Leon [20] and Strang [34]. Computational aspects of numerical linear algebra will be treated in depth in Volume II. Gaussian elimination and iterative methods for linear systems are covered in Volume II, Chapters 7 and 10, respectively. The numerical solution of ordinary and partial differential equations are treated in Volume III.

A comprehensive source of information on all aspects of random numbers is given by Knuth [17]. A good reference on the current state of the art is the monograph by Niederreiter [28, 1992]. A more application oriented overview is

found in Press et al. [31, Chapter 7]. Guidelines for choosing a good random number generator are given in Marsaglia [23] and L'Ecuyer [18]. Hellekalek [14] explains the art to access random number generators for practitioners.

# Bibliography


[1] Milton Abramowitz and Irene A. Stegun (eds.). *Handbook of Mathematical Functions*. Dover, New York, NY, 1965.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, editors. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, third edition, 1999.

[3] N. T. J. Bailey. A study of queues and appointment systems in hospital outpatient departments, with special reference to waiting times. *J. Roy. Stat. Soc.*, 3:14:185ff, 1951.

[4] B. J. C. Baxter and Arieh Iserles. On the foundations of computational mathematics. In P. G. Ciarlet and F. Cucker, editors, *Handbook of Numerical Analysis*, pages 3–34. North Holland Elsevier, Amsterdam, 2002.

[5] Claude Brezinski and Luc Wuytack. Numerical analysis in the twentieth century. In Claude Brezinski and L. Wuytack, editors, *Numerical Analysis: Historical Developments in the 20th Century*, pages 1–40. North Holland Elsevier, Amsterdam, 2001.

[6] RAND Corporation. *A Million Random Digits and 100,000 Normal Deviates*. Free Press, Glencoe, IL, 1955.

[7] Germund Dahlquist. Preliminär rapport om premieobligationsdragning med datamaskin. (in swedish), Riksgäldskontoret, Stockholm, 1962.

[8] Germund Dahlquist and Åke Björck. *Numerical Methods*. Dover, Mineola, NY, 2004.

[9] Terje O. Espelid. On floating-point summation. *SIAM Review*, 37:603–607, 1995.

[10] George E. Forsythe and Cleve B. Moler. *Computer Solution of Linear Algebraic Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1967.

[11] H. H. Goldstine. *A History of Numerical Analysis from the 16th through the 19th Century*. Springer-Verlag, New York, 1977.

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[13] J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods*. Methuen, London, UK, 1964.

[14] Peter Hellekalek. Good random number generators are (not so) easy to find. *Math. Comput. Simulation*, 46:485–505, 1998.

[15] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, second edition, 2002.

[16] David Kahaner, Cleve B. Moler, and Stephen Nash. *Numerical Methods and Software*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[17] Donald E. Knuth. *The Art of Computer Programming, Vol. 2. Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1997.

[18] Pierre L'Ecuyer. Efficient and portable combined random number generators. *Comm. ACM*, 31:6:742–774, 1988.

[19] Pierre L'Ecuyer. Software for uniform random number generation: Distinguishing the good and bad. In *Proc. 2001 Winter Simulation Conference*, pages 95–105. IEEE Press, Pistacaway, NJ, 2001.

[20] Steven J. Leon. *Linear Algebra with Applications*. Macmillan, New York, fourth edition, 1994.

[21] George Marsaglia. Expressing a random variable in terms of uniform random variables. *Ann. Math. Stat.*, 32:894–898, 1961.

[22] George Marsaglia. Random numbers falls mainly in the planes. *Proc. Nat. Acad. Sci.*, 60:5:25–28, 1968.

[23] George Marsaglia. A current view of random number generators. In L. Billard, editor, *Computer Science and Statistics: The Interface*, pages 3–10. Elsevier Science Publishers, Amsterdam, 1985.

[24] George Marsaglia and W. W. Tsang. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM J. Sci. Stat. Comput.*, 5:2:349–360, 1984.

[25] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Modeling Comput. Software*, 8:1:3–30, 1998.

[26] Cleve Moler. Random thoughts, $10^{435}$ years is a very long time. *MATLAB News and Notes*, Fall, 1995.

[27] Cleve Moler. Normal behavior. *MATLAB News and Notes*, Spring, 2001.

[28] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, Philadelphia, PA, 1992.

[29] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Comm. ACM*, 22:1192–1201, 1988.

[30] K. Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Phil. Mag. Series 5*, 50:p. 157–175, 1900.

[31] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in Fortran; The Art of Scientific Computing*. Cambridge University Press, Cambridge, GB, second edition, 1992.

[32] Lewis F. Richardson. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with application to the stress in a masonry dam. *Philos. Trans. Roy. Soc.*, A210:307–357, 1910.

[33] George W. Stewart. *Matrix Algorithms Volume I: Basic Decompositions*. SIAM, Philadelphia, PA, 1998.

[34] Gilbert Strang. *Linear Algebra and Its Applications*. Academic Press, New York, fourth edition, 2005.

[35] V. Strassen. Gaussian elimination is not optimal. *Numer. Math..*, 13:354–356, 1969.

# Index