Chapter 1

Principles of Numerical Calculations

It is almost impossible to identify a mathematical theory no matter how "pure," that has never influenced numerical reasoning.
—B. J. C. Baxter and Arieh Iserles

1.1 Common Ideas and Concepts

Although numerical mathematics has been used for centuries in one form or another within many areas of science and industry,[1] modern scientific computing using electronic computers has its origin in research and developments during the Second World War. In the late 1940s and early 1950s the foundation of numerical analysis was laid as a separate discipline of mathematics. The new capability of performing billions of arithmetic operations cheaply has led to new classes of algorithms, which need careful analysis to ensure their accuracy and stability.

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. In many cases, one thereby reduces the problem to a sequence of linear problems—for example, linear systems of differential equations. Such an approach can quite often lead to concepts and points of view which, at least qualitatively, can be used even in the unreduced problems.

Recent developments have enormously increased the scope for using numerical methods. Gains in problem solving capabilities mean that today one can treat much more complex and less simplified problems through massive amounts of numerical calculations. This has increased the interaction of mathematics with science and technology.

In most numerical methods, one applies a small number of general and relatively simple ideas. These are then combined with one another in an inventive way and with such

[1] The Greek mathematician Archimedes (287–212 B.C.), Isaac Newton (1642–1727), and Carl Friedrich Gauss (1777–1883) were pioneering contributors to numerical mathematics.

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 orders 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. These may occur as subproblems or computational details of larger problems, though as a rule they more often occur in a less pure form and on a larger scale than they do here. When we present and analyze numerical methods, to some degree we use the same approach which was first mentioned 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 us in more difficult problems.

It is important to keep 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 are needed.

1.1.1 Fixed-Point Iteration

One of the most frequently occurring ideas in numerical calculations is **iteration** (from the Latin *iterare*, "to plow once again") 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 (usually) 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

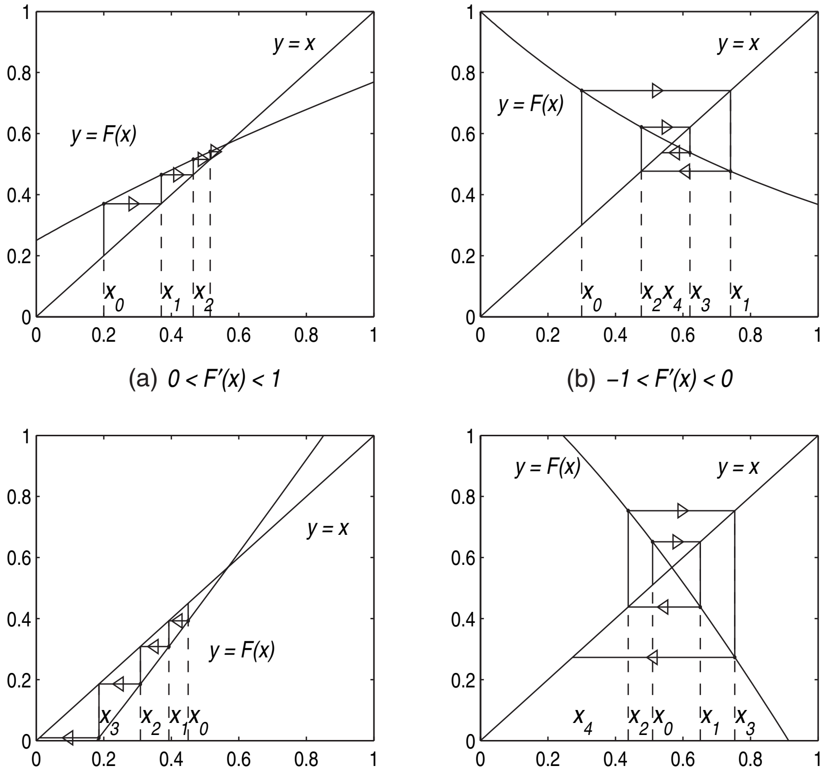
$$x_1 = F(x_0), \qquad x_2 = F(x_1), \qquad x_3 = F(x_2), \dots . \tag{1.1.2}$$

Each computation of the type $x_{n+1} = F(x_n)$, $n = 0, 1, 2, \dots,$ is called a **fixed-point iteration**. As n grows, we would like the numbers x_n to be better and better estimates of the desired root. If the sequence $\{x_n\}$ converges to a limiting value α, then we have

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

and thus $x = \alpha$ satisfies the equation $x = F(x)$. One can then stop the iterations when the desired accuracy has been attained.

A geometric interpretation of fixed point iteration is shown in Figure 1.1.1. A root of (1.1.1) is given by the abscissa (and ordinate) of an intersecting point of the curve $y = F(x)$ and the line $y = x$. Starting from x_0, the point $x_1 = F(x_0)$ 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); from there we draw a vertical line to $(x_1, F(x_1)) = (x_1, x_2)$,



(a) $0 < F'(x) < 1$ (b) $-1 < F'(x) < 0$

(c) $F'(x) > 1$ (d) $F'(x) < -1$

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

and so on in a "staircase" pattern. In Figure 1.1.1(a) it is obvious that the sequence $\{x_n\}$ converges monotonically to the root α. Figure 1.1.1(b) shows a case where F is a decreasing function. There we also have convergence, but not monotone convergence; the successive iterates x_n lie alternately to the right and to the left of the root α. In this case the root is bracketed by any two successive iterates.

There are also two divergent cases, exemplified by Figures 1.1.1(c) and (d). 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 of calculus we have

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

where ξ_n lies between x_n and α. 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 α is called a **point of attraction**. The convergence is faster the smaller $|F'(\alpha)|$ is.

If $|F'(\alpha)| > 1$, then α is a **point of repulsion** and the iteration diverges.

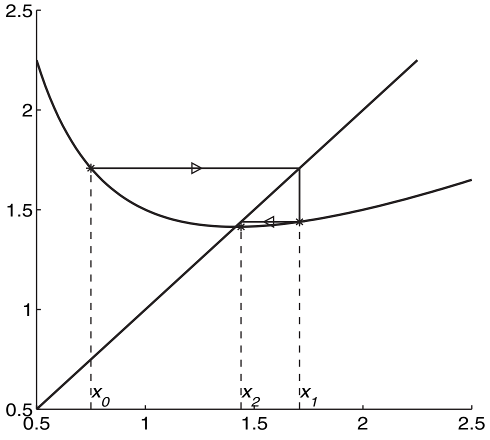


Figure 1.1.2. *The fixed-point iteration* $x_{n+1} = (x_n + c/x_n)/2$, $c = 2$, $x_0 = 0.75$.

Example 1.1.1.

The square root of $c > 0$ satisfies the equation $x^2 = c$, which can also be written $x = c/x$ or $x = 1/2(x + c/x)$. This suggests the fixed-point iteration

$$x_{n+1} = \frac{1}{2}\left(x_n + c/x_n\right), \quad n = 1, 2, \dots, \tag{1.1.3}$$

which is the widely used **Heron's rule**.[2] The curve $y = F(x)$ is in this case a hyperbola (see Figure 1.1.2).

From (1.1.3) follows

$$x_{n+1} \pm \sqrt{c} = \frac{1}{2}\left(x_n \pm 2\sqrt{c} + \frac{c}{x_n}\right) = \frac{(x_n \pm \sqrt{c})^2}{2x_n};$$

that is,

$$\frac{x_{n+1} - \sqrt{c}}{x_{n+1} + \sqrt{c}} = \left(\frac{x_n - \sqrt{c}}{x_n + \sqrt{c}}\right)^2. \tag{1.1.4}$$

We can take $e_n = \frac{x_n - \sqrt{c}}{x_n + \sqrt{c}}$ to be a measure of the error in x_n. Then (1.1.4) reads $e_{n+1} = e_n^2$ and it follows that $e_n = e_0^{2^n}$. If $|x_0 - \sqrt{c}| \neq |x_0 + \sqrt{c}|$, then $e_0 < 1$ and x_n converges to a square root of c when $n \to \infty$. Note that the iteration (1.1.3) can also be used for complex values of c.

For $c = 2$ and $x_0 = 1.5$, we get $x_1 = 1/2(1.5 + 2/1.5) = 15/12 = \mathbf{1.41}66666\dots$, and

$$x_2 = \mathbf{1.414215}\,686274, \qquad x_3 = \mathbf{1.414213\,56237}5$$

(correct digits shown in boldface). This can be compared with the exact value $\sqrt{2} = 1.414213\,562373\dots$. As can be seen from Figure 1.1.2, a rough value for x_0 suffices. The

[2] Heron made important contributions to geometry and mechanics. He is believed to have lived in Alexandria, Egypt, during the first century A.D.

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

$$\lim_{n\to\infty} \frac{|x_{n+1} - \sqrt{c}|}{|x_n - \sqrt{c}|^2} = C$$

for some constant $0 < C < \infty$, which is an example of what is known as **quadratic convergence**. Roughly, if x_n has t correct digits, then x_{n+1} will have at least $2t - 1$ correct digits.

The above iteration method is used quite generally on both pocket calculators and larger computers for calculating square roots.

Iteration is one of the most important aids for 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 (1.1.1) is frequently called the **fixed-point form**, since the root α is a fixed point of the mapping F. An equation may not be given in this form originally. One has a certain amount of choice in the rewriting of an 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 $x_{n+1} = c/x_n$ gives a sequence which alternates between x_0 (for even n) and c/x_0 (for odd n)—the sequence does not converge for any $x_0 \neq \sqrt{c}$!

1.1.2 Newton's Method

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 α is not known, this cannot be applied literally. But if we use x_n as an approximation, this leads to the choice $F(x) = x - f(x)/f'(x)$, or the iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{1.1.5}$$

This is the celebrated **Newton's method**.[3] We shall derive it in another way below.

[3] Isaac Newton (1642–1727), English mathematician, astronomer, and physicist invented infinitesimal calculus independently of the German mathematician and philosopher Gottfried W. von Leibniz (1646–1716).

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

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n} = \frac{1}{2}\Big(x_n + \frac{c}{x_n}\Big), \quad n = 0, 1, 2, \dots, \tag{1.1.6}$$

which is the fast method in Example 1.1.1. More generally, Newton's method applied to the equation $f(x) = x^p - c = 0$ can be used to compute $c^{1/p}$, $p = \pm 1, \pm 2, \dots$, from the iteration

$$x_{n+1} = x_n - \frac{x_n^p - c}{p x_n^{p-1}}.$$

This can be written as

$$x_{n+1} = \frac{1}{p}\left((p-1)x_n + \frac{c}{x_n^{p-1}}\right) = \frac{x_n}{(-p)}[(1-p) - c x_n^{-p}]. \tag{1.1.7}$$

It is convenient to use the first expression in (1.1.7) when $p > 0$ and the second when $p < 0$. With $p = 2, 3$, and -2, respectively, this iteration formula is used for calculating $\sqrt{c}$, $\sqrt[3]{c}$, and $1/\sqrt{c}$. Also $1/c$, $(p = -1)$ can be computed by the iteration

$$x_{n+1} = x_n + x_n(1 - c x_n) = x_n(2 - c x_n),$$

using only multiplication and addition. In some early computers, which lacked division in hardware, this iteration was used to implement division, i.e., b/c was computed as $b(1/c)$.

Example 1.1.2.

We want to construct an algorithm based on Newton's method for the efficient calculation of the square root of any given floating-point number a. If we first shift the mantissa so that the exponent becomes even, $a = c \cdot 2^{2e}$, and $1/2 \le c < 2$; then

$$\sqrt{a} = \sqrt{c} \cdot 2^e.$$

We need only consider the reduced range $1/2 \le c \le 1$ since for $1 < c \le 2$ we can compute $\sqrt{1/c}$ and invert.[4]

To find an initial approximation x_0 to start the Newton iterations when $1/2 \le c < 1$, we can use linear interpolation of $x = \sqrt{c}$ between the endpoints $1/2, 1$, giving

$$x_0(c) = \sqrt{2}(1 - c) + 2(c - 1/2)$$

($\sqrt{2}$ is precomputed). The iteration then proceeds using (1.1.6).

For $c = 3/4$ ($\sqrt{c} = 0.86602540378444$) the result is $x_0 = (\sqrt{2} + 2)/4$ and (correct digits are in boldface)

$$x_0 = \mathbf{0.8}5355339059327, \qquad x_1 = \mathbf{0.866}11652351682,$$
$$x_2 = \mathbf{0.86602540}857756, \qquad x_3 = \mathbf{0.86602540378444},$$

[4] Since division is usually much slower than addition and multiplication, this may not be optimal.

The quadratic rate of convergence is apparent. Three iterations suffice to give about 16 digits of accuracy for all $x \in [1/2, 1]$.

Newton's method is based on **linearization**. This means that *locally*, i.e., in a small neighborhood of a point, *a more complicated function is approximated with a linear function*. In the solution of the equation $f(x) = 0$, this means geometrically that we seek the intersection point between the x-axis and the curve $y = f(x)$; see Figure 1.1.3. Assume that 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

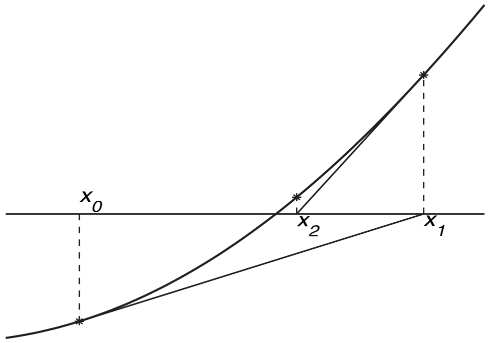


Figure 1.1.3. *Geometric interpretation of Newton's method.*

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

by setting $y = 0$ we obtain 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. But 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, 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 α, the iterations will converge rapidly (see Figure 1.1.3), but there are also cases of divergence.

An alternative to 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 Figure 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 in more detail in Sec. 6.3.1.

Newton's method can be generalized to yield a quadratically convergent method for solving a **system of nonlinear equations**

$$f_i(x_1, x_2, \ldots, x_n) = 0, \quad i = 1 : n. \tag{1.1.8}$$

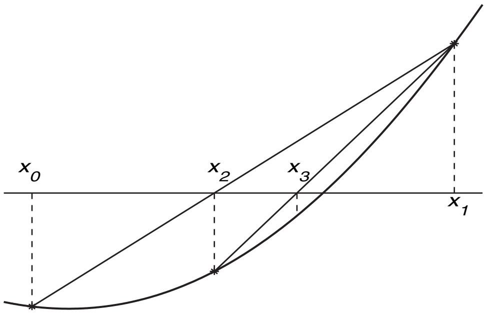


Figure 1.1.4. *Geometric interpretation of the secant method.*

Such systems arise in many different contexts in scientific computing. Important examples are the solution of systems of differential equations and optimization problems. We can write (1.1.8) as $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, where $\mathbf{f}$ and $\mathbf{x}$ are vectors in $\mathbf{R}^n$. The vector-valued function $\mathbf{f}$ is said to be differentiable at the point $\mathbf{x}$ if each component is differentiable with respect to all the variables. The matrix of partial derivatives of $\mathbf{f}$ with respect to $\mathbf{x}$,

$$J(\mathbf{x}) = \mathbf{f}'(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \dots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \in \mathbf{R}^{n\times n}, \tag{1.1.9}$$

is called the **Jacobian** of $\mathbf{f}$.

Let $\mathbf{x}_k$ be the current approximate solution and assume that the matrix $\mathbf{f}'(\mathbf{x}_k)$ is nonsingular. Then in **Newton's method** for the system (1.1.8), the next iterate $\mathbf{x}_{n+1}$ is determined from the unique solution to the system of linear equations

$$J(\mathbf{x}_k)(\mathbf{x}_{n+1} - \mathbf{x}_k) = -\mathbf{f}(\mathbf{x}_k). \tag{1.1.10}$$

The linear system (1.1.10) can be solved by computing the LU factorization of the matrix $J(\mathbf{x})$; see Sec. 1.3.2.

Each step of Newton's method requires the evaluation of the n^2 entries of the Jacobian matrix $J(\mathbf{x}_k)$. This may be a time consuming task if n is large. If either the iterates or the Jacobian matrix are not changing too rapidly, it is possible to reevaluate $J(\mathbf{x}_k)$ only occasionally and use the same Jacobian in several steps. This has the further advantage that once we have computed the LU factorization of the Jacobian matrix, the linear system can be solved in only $O(n^2)$ arithmetic operations; see Sec. 1.3.2.

Example 1.1.3.

The following example illustrates the quadratic convergence of Newton's method for simple roots. The nonlinear system

$$\begin{aligned} x^2 + y^2 - 4x &= 0, \\ y^2 + 2x - 2 &= 0 \end{aligned}$$

has a solution close to $x_0 = 0.5$, $y_0 = 1$. The Jacobian matrix is

$$J(x, y) = \begin{pmatrix} 2x - 4 & 2y \\ 2 & 2y \end{pmatrix},$$

and Newton's method becomes

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - J(x_k, y_k)^{-1} \begin{pmatrix} x_k^2 + y_k^2 - 4x_n \\ y_k^2 + 2x_k - 2 \end{pmatrix}.$$

We get the following results:

k	x_k	y_k
1	**0.35**	**1.1**5
2	**0.35424**528301887	**1.136**52584085316
3	**0.35424868893**322	**1.13644297**217273
4	**0.35424868893541**	**1.13644296914943**

All digits are correct in the last iteration. The quadratic convergence is obvious; the number of correct digits approximately doubles in each iteration.

Often, the main difficulty in solving a nonlinear system is to find a sufficiently good starting point for the Newton iterations. Techniques for modifying Newton's method to ensure **global convergence** are therefore important in several dimensions. These must include techniques for coping with ill-conditioned or even singular Jacobian matrices at intermediate points. Such techniques will be discussed in Volume II.

1.1.3 Linearization and Extrapolation

The secant approximation is useful in many other contexts; for instance, it is 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**, i.e., in the approximate calculation of a definite integral,

$$I = \int_a^b y(x)\, dx, \tag{1.1.11}$$

(see Figure 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 Figure 1.1.5, we have

$$T(h) = h\frac{1}{2}\sum_{i=0}^{n-1}(y_i + y_{i+1}), \quad h = \frac{b-a}{n}. \tag{1.1.12}$$

(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 arbitrarily high accuracy by choosing h sufficiently small. But the computational work involved is roughly

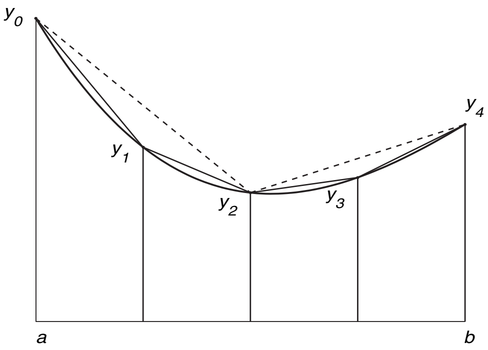


Figure 1.1.5. *Numerical integration by the trapezoidal rule* ($n = 4$).

proportional to the number of points where $y(x)$ must be computed, and thus inversely proportional to h. Hence the computational work grows rapidly as one demands higher accuracy (smaller h).

Numerical integration is a fairly common problem because only seldom can the "primitive" function be analytically calculated in a finite expression containing only elementary functions. It is not possible for such simple functions as e^{x^2} or $(\sin x)/x$. In order to obtain *higher accuracy* with significantly 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 $h = 0$, the so-called **Richardson extrapolation**[5] or **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, including 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 (1.1.12) 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

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

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

Thus, by adding the corrective term $\frac{1}{3}(T(h) - T(2h))$ to $T(h)$, one should get an estimate of I which is typically far more accurate than $T(h)$. In Sec. 3.4.6 we shall see that the improvement is in most cases quite striking. The result of the Richardson extrapolation is in this case equivalent to the classical **Simpson's rule** 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 problems other 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**; see Sec. 5.2.2.

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 1.1.4.

The integral

$$\int_{10}^{12} f(x)\,dx$$

is computed for $f(x) = x^3$ by the trapezoidal method. With $h = 1$ we obtain $T(h) = 2695$, $T(2h) = 2728$, and extrapolation gives $T = 2684$, equal to the exact result.

Similarly, for $f(x) = x^4$ we obtain $T(h) = 30{,}009$, $T(2h) = 30{,}736$, and with extrapolation $T \approx 29{,}766.7$ (exact $29{,}766.4$).

1.1.4 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. Figure 1.1.6 shows the graph of a function

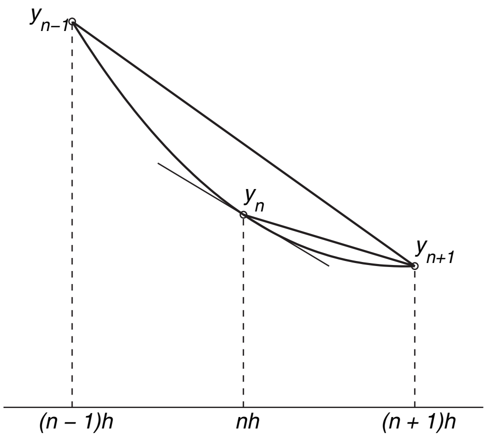


Figure 1.1.6. *Centered finite difference quotient.*

$y(x)$ 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,

$$y'(x_n) \approx \frac{y_{n+1} - y_n}{h}, \tag{1.1.13}$$

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

But it is conceivable that the **centered difference** approximation

$$y'(x_n) \approx \frac{y_{n+1} - y_{n-1}}{2h} \tag{1.1.14}$$

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

$$y(x+h) - y(x) = y'(x)h + y''(x)h^2/2 + y'''(x)h^3/6 + \dots, \tag{1.1.15}$$

$$-y(x-h) + y(x) = y'(x)h - y''(x)h^2/2 + y'''(x)h^3/6 - \dots. \tag{1.1.16}$$

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

$$y'(x_n) = \frac{y_{n+1} - y_{n-1}}{2h} - \frac{h^2}{6}y'''(x_n) - \dots. \tag{1.1.17}$$

In what follows we 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 (1.1.13) *is exact only for a linear function, and it is only first order accurate in the general case. The centered difference approximation* (1.1.14) *is exact also for a quadratic function, and is second order accurate in the general case.*

For the above reason the approximation (1.1.14) is, in most situations, preferable to (1.1.13). But 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 examine this phenomenon 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 can be approximated with **higher differences**, that is, differences 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 Δx instead of by h, we get the following formulas, which are easily remembered:

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x}, \qquad \frac{d^2 y}{dx^2} \approx \frac{\Delta^2 y}{(\Delta x)^2}, \tag{1.1.18}$$

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 (1.1.13) and (1.1.14).

Taking the difference of the Taylor expansions (1.1.15)–(1.1.16) 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**

$$\delta y_n = y\left(x_n + \frac{1}{2}h\right) - y\left(x_n - \frac{1}{2}h\right) \tag{1.1.19}$$

and neglecting higher order terms we get

$$y''(x_n) \approx \frac{1}{h^2}\delta^2 y_n - \frac{h^2}{12} y^{iv}(x_n). \tag{1.1.20}$$

The approximation of (1.1.14) can be interpreted as an application of (1.1.18) with $\Delta x = 2h$, or as the mean of the estimates which one gets according to (1.1.18) 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 smaller h is. Thus if one wishes to compute the derivatives of a function one should be careful not to use too small a step length; see Sec. 3.3.4.

Example 1.1.5.

Assume that for $y = \cos x$, function values correct to six decimal digits are known at equidistant points:

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

,

where the differences are expressed in units of 10^{-6}. This arrangement of the numbers is called a **difference scheme**. Using (1.1.14) and (1.1.18) one gets

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

The correct results are, with six decimals,

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

In y'' we got only two correct decimal digits. This is due to **cancellation**, which is an important cause of loss of accuracy; see Sec. 2.3.4. Better accuracy can be achieved by *increasing* the step h; see Problem 1.1.5 at the end of this section.

A very important equation of mathematical physics is **Poisson's equation**:[6]

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y), \quad (x, y) \in \Omega. \tag{1.1.21}$$

Here the function $f(x, y)$ is given together with some boundary condition on $u(x, y)$. Under certain conditions, gravitational, electric, magnetic, and velocity potentials satisfy **Laplace's equation**[7] which is (1.1.21) with $f(x, y) = 0$.

Finite difference approximations are useful for partial derivatives. Suppose that Ω is a rectangular region and introduce a **rectangular grid** that covers the rectangle. With grid spacing h and k, respectively, in the x and y directions, respectively, this consists of the points

$$x_i = x_0 + ih, \quad i = 0 : M, \qquad y_j = y_0 + jk, \quad j = 0 : N.$$

By (1.1.20), a second order accurate approximation of Poisson's equation is given by the **five-point operator**

$$\nabla_5^2 u = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}.$$

For $k = h$

$$\nabla_5^2 u = \frac{1}{h^2}\big(u_{i,j+1} + u_{i-1,j} - 4u_{i,j} + u_{i+1,j} + u_{i,j-1}\big),$$

[6] Siméon Denis Poisson (1781–1840), professor at École Polytechnique. He has also given his name to the Poisson distribution in probability theory.

[7] Pierre-Simon, Marquis de Laplace (1749–1827), professor at École Militaire. Laplace was one of the most influential scientists of his time and did major work in probability and celestial mechanics.

which corresponds to the “computational molecule”

$$\frac{1}{h^2}\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}.$$

If this is superimposed on each grid point we get one equation for the unknown values $u(x_i, y_j)$, $i = 1 : M - 1$, $j = 1 : N - 1$, at each interior point of the grid.

To get a solution we also need prescribed boundary conditions on u or $\partial u/\partial n$ on the boundary. The solution can then be obtained in the interior by solving a system of linear equations.

Review Questions

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

1.1.2 Discuss the convergence condition and the rate of convergence of the fixed-point iteration method for solving a nonlinear equation $x = F(x)$.

1.1.3 What is meant by quadratic convergence of an iterative method for solving a nonlinear equation?

1.1.4 What is the trapezoidal rule? What is said about the dependence of its error on the step length?

1.1.5 How can Richardson extrapolation be used to improve the accuracy of the trapezoidal rule?

Problems and Computer Exercises

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

1.1.2 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 by Newton’s method until you trust that the result is correct to six decimal places.

1.1.3 The equation $x^3 - x = 0$ has three roots, $-1, 0, 1$. We shall study the behavior of Newton’s method on this equation, with the notations used in Sec. 1.1.1 and Figure 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 one 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 $x_0 \in (a_i, a_{i+1})$;
$a_1 = 0.577$, $a_2 = 0.462$, $a_3 = 0.450$, $a_4 \approx \lim_{i\to\infty} a_i = 1/\sqrt{5} = 0.447$.

1.1.4 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 lengths $h = 1/2$ and $h = 1/4$.

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

1.1.5 In Example 1.1.5 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.

1.1.6 Give an approximate expression of the form $ah^b f^{(c)}(0)$ for the error of the estimate of the integral $\int_{-h}^{h} f(x)dx$ obtained by Richardson extrapolation (according to Sec. 1.1.3) from the trapezoidal values $T(h)$ and $T(2h)$.

1.2 Some Numerical Algorithms

For a given numerical problem one can consider many different algorithms. Even if they just differ in small details they can differ in efficiency and reliability and give approximate answers 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 Solving a Quadratic Equation

An early example of pitfalls in computation studied by G. E. Forsythe [118] is the following. For computing the roots of the quadratic equation $ax^2 + bx + c = 0$, $a \neq 0$, elementary textbooks usually give the well-known formula

$$r_{1,2} = \bigl(-b \pm \sqrt{b^2 - 4ac}\bigr)\big/(2a).$$

Using this for the quadratic equation $x^2 - 56x + 1 = 0$, we get the two approximate real roots

$$r_1 = 28 + \sqrt{783} \approx 28 + 27.982 = 55.982 \pm \frac{1}{2}10^{-3},$$

$$r_2 = 28 - \sqrt{783} \approx 28 - 27.982 = 0.018 \pm \frac{1}{2}10^{-3}.$$

In spite of the fact that the square root used is given to five digits of accuracy, we get only two significant digits in r_2, while the relative error in r_1 is less than 10^{-5}. This shows that *there can be very poor relative accuracy in the difference between two nearly equal numbers*. This phenomenon is called **cancellation of terms**. It is a very common reason for poor accuracy in numerical calculations.

Notice that the subtraction in the calculation of r_2 was carried out exactly. *The cancellation in the subtraction only gives an indication of the unhappy consequence of a loss of information in previous steps, due to the rounding of one of the operands, and is not the cause of the inaccuracy.*

In numerical calculations, if possible one should try to avoid formulas that give rise to cancellation, as in the above example. For the quadratic equation this can be done by *rewriting of the formulas*. Comparing coefficients on both sides of

$$x^2 + (b/a)x + c/a = (x - r_1)(x - r_2) = x^2 - (r_1 + r_2)x + r_1 r_2,$$

we get the relation between coefficients and roots

$$r_1 + r_2 = -b/a, \qquad r_1 r_2 = c/a. \tag{1.2.1}$$

A more accurate value of the root of *smaller magnitude* is obtained by computing this root from the latter of these relations. We then get

$$r_2 = 1/55.982 = 0.0178629 \pm 0.0000002.$$

Five significant digits are now obtained also for this root.

1.2.2 Recurrence Relations

A common computational task is the evaluation of a polynomial

$$p(x) = a_0 x^n + a_1 x^2 + \cdots + a_{n-1} x + a_n$$

at a given point. This can be reformulated as

$$p(x) = (\cdots((a_0 x + a_1)x + a_2)x + \cdots + a_{n-1})x + a_n,$$

and written as a **recurrence relation**:

$$b_i(x) = b_{i-1}(x)x + a_i, \quad i = 1 : n. \tag{1.2.2}$$

We note that this recurrence relation can be used in two different ways:

- it can be used *algebraically* to generate a sequence of Horner polynomials $b_i(x)$ such that $b_n(x) = p(x)$;
- it can be used *arithmetically* with a specific value $x = x_1$, which is **Horner's rule** for evaluating $p(x_1) = b_n(x_1)$.

Horner's rule requires n additions and multiplications for evaluating $p(x)$ for $x = x_1$. Note that if the powers are calculated recursively by $x_1^i = x_1 \cdot x_1^{i-1}$ and subsequently multiplied by a_{n-i}, this requires twice as many multiplications.

When a polynomial $p(x)$ is divided by $x - x_1$ the remainder equals $p(x_1)$; i.e., $p(x) = (x - x_1)q(x) + p(x_1)$. The quantities $b_i(x_1)$ from the Horner scheme (1.2.2) are of intrinsic interest because they are the coefficients of the quotient polynomial $q(x)$. This algorithm therefore performs the **synthetic division**

$$\frac{p(x) - p(x_1)}{x - x_1} = \sum_{i=0}^{n-1} b_i(x_1)x^{n-1-i}. \tag{1.2.3}$$

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. 6.5.4. As emphasized there, some care is necessary in the numerical application of this idea to prevent the propagation of roundoff errors.

The proof of the following useful relation is left as an exercise for the reader.

Lemma 1.2.1.

Let b_i be defined by (1.2.2) *and*

$$c_0 = b_0, \qquad c_i = b_i + xc_{i-1}, \quad i = 1 : n - 1. \tag{1.2.4}$$

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

Due to their intrinsic constructive quality, recurrence relations are one of the basic mathematical tools of computation. There is hardly a computational task which does not use recursive techniques. One of the most important and interesting parts of the preparation of a problem for a computer is therefore to find a recursive description of the task. Often an enormous amount of computation can be described by a small set of recurrence relations.

Although recurrence relations are a powerful tool they are also susceptible to error growth. Each cycle of a recurrence relation not only generates its own errors but also inherits errors committed in all previous cycles. If conditions are unfavorable, the result may be disastrous. This aspect of recurrence relations and its prevention is therefore of great importance in computations and has been studied extensively; see [136].

Example 1.2.1.

Unless used in the right way, errors committed in a recurrence relation can grow exponentially and completely ruin the results. To compute the integrals

$$I_n = \int_0^1 \frac{x^n}{x+5}\,dx, \quad i = 1 : N,$$

one can use the recurrence relation

$$I_n + 5I_{n-1} = \frac{1}{n}, \tag{1.2.5}$$

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

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

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 roundoff error ϵ 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 -5ϵ. That error produces an error in I_2 of $5^2\epsilon$, and so forth. 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 are the roundoff 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 an unpleasant phenomenon, called **numerical instability**. In this simple case, one can avoid the numerical instability by reversing the direction of the recursion.

Example 1.2.2.

If we use the recurrence relation in the other direction,

$$I_{n-1} = (1/n - I_n)/5, \tag{1.2.6}$$

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 encouraged 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,

$$I_8 = 0.018838, \quad I_7 = 0.021232, \quad I_6 = 0.024325, \quad I_5 = 0.028468,$$
$$I_4 = 0.034306, \quad I_3 = 0.043139, \quad I_2 = 0.058039, \quad I_1 = 0.088392,$$

and finally $I_0 = 0.182322$. Correct!

If one instead simply takes $I_{12} = 0$ as the starting value, 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 $I_5, \dots, I_0$ have the same values as above. The difference in the values for I_{11} is 0.002778. The subsequent values of $I_{10}, I_9, \dots, I_0$ 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 $n \le 8$.

One should not draw erroneous conclusions from the above example. The use of a recurrence relation "backwards" is not a universal recipe, as will be seen later. Compare also Problems 1.2.7 and 1.2.8.

In Sec. 3.3.5 we will study the general linear homogeneous difference equation of kth order

$$y_{n+k} + a_1 y_{n+k-1} + \cdots + a_k y_n = 0, \tag{1.2.7}$$

with real or complex constant coefficients $a_1, \dots, a_k$. The stability properties of this type of equation are fundamental, since they arise in the numerical solution of ordinary and partial differential equations.

1.2.3 Divide and Conquer Strategy

A powerful strategy for solving large scale problems is the **divide and conquer** strategy (one of the oldest military strategies). This is one of the most powerful algorithmic paradigms for designing efficient algorithms. The idea is to split a high-dimensional problem into multiple problems (typically two for sequential algorithms) of lower dimension. Each of these is then again split into smaller subproblems, and so forth, until a number of sufficiently small problems are obtained. The solution of the initial problem is then obtained by combining the solutions of the subproblems working backward 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 $n = 2^3 = 8$:

$$\begin{tikzcd}
a_1 \arrow[dr] & & a_2 \arrow[dl] & & a_3 \arrow[dr] & & a_4 \arrow[dl] & & a_5 \arrow[dr] & & a_6 \arrow[dl] & & a_7 \arrow[dr] & & a_8 \arrow[dl] \\
& s_{1:2} \arrow[drr] & & & & s_{3:4} \arrow[dll] & & & & s_{5:6} \arrow[drr] & & & & s_{7:8} \arrow[dll] & , \\
& & & s_{1:4} \arrow[drrrr] & & & & & & & & s_{5:8} \arrow[dllll] & & & \\
& & & & & & & s_{1:8} & & & & & & &
\end{tikzcd}$$

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 kth step we compute the final sum.

This summation algorithm uses the same number of additions as the first one. But it has the advantage that it splits the task into *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).

The algorithm can also be described in another way. Consider the summation algorithm

```
$sum = s(i, j)$;
**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**
```

for computing the sum $s(i, j) = a_i + \cdots + a_j$, $j > i$. (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. Espelid [112] gives an interesting discussion of such summation algorithms.

The function above is an example of a **recursive algorithm**—it calls itself. Many computer languages (for example, 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.

Example 1.2.3.

Sorting the items of a one-dimensional array in ascending or descending order is one of the most important problems in computer science. In numerical work, sorting is frequently needed when data need to be rearranged. One of the best known and most efficient sorting algorithms, **quicksort** by Hoare [198], is based on the divide and conquer paradigm. To sort an array of n items, $a[0 : n - 1]$, it proceeds as follows:

1. Select an element $a(k)$ to be the pivot. Commonly used methods are to select the pivot randomly or select the median of the first, middle, and last element in the array.
2. Rearrange the elements of the array a into a left and right subarray such that no element in the left subarray is larger than the pivot and no element in the right subarray is smaller than the pivot.
3. Recursively sort the left and right subarray.

The partitioning of a subarray $a[l : r]$, $l < r$, in step 2 can proceed as follows. Place the pivot in $a[l]$ and initialize two pointers $i = l$, $j = r + 1$. The pointer i is incremented until an element $a(i)$ is encountered which is larger than the pivot. Similarly, the pointer j is decremented until an element $a(j)$ is encountered which is smaller than the pivot. At this point the elements $a(i)$ and $a(j)$ are exchanged. The process continues until the pointers cross each other. Finally, the pivot element is placed in its correct position.

It is intuitively clear that this algorithm sorts the entire array and that no merging phase is needed.

There are many other examples of the power of the divide and conquer approach. It underlies the fast Fourier transform (Sec. 4.6.3) and is used in efficient automatic parallelization of many tasks, such as matrix multiplication; see [109].

1.2.4 Power Series Expansions

In many problems of applied mathematics, the solution of a given problem can be obtained as a power series expansion. Often the convergence of these series is quite fast. As an example we consider the task of computing, to five decimals, $y(0.5)$, where $y(x)$ is the solution to the differential equation

$$y'' = -xy,$$

with initial conditions $y(0) = 1$, $y'(0) = 0$. The solution cannot be simply expressed in terms of elementary functions. We shall use the **method of undetermined coefficients**. Thus we try substituting a series of the form:

$$y(x) = \sum_{n=0}^{\infty} c_n x^n = c_0 + c_1 x + c_2 x^2 + \cdots.$$

Differentiating twice we get

$$\begin{aligned} y''(x) &= \sum_{n=0}^{\infty} n(n-1)c_n x^{n-2} \\ &= 2c_2 + 6c_3 x + 12c_4 x^2 + \cdots + (m+2)(m+1)c_{m+2}x^m + \cdots, \\ -xy(x) &= -c_0 x - c_1 x^2 - c_2 x^3 - \cdots - c_{m-1}x^m - \cdots. \end{aligned}$$

Equating coefficients of x^m in these series gives

$$c_2 = 0, \qquad (m+2)(m+1)c_{m+2} = -c_{m-1}, \quad m \geq 1.$$

It follows from the initial conditions that $c_0 = 1$, $c_1 = 0$. Thus $c_n = 0$, if n is not a multiple of 3, and using the recursion we obtain

$$y(x) = 1 - \frac{x^3}{6} + \frac{x^6}{180} - \frac{x^9}{12\,960} + \cdots. \tag{1.2.8}$$

This gives $y(0.5) \approx 0.97925$. The x^9 term is ignored, since it is less than $2 \cdot 10^{-7}$. In this example also the first neglected term gives a rigorous bound for the error (i.e., for the remaining terms), since the absolute value of the term decreases, and the terms alternate in sign.

Since the calculation was based on a trial substitution, one should, strictly speaking, prove that the series obtained defines a function which satisfies the given problem. Clearly, the series converges at least for $|x| < 1$, since the coefficients are bounded. (In fact the series converges for all x.) Since a power series can be differentiated term by term in the interior of its interval of convergence, the proof presents no difficulty. Note, in addition, that the finite series obtained for $y(x)$ by breaking off after the x^9-term is the exact solution to the following *modified differential equation*:

$$y'' = -xy - \frac{x^{10}}{12\,960}, \qquad y(0) = 1, \qquad y'(0) = 0,$$

where the "perturbation term" $-x^{10}/12\,960$ has magnitude less than 10^{-7} for $|x| \leq 0.5$. It is possible to find rigorous bounds for the difference between the solutions of a differential system and a modified differential system.

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

A different approximation problem, which occurs in many variants, is to approximate a function f specified at a one- or two-dimensional grid by a member f^* of a class of functions which is easy to work with mathematically. Examples are (piecewise) 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. For this purpose **spline functions** are now used extensively with important applications in the aircraft and automotive industries; see Sec. 4.4. The name **spline** comes from a very old technique in drawing smooth curves, in which a thin strip of wood or rubber, called a draftsman's spline, is bent so that it passes through 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 shipbuilding and other engineering designs.

Review Questions

1.2.1 What is a common cause of loss of accuracy in numerical calculations?

1.2.2 Describe Horner's rule and synthetic division.

1.2.3 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.

1.2.4 Describe the basic 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?

Problems and Computer Exercises

1.2.1 (a) Use Horner's scheme to compute for $x = 2$

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

(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 $x^j = x \cdot x^{j-1}$ and subsequently multiplied by a_{n-j}.

1.2.2 $P(x) = 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4$ is a polynomial approximation to $\cos x$ for small values of $|x|$. Estimate the errors of

$$P(x), \quad P'(x), \quad \frac{1}{x}\int_0^x P(t)\,dt,$$

and compare them for $x = 0.1$.

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

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

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

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

1.2.4 (a) Show that the transformation made in Problem 1.2.3 can also be expressed by means of the matrix-vector equation

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

where $a = [a_1, a_2, \dots, a_n]^T$, $c = [c_1, c_2, \dots, c_n]^T$, and $\operatorname{diag}(1, z, \dots, z^{n-1})$ is a diagonal matrix with elements z^{j-1}, $j = 1 : n$. The matrix $P \in \mathbf{R}^{n\times n}$ has elements

$$p_{i,j} = \begin{cases} \binom{j-1}{i-1} & \text{if } j \geq i, \\ 0 & \text{otherwise.} \end{cases}$$

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 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.

1.2.5 Derive recurrence relations and write a program for computing the coefficients of the *product* r of two polynomials p and q:

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

1.2.6 Let a, b be nonnegative integers, with $b \neq 0$. The division a/b yields the quotient q and the remainder r. Show that if a and b have a common factor, then that number is a divisor of r as well. Use this remark to derive the **Euclidean algorithm** for the determination of the greatest common divisor of a and b.

1.2.7 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 the forward recurrence stable and the backward recurrence unstable in this case?

1.2.8 (a) Solve Example 1.2.1 on a computer, with the following changes: Start the recursion (1.2.5) with $I_0 = \ln 1.2$, and compute and print the sequence $\{I_n\}$ until I_n becomes negative for the first time.

(b) Start the recursion (1.2.6) 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 part (a).

1.2.9 (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.,

$$A(x) = Q(x)B(x) + R(x), \quad \deg R(x) < \deg B(x).$$

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

1.2.10 (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 choose only 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.

1.3 Matrix Computations

Matrix computations are ubiquitous in scientific computing. A survey of basic notation and concepts in matrix computations and linear vector spaces is given in Online Appendix A. This is needed for several topics treated in later chapters of this book. A fuller treatment of this topic will be given in Volume 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 savings to solve new problems involving the original matrix. Another is the rapid development of sophisticated iterative methods, which are becoming increasingly important as systems increase in size.

1.3.1 Matrix Multiplication

A **matrix**[8] 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 $C = AB \in \mathbf{R}^{m\times n}$, where

$$c_{ij} = \sum_{k=1}^{p} a_{ik}b_{kj}, \quad 1 \le i \le m, \quad 1 \le j \le n. \tag{1.3.1}$$

The product BA is defined only if $m = n$ and then $BA \in \mathbf{R}^{p\times p}$. Clearly, matrix multiplication is in general *not commutative*. In the exceptional case where $AB = BA$ holds, the matrices A and B are said to **commute**.

Matrix multiplication satisfies the associative and 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.

Let the three matrices $A \in \mathbf{R}^{m\times p}$, $B \in \mathbf{R}^{p\times n}$, and $C \in \mathbf{R}^{n\times q}$ be given. Then computing the product ABC as $(AB)C$ requires $mn(p+q)$ multiplications, whereas $A(BC)$ requires $pq(m + n)$ multiplications.

[8] The first to use the term "matrix" was the English mathematician James Sylvester in 1850. Arthur Cayley then published *Memoir on the Theory of Matrices* in 1858, which spread the concept.

If A and B are square $n \times n$ matrices and $C = x \in \mathbf{R}^{n\times 1}$, a column vector of length n, then computing $(AB)x$ requires $n^2(n+1)$ multiplications, whereas $A(Bx)$ only requires $2n^2$ multiplications. 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 *noncommuting 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 2×2 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} in the product $C = AB$ 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 [166])a flop is defined as one floating-point operation, doubling the older flop counts.[9] Hence, multiplication $C = AB$ of two square matrices of order n requires $2n^3$

[9] Stewart [328, p. 96] uses **flam** (floating-point addition and multiplication) to denote an "old" flop.

flops. The matrix-vector multiplication $y = Ax$, where $A \in \mathbf{R}^{n\times n}$ and $x \in \mathbf{R}^n$, requires $2mn$ flops.[10]

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 usually ignored is the fact that on many computers a division is five to ten times slower than a multiplication.

An operation count still provides useful information and can serve as an initial basis of comparison of different algorithms. It implies that the running time for multiplying two square matrices on a computer will increase roughly cubically with the dimension n. Thus, doubling n will approximately increase the work by a factor of eight; this is also apparent from (1.3.2).

A faster method for matrix multiplication would give more efficient algorithms for many linear algebra problems such as inverting matrices, solving linear systems, and solving eigenvalue problems. 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 [334] 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 $n^{\log_2 7} = n^{2.807\cdots}$. The key observation behind the algorithm is that the block matrix multiplication (1.3.2) can be performed with only *seven* block matrix multiplications and eighteen block matrix additions. Since for large dimensions matrix multiplication is much more expensive ($2n^3$ flops) than addition ($2n^2$ flops), this will lead to a savings in operations.

It is still an open (difficult!) question what the minimum exponent ω is such that matrix multiplication can be done in $O(n^\omega)$ operations. The fastest known algorithm, devised in 1987 by Coppersmith and Winograd [78], has $\omega < 2.376$. Many believe that an optimal algorithm can be found which reduces the number to essentially n^2. For a review of recent efforts in this direction using group theory, see Robinson [299]. (Note that for many of the theoretically "fast" methods large constants are hidden in the O notation.)

1.3.2 Solving Linear Systems by LU Factorization

The solution of **linear systems of equations** is the most frequently encountered task 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, $1 \le i \le m$, $1 \le j \le n$, are known input data and the task is to compute the unknowns x_j, $1 \le j \le n$. More compactly we write $Ax = b$, 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.

[10] To add to the confusion, in computer literature "flops" means floating-point operations per second.

Solving linear systems by **Gaussian elimination**[11] is taught in elementary courses in linear algebra. Although in theory this algorithm seems deceptively simple, the practical solution of large linear systems is far from trivial. In the 1940s, at the beginning of the computer age, there was a mood of pessimism among mathematicians about the possibility of accurately solving systems even of modest order, say $n = 100$. Today there is a deeper understanding of how Gaussian elimination performs in finite precision arithmetic. Linear systems with hundred of thousands of unknowns are now routinely solved in scientific computing.

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 ($m = n$),

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

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:

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

Since the unknowns are solved in reverse order this is called **back-substitution**. Thus the solution of a triangular system of order n can be computed in exactly n^2 flops; this is the same amount of work as required for *multiplying* a vector by a triangular matrix.

Similarly, a square linear system of **lower triangular** form $Lx = b$,

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

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

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

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

[11] Named after the German mathematician Carl Friedrich Gauss (1777–1855), but known already in China as early as the first century B.C. Gauss was one of the greatest mathematician of the nineteenth century. He spent most of his life in Göttingen, where in his dissertation he gave the first proof of the fundamental theorem of algebra. He made fundamental contributions to number theory, differential geometry, celestial mechanics, and geodesy. He introduced the method of least squares and put it on a solid foundation.

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 are 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 (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 1.1. *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 $i = n : (-1) : 1$
    $s := \sum_{k=i+1}^{n} u_{ik} b_k;$
    $b_i := (b_i - s)/u_{ii};$
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.

In the above algorithm the elements in U are accessed in a rowwise manner. In another possible sequencing of the operations the elements in U are accessed columnwise. This gives the following algorithm:

```
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
```

Such differences in the sequencing of the operations can influence the efficiency of the implementation depending on how the elements in the matrix U are stored.

Gaussian elimination uses the following elementary operations, which can be performed without changing the set of solutions:

- interchanging two equations,
- multiplying an equation by a nonzero scalar α,
- adding a multiple α of the ith equation to the jth equation, $j \neq i$.

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 the system $Ax = b$ can be transformed into an upper triangular system which, as shown above, can be solved by recursive substitution.

In 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 by subtracting from the ith equation the multiple $l_{i1} = a_{i1}/a_{11}$, of the first equation. The last $(n-1)$ equations then 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

$$a_{ij}^{(2)} = a_{ij} - \frac{a_{i1}a_{1j}}{a_{11}} = a_{ij} - l_{i1}a_{1j},$$

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

This is a system of $(n-1)$ equations in the $(n-1)$ unknowns $x_2, \dots, x_n$. All following steps are similar. In step k, $k = 1 : n-1$, if $a_{kk}^{(k)} \neq 0$, we eliminate x_k from the last $(n-k)$ equations giving a system containing only $x_{k+1}, \dots, x_n$. We take $l_{ik} = a_{ik}^{(k)}/a_{kk}^{(k)}$, and the elements of the new system are given by

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)}a_{kj}^{(k)}}{a_{kk}^{(k)}} = a_{ij}^{(k)} - l_{ik}a_{kj}^{(k)}, \tag{1.3.7}$$

$$b_i^{(k+1)} = b_i^{(k)} - l_{ik}b_k^{(k)}, \quad i, j = k+1 : n. \tag{1.3.8}$$

The diagonal elements $a_{11}, a_{22}^{(2)}, \dots, a_{n,n}^{(n)}$, which appear in the denominator in (1.3.7) 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

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots \\ & & & a_{nn}^{(n)} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(n)} \end{pmatrix}, \tag{1.3.9}$$

where we have introduced the notations $a_{ij}^{(1)} = a_{ij}$, $b_i^{(1)} = b_i$ for the coefficients in the original system. Thus (1.3.4) has been reduced to an equivalent nonsingular, upper triangular system (1.3.9), which can be solved by back-substitution.

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 several linear systems with the same matrix A but different right-hand sides,*

$$AX = B, \quad X = (x_1, \dots, x_p), \quad B = (b_1, \dots, b_p),$$

the operations on A only have to be carried out once.

We now show another interpretation of Gaussian elimination. For notational convenience we assume that $m = n$ and that Gaussian elimination can be carried out without pivoting. Then Gaussian elimination can be interpreted as computing the factorization $A = LU$ of the matrix A into the product of a unit lower triangular matrix L and an upper triangular matrix U.

Depending on whether the element a_{ij} lies on, above, or below the principal diagonal we have

$$a_{ij}^{(n)} = \begin{cases} \dots = a_{ij}^{(i+1)} = a_{ij}^{(i)}, & i \le j; \\ \dots = a_{ij}^{(j+1)} = 0, & i > j. \end{cases}$$

Thus the elements a_{ij}, $1 \le i, j \le n$, are transformed according to

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik}a_{kj}^{(k)}, \quad k = 1 : p, \quad p = \min(i-1, j). \tag{1.3.10}$$

If these equations are summed for $k = 1 : p$, we obtain

$$\sum_{k=1}^{p}\left(a_{ij}^{(k+1)} - a_{ij}^{(k)}\right) = a_{ij}^{(p+1)} - a_{ij} = -\sum_{k=1}^{p} l_{ik}a_{kj}^{(k)}.$$

This can also be written

$$a_{ij} = \begin{cases} a_{ij}^{(i)} + \displaystyle\sum_{k=1}^{i-1} l_{ik}a_{kj}^{(k)}, & i \le j, \\ 0 + \displaystyle\sum_{k=1}^{j} l_{ik}a_{kj}^{(k)}, & i > j, \end{cases}$$

or, if we define $l_{ii} = 1$, $i = 1 : n$,

$$a_{ij} = \sum_{k=1}^{r} l_{ik}u_{kj}, \quad u_{kj} = a_{kj}^{(k)}, \quad r = \min(i, j). \tag{1.3.11}$$

However, these equations are equivalent to the matrix equation

$$A = LU, \quad L = (l_{ik}), \quad U = (u_{kj}).$$

Here L and U are lower and upper triangular matrices, respectively. Hence Gaussian elimination computes a factorization of A into a product of a lower and an upper triangular matrix, the **LU factorization** of A. Note that since the unit diagonal elements in L need not be stored, it is possible to store the L and U factors in an array of the same dimensions as A.

Algorithm 1.2. *LU Factorization.*

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 (1.3.9). 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
    end
end
```

Although the LU factorization is just a different interpretation of Gaussian elimination it turns out to have important conceptual advantages. It divides the solution of a linear system into two independent steps:

1. the factorization $A = LU$,
2. solution of the systems $Ly = b$ and $Ux = y$.

The LU factorization is a prime example of *the decompositional approach to matrix computation*. This approach came into favor in the 1950s and early 1960s and has been named as one of the ten algorithms having the most influence on science and engineering in the twentieth century. This interpretation of Gaussian elimination has turned out to be very 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 (see Forsythe and Moler [121]) that "Almost anything you can do with A^{-1} can be done without it."

Another example is the problem of solving the transposed system $A^Tx = b$. Since

$$A^T = (LU)^T = U^TL^T,$$

we have that $A^T x = U^T(L^T x) = b$. It follows that x can be computed by solving the two triangular systems

$$U^T c = b, \qquad L^T x = c. \tag{1.3.12}$$

In passing we remark that Gaussian elimination is also an efficient algorithm for computing the **determinant** of a matrix A. It can be shown that the value of the determinant is unchanged if a row (column) multiplied by a scalar is added to another row (column) (see (A.2.4) in the online Appendix). Further, if two rows (columns) are interchanged, the value of the determinant is multiplied by (-1). Since the determinant of a triangular matrix equals the product of the diagonal elements it follows that

$$\det(A) = \det(L)\det(U) = \det(U) = (-1)^q a_{11}^{(1)} a_{22}^{(2)} \cdots a_{nn}^{(n)}, \tag{1.3.13}$$

where q is the number of row interchanges performed.

From Algorithm 1.2 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 number of flops required for the reduction of $Ax = b$ to a triangular system by Gaussian elimination is

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

for the transformation of A and the right-hand side b, respectively. 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.

Linear systems where the matrix A has only a few nonzero diagonals often arise. Such matrices are called **band matrices**. In particular, band matrices of the form

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ a_1 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & a_{n-1} & b_n \end{pmatrix} \tag{1.3.14}$$

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 and U are obtained from the recursion: Set $\beta_1 = b_1$, and

$$\gamma_k = a_k/\beta_k, \qquad \beta_{k+1} = b_{k+1} - \gamma_k c_k, \quad k = 1 : n-1. \tag{1.3.15}$$

(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 $y = Ux$,

$$y_1 = g_1, \qquad y_{k+1} = g_{k+1} - \gamma_k y_k, \quad k = 1 : n-1, \tag{1.3.16}$$

followed by a backward recursion for x,

$$x_n = y_n/\beta_n, \qquad x_k = (y_k - c_k x_{k+1})/\beta_k, \quad k = n-1 : -1 : 1. \tag{1.3.17}$$

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

Stability of Gaussian Elimination

If A is nonsingular, then Gaussian elimination can always be carried through provided row interchanges are allowed. In this more general case, Gaussian elimination computes an LU factorization of the matrix $\tilde{A}$ obtained by carrying out all row interchanges on A. In practice row interchanges are needed to ensure the numerical stability of Gaussian elimination. We now consider how the LU factorization has to be modified when such interchanges are incorporated.

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 if the row interchanges are first performed on A and the Gaussian elimination performed without interchanges. Row interchanges can be expressed as premultiplication with certain matrices, which we now introduce.

A **permutation matrix** $P \in \mathbf{R}^{n\times n}$ is a matrix whose columns are a permutation of the columns of the unit matrix, that is,

$$P = (e_{p_1}, \dots, e_{p_n}),$$

where $(p_1, \dots, p_n)$ is a permutation of $(1, \dots, n)$. Notice that in a permutation matrix every row and every column contains just one unity element. The transpose P^T of a permutation matrix is therefore again a permutation matrix. A permutation matrix is orthogonal $P^T P = I$, and hence P^T affects the reverse permutation.

A **transposition matrix** is a special permutation matrix,

$$I_{ij} = (\ldots, e_{i-1}, e_j, e_{i+1}, \ldots, e_{j-1}, e_i, e_{j+1}),$$

which equals the identity matrix except with columns i and j interchanged. By construction it immediately follows that I_{ij} is symmetric. $I_{ij}^2 = I$ and hence $I_{ij}^{-1} = I_{ij}$. If a matrix A is premultiplied by I_{ij}, this results in the interchange of *rows* i and j. Any permutation matrix can be expressed as a product of transposition matrices.

If P is a permutation matrix, then PA is the matrix A with its rows permuted. Hence, Gaussian elimination with row interchanges produces a factorization, which in matrix notations can be written

$$PA = LU,$$

where P is a permutation matrix. Note that P is uniquely represented by the integer vector $(p_1, \ldots, p_n)$ and need never be stored as a matrix.

Assume that in the kth step, $k = 1 : n-1$, we select the pivot element from row p_k, and interchange the rows k and p_k. Notice that in these row interchanges also, previously computed multipliers l_{ij} must take part. At completion of the elimination, we have obtained lower and upper triangular matrices L and U. We now make the important observation that these are the same triangular factors that are obtained if we *first* carry out the row interchanges $k \leftrightarrow p_k$, $k = 1 : n-1$, on the *original matrix* A to get a matrix PA, where P is a permutation matrix, and then perform Gaussian elimination on PA *without any interchanges*. This means that Gaussian elimination with row interchanges computes the LU factors of the matrix PA. We now summarize the results and prove the uniqueness of the LU factorization.

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

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 $x_1 = -x_2 = -1/(1-\epsilon)$. But when $\epsilon = 0$ 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 $x = (-1.0001, 1.0001)^T$. But 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 are of the same size as in the original matrix.

In general an algorithm is said to be **backward stable** (see Definition 2.4.19) if the computed solution $\overline{w}$ equals *the exact solution* of a problem with "slightly perturbed data." The more or less final form of error analysis of Gaussian elimination was given by J. H. Wilkinson [368].[12]

Wilkinson showed that the computed triangular factors L and U of A obtained by Gaussian elimination *are the exact triangular factors of a slightly perturbed matrix* $A + E$. He further gave bounds on the elements of E. The essential condition for stability is that *no substantial growth occurs in the elements in* L *and* U; see Theorem 2.4.12. Although matrices can be constructed for which an element growth factor in Gaussian elimination with partial pivoting equals 2^{n-1}, quoting Kahan [214] we say that: "Intolerable pivot-growth (with partial pivoting) is a phenomenon that happens only to numerical analysts who are looking for that phenomenon." Why large element growth rarely occurs in practice with partial pivoting is a subtle and still not fully understood phenomenon. Trefethen and Schreiber [353] show that for certain distributions of random matrices the average element growth is close to $n^{2/3}$ for partial pivoting.

It is important to note that the fact that a problem has been solved by a backward stable algorithm *does not mean that the error in the computed solution is small.* If the matrix A is close to a singular matrix, then the solution is very sensitive to perturbations in the data. This is the case when the rows (columns) of A are almost linearly dependent. But 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. 2.4.2.

An important special case of LU factorization is when the matrix A is symmetric, $A^T = A$, and **positive definite**, i.e.,

$$x^T A x > 0 \quad \forall x \in \mathbf{R}^n, \quad x \neq 0. \tag{1.3.18}$$

Similarly A is said to be **positive semidefinite** if $x^T A x \geq 0$ for all $x \in \mathbf{R}^n$. Otherwise it is called **indefinite**.

[12] James Hardy Wilkinson (1919–1986), English mathematician who graduated from Trinity College, Cambridge. He became Alan Turing's assistant at the National Physical Laboratory in London in 1946, where he worked on the ACE computer project. He did pioneering work on numerical methods for solving linear systems and eigenvalue problems, and developed software and libraries of numerical routines.

For symmetric positive definite matrices there always exists a unique factorization

$$A = R^T R, \tag{1.3.19}$$

where R is an upper triangular matrix with positive diagonal elements. An important fact is that *no pivoting is needed for stability. Indeed, unless the pivots are chosen from the diagonal, pivoting is harmful since it will destroy symmetry.*

This is called the **Cholesky factorization**.[13] The elements in the Cholesky factor $R = (r_{ij})$ can be determined directly. The matrix equation $A = R^T R$ with R upper triangular can be written elementwise as

$$a_{ij} = \sum_{k=1}^{i} r_{ki} r_{kj} = \sum_{k=1}^{i-1} r_{ki} r_{kj} + r_{ii} r_{ij}, \quad 1 \le i \le j \le n. \tag{1.3.20}$$

These are $n(n+1)/2$ equations for the unknown elements in R. Solving for r_{ij} from the corresponding equation in (1.3.20), we obtain

$$r_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj} \right) \Big/ r_{ii}, \quad i < j, \qquad r_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2 \right)^{1/2}. \tag{1.3.21}$$

If properly sequenced, these equations can be used in a recursive fashion to compute the elements in R, for example, one row at a time. The resulting algorithm requires n square roots and approximately $n^3/3$ flops, which is about half the work of an LU factorization.

We remark that the Cholesky factorization can be carried out also for a symmetric indefinite matrix, if at each step a positive pivot is chosen from the diagonal. However, for a symmetric *indefinite* matrix, such as, the matrix in Example 1.3.4 with $\epsilon < 1$, no Cholesky factorization can exist.

1.3.3 Sparse Matrices and Iterative Methods

Following Wilkinson and Reinsch [374], a matrix A will be called **sparse** if the percentage of zero elements is large and its distribution is such that it is economical to take advantage of their presence. The nonzero elements of a sparse matrix may be concentrated on a narrow band centered on the diagonal. Alternatively, they may be distributed in a less systematic manner.

Large sparse linear systems arise in numerous areas of application, such as the numerical solution of partial differential equations, mathematical programming, structural analysis, chemical engineering, electrical circuits, and networks. Large could imply a value of n in the range 1000–1,000,000. Figure 1.3.1 shows a sparse matrix of order $n = 479$ with 1887 nonzero elements (or 0.9%) that arises from a model of an eight stage chemical distillation column.

The first task in solving a sparse system by Gaussian elimination is to permute the rows and columns so that not too many new nonzero elements are created during the elimination.

[13] André-Louis Cholesky (1875–1918), a French military officer involved in geodesy and surveying in Crete and North Africa just before World War One.

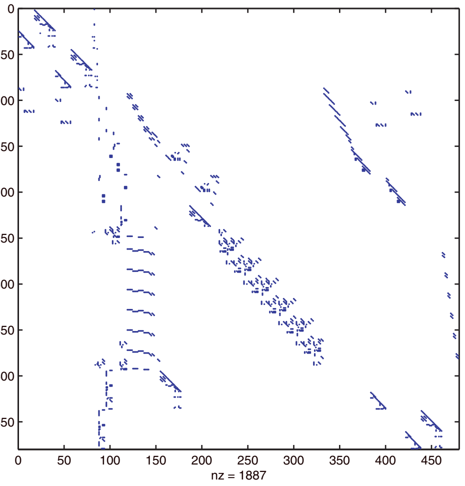


Figure 1.3.1. *Nonzero pattern of a sparse matrix from an eight stage chemical distillation column.*

Equivalently, we want to choose permutation matrices P and Q such that the LU factors of PAQ are as sparse as possible. Such a reordering will usually nearly minimize the number of arithmetic operations.

To find an *optimal* ordering which minimizes the number of nonzero elements in L and U is unfortunately an intractable problem, because the number of possible orderings of rows and columns are $(n!)^2$. Fortunately, there are heuristic ordering algorithms which do a good job. In Figure 1.3.2 we show the reordered matrix PAQ and its LU factors. Here L and U contain together 5904 nonzero elements or about 2.6%. The column ordering was obtained using a MATLAB version of the so-called column minimum degree ordering.

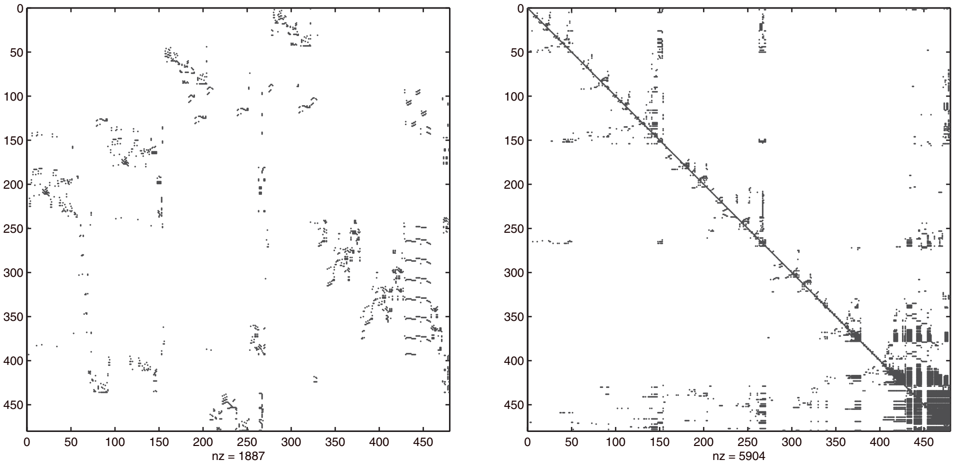


Figure 1.3.2. *Nonzero structure of the matrix A (left) and L + U (right).*

For the origin and details of this code we refer to Gilbert, Moler, and Schreiber [153]. We remark that, in general, some kind of stability check on the pivot elements must be performed during the factorization.

For many classes of sparse linear systems **iterative methods** are more efficient to use than **direct methods** such as Gaussian elimination. Typical examples are those arising when a differential equation in two or three dimensions 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 have the added advantage of requiring only extra storage for a few vectors.

In a classical iterative method due to Richardson [295], starting from $x^{(0)} = 0$, a sequence $x^{(k)}$ is defined by

$$x^{(k+1)} = x^{(k)} + \omega(b - Ax^{(k)}), \quad k = 0, 1, 2, \dots, \tag{1.3.22}$$

where $\omega > 0$ is a parameter to be chosen. It follows easily from (1.3.22) that the error in $x^{(k)}$ satisfies $x^{(k+1)} - x = (I - \omega A)(x^{(k)} - x)$, and hence

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

It can be shown that, if all the eigenvalues λ_i of A are real and satisfy

$$0 < a \leq \lambda_i \leq b,$$

then $x^{(k)}$ will converge to the solution, when $k \to \infty$, for $0 < \omega < 2/b$.

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 orders of magnitude 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. In a typical problem for the Poisson equation (1.1.21) the function is to be determined in a plane domain D, when the values of u are given on the boundary ∂D. Such **boundary value problems** occur in the study of steady states in most branches of physics, such as electricity, elasticity, heat flow, and fluid mechanics (including meteorology). Let D be a square grid with grid size h, i.e., $x_i = x_0 + ih$, $y_j = y_0 + jh$, $0 \leq i \leq N + 1$, $0 \leq j \leq N + 1$. Then the difference approximation yields

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

$(1 \leq i, j \leq N)$. This is a huge system of linear algebraic equations; one equation for each interior gridpoint, altogether N^2 unknowns 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,

$$u = (u_{1,1}, \dots, u_{1,N}, u_{2,1}, \dots, u_{2,N}, \dots, u_{N,1}, \dots, u_{N,N}),$$

the so-called natural ordering. 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 1.3.3 (left).

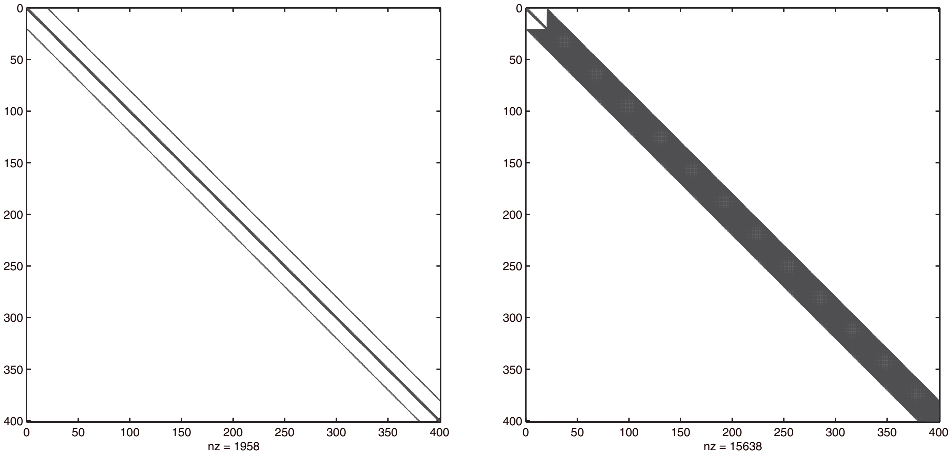


Figure 1.3.3. *Structure of the matrix A (left) and $L + U$ (right) for the Poisson problem, $N = 20$ (rowwise ordering of the unknowns).*

In principle Gaussian elimination can be used to solve such systems. But even taking symmetry and the banded structure into account, this would require $\frac{1}{2} \cdot N^4$ multiplications, since in the LU factors the zero elements inside the outer diagonals will fill in during the elimination, as shown in Figure 1.3.3 (right).

The linear system arising from the Poisson equation has several features common to boundary value problems for other linear partial differential equations. One of these is that only a tiny fraction of the elements in each row of A are nonzero. Therefore, each iteration in Richardson's method requires only about kN^2 multiplications, i.e., k 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.

As early as 1954, a simple atmospheric model was used for weather forecasting on an electronic computer. The net covered most of North America and Europe. During a 48 hour forecast, the computer solved (among other things) 48 Poisson equations (with different right-hand sides). This would have been impossible at that time if the special features of the system had not been used.

1.3.4 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, as well as a column oriented back-substitution rather than that in Algorithm 1.1.

The first collection of high quality linear algebra software was a series of algorithms written in Algol 60 that appeared in the handbook [374]. This contains 11 subroutines for linear systems, least squares, and linear programming, and 18 routines for the algebraic eigenvalue problem.

The Basic Linear Algebra Subprograms (BLAS) have become an important tool for structuring linear algebra computations. 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 [234], introduced in 1979, identified frequently occurring vector operations in matrix computation such as scalar product, adding of a multiple of one vector to another, etc. For example, the operation

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

in single precision is named SAXPY. By carefully optimizing them for each specific computer, performance was enhanced without sacrificing portability. These BLAS were adopted in the collections of Fortran subroutines LINPACK (see [99]) for linear systems and EISPACK (see [130]) for eigenvalue problems.

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

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

Level 3 BLAS use $O(n^2)$ data but perform $O(n^3)$ arithmetic operations. This gives a surface-to-volume effect for the ratio of data movement to operations.

LAPACK (see [6]) is a linear algebra package initially released in 1992. LAPACK was designed to supersede and integrate the algorithms in both LINPACK and EISPACK. It achieves close to optimal performance on a large variety of computer architectures by expressing as much of the algorithm as possible as calls to level 3 BLAS. This is also an aid to clarity, portability, and modularity. LAPACK today is the backbone of the interactive matrix computing system MATLAB.

Example 1.3.5.

In 1974 the authors wrote in [87, Sec. 8.5.3] that "a full 1000 × 1000 system of equations is near the limit at what can be solved at a reasonable cost." Today systems of this size can easily be handled on a personal computer. The benchmark problem for the Japanese Earth Simulator, one of the world's fastest computers in 2004, was the solution of a system of size 1,041,216 on which a speed of 35.6×10^{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

1.3.1 How many operations are needed (approximately) for

(a) the multiplication of two square matrices $A, B \in \mathbf{R}^{n\times n}$?

(b) the LU factorization of a matrix $A \in \mathbf{R}^{n\times n}$?

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

1.3.2 Show that if the kth diagonal entry of an upper triangular matrix is zero, then its first k columns are linearly dependent.

1.3.3 What is the LU factorization 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.

1.3.4 (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?

1.3.5 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

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

Hint: Write k in the binary number system and compute $A^2, A^4, A^8, \dots,$ by successive squaring; e.g., $13 = (1101)_2$ and $A^{13} = A^8A^4A$.

1.3.2 (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 $R^n = 0$.

1.3.3 Show that there cannot exist an LU 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.

1.3.4 (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 $e_n = (0, 0, \dots, 0, 1)^T$ is the nth unit vector.

(b) Show that the inverse of an upper triangular matrix is also upper triangular. Determine for $n = 3$ the inverse of $U_n(a)$. Try also to determine $U_n(a)^{-1}$ for an arbitrary n. *Hint:* Note that $UU^{-1} = U^{-1}U = I$, the identity matrix.

1.3.5 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

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

(a) Determine the approximate number of operations needed to compute the LU factorization of H_n without pivoting.

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

1.3.6 Compute the product $|L|\,|U|$ for the LU factors with and without pivoting of the matrix in Example 1.3.4. (Here $|A|$ denotes the matrix with elements $|a_{ij}|$.)

1.3.7 Let $A \in \mathbf{R}^{n\times n}$ be a given matrix. Show that if $Ax = y$ has *at least one* solution for any $y \in \mathbf{R}^n$, then it has *exactly one* solution for any $y \in \mathbf{R}^n$. (This is a useful formulation for showing uniqueness of approximation formulas.)

1.4 The Linear Least Squares Problem

A basic problem in science is to fit a mathematical model to given observations subject to errors. As an example, consider observations (t_i, y_i), $i = 1 : m$, to be fitted to a model described by a scalar function $y(t) = f(c, t)$, where $c \in \mathbf{R}^n$ is a parameter vector to be determined. There are two types of shortcomings to take into account: errors in the input data, and approximations made in the particular model (class of functions, form). We shall call these **measurement errors** and **errors in the model**, respectively.

Clearly the more observations that are available the more accurately will it be possible to determine the parameters in the model. One can also see this problem 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 of approximation theory.

A simple example is when the model is *linear* in c and of the form

$$y(t) = \sum_{j=1}^{n} c_j \phi_j(t),$$

where $\phi_j(t)$ are given (possibly nonlinear) functions. Given $m > n$ measurements the resulting equations

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

form an **overdetermined** linear system. If we set $A \in \mathbf{R}^{m\times n}$, $a_{ij} = \phi_j(t_i)$, then the system can be written in matrix form as $Ac = y$. In general such a system is inconsistent and has no solution. But we can try to find a vector $c \in \mathbf{R}^n$ such that Ac is the "best" approximation to y. This is equivalent to minimizing the size of the **residual vector** $r = y - Ac$.

There are many possible ways of defining the best solution to such an inconsistent linear system. A choice which can often be motivated for statistical reasons, and which also leads to a simple computational problem, is to take as the solution a vector c, which minimizes the sum of the squared residuals, that is,

$$\min_x \sum_{i=1}^{m} r_i^2 = \min_x \|y - Ac\|_2^2, \tag{1.4.1}$$

where we have used the notation

$$\|x\|_2 = (|x_1|^2 + \cdots + |x_n|^2)^{1/2} = (x^T x)^{1/2}$$

for the Euclidean length of a vector x (see Online Appendix A). We call (1.4.1) a **linear least squares problem** and any minimizer x a **least squares solution** of the system $Ax = b$.

Gauss claims to have discovered the method of least squares in 1795 when he was 18 years old. He used it in 1801 to successively predict the orbit of the asteroid Ceres.

1.4.1 Basic Concepts in Probability and Statistics

We now introduce, without proofs, some basic concepts, formulas, and results from statistics which will be used later. Proofs may be found in most texts on these subjects.

The **distribution function** of a random variable X is denoted by the nonnegative and nondecreasing function

$$F(x) = Pr\{X \le x\}, \quad F(-\infty) = 0, \quad F(\infty) = 1.$$

If $F(x)$ is differentiable, the (probability) **density function**[14] is $f(x) = F'(x)$. Note that

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

and

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

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 $p_i \ge 0$ and $\sum_i p_i = 1$.

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

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

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

The **variance** of X equals

$$\sigma^2 = \text{var}(X) = E((X-\mu)^2), \quad \mu = E(X),$$

and $\sigma = \sqrt{\text{var}(X)}$ is the **standard deviation**. The mean and standard deviation are frequently used as measures of the center and spread of a distribution.

Let X_k, $k = 1 : n$, be random variables with mean values μ_k. Then the **covariance matrix** is $V = \{\sigma_{jk}\}_{j,k=1}^{n}$, where

$$\begin{aligned} \sigma_{jk} = \text{cov}(X_j, X_k) &= E((X_j - \mu_j)(X_k - \mu_k)) \\ &= E(X_j)E(X_k) - \mu_j \mu_k. \end{aligned}$$

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

If the random variables X_k, $k = 1 : n$, are mutually uncorrelated, then V is a diagonal matrix.

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.

1.4.2 Characterization of Least Squares Solutions

Let $y \in \mathbf{R}^m$ be a vector of observations that is related to a parameter vector $c \in \mathbf{R}^n$ by the linear relation

$$y = Ac + \epsilon, \quad A \in \mathbf{R}^{m\times n}, \tag{1.4.2}$$

where A is a known matrix of full column rank and $\epsilon \in \mathbf{R}^m$ is a vector of random errors. We assume here that ϵ_i, $i = 1 : m$, has zero mean and that ϵ_i and ϵ_j, $i \neq j$, are uncorrelated, i.e.,

$$E(\epsilon) = 0, \qquad V(\epsilon) = \sigma^2 I.$$

The parameter c is then a random vector which we want to estimate in terms of the known quantities A and y. Let $y^T c$ be a linear functional of the parameter c in (1.4.2). We say that $\theta = \theta(A, y)$ is an unbiased linear estimator of $y^T c$ if $E(\theta) = y^T c$. It is a **best linear unbiased estimator** if θ has the smallest variance among all such estimators.

The following theorem[15] places the method of least squares on a sound theoretical basis.

[15] This theorem is originally due to C. F. Gauss (1821). His contribution was somewhat neglected until rediscovered by the Russian mathematician A. A. Markov in 1912.

Theorem 1.4.1 (*Gauss–Markov Theorem*).

Consider a linear model (1.4.2)*, where ϵ is an uncorrelated random vector with zero mean and covariance matrix $V = \sigma^2 I$. Then the best linear unbiased estimator of any linear functional $y^T c$ is $y^T \hat{c}$, where*

$$\hat{c} = (A^T A)^{-1} A^T y \tag{1.4.3}$$

is the least squares estimator obtained by minimizing the sum of squares $\|f - Ac\|_2^2$. Furthermore, the covariance matrix of the least squares estimate $\hat{c}$ equals

$$V(\hat{c}) = \sigma^2 (A^T A)^{-1}, \tag{1.4.4}$$

and

$$s^2 = \|y - A\hat{c}\|_2^2/(m-n) \tag{1.4.5}$$

is an unbiased estimate of σ^2, i.e. $E(s^2) = \sigma^2$.

Proof. See Zelen [382, pp. 560–561]. □

The set of all least squares solutions can also be characterized geometrically. For this purpose we introduce two fundamental subspaces of $\mathbf{R}^m$, the **range** of A and the **null space** of A^T, defined by

$$\mathcal{R}(A) = \{z \in \mathbf{R}^m \mid z = Ax,\ x \in \mathbf{R}^n\}, \tag{1.4.6}$$

$$\mathcal{N}(A^T) = \{y \in \mathbf{R}^m \mid A^T y = 0\}. \tag{1.4.7}$$

If $z \in \mathcal{R}(A)$ and $y \in \mathcal{N}(A^T)$, then $z^T y = x^T A^T y = 0$, which shows that $\mathcal{N}(A^T)$ is the orthogonal complement of $\mathcal{R}(A)$.

By the Gauss–Markov theorem any least squares solution to an overdetermined linear system $Ax = b$ satisfies the **normal equations**

$$A^T A x = A^T b. \tag{1.4.8}$$

The normal equations are always consistent, since the right-hand side satisfies

$$A^T b \in \mathcal{R}(A^T) = \mathcal{R}(A^T A).$$

Therefore, a least squares solution always exists, although it may not be unique.

Theorem 1.4.2.

The vector x minimizes $\|b - Ax\|_2$ if and only if the residual vector $r = b - Ax$ is orthogonal to $\mathcal{R}(A)$ or, equivalently,

$$A^T (b - Ax) = 0. \tag{1.4.9}$$

Proof. Let x be a vector for which $A^T(b - Ax) = 0$. For any $y \in \mathbf{R}^n$, it holds that $b - Ay = (b - Ax) + A(x - y)$. Squaring this and using (1.4.9) we obtain

$$\|b - Ay\|_2^2 = \|b - Ax\|_2^2 + \|A(x - y)\|_2^2 \geq \|b - Ax\|_2^2,$$

where equality holds only if $A(x - y) = 0$.

Now assume that $A^T(b - Ax) = z \neq 0$. If $x - y = -\epsilon z$, we have for sufficiently small $\epsilon \neq 0$

$$\begin{aligned}\|b - Ay\|_2^2 &= \|b - Ax\|_2^2 + \epsilon^2\|Az\|_2^2 - 2\epsilon(Az)^T(b - Ax)\\ &= \|b - Ax\|_2^2 + \epsilon^2\|Az\|_2^2 - 2\epsilon\|z\|_2^2 < \|b - Ax\|_2^2,\end{aligned}$$

and thus x does not minimize $\|b - Ax\|_2$. □

From Theorem 1.4.2 it follows that a least squares solution x decomposes the right-hand side into two orthogonal components

$$b = Ax + r, \quad r = b - Ax \in \mathcal{N}(A^T), \quad Ax \in \mathcal{R}(A). \tag{1.4.10}$$

This geometric interpretation is illustrated in Figure 1.4.1. Note that although the solution x to the least squares problem may not be unique, the decomposition (1.4.10) always is unique.

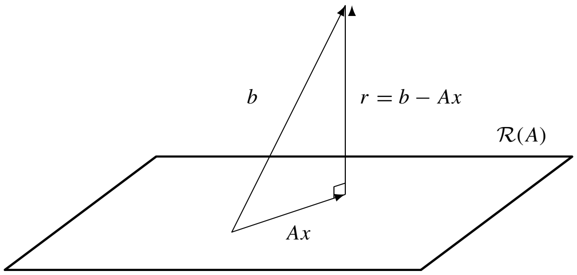


Figure 1.4.1. *Geometric characterization of the least squares solution.*

We now give a necessary and sufficient condition for the least squares solution to be unique.

Theorem 1.4.3.

The matrix A^TA is positive definite and hence nonsingular if and only if the columns of A are linearly independent, that is, when rank $(A) = n$. *In this case the least squares solution x is unique and given by*

$$x = (A^TA)^{-1}A^Tb. \tag{1.4.11}$$

Proof. If the columns of A are linearly independent, then $x \neq 0 \Rightarrow Ax \neq 0$. Therefore, $x \neq 0 \Rightarrow x^TA^TAx = \|Ax\|_2^2 > 0$, and hence A^TA is positive definite.

On the other hand, if the columns are linearly dependent, then for some $x_0 \neq 0$ we have $Ax_0 = 0$. Then $x_0^TA^TAx_0 = 0$, and therefore A^TA is not positive definite. When A^TA is positive definite it is also nonsingular and (1.4.11) follows. □

When A has full column rank A^TA is symmetric and positive definite and the normal equations can be solved by computing the Cholesky factorization $A^TA = R^TR$. The normal equations then become $R^TRx = A^Tb$, which decomposes as

$$R^T z = A^T b, \qquad Rx = z.$$

The first system is lower triangular and z is computed by forward-substitution. Then x is computed from the second upper triangular system by back-substitution. For many practical problems this **method of normal equations** is an adequate solution method, although its numerical stability is not the best.

Example 1.4.1.

The comet Tentax, discovered in 1968, 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
ϕ	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 $e = px_2 = 0.7027$.

By (1.4.10), if x is a least squares solution, then Ax is the orthogonal projection of b onto $\mathcal{R}(A)$. Thus orthogonal projections play a central role in least squares problems. In general, a matrix $P_1 \in \mathbf{R}^{m\times m}$ is called a **projector** onto a subspace $S \subset \mathbf{R}^m$ if and only if it holds that

$$P_1 v = v \ \ \forall v \in S, \qquad P_1^2 = P_1. \tag{1.4.12}$$

An arbitrary vector $v \in \mathbf{R}^m$ can then be decomposed as $v = P_1 v + P_2 v \equiv v_1 + v_2$, where $P_2 = I - P_1$. In particular, if P_1 is symmetric, $P_1 = P_1^T$, we have

$$P_1^T P_2 v = P_1^T (I - P_1) v = (P_1 - P_1^2) v = 0 \quad \forall v \in \mathbf{R}^m,$$

and it follows that $P_1^T P_2 = 0$. Hence $v_1^T v_2 = v^T P_1^T P_2 v = 0$ for all $v \in \mathbf{R}^m$, i.e., $v_2 \perp v_1$. In this case P_1 is the **orthogonal projector** onto S, and $P_2 = I - P_1$ is the orthogonal projector onto $S^{\perp}$.

In the full column rank case, rank $(A) = n$, of the least squares problem, the residual $r = b - Ax$ can be written $r = b - P_{\mathcal{R}(A)} b$, where

$$P_{\mathcal{R}(A)} = A(A^T A)^{-1} A^T \tag{1.4.13}$$

is the orthogonal projector onto $\mathcal{R}(A)$. If rank $(A) < n$, then A has a nontrivial null space. In this case if $\hat{x}$ is any vector that minimizes $\|Ax - b\|_2$, then the set of all least squares solutions is

$$\mathcal{S} = \{x = \hat{x} + y \mid y \in \mathcal{N}(A)\}. \tag{1.4.14}$$

In this set there is a unique solution of minimum norm characterized by $x \perp \mathcal{N}(A)$, which is called the pseudoinverse solution.

1.4.3 The Singular Value Decomposition

In the past the conventional way to determine the rank of a matrix A was to compute the row echelon form by Gaussian elimination. This would also show whether a given linear system is consistent or not. However, in floating-point calculations it is difficult to decide if a pivot element, or an element in the transformed right-hand side, should be considered as zero or nonzero. Such questions can be answered in a more satisfactory way by using the **singular value decomposition** (SVD), which is of great theoretical and computational importance.[16]

The geometrical significance of the SVD is as follows: The rectangular matrix $A \in \mathbf{R}^{m \times n}$, $m \geq n$, represents a mapping $y = Ax$ from $\mathbf{R}^n$ to $\mathbf{R}^m$. The image of the unit sphere $\|x\|_2 = 1$ is a hyper ellipse in $\mathbf{R}^m$ with axes equal to $\sigma_1 \geq \sigma_2 \cdots \geq \sigma_n \geq 0$. In other words, the SVD gives orthogonal bases in these two spaces such that the mapping is represented by the generalized diagonal matrix $\Sigma \in \mathbf{R}^{m \times n}$. This is made more precise in the following theorem, a constructive proof of which will be given in Volume II.

Theorem 1.4.4 (*Singular Value Decomposition*).

Any matrix $A \in \mathbf{R}^{m \times n}$ of rank r can be decomposed as

$$A = U \Sigma V^T, \qquad \Sigma = \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} \in \mathbf{R}^{m \times n}, \tag{1.4.15}$$

where $\Sigma_r = \operatorname{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$ is diagonal and

$$U = (u_1, \ldots, u_m) \in \mathbf{R}^{m \times m}, \qquad V = (v_1, \ldots, v_n) \in \mathbf{R}^{n \times n} \tag{1.4.16}$$

[16] The SVD was published by Eugenio Beltrami in 1873 and independently by Camille Jordan in 1874. Its use in numerical computations is much more recent, since a stable algorithm for computing the SVD did not become available until the publication of Golub and Reinsch [164] in 1970.

are square orthogonal matrices, $U^TU = I_m$, $V^TV = I_n$. *Here*

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$$

are the $r \leq \min(m, n)$ *nonzero* **singular values** *of* A. *The vectors* u_i, $i = 1 : m$, *and* v_j, $j = 1 : n$, *are left and right* **singular vectors**. *(Note that if* $r = n$ *and/or* $r = m$, *some of the zero submatrices in* Σ *disappear.)*

The singular values of A are uniquely determined. For any distinct singular value $\sigma_j \neq \sigma_i$, $i \neq j$, the corresponding singular vector v_j is unique (up to a factor ± 1). For a singular value of multiplicity p the corresponding singular vectors can be chosen as any orthonormal basis for the unique subspace of dimension p that they span. Once the singular vectors v_j, $1 \leq j \leq r$, have been chosen, the vectors u_j, $1 \leq j \leq r$, are uniquely determined, and vice versa, by

$$u_j = \frac{1}{\sigma_j} A v_j, \qquad v_j = \frac{1}{\sigma_j} A^T u_j, \quad j = 1 : r. \tag{1.4.17}$$

By transposing (1.4.15) we obtain $A^T = V\Sigma^T U^T$, which is the SVD of A^T. Expanding (1.4.15), the SVD of the matrix A can be written as a sum of r matrices of rank one,

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T. \tag{1.4.18}$$

The SVD gives orthogonal bases for the range and null space of A and A^T. Suppose that the matrix A has rank $r < \min(m, n)$. It is easy to verify that

$$\mathcal{R}(A) = \text{span}\,(u_1, \ldots, u_r), \qquad \mathcal{N}(A^T) = \text{span}\,(u_{r+1}, \ldots, u_m), \tag{1.4.19}$$

$$\mathcal{R}(A^T) = \text{span}\,(v_1, \ldots, v_r), \qquad \mathcal{N}(A) = \text{span}\,(v_{r+1}, \ldots, v_n). \tag{1.4.20}$$

It immediately follows that

$$\mathcal{R}(A)^\perp = \mathcal{N}(A^T), \qquad \mathcal{N}(A)^\perp = \mathcal{R}(A^T); \tag{1.4.21}$$

i.e., $\mathcal{N}(A^T)$ is the orthogonal complement to $\mathcal{R}(A)$, and $\mathcal{N}(A)$ is the orthogonal complement to $\mathcal{R}(A^T)$. This result is sometimes called the fundamental theorem of linear lgebra.

We remark that the SVD generalizes readily to complex matrices. The SVD of a matrix $A \in \mathbf{C}^{m\times n}$ is

$$A = (U_1\ U_2)\Sigma \begin{pmatrix} V_1^T \\ V_2^T \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} \in \mathbf{R}^{m\times n}, \tag{1.4.22}$$

where the singular values $\sigma_1, \sigma_2, \ldots, \sigma_r$ are real and nonnegative, and U and V are square unitary matrices, $U^HU = I_m$, $V^HV = I_n$. (Here A^H denotes the conjugate transpose of A.)

The SVD can also be used to solve linear least squares problems in the case when the columns in A are linearly dependent. If there is a vector $c \neq 0$ such that $Ac = 0$ and the least squares solution is not unique, then there exists a *unique least squares solution of minimum Euclidean length*, which solves the least squares problem

$$\min_{x\in S} \|x\|_2, \qquad S = \{x \in \mathbf{R}^n |\ \|b - Ax\|_2 = \min\}. \tag{1.4.23}$$

In terms of the SVD (1.4.22) of A the solution to (1.4.23) can be written $x = A^\dagger b$, where the matrix $A^\dagger$ is

$$A^\dagger = (V_1 \; V_2)\Sigma^\dagger \begin{pmatrix} U_1^T \\ U_2^T \end{pmatrix}, \quad \Sigma^\dagger = \begin{pmatrix} \Sigma_r^{-1} & 0 \\ 0 & 0 \end{pmatrix} \in \mathbf{R}^{n\times m}. \tag{1.4.24}$$

The matrix $A^\dagger$ is unique and called the **pseudoinverse** of A, and $x = A^\dagger b$ is the **pseudoinverse solution**. Note that problem (1.4.23) includes as special cases the solution of both overdetermined and underdetermined linear systems.

The pseudoinverse $A^\dagger$ is often called the **Moore–Penrose inverse**. Moore developed the concept of the general reciprocal in 1920. In 1955 Penrose [283] gave an elegant algebraic characterization and showed that $X = A^\dagger$ is uniquely determined by the four **Penrose conditions**:

$$(1) \quad AXA = A, \qquad (2) \quad XAX = X, \tag{1.4.25}$$

$$(3) \quad (AX)^T = AX, \qquad (4) \quad (XA)^T = XA. \tag{1.4.26}$$

It can be directly verified that $X = A^\dagger$ given by (1.4.24) satisfies these four conditions. In particular this shows that $A^\dagger$ does not depend on the particular choices of U and V in the SVD.

The orthogonal projections onto the four fundamental subspaces of A have the following simple expressions in terms of the SVD:

$$\begin{aligned} P_{\mathcal{R}(A)} &= AA^\dagger = U_1U_1^T, & P_{\mathcal{N}(A^T)} &= I - AA^\dagger = U_2U_2^T, \\ P_{\mathcal{R}(A^T)} &= A^\dagger A = V_1V_1^T, & P_{\mathcal{N}(A)} &= I - A^\dagger A = V_2V_2^T. \end{aligned} \tag{1.4.27}$$

These first expressions are easily verified using the definition of an orthogonal projection and the Penrose conditions.

1.4.4 The Numerical Rank of a Matrix

Let A be a matrix of rank $r < \min(m, n)$, and E a matrix of small random elements. Then it is most likely that the perturbed matrix $A + E$ has maximal rank $\min(m, n)$. However, since $A + E$ is close to a rank deficient matrix, it should be considered as having **numerical rank** equal to r. In general, the numerical rank assigned to a matrix should depend on some tolerance δ, which reflects the error level in the data and/or the precision of the arithmetic used.

It can be shown that perturbations of an element of a matrix A result in perturbations of the same, or smaller, magnitude in its singular values. This motivates the following definition of numerical rank.

Definition 1.4.5.

A matrix $A \in \mathbf{R}^{m\times n}$ is said to have numerical δ-rank equal to k if

$$\sigma_1 \geq \cdots \geq \sigma_k > \delta \geq \sigma_{k+1} \geq \cdots \geq \sigma_p, \quad p = \min(m, n), \tag{1.4.28}$$

where σ_i are the singular values of A. Then the right singular vectors $(v_{k+1}, \dots, v_n)$ form an orthogonal basis for the numerical null space *of A.*

Definition 1.4.5 assumes that there is a well-defined gap between σ_{k+1} and σ_k. When this is not the case the numerical rank of A is not well defined.

Example 1.4.2.

Consider an integral equation of the first kind,

$$\int_0^1 k(s,t)f(s)\,ds = g(t), \quad k(s,t) = e^{-(s-t)^2},$$

on $-1 \le t \le 1$. In order to solve this equation numerically it must first be discretized. We introduce a uniform mesh for s and t on $[-1, 1]$ with step size $h = 2/n$, $s_i = -1 + ih$, $t_j = -1 + jh$, $i, j = 0 : n$. Approximating the integral with the trapezoidal rule gives

$$h \sum_{i=0}^{n} w_i k(s_i, t_j) f(t_i) = g(t_j), \quad j = 0 : n,$$

where $w_i = 1$, $i \neq 0, n$, and $w_0 = w_n = 1/2$. These equations form a linear system

$$Kf = g, \quad K \in \mathbf{R}^{(n+1)\times(n+1)}, \quad f, g \in \mathbf{R}^{n+1}.$$

For $n = 100$ the singular values σ_k of the matrix K were computed in IEEE double precision with a unit roundoff level of $1.11 \cdot 10^{-16}$ (see Sec. 2.2.3). They are displayed in logarithmic scale in Figure 1.4.2. Note that for $k > 30$ all σ_k are close to roundoff level, so the numerical rank of K certainly is smaller than 30. This means that the linear system $Kf = g$ is *numerically underdetermined* and has a meaningful solution only for special right-hand sides g.

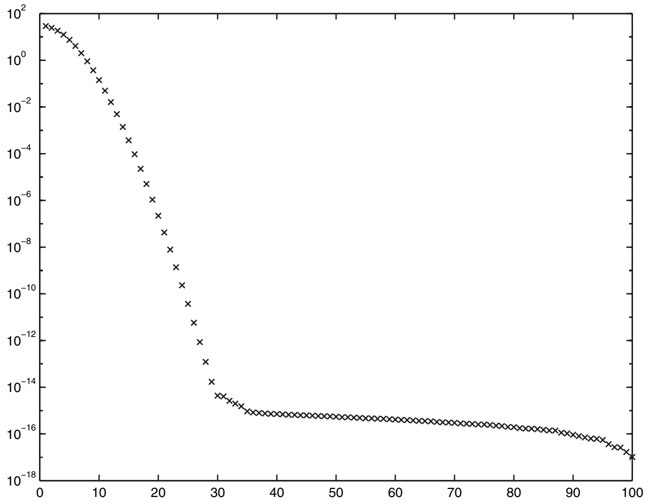


Figure 1.4.2. *Singular values of a numerically singular matrix.*

Equation (1.4.28) is a **Fredholm integral equation** of the first kind. It is known that such equations are **ill-posed** in the sense that the solution f does not depend continuously on the right-hand side g. This example illustrate how this inherent difficulty in the continuous problem carries over to the discretized problem.

Review Questions

1.4.1 State the Gauss–Markov theorem.

1.4.2 Show that the matrix $A^TA \in \mathbf{R}^{n\times n}$ of the normal equations is symmetric and positive semidefinite, i.e., $x^T(A^TA)x \geq 0$, for all $x \neq 0$.

1.4.3 Give two geometric conditions which are necessary and sufficient conditions for x to be the pseudoinverse solution of $Ax = b$.

1.4.4 (a) Which are the four fundamental subspaces of a matrix? Which relations hold between them?

(b) Show, using the SVD, that $P_{\mathcal{R}(A)} = AA^\dagger$ and $P_{\mathcal{R}(A^T)} = A^\dagger A$.

1.4.5 (a) Construct an example where $(AB)^\dagger \neq B^\dagger A^\dagger$.

(b) Show that if A is an $m\times r$ matrix, B is an $r \times n$ matrix, and rank $(A) =$ rank $(B) = r$, then $(AB)^\dagger = B^\dagger A^\dagger$.

Problems and Computer Exercises

1.4.1 In order to estimate the height above sea level for three points A, B, and C, the difference in altitude was measured between these points and points D, E, and F at sea level. The measurements obtained form a linear system in the heights x_A, x_B, and x_C of A, B, and C:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_A \\ x_B \\ x_C \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 1 \\ 2 \\ 1 \end{pmatrix}.$$

Determine the least squares solution and verify that the residual vector is orthogonal to all columns in A.

1.4.2 Consider the least squares problem $\min_x \|Ax - b\|_2^2$, where A has full column rank. Partition the problem as

$$\min_{x_1,x_2} \left\| (A_1\ A_2) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - b \right\|_2^2.$$

By a geometric argument show that the solution can be obtained as follows. First compute x_2 as solution to the problem

$$\min_{x_2} \| P_{A_1}^{\perp}(A_2 x_2 - b)\|_2^2,$$

where $P_{A_1}^{\perp} = I - P_{A_1}$ is the orthogonal projector onto $\mathcal{N}(A_1^T)$. Then compute x_2 as solution to the problem

$$\min_{x_1} \|A_1 x_1 - (b - A_2 x_2)\|_2^2.$$

1.5 Numerical Solution of Differential Equations

1.5.1 Euler's Method

The 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 lead to systems of ordinary (or partial) differential equations. For the **step by step simulation** of such a system a **mathematical model** is first set up; i.e., **state variables** are set up which describe the essential features of the state of the system. 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 quality 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.

The construction of a mathematical model is not trivial. Knowledge of numerical methods and programming helps 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 of results obtained by applying 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 using a particular example.

An **initial value problem** for an ordinary differential equation is to find $y(t)$ 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 eighteenth century by Euler[17]) is that one studies the solution for only certain

[17] Leonhard Euler (1707–1783), incredibly prolific Swiss mathematician. He gave fundamental contributions to many branches of mathematics and to the mechanics of rigid and deformable bodies, as well as to fluid mechanics.

values of $t = t_n = nh$, $n = 0, 1, 2, \ldots,$ (h is called the “time step” or “step length”) and assumes that dy/dt is constant between the points. In this way the solution is approximated by a polygon (Figure 1.5.1) which joins the points (t_n, y_n), $n = 0, 1, 2, \ldots,$ where

$$y_0 = c, \qquad \frac{y_{n+1} - y_n}{h} = f(t_n, y_n). \tag{1.5.1}$$

Thus we have the simple difference equation known as **Euler’s method**:

$$y_0 = c, \qquad y_{n+1} = y_n + hf(t_n, y_n), \quad n = 0, 1, 2, \ldots. \tag{1.5.2}$$

During the computation, each y_n occurs first on the left-hand side, then *recurs* later on the right-hand side of an equation. (One could also call (1.5.2) 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.)

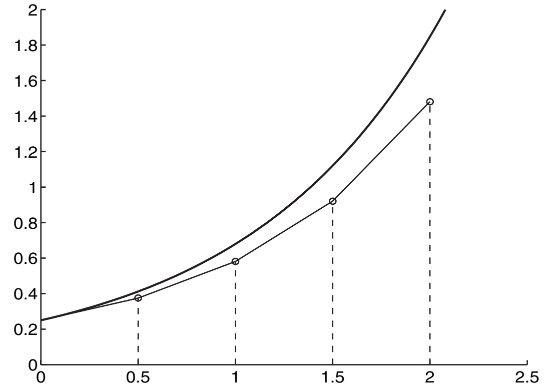


Figure 1.5.1. *Approximate solution of the differential equation* $dy/dt = y$*,* $y_0 = 0.25$*, by Euler’s method with* $h = 0.5$.

1.5.2 An Introductory Example

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 as well as the curvature and the rotation of the Earth. This means that we forsake serious applications to, for example, satellites. 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 as motion in a plane, but we have to forsake the application to, for example, table tennis, baseball, 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 upward. Assume that the air resistance is a force P such that the direction is opposite to the velocity, and the strength z 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

$$P_x = -mzu, \qquad P_y = -mzv, \qquad z = \frac{cR^2}{m}\sqrt{u^2 + v^2}, \tag{1.5.3}$$

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 with 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

$$m\,du/dt = P_x, \qquad m\,dv/dt = -mg + P_y, \tag{1.5.4}$$

where the term $-mg$ is the force of gravity. Inserting (1.5.3) into (1.5.4) and dividing by m we get

$$du/dt = -zu, \qquad dv/dt = -g - zv, \tag{1.5.5}$$

and by the definition of velocity,

$$dx/dt = u, \qquad dy/dt = v. \tag{1.5.6}$$

Equations (1.5.5) and (1.5.6) 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 us 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 of the motion of the ball means that, at a sequence of time instances $t_n, n = 0, 1, 2, \dots,$ 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. Therefore, set $t_n = nh$. 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**:

$$\begin{aligned} x_{n+1} &= x_n + hu_n, \qquad y_{n+1} = y_n + hv_n,\\ u_{n+1} &= u_n - hz_n u_n,\\ v_{n+1} &= v_n - h(g + z_n v_n), \end{aligned} \tag{1.5.7}$$

where

$$z_n = \frac{cR^2}{m}\sqrt{u_n^2 + v_n^2}.$$

From this x_{n+1}, y_{n+1}, u_{n+1}, v_{n+1}, etc. are solved, step by step, for $n = 0, 1, 2, \dots,$ using the provided initial values x_0, y_0, u_0, and v_0.

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

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

Curves obtained for $h = 0.01$ and $cR^2/m = 0.25i \cdot 10^{-3}$, $i = 0 : 4$, are shown in Figure 1.5.2. There is, in this graphical representation, also an error due to the limited resolution of the plotting device.

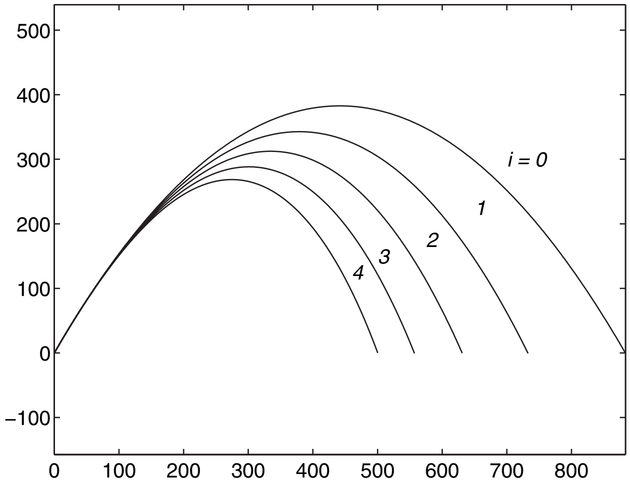


Figure 1.5.2. *Approximate trajectories computed with Euler's method with* $h = 0.01$.

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.,

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

By setting $y = 0$ we obtain

$$x^* = x_N - y_N\frac{x_{N+1} - x_N}{y_{N+1} - y_N}. \tag{1.5.8}$$

This is called (linear) **inverse interpolation**; see Sec. 4.2.2. The error from the linear approximation in (1.5.8) 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

$$x^* = 2u_0v_0/9.81 = 5000 \cdot \sqrt{3}/9.81 \approx 882.7986.$$

The computer produced $x^* \approx 883.2985$ for $h = 0.01$, and $x^* \approx 883.7984$ 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 small if reasonable accuracy is desired. In order to improve the method we can apply another idea mentioned previously, namely Richardson extrapolation. (The application differs a little from the one we saw previously, 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 $x^* \approx 500.2646$ and $x^* \approx 500.3845$ for, respectively, $h = 0.01$ and $h = 0.02$. Now let x^* denote the *exact* horizontal 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 coordinate. By a more accurate integration method we obtained 500.1440. Thus, in this case we gained more than two decimal digits by the use of Richardson extrapolation.

The simulations shown in Figure 1.5.2 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 with most serious applications. So we would like to have a method that allows *much larger time steps* than Euler's method.

1.5.3 Second Order Accurate Methods

In step by step computations we have to distinguish between the **local error**, the error that is committed at a single step, and the **global error**, 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 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. In the above ball problem this means $\sqrt{1500} \approx 40$ steps. Since the amount of work is closely proportional to the number of steps this is an enormous savings!

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 sizes* 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[18]) obtained by 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

$$d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(a) = \mathbf{c}, \tag{1.5.9}$$

where $\mathbf{y}$ is a column vector that contains all the state variables.[19] 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.

For the ball example, we have $a = 0$, and by (1.5.5) and (1.5.6),

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} \equiv \begin{pmatrix} x \\ y \\ u \\ v \end{pmatrix}, \qquad \mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} y_3 \\ y_4 \\ -zy_3 \\ -g - zy_4 \end{pmatrix}, \qquad \mathbf{c} = 10^2 \begin{pmatrix} 0 \\ 0 \\ \cos\phi \\ \sin\phi \end{pmatrix},$$

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:

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

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 second order accurate method. It is left as an exercise (Problem 1.5.2) to verify that *this scheme is identical to the following somewhat simpler scheme* known as **Runge's second order method**:

$$\begin{aligned} \mathbf{k}_1 &= h_n\mathbf{f}(t_n, \mathbf{y}_n), \\ \mathbf{k}_2 &= h_n\mathbf{f}(t_n + h_n/2, \mathbf{y}_n + \mathbf{k}_1/2), \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \mathbf{k}_2, \end{aligned} \tag{1.5.10}$$

[18] Carle David Tolmé Runge (1856–1927), German mathematician. Runge was chair of applied mathematics at Göttingen from 1904 until his death.

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

where we have replaced h by h_n in order to include the use of variable step size. Another explanation of the second 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 *midpoint* of the time step. Sometimes this method is called the improved Euler method or Heun's method, but these names are also used to denote other second order accurate methods.

1.5.4 Adaptive Choice of Step Size

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 an error he tolerates in values of variables (relative to the values themselves).[20] Compute

$$\delta = \max_i \delta_i, \quad \delta_i = |k_{2,i} - k_{1,i}|/|3y_i|,$$

where δ_i is related to the *relative* error of the ith component of the vector $\mathbf{y}$ at the current step; see below.

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

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

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 second order accurate; i.e., the global error is almost proportional to the square of the step size and δ is approximately proportional to h^2.

A step is *rejected* if $\delta >$ tol, and *recomputed* with the step size

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

The program needs a suggestion for the size of the first step. This can 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 $y'(a) \neq 0$, 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 δ by*

$$\bar{y}_i = \max\{|y_i|, 0.001\}.$$

(You may sometimes have to replace the default value 0.001 by something else.)

[20] 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.

It is a good habit to make a second run with a predetermined sequence of step sizes (if your program allows this) instead of adaptive control. Suppose that the sequence of time instances used in the first run is $t_0, t_1, t_2, \ldots$. Divide each subinterval $[t_n, t_{n+1}]$ into two steps of equal length. Thus, 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 four 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 such as $\text{tol}^{-1/2}$. It is, however, a nontrivial matter to find an error curve that strictly and realistically tells us how good the extrapolated results are. 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 this is inferior is that the two runs do not "keep in step," and then 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 second order method is, however, no universal recipe. There are special classes of problems, notably the problems which are called "stiff," which need special methods.

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 in the postprocessing of numerical results. (After you have used a more than second order accurate integration method, it may be necessary to use more sophisticated interpolation at the graphical or numerical treatment of the results.) We suggest that you write or try to find a program that can be used for systems with (in principle) any number of equations; see the preface.

Example 1.5.1.

The differential equation

$$dy/dt = -\frac{1}{2}y^3,$$

with initial condition $y(1) = 1$, was treated by a program, essentially constructed as described above, with $\text{tol} = 10^{-4}$ until $t = 10^4$. When comparing the result with the exact solution $y(t) = t^{-1/2}$, it was found that the actual relative error remained at 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 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 1.5.2.

The example of the motion of a ball was treated by Runge's second order method with the constant step size $h = 0.9$. The x-coordinate of the landing point became $x^* \approx 500.194$, 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 upstage 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. Recent developments in hardware as well as software has created a far better environment for such work.

Review Questions

1.5.1 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 of a method?

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

1.5.3 Discuss some strategies for the adaptive control of step length and estimation of global accuracy in the numerical solution of differential equations.

Problems and Computer Exercises

1.5.1 (a) Integrate numerically using Euler's method the differential equation $dy/dt = y$, with initial conditions $y(0) = 1$, to $t = 0.4$:, 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)?

1.5.2 (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., $n = N$ and $n = N + 1$) as well as the x-coordinate of the landing point. Take $h = 0.05$ and $h = 0.1$. As postprocessing, improve the estimates of x^* by Richardson extrapolation, and estimate the error by comparison with the results given in the text above.

(b) In (1.5.7), in the equations for x_{n+1} and y_{n+1}, replace 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?

1.5.3 Verify that Runge's second order method, as described by (1.5.10), is equivalent to the scheme described a few lines earlier (with Euler steps and Richardson extrapolation).

1.5.4 Write a program for Runge's second order method with automatic step size control that can be applied to a system of differential equations. Store the results so that they can be processed afterward, for example, to make a table of the results; draw curves showing $y(t)$ versus t, or (for a system) y_2 versus y_1; or draw some other interesting curves.

Apply the program to Examples 1.5.1 and 1.5.2, and to the circle test, that is,

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

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 not become "thick."

1.6 Monte Carlo Methods

1.6.1 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 a **numerical game of chance**, the mathematical analysis of which leads to the same equations as the given problem for, e.g., 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 proportionally to* $\sqrt{N}$.

The idea behind the Monte Carlo method was used by the Italian physicist Enrico Fermi to study 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 the method was pioneered by von Neumann,[21] Ulam,[22] 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 numerical simulations 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, statistical experiments were used to check the adequacy of certain theoretical probability laws that had been derived mathematically by the eminent scientist W. S. Gosset. (He used the pseudonym "Student" when he wrote on probability.)

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 Figure 1.6.1).

(b) Technical problems concerning traffic (in telecommunication systems and railway networks; in the regulation of traffic lights, and in 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 to 100) integrals over complicated regions. Such integrals occur in such diverse areas as

[21] John von Neumann was born János Neumann in Budapest 1903, and died in Washington D.C. 1957. He studied under Hilbert in Göttingen in 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.

[22] Stanislaw Marcin Ulam, born in Lemberg, Poland (now Lwow, Ukraine) 1909, and 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 Harvard 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.

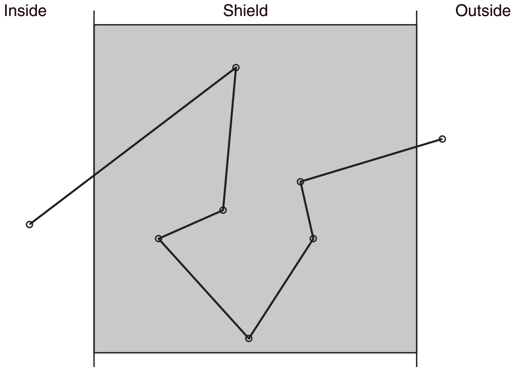


Figure 1.6.1. *Neutron scattering.*

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; see Sec. 5.4.5.

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.[23]

1.6.2 Generating and Testing Pseudorandom Numbers

In the beginning, coins, dice, and roulettes were used for creating the randomness. For example, the sequence of 20 digits

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

is a record of 20 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. We also assume that the tosses of the coin are made in a statistically independent way.[24]

Similarly, decimal random digits could in principle be obtained by using a well-made icosahedral (20 sided) die and assigning each decimal digit to two of its sides. Such

[23] One notable early example is the SIMULA programming language designed and built by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo 1962–67. It was originally built as a language for discrete event simulation, but was also influential because it introduced object-oriented programming concepts.

[24] Of course, these assumptions cannot be obtained in practice, as shown in theoretical and experimental studies by Persi Diaconis, Stanford University.

mechanical (or analogous electronic) 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 $0, 1, 2, \dots, 9$, 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" ([289]). The wheel had 32 slots, of which 12 were ignored; the others were numbered from zero to nine twice. To test the quality of the randomness several tests were applied. Every block of a 1000 digits in the tables (and also the table as a whole) were tested. The *Handbook of Mathematical Functions* [1, Table 26.11][25] provides 2500 five-digit random numbers compiled from this set.

Example 1.6.1.

The random number generator, used for drawing of prizes of Swedish Premium Saving Bonds, was developed in 1962 by Dahlquist [84]. Speed is not a major concern for this application, 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 other means. The length of this 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 to generate a much longer secondary series of prime numbers in a way that is best described by a mechanical analogy. Think of k cogwheels 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 cogwheel. Then each wheel is turned one cog clockwise 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 drawing from a tombola.

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

$$55693 \;\; 02945 \;\; 81723 \;\; 43588 \;\; 81350 \;\; 76302 \;\; \dots$$

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

$$0.556935, 0.029455, 0.817235, 0.435885, 0.813505, 0.763025, \dots, \qquad (1.6.1)$$

which can be considered a good approximation to a sequence of independent observations of a variable which is a sequence of uniform deviates in the interval $[0, 1)$. The 5 in the

[25] This classical *Handbook of Mathematical Functions*, edited by Milton Abramowitz and Irene A. Stegun, is used as a reference throughout this book. We will often refer to it as just "the Handbook."

sixth decimal place is added in order to get the correct mean (without this the mean would be 0.499995 instead of 0.5).

In a computer it is usually not appropriate to store a large table of random numbers. Several physical devices for random number generation have been proposed, using for instance electronic or radioactive noise, but very few seem to have been inserted in an actual computer system. Instead random numbers are usually produced by arithmetic methods, so-called **random number generators** (RNGs). The aim of a random number generator is to generate a sequence of numbers $u_1, u_2, u_3, \dots$ that imitates the abstract mathematical concept of a sequence of mutually independent random variables uniformly distributed over the interval $[0, 1)$. Sequences obtained in this way are *uniquely determined* by one or more starting values called **seeds**, to be given by the user (or some default values). Random number generators should be analyzed theoretically and be backed by practical evidence from extensive statistical testing. According to a much quoted statement by D. H. Lehmer,[26]

> 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. For this and other reasons, a sequence generated like this is called a **pseudorandom** sequence. But 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: that they are impossible and that they are trivial. We have seen that the first myth is correct, unless we add the prefix "pseudo."[27] The second myth, however, is completely false.

In a computer the fundamental concept is not a sequence of decimal random *digits*, but **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. For example, 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. Algorithms for generating deviates for several other distributions are given in Sec. 1.6.3.

[26] Some readers may think that Lehmer's definition is too vague. There have been many deep attempts for more precise formulation. See Knuth [225, pp. 149–179], who catches the flavor of the philosophical discussion of these matters and contributes to it himself.

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

The most widely used generators for producing pseudorandom numbers are **multiple recursive generators**. These are based on a linear recurrence of order k,

$$x_n = \lambda_1 x_{n-1} + \cdots + \lambda_k x_{n-k} + c \mod P, \tag{1.6.2}$$

i.e., x_n is the remainder obtained when the right-hand side is divided by the modulus m. Here P is a positive integer and the coefficients $\lambda_1, \dots, \lambda_k$ belong to the set $\{0, 1, \dots, m-1\}$. The state at step n is $s_n = (x_{n-k+1}, \dots, x_n)$ and the generator is started from a seed $s_{k-1} = (x_0, \dots, x_{k-1})$. When m is large the output can be taken as the number $u_n = x_n/m$. For $k = 1$ we obtain the classical **mixed congruential method**

$$x_n = \lambda x_{n-1} + c \mod P.$$

An important characteristic of an 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_n depends only on x_{n-1}, then the entire sequence repeats once the seed x_0 is duplicated. One can show that if $P = 2^t$ (which is natural on a binary computer) the period of the mixed congruential method is equal to 2^t, assuming that c is odd and that λ gives remainder 1 when divided by four. Also, if P is a prime number and if the coefficients λ_j satisfy certain conditions, then the generated sequence has the maximal period $m^k - 1$; see Knuth [225].

A good RNG should have a period that is guaranteed to be extremely long to make sure that no wrap-around can occur in practice. The linear congruential generator defined by

$$x_n = 16807 x_{n-1} \mod (2^{31} - 1), \tag{1.6.3}$$

with period $(2^{31} - 2)$, 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 [278] 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 [253] pointed out a theoretical weakness of all linear congruential generators. He showed that if k successive random numbers $(x_{i+1}, \dots, x_{i+k})$ 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; see Figure 1.6.2 (left). More precisely, the values will lie on a set of at most $(k!m)^{1/k} \approx (k/e)m^{1/k}$ 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. [288, Chapter 7.1].

A good generator should have been analyzed theoretically and be supported by practical evidence from extensive statistical and other tests. Knuth [225, Chapter 3] points out important ideas, concepts, and facts of the topic, but also mentions some scandalously poor RNGs that were in widespread daily use for decades as standard tools in computer libraries. Although the generators in daily use have improved, *many are still not satisfactory*. He ends this 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 being too shocked at what you find."

L'Ecuyer [239] 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.

L'Ecuyer reports on tests of RNGs used in some popular software products. Microsoft Excel used 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 used 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 2^{48} and sets $u_i = x_i/2^{48}$. The Java standard library uses the same recurrence but constructs random deviates u_i from x_{2i} and x_{2i+1}.

In MATLAB 5 and later versions the previous linear congruential generator has been replaced with a much better generator, based on ideas of Marsaglia; see Figure 1.6.2 (right). 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 [261]. If one generates one million random numbers a second it would take 10^{435} years before it repeats itself!

Some modern linear RNGs can generate huge samples of pseudorandom 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 [256], the "world champion" of RNGs in the year 2000, has a period length of $2^{19,937} - 1$!

Many statistical tests have been adapted and extended for the examination of *arithmetic* methods of (pseudo)random number generation, in current or proposed use for digital

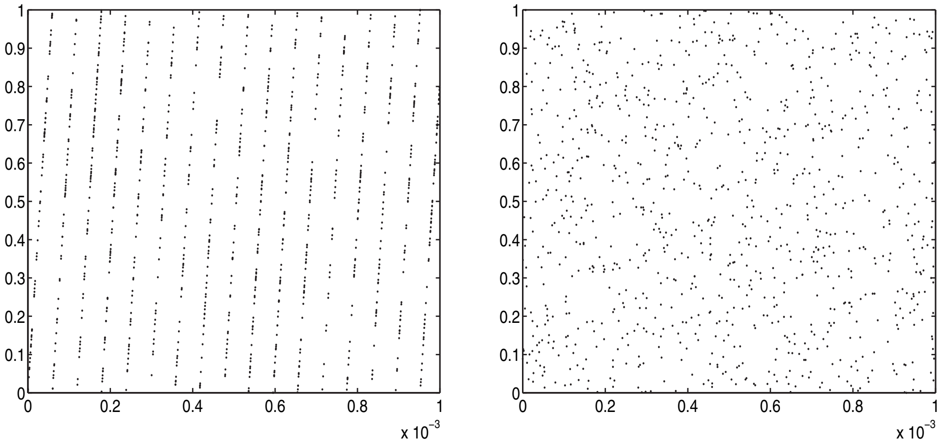


Figure 1.6.2. *Plots of pairs of* 10^6 *random uniform deviates* (U_i, U_{i+1}) *such that* $U_i < 0.0001$. *Left: MATLAB* 4*; Right: MATLAB* 5.

computers. In these, the observed frequencies (a histogram) for some random variable associated with the test are 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 χ^2-test of K. Pearson [282],[28] which we now describe.

Suppose that the space S of the random variable is divided into a finite number r of nonoverlapping parts $S_1, \dots, S_r$. These parts may be groups into which the sample values have been arranged for tabulation purposes. Let the corresponding group probabilities be

$$p_i = Pr(S_i), \quad i = 1, \dots, r, \quad \sum_{i=1}^{r} p_i = 1.$$

We now form a measure of the deviation of the observed frequencies $\nu_1, \dots, \nu_r$, $\sum_i \nu_i = n$, from the expected frequencies

$$\chi^2 = \sum_{i=1}^{r} \frac{(\nu_i - np_i)^2}{np_i} = \sum_{i=1}^{r} \frac{\nu_i^2}{np_i} - n. \tag{1.6.4}$$

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

Let χ_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

[28] This paper, published in 1900 by the English mathematician Karl Pearson (1857–1936), is considered one of the foundations of modern statistics. In it he gave several examples, such as proving that some runs at roulette that he had observed during a visit to Monte Carlo were so far from expectations that the odds against an honest wheel were about 10^{29} to one.

hypothesis is rejected if the observed value of χ^2 is larger than χ_p^2. Often a rejection level of 5% or 1% is used.

Example 1.6.2.

In an experiment consisting of $n = 4040$ throws with a coin, $\nu_1 = 2048$ heads were obtained and hence $\nu_2 = n - \nu_1 = 1992$ tails. Is this consistent with the hypothesis that there is a probability of $p_1 = 1/2$ of throwing tails? Computing

$$\chi^2 = \frac{(\nu_1 - np_1)^2}{np_1} + \frac{(n - \nu_1 - np_1)^2}{np_1} = 2\frac{(2048 - 2020)^2}{2020} = 0.776$$

and using a rejection level of 5%, we find from a table of the χ^2-distribution with one degree of freedom that $\chi_5^2 = 3.841$. Hence the hypothesis is accepted at this level.

Several tests that have been used for testing RNGs are described in Knuth [225, Sec. 3.3]. Some of them are the following:

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

2. **Poker test**. This test applies to generated digits, which are divided into nonoverlapping groups of five 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 α and β are two numbers with $0 \leq \alpha < \beta \leq 1$, we consider the length of consecutive subsequences $U_j, U_{j+1}, \dots, U_{j+r}$ in which U_{j+r} lies between α and β but $U_j, U_{j+1}, \dots, U_{j+r-1}$ does not. This subsequence then represents a gap of length r.

   The special cases $(\alpha, \beta) = (0, 1/2)$ or $(1/2, 1)$ give rise to tests called "runs above the mean" and "runs below the mean," respectively.

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

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

Example 1.6.3.

To test the two-dimensional behavior of an RNG we generated 10^6 pseudorandom numbers U_i. We then placed the numbers (U_i, U_{i+1}) in the unit square of the plot. A thin slice of the surface of the square, 0.0001 wide by 1.0 high, was then 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 statistical test studied by Knuth [225] is the **collision test**. In this test the interval useful $[0, 1)$ is first cut into n equal intervals, for some positive integer n. This partitions the hypercube $[0, 1)^d$ into $k = n^d$ cubic boxes. Then N random points are generated in $[0, 1)^d$ and we record the number of times C that a point falls in a box that already has a point in it. The expectation of the random number C is known to be of very good approximation when N is large. Indeed, C follows approximatively a Poisson distribution with mean equal to $N^2/(2k)$.

For this and other similar tests it has been observed that when the sample size N is increased the test starts to fail when N reaches a critical value N_0, and the failure is clear for all larger values of N. For the collision test it was observed by L'Ecuyer [239] that $N_0 \approx 16\rho^{1/2}$ for good linear congruential generators,where ρ is the period of the RNG. For another statistical test called the *birthday spacing* test the relation was $N_0 \approx 16\rho^{1/3}$.

From such tests is can be concluded that when large sample sizes are needed many RNGs are unsafe to use and can fail decisively. A period 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.

1.6.3 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 the random numbers

$$S = a + (b - a)U$$

will be uniformly distributed on $[a, b)$.

Monte Carlo methods often call for other kinds of distributions. We shall show here how to use uniform deviates to generating random deviates X for several other distributions. Many of the tricks used were originally suggested by John von Neumann in the early 1950s, but have since been improved and refined.

Discrete Distributions

Making a random choice from a finite number k of *equally probable* possibilities is equivalent to generating a random integer X between 1 and k. To do this we take a random deviate

U uniformly distributed on [0, 1), multiply it by k, and take the integer part

$$X = \lceil kU \rceil;$$

here $\lceil x \rceil$ denotes the smallest integer larger than or equal to x. There will be 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 to the values $X = x_i, i = 1 : k$ the probabilities $p_i, i = 1 : k$; note that $\sum p_i = 1$. We can then 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, it may require some thought how to find x quickly for a given u. See the analogous question of finding a first guess to the root of (1.6.5) below and the discussion in Knuth [225, Sec. 3.4.1].

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

$$F(X) = U \quad \text{or, equivalently,} \quad X = F^{-1}(U); \tag{1.6.5}$$

see Figure 1.6.3. Because $F(x)$ is a nondecreasing function, and $Pr\{U \le u\} = u$ for all $u \in [0, 1]$, (1.6.5) 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 (1.6.5) efficiently is the main problem with this method. For some distributions we shall describe better methods below.

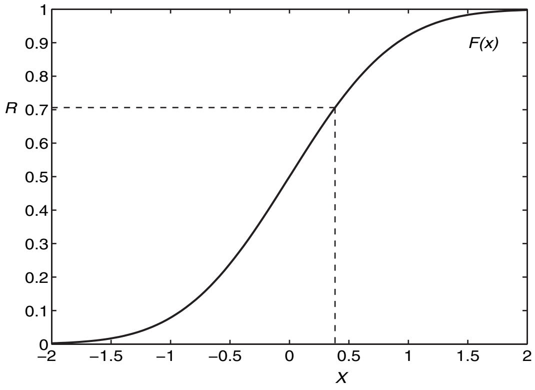


Figure 1.6.3. *Random number with distribution $F(x)$.*

Exponential Deviates

The exponential distribution with parameter $\lambda > 0$ occurs in queuing problems, for example, in telecommunications, 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

$$F(x) = \int_0^x \lambda e^{-\lambda t}\, dt = 1 - e^{-\lambda x}. \tag{1.6.6}$$

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 telecommunication systems and other service systems. A Poisson process with frequency parameter λ 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. An "event" can mean a call on a telephone line, the arrival of a customer to 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 $N = N(0, 1)$ with zero mean and unit standard deviation has the density function

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

Then $\mu + \sigma N$ is a normal deviate with mean μ and standard deviation σ with density function $\frac{1}{\sigma} f((x - \mu)/\sigma)$. Since the normal distribution function

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt \tag{1.6.7}$$

is not an elementary function, solving equation (1.6.5) would be time consuming.

Fortunately, random normal deviates can be obtained in easier ways. In the **polar algorithm** a random point in the unit disk is first 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 $[-1, 1] \times [-1, 1]$. If we compute $S = V_1^2 + V_2^2$ and reject the point if it is outside the unit circle, i.e., if $S > 1$, remaining points will be uniformly distributed on the unit disk. For each accepted point we then form

$$N_1 = \tau V_1, \qquad N_2 = \tau V_2, \quad \tau = \sqrt{\frac{-2 \log S}{S}}. \tag{1.6.8}$$

It can be proved that N_1, N_2 are two independent normally distributed random numbers with zero mean and unit standard deviation.

We point out that N_1, N_2 can be considered to be rectangular coordinates of a point whose polar coordinates (r, ϕ) are determined by the equations

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

The correctness of the above procedure follows from the fact 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. For a proof of this, see Knuth [225, p. 123].

The polar algorithm (used in MATLAB 4) is not optimal. First, about $1 - \pi/4 \approx 21.5\%$ of the uniform numbers are rejected because the generated point falls outside the unit disk. Further, the calculation of the logarithm contributes significantly to the cost. From MATLAB version 5 and later, a more efficient table look-up algorithm developed by Marsaglia and Tsang [255] is used. This is called the "ziggurat" algorithm after the name of ancient Mesopotamian terraced temple mounds which look like two-dimensional step functions. A popular description of the ziggurat algorithm is given by Moler [262]; see also [215].

Example 1.6.4.

To simulate a two-dimensional Brownian motion, trajectories are generated as follows. Initially the particle is located at the origin $w_0 = (0, 0)^T$. At each time step the particle moves randomly,

$$w_{k+1} = w_k + h \begin{pmatrix} N_{1k} \\ N_{2k} \end{pmatrix}, \quad k = 0 : n,$$

where N_{1k} and N_{2k} are normal random deviates generated according to (1.6.8). Figure 1.6.4 shows plots of 32 simulated paths with $h = 0.1$, each consisting of $n = 64$ time steps.

Chi-Square Distribution

The **chi-square distribution** function $P(\chi^2, n)$ is related to the incomplete gamma function (see Abramowitz and Stegun [1, Sec. 6.5]):

$$P(\chi^2, n) = \gamma(n/2, \chi^2/2). \tag{1.6.9}$$

Its complement $Q(\chi^2, n) = 1 - P(\chi^2, n)$ is the probability that the observed chi-square will exceed the value χ^2, even for a correct model. Subroutines for evaluating the χ^2-distribution function as well as other important statistical distribution functions are given in [288, Sec. 6.2–6.3].

Numbers belonging to the chi-square distribution can also be obtained by using the definition of the distribution. If $N_1, N_2, \dots, N_n$ are normal deviates with zero mean and unit variance, the number

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

is distributed as χ^2 with n degrees of freedom.

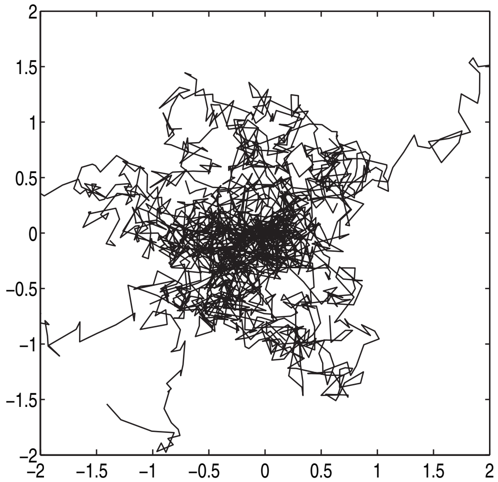


Figure 1.6.4. *Simulated two-dimensional Brownian motion. Plotted are* 32 *simulated paths with* $h = 0.1$*, each consisting of* 64 *steps.*

Other Methods

Several other methods to generate random deviates with Poisson, gamma, and binomial distribution are described in Knuth [225, Sec. 3.4]) and Press et al. [288, Chapter 7.3]. The **rejection method** is based on ideas of von Neumann (1951). A general method introduced by Marsaglia [252] is the **rectangle-wedge-tail method**; see references in Knuth [225]. Powerful combinations of rejection methods and the rectangle-wedge-tail method have been developed.

1.6.4 Reduction of Variance

From statistics, we know that if one makes n independent observations of a quantity whose standard deviation is σ, 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.

Often a more efficient way than increasing the number of samples is to try to decrease the value of σ 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 σ_1 and σ_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

$$n_1/n_2 = \sigma_1^2/\sigma_2^2. \qquad (1.6.10)$$

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.

Example 1.6.5.

In 1777 Buffon[29] 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 the lines between the tiles. He stated that the favorable 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 π. Take the distance δ between the center of the needle and the lines and the angle ϕ 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 π after n throws. In the left part of Figure 1.6.5 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

$$\sqrt{\frac{2}{\pi}\left(1-\frac{2}{\pi}\right)\frac{1}{n}}$$

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 RNG in MATLAB is behaving correctly! (The spikes, directed downward in the figure, typically indicate where $m/n - 2/\pi$ changes sign.)

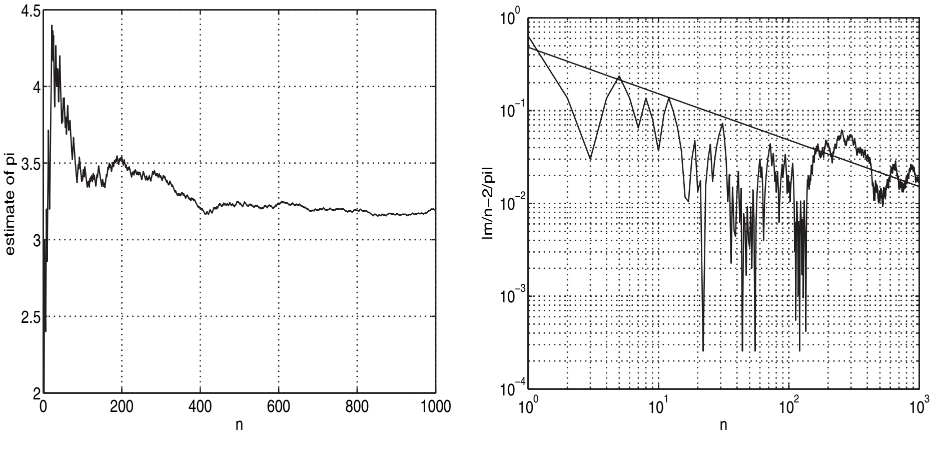


Figure 1.6.5. *The left part shows how the estimate of π varies with the number of throws. The right part compares $|m/n - 2/\pi|$ with the standard deviation of m/n.*

[29] Comte de Buffon (1707–1788), French natural scientist who 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.

An important means of reducing the variance of estimates obtained from the Monte Carlo method is to use **antithetic sequences**. If U_i, $i = 1 : n$, is a sequence of random uniform deviates on [0, 1], then $U_i' = 1 - U_i$, $i = 1 : n$, is an antithetic uniformly distributed sequence. From the sequence in (1.6.1) we get the antithetic sequence

$$0.443065, 0.970545, 0.182765, 0.564115, 0.186495, 0.236975, \ldots. \tag{1.6.11}$$

Antithetic sequences of normally distributed numbers with zero mean 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 1.6.6.

Suppose the numbers x_i are the results of statistically independent measurements of a quantity with expected value μ, and standard deviation σ. Set

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

Then $\bar{x}$ is an estimate of μ, and s is an estimate of σ.

In ten simulations 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. Using an antithetic series, the following values were obtained:

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

The series means are thus

$$708 \quad 783 \quad 651.5 \quad 662.5 \quad 774 \quad 654.5 \quad 641 \quad 666.5 \quad 688.5 \quad 710.5\,,$$

from which one gets the estimate 694.0 ± 15.9.

When one instead supplements the first sequence with 10 values using independent random numbers, the estimate 704 ± 36 using all 20 values is obtained. These results indicate that, in this example, using antithetical sequence produces the desired accuracy with $(15.9/36)^2 \approx 1/5$ 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 1.6.7.

Monte Carlo methods have been used successfully to study queuing problems. A well-known example is a study by Bailey [12] 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 the results that he wanted; this situation is not uncommon when numerical 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.) We use the following numbers which are taken from a table of exponentially distributed random numbers with mean 100:

$$211 \quad 3 \quad 53 \quad 159 \quad 24 \quad 35 \quad 54 \quad 39 \quad 44 \quad 13\,.$$

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.* s

The computations are shown in Table 1.6.1. The following abbreviations are used in what follows: 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 given previously for booking patients. The treatment time T_{time} equals $R/2$, where R are exponentially distributed numbers with mean 100 taken from a table, i.e., the mean treatment time is 50. T_{beg} equals the larger of the number P_{arr} (on the same row) and T_{end} (in the row just above), where $T_{end} = T_{beg} + T_{time}$.

Table 1.6.1. *Simulation of waiting times for 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
Σ	2250			319	2663	1800	2407

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 $P = 2663 - 2250 - 319 = 94$. For $k = 2$ the corresponding waiting times were $D = 406-319 = 87$ and $P = 2407-1800-319 = 288$. Similar calculations for $k = 3$ gave $D = 28$ and $P = 553$ (see Figure 1.6.6). For $k \geq 4$ the doctor never needs to wait.

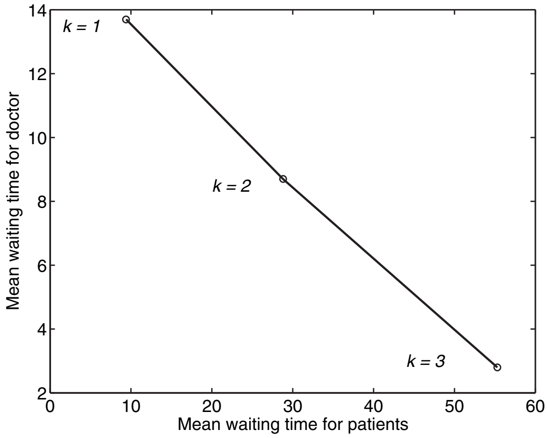


Figure 1.6.6. *Mean waiting times for doctor and patients at polyclinic.*

One cannot, of course, draw any tenable conclusions from one experiment. More experiments should be done 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 enormous quantities of unintelligible results.

Two methods for **reduction of variance** have been introduced here: *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, for example, the parameter k in Example 1.6.7. Note that for this we need to be able to restart the RNG using the same seed. Other effective methods for reducing variance are **importance sampling** and **splitting techniques**; see Hammersley and Handscomb [179].

Review Questions

1.6.1 What is meant by the Monte Carlo method? Describe the origin of the method and give some typical applications. In general, how fast does the error decrease in estimates obtained with the Monte Carlo method?

1.6.2 Describe a linear congruential generator for generating a sequence of uniformly distributed pseudorandom numbers. What are some important properties of such a generator?

1.6.3 Describe a general method for obtaining pseudorandom numbers with a given discrete or continuous distribution from uniformly distributed random numbers. Give examples of its use.

1.6.4 Describe some statistical tests which can be applied to a pseudorandom number generator.

1.6.5 What are the most important properties of a Poisson process? How can one generate a Poisson process with the help of random numbers?

1.6.6 What is the mixed congruential method for generating pseudorandom numbers? What important difference is there between the numbers generated by this method and "genuine" random numbers?

1.6.7 Explain what is meant by *reduction of variance* in estimates made with the Monte Carlo method. Give three methods for reduction of variance. What is the quantitative connection between reducing variance and decreasing the amount of computation needed in a given problem?

Problems and Computer Exercises

1.6.1 (C. Moler) Consider the toy RNG, $x_i = ax_i \bmod m$, 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 period $m - 1 = 30$, equal to the maximum possible.

1.6.2 Simulate (say) 360 throws with two standard dice. Denote the sum of the number of dots on the two dice on the nth throw by Y_n, $2 \leq Y_n \leq 12$. Tabulate or draw a histogram of the (absolute) frequency of the occurrence of j dots versus j, $j = 2 : 12$. Make a conjecture about the true value of $Pr(Y_n = j)$. Try to confirm it by repeating the experiment with fresh uniform random numbers. When you have found the right conjecture, you will find that it is not hard to prove.

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

(b) Conduct an antithetic experiment, and take the average of the two results. Is the average better than one could 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 $\pi^2/2$.)

1.6.4 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." (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 ν 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 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 drawing beautiful hearts, clubs, etc.

1.6.5 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_i), 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.

1.6.6 Repeat the simulation in the queuing problem in Example 1.6.7 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 1.6.7. Compute the mean of the waiting times for the doctor and for all patients for this and the previous experiment.

1.6.7 A target with depth $2b$ and very large width is to be shot at with a cannon. (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 random variable $X = N(0, \sigma_1)$.

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 $N(0, \sigma_2)$; 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 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 pseudorandom numbers to shoot 10 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.

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

Notes and References

The methods and problems presented in this introductory chapter will be studied in greater detail later in this volume and in Volume II. In particular, numerical quadrature methods are studied in Chapter 5 and methods for solving a single nonlinear equation in Chapter 6. For a survey of sorting algorithms we refer to [288, Chapter 8]. A comprehensive treatment of sorting and searching is given in Knuth [226].

Although the history of Gaussian elimination goes back at least to Chinese mathematicians (about 250 B.C.), there was no practical experience of solving large linear systems until the advent of computers in the 1940s. Gaussian elimination was the first numerical algorithm to be subjected to a rounding error analysis. In 1946 there was a mood of pessimism about the stability of Gaussian elimination. Bounds had been produced showing that the error in the solution would be proportional to 4^n. This suggested that it would be impossible to solve even systems of modest order. A few years later J. von Neumann and H. H. Goldstein published more relevant error bounds. In 1948 A. M. Turing wrote a remarkable paper [355], in which he formulated the LU factorization and introduced matrix condition numbers.

Several of the great mathematicians at the turn of the nineteenth century worked on methods for solving overdetermined linear systems. In 1799, Laplace used the principle of minimizing the sum of absolute errors $|r_i|$, with the added conditions that the errors sum to zero. This leads to a solution x that satisfies at least n equations exactly. The method of least squares was first published as an algebraic procedure by Legendre in 1805 [240]. Gauss justified the least squares principle as a statistical procedure in [135], where he claimed to have used the method since 1795. This led to one of the most famous priority disputes in the history of mathematics. Gauss further developed the statistical aspects in 1821–1823. For an interesting accounts of the history of the invention of least squares, see Stigler [330].

For a comprehensive treatment of all aspects of random numbers we refer to Knuth [225]. Another good reference on the state of the art is the monograph by Niederreiter [270]. Guidelines for choosing a good RNG are given in Marsaglia [254], the monograph by Gentle [149], and in the two surveys L'Ecuyer [237, 238]. Hellekalek [187] explains how to access RNGs for practitioners. An introduction to Monte Carlo methods and their applications is given by Hammersley and Handscomb [179]. There is a close connection between random number generation and data encryptation; see Press et al. [288, Chapter 7.5].

Some later chapters in this book assume a working knowledge in numerical linear algebra. Online Appendix A gives a brief survey of matrix computations. A more in-depth treatment of direct and iterative methods for linear systems, least squares, and eigenvalue problems is planned for Volume II. Some knowledge of modern analysis including analytic functions is also needed for some more advanced parts of the book. The classical textbook by Apostol [7] is highly recommended as a suitable reference.

The *James & James Mathematics Dictionary* [206] is a high-quality general mathematics dictionary covering arithmetic to calculus, and it includes a multilingual index. *CRC Concise Encyclopedia of Mathematics* [363] is a comprehensive compendium of mathematical definitions, formulas, and references. A free Web encyclopedia containing surveys and references is Eric Weisstein's MathWorld at `mathworld.wolfram.com`.

The development of numerical analysis during the period when the foundation was laid in the sixteenth through the nineteenth century is traced in Goldstine [157]. Essays on the history of scientific computing can be found in Nash [269]. An interesting account of the developments in the twentieth century is given in [55]. An eloquent essay on the foundations of computational mathematics and its relation to other fields is given by Baxter and Iserles [21].

In 2000–2001, the *Journal of Computational and Applied Mathematics* published a series of papers on Numerical Analysis of the 20th Century, with the aim of presenting the historical development of numerical analysis and review current research. The papers were arranged in seven volumes.

We give below a selection of textbooks and review papers on numerical methods. Even though the selection is by no means complete and reflects a subjective choice, we hope it can serve as a guide for a reader who, out of interest (or necessity!), wishes to deepen his or her knowledge. Both recent textbooks and older classics are included. Note that reviews of new books can be found in *Mathematical Reviews* as well as in the journals *SIAM Review* and *Mathematics of Computation*. A more complete guide to relevant literature and software is given in Online Appendix C.

Many outstanding textbooks in numerical analysis were originally published in the 1960s and 70s. The classical text by Hildebrand [197] can still be used as an introduction. Isaacson and Keller [204] give a rigorous mathematical treatment of classical topics, including differential equations and orthogonal polynomials. The present authors' textbook [87] was used at many universities in the USA and is still available.

Hamming [180] is a more applied text and aims at combining mathematical theory, heuristic analysis, and computing methods. It emphasizes the message that *"the purpose of computing is insight, not numbers."* An in-depth treatment of several areas such as numerical quadrature and approximation is found in the comprehensive book by Ralston and Rabinowitz [290]. This book also contains a large number of interesting and fairly advanced problems.

The book by Forsythe, Malcolm, and Moler [120] is notable in that it includes a set of Fortran subroutines of unusually high quality. Kahaner, Moler, and Nash [215] comes with a disk containing software. A good introduction to scientific computing is given by Golub and Ortega [163]. A matrix-vector approach is used in the MATLAB oriented text of Van Loan [360]. Analysis is complemented with computational experiments using a package of more than 200 m-files. Cheney and Kincaid [67] is an undergraduate text with many examples and exercises. Kincaid and Cheney [221] is a related textbook but more mathematically oriented. Two other good introductory texts are Eldén, Wittmeyer-Koch, and Nielsen [107] and Süli and Mayers [337].

Heath [186] is a popular, more advanced, and comprehensive text. Gautschi [144] is an elegant introductory text containing a wealth of computer exercises. Much valuable and hard-to-find information is included in notes after each chapter. The bestseller by Press et al. [288] surveys contemporary numerical methods for the applied scientist, but is weak on analysis.

Several good textbooks have been translated from German, notably the excellent book by Stoer and Bulirsch [331]. This is particularly suitable for a reader with a good mathematical background. Hämmerlin and Hoffmann [178] and Deuflhard and Hohmann [94] are less comprehensive but with a modern and careful treatment. Schwarz [311] is a mathematically oriented text which also covers ordinary and partial differential equations. Rutishauser [305] is an annotated translation of a highly original textbook by one of the pioneers of numerical analysis. Though brief, the book by Tyrtychnikov [356] is original and thorough. It also contains references to Russian literature unavailable in English.

Since numerical analysis is still in a dynamic stage it is important to keep track of new developments. An excellent source of survey articles on topics of current interest can be found in *Acta Numerica*, a Cambridge University Press Annual started in 1992. The journal *SIAM Review* also publishes high-quality review papers.

Another collection of outstanding survey papers on special topics is being published in a multivolume sequence in the *Handbook of Numerical Analysis* [69], edited by Philippe G. Ciarlet and Jacques-Louis Lions. It offers comprehensive coverage in all areas of numerical analysis as well as many actual problems of contemporary interest; see section C3 in Online Appendix C.