CHAPTER TWO

Computer Arithmetic

2.1 Floating-Point Numbers and Roundoff Errors
2.2 Absolute and Relative Errors: Loss of Significance
2.3 Stable and Unstable Computations: Conditioning
CHAPTER THREE
Solution of Nonlinear Equations

3.1 Bisection (Interval Halving) Method
3.2 Newton’s Method
3.3 Secant Method
*3.4 Fixed Points and Functional Iteration
*3.5 Computing Zeros of Polynomials
*3.6 Homotopy and Continuation Methods
CHAPTER FOUR

Solving Systems of Linear Equations

4.1 Matrix Algebra
4.2 The LU and Cholesky Factorizations
4.3 Pivoting and Constructing an Algorithm
4.4 Norms and the Analysis of Errors
4.5 Neumann Series and Iterative Refinement
4.6 Solution of Equations by Iterative Methods
4.7 Steepest Descent and Conjugate Gradient Methods
4.8 Analysis of Roundoff Error in the Gaussian Algorithm
CHAPTER FIVE
Selected Topics in Numerical Linear Algebra

5.1 Matrix Eigenvalue Problem: Power Method
5.2 Schur’s and Gershgorin’s Theorems
*5.3 Orthogonal Factorizations and Least-Squares Problems
*5.4 Singular-Value Decomposition and Pseudoinverses
*5.5 The $QR$-Algorithm of Francis for the Eigenvalue Problem
CHAPTER SIX

Approximating Functions

6.1 Polynomial Interpolation
6.2 Divided Differences
6.3 Hermite Interpolation
6.4 Spline Interpolation
*6.5 The B-Splines: Basic Theory
*6.6 The B-Splines: Applications
6.7 Taylor Series
*6.8 Best Approximation: Least-Squares Theory
*6.9 Best Approximation: Chebyshev Theory
*6.10 Interpolation in Higher Dimensions
*6.11 Continued Fractions
*6.12 Trigonometric Interpolation
*6.13 Fast Fourier Transform
6.14 Adaptive Approximation
CHAPTER SEVEN
Numerical Differentiation and Integration

7.1 Numerical Differentiation and Richardson Extrapolation
7.2 Numerical Integration Based on Interpolation
7.3 Gaussian Quadrature
7.4 Romberg Integration
7.5 Adaptive Quadrature
*7.6 Sard’s Theory of Approximating Functionals
*7.7 Bernoulli Polynomials and the Euler-Maclaurin Formula
CHAPTER EIGHT

Numerical Solution of Ordinary Differential Equations

8.1 The Existence and Uniqueness of Solutions
8.2 Taylor-Series Method
8.3 Runge-Kutta Methods
8.4 Multistep Methods
*8.5 Local and Global Errors: Stability
8.6 Systems and Higher-Order Ordinary Differential Equations
*8.7 Boundary-Value Problems
8.8 Boundary-Value Problems: Shooting Methods
8.9 Boundary-Value Problems: Finite-Difference Methods
*8.10 Boundary-Value Problems: Collocation
*8.11 Linear Differential Equations
8.12 Stiff Equations
CHAPTER NINE
Numerical Solution of Partial Differential Equations

9.1 Parabolic Equations: Explicit Methods
9.2 Parabolic Equations: Implicit Methods
9.3 Problems Without Time Dependence: Finite-Difference Methods
*9.4 Problems Without Time Dependence: Galerkin and Ritz Methods
*9.5 First-Order Partial Differential Equations: Characteristic Curves
*9.6 Quasilinear Second-Order Equations: Characteristics
*9.7 Other Methods for Hyperbolic Problems
9.8 Multigrid Method
*9.9 Fast Methods for Poisson’s Equation
CHAPTER TEN

Linear Programming and Related Topics

*10.1 Convexity and Linear Inequalities
*10.2 Linear Inequalities
10.3 Linear Programming
10.4 The Simplex Algorithm
2.1 Floating-Point Numbers and Roundoff Errors
2.2 Absolute and Relative Errors: Loss of Significance
2.3 Stable and Unstable Computations: Conditioning
Hence, nonzero normalized machine numbers are bit strings whose values are decoded as follows:

\[ x = (-1)^s q \times 2^m \]  

(5)

where

\[ q = (1.f)_2 \quad \text{and} \quad m = e - 127 \]

Here \( 1 \leq q < 2 \) and the most significant bit in \( q \) is 1 and is not explicitly stored. Also, here \( s \) is the bit representing the sign of \( x \) (positive: bit 0, negative: bit 1), \( m = e - 127 \) is the 8-bit biased exponent, and \( f \) is the 23-bit fractional part of the real number \( x \) that, together with an implicit leading bit 1, yields the significant digit field \((1.\Box\Box\Box \cdots \Box\Box\Box)_2\).

A real number expressed as in Equation (5) is said to be in \textbf{normalized floating-point form}. If it can then be represented with \( |m| \) occupying 8 bits and \( q \) occupying 23 bits, it is a \textbf{machine number} in the \textit{Marc-32}. That is, it can be precisely represented within this particular computer. Most real numbers are not precisely representable within the \textit{Marc-32}. When such a number occurs as an input datum or as the result of a computation, an inevitable error will arise in representing it as accurately as possible by a \textit{machine} number.

The restriction that \( |m| \) require no more than 8 bits means that

\[ 0 < e < (1111111)_2 = 2^8 - 1 = 255 \]
Zero, Infinity, NaN

In IEEE standard arithmetic, there are two forms of 0: +0 and −0, in single precision, are represented in the computer by the words [00000000]₁₆ and [80000000]₁₆, respectively. Most arithmetic operations that result in a 0 value are given the value +0. A very tiny negative number that is 0 to machine precision is given the value −0.

Similarly, there are two forms of infinity: +∞ and −∞, in single precision, are represented by the computer words [7F800000]₁₆ and [FF800000]₁₆, respectively. Usually, infinity is treated like a very large number whenever it makes sense to do so. For example, suppose that \( x \) is a floating-point number in the range 0 < \( x \) < ∞; then each of the computations \( x + ∞, x \times ∞, \) and \( ∞ / x \) is given the value +∞, whereas \( x / ∞ \) becomes +0. Here ∞ is understood to be +∞. Similar results hold for −∞.

NaN means **Not a Number** and results from an indeterminate operation such as 0/0, ∞ − ∞, \( x + \text{NaN} \), and so on. NaN’s are represented by computer words with e = 255 and f ≠ 0.

Machine Rounding

In addition to rounding input data, rounding is needed after most arithmetic operations. The result of an arithmetic operation resides in a long 80-bit computer register and must be rounded to single-precision before being placed in memory. An analogous situation occurs for double-precision operations.

The usual (default) rounding mode is **round to nearest**: The closer of the two machine numbers on the left and right of the real number is selected. In the case of a tie, **round to even** is used: If the real number is exactly halfway between the machine numbers to its left and right, then the even machine number is chosen. With this default rounding scheme (round to nearest plus round to even), the maximum error is half a unit in the least significant place.

Other modes of rounding are **directed rounding** such as **round toward 0** (also known as **truncation**), **round toward +∞**, and **round toward −∞**.
TABLE 2.1 Results from some Fortran 90 Intrinsic Procedures: Sun 4

<table>
<thead>
<tr>
<th>X</th>
<th>single precision</th>
<th>double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGITS(X)</td>
<td>24</td>
<td>53</td>
</tr>
<tr>
<td>EPSILON(X)</td>
<td>$1.19 \times 10^{-7} \approx 2^{-23}$</td>
<td>$2.22 \times 10^{-16} \approx 2^{-52}$</td>
</tr>
<tr>
<td>HUGE(X)</td>
<td>$3.40 \times 10^{38} \approx (2 - 2^{-23})2^{127}$</td>
<td>$1.80 \times 10^{308} \approx (2 - 2^{-23})2^{1023}$</td>
</tr>
<tr>
<td>PRECISION(X)</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>TINY(X)</td>
<td>$1.18 \times 10^{-38} \approx 2^{-126}$</td>
<td>$2.23 \times 10^{-308} \approx 2^{-1022}$</td>
</tr>
</tbody>
</table>

Fortran 90

In Fortran 90, a large number of generic intrinsic procedures are available for determining the numerical environment in the computer being used. In general, they return a number of the same type (real, integer, complex, etc.) and kind (single precision, double precision, etc.) as the argument. For example, some of these procedures related to floating-point numbers are: DIGITS—number of significant (binary) digits, EPSILON—a positive number that is almost negligible compared to unity (smallest floating-point number $\varepsilon$ such that $1 + \varepsilon \neq 1$), HUGE—largest number, PRECISION—decimal precision, and TINY—smallest positive number. Here we assume the machine is a binary computer.

Table 2.1 is constructed from the results of calling these procedures on a 32-bit-word workstation with IEEE standard arithmetic (Sun 4). For X integer, DIGITS(X) is 31 and HUGE(X) is $2147483647 \approx 2^{31} - 1$.

Table 2.2 is constructed from the results of calling these procedures on a 64-bit-word supercomputer without IEEE standard arithmetic (Cray Y-MP). For X integer, DIGITS(X) is 46 and HUGE(X) is $70368744177663 \approx 2^{46} - 1$. 
Q1. In floating point arithmetic, the machine epsilon $\varepsilon_M$ (also called machine precision) is, for a particular floating point unit, the difference between 1 and the smallest exactly representable number greater than one. To compute the value of the machine epsilon $\varepsilon_M$ on your computer.

Q2. (a) Find any floating-point number $x$ in the range $1 < x < 2$ such that $x^*(1/x) \neq 1$; that is, $\text{fl}(x \text{fl}(1/x))$ is not exactly 1.
(b) To find out the smallest such number.

Q3. To compute

\[
\begin{align*}
  f(x) &= \sqrt{x^2 + 1} - 1 \\
  g(x) &= x^2 / (\sqrt{x^2 + 1} + 1)
\end{align*}
\]

for a succession of values of $x$ such as $8^{-1}, 8^{-2}, 8^{-3}, \ldots$. Although $f=g$, the computer will produce different results. Which results are reliable and which are not?

Q4. To compute the dot product of the following two vectors:
\[
x = [ 2.718281828, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957 ]
\[
y = [ 1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049 ]
\]
Compute the summation in four ways:
(a) Forward order $\sum_{i=1}^{5} x_i y_i$
(b) Reverse order $\sum_{i=5}^{1} x_i y_i$
(c) Largest-to-smallest order (add positive numbers in order from largest to smallest, then add negative numbers in order from smallest to largest, and then add the two partial sums)
(d) Smallest-to-largest order (reverse the order of adding in the previous method)
CHAPTER THREE

Solution of Nonlinear Equations

3.1 Bisection (Interval Halving) Method
3.2 Newton’s Method
3.3 Secant Method
*3.4 Fixed Points and Functional Iteration
*3.5 Computing Zeros of Polynomials
*3.6 Homotopy and Continuation Methods
3.1 Bisection (Interval Halving) Method

If \( f \) is a continuous function on the interval \([a, b]\) and if \( f(a)f(b) < 0 \), then \( f \) must have a zero in \((a, b)\). Since \( f(a)f(b) < 0 \), the function \( f \) changes sign on the interval \([a, b]\) and, therefore, it has at least one zero in the interval. This is a consequence of the following fundamental theorem.

**THEOREM 1** Intermediate-Value Theorem for Continuous Functions  If \( f \) is continuous on \([a, b]\), and if \( f(a) < y < f(b) \), then \( f(x) = y \) for some \( x \in (a, b) \).

The bisection method exploits this idea in the following way. If \( f(a)f(b) < 0 \), then we compute \( c = \frac{1}{2}(a + b) \) and test whether \( f(a)f(c) < 0 \). If this is true, then \( f \) has a zero in \([a, c]\). So we rename \( c \) as \( b \) and start again with the new interval \([a, b]\), which is half as large as the original interval. If \( f(a)f(c) > 0 \), then \( f(c)f(b) < 0 \), and in this case we rename \( c \) as \( a \). In either case, a new interval containing a zero of \( f \) has been produced, and the process can be repeated. Figures 3.2(a) and (b) show the two cases, assuming \( f(a) > 0 > f(b) \). These figures makes it clear why the bisection method finds one zero but not all the zeros in the interval \([a, b]\). Of

```
3.1 Bisection (Interval Halving) Method

input a, b, M, δ, ε
u ← f(a)
v ← f(b)
e ← b - a
output a, b, u, v
if sgn(u) = sgn(v) then stop
for k = 1 to M do
    e ← e/2
    c ← a + e
    w ← f(c)
    output k, c, w, e
    if |e| < δ or |w| < ε then stop
    if sgn(w) ≠ sgn(u) then
        b ← c
        v ← w
    else
        a ← c
        u ← w
    end if
end do
```
Write and test a subprogram or procedure to implement the bisection algorithm. Test the program on these functions and intervals:

(a) $x^{-1} - \tan x$ on $[0, \pi/2]$
(b) $x^{-1} - 2^x$ on $[0, 1]$
(c) $2^{-x} + e^x + 2 \cos x - 6$ on $[1, 3]$
(d) $(x^3 + 4x^2 + 3x + 5)/(2x^3 - 9x^2 + 18x - 2)$ on $[0, 4]$
Newton’s Algorithm

A simple algorithm that applies $M$ steps of Newton’s method, starting with an initial value for $x$, is:

```plaintext
input $x, M$
$y \leftarrow f(x)$
output $0, x, y$
for $k = 1$ to $M$ do
    $x \leftarrow x - y/f'(x)$
    $y \leftarrow f(x)$
    output $k, x, y$
end do
```

A more detailed pseudocode including stopping criteria follows:

```plaintext
input $x_0, M, \delta, \epsilon$
$v \leftarrow f(x_0)$
output $0, x_0, v$
if $|v| < \epsilon$ then stop
for $k = 1$ to $M$ do
    $x_1 \leftarrow x_0 - v/f'(x_0)$
    $v \leftarrow f(x_1)$
    output $k, x_1, v$
    if $|x_1 - x_0| < \delta$ or $|v| < \epsilon$ then stop
    $x_0 \leftarrow x_1$
end do
```
Example 1 Use Newton’s method, with double-precision computation, to find the negative zero of the function $f(x) = e^x - 1.5 - \tan^{-1} x$.

Solution The preceding algorithm was executed with double precision on a machine with 48-bit mantissas in single precision. (Double-precision machine numbers have 96 bits, corresponding to about 28 decimal places.) The function $f'(x) = e^x - (1 + x^2)^{-1}$ as well as $f$ had to be programmed. A starting point of $x_0 = -7$ was chosen. The output from the computer program is shown here.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$x$</th>
<th>$f(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.00000 00000 00000 00000 00000 00000 00000 00000 0</td>
<td>$-0.702 \times 10^1$</td>
</tr>
<tr>
<td>1</td>
<td>-10.67709 61766 40013 99296 98438 6</td>
<td>$-0.226 \times 10^{-1}$</td>
</tr>
<tr>
<td>2</td>
<td>-13.27916 73756 32712 90859 78631 9</td>
<td>$-0.437 \times 10^{-2}$</td>
</tr>
<tr>
<td>3</td>
<td>-14.05365 58542 69238 73474 83175 3</td>
<td>$-0.239 \times 10^{-3}$</td>
</tr>
<tr>
<td>4</td>
<td>-14.10110 99568 66413 47616 31270 6</td>
<td>$-0.800 \times 10^{-6}$</td>
</tr>
<tr>
<td>5</td>
<td>-14.10126 97709 39415 94621 57950 6</td>
<td>$-0.901 \times 10^{-11}$</td>
</tr>
<tr>
<td>6</td>
<td>-14.10126 97727 39968 42508 30031 4</td>
<td>$-0.114 \times 10^{-20}$</td>
</tr>
<tr>
<td>7</td>
<td>-14.10126 97727 39968 42531 15512 2</td>
<td>0.000</td>
</tr>
<tr>
<td>8</td>
<td>-14.10126 97727 39968 42531 15512 2</td>
<td>0.000</td>
</tr>
</tbody>
</table>

The output shows rapid convergence of the iterates; in fact, in each step the number of correct digits in the approximation seems to double. Our analysis will reveal why this is true.
Secant Algorithm

An algorithm for the secant method can be written as follows. We modify the standard secant method slightly in order to obtain nonincreasing function values.

\[
\text{input } a, b, M, \delta, \varepsilon \\
fa \leftarrow f(a); \ fb \leftarrow f(b) \\
\text{output } 0, a, fa \\
\text{output } 1, b, fb \\
\text{for } k = 2 \text{ to } M \text{ do} \\
\quad \text{if } |fa| > |fb| \text{ then} \\
\quad \quad a \leftarrow b; \ fa \leftarrow fb \\
\quad \text{end if} \\
\quad s \leftarrow (b - a)/(fb - fa) \\
\quad b \leftarrow a \\
\quad fb \leftarrow fa \\
\quad a \leftarrow a - fa \ast s \\
\quad fa \leftarrow f(a) \\
\text{output } k, a, fa \\
\quad \text{if } |fa| < \varepsilon \text{ or } |b - a| < \delta \text{ then stop} \\
\text{end do}
\]

Notice that in the pseudocode the endpoints of \([a, b]\) may be interchanged to keep \(|f(a)| \leq |f(b)|\). Thus, the pair \(\{x_n, x_{n-1}\}\) has \(|f(x_n)| \leq |f(x_{n-1})|\), and the next pair \(\{x_{n+1}, x_n\}\) has \(|f(x_{n+1})| \leq |f(x_n)|\). This ensures that the absolute value of the function is nonincreasing beginning in step 2.
Example 1  Use the secant method to find a zero of the function

\[ f(x) = x^3 - \sinh x + 4x^2 + 6x + 9 \]

Solution  A rough plot suggests that there is a zero between 7 and 8. We take these points as \( x_0 \) and \( x_1 \) in the algorithm. When this code was run on a machine similar to the

![Secant line](image)

**FIGURE 3.6** Geometric interpretation of secant method

Marc-32, the following results were obtained:

<table>
<thead>
<tr>
<th>( n )</th>
<th>( x_n )</th>
<th>( f(x_n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8.00000</td>
<td>(-0.665 \times 10^3)</td>
</tr>
<tr>
<td>1</td>
<td>7.00000</td>
<td>(0.417 \times 10^2)</td>
</tr>
<tr>
<td>2</td>
<td>7.05895</td>
<td>(0.208 \times 10^2)</td>
</tr>
<tr>
<td>3</td>
<td>7.11764</td>
<td>(-0.183 \times 10^1)</td>
</tr>
<tr>
<td>4</td>
<td>7.11289</td>
<td>(0.710 \times 10^{-1})</td>
</tr>
<tr>
<td>5</td>
<td>7.11306</td>
<td>(0.244 \times 10^{-3})</td>
</tr>
<tr>
<td>6</td>
<td>7.11306</td>
<td>(0.191 \times 10^{-4})</td>
</tr>
</tbody>
</table>
Chapter Four

Solving Systems of Linear Equations

4.1 Matrix Algebra
4.2 The LU and Cholesky Factorizations
4.3 Pivoting and Constructing an Algorithm
4.4 Norms and the Analysis of Errors
4.5 Neumann Series and Iterative Refinement
*4.6 Solution of Equations by Iterative Methods
*4.7 Steepest Descent and Conjugate Gradient Methods
*4.8 Analysis of Roundoff Error in the Gaussian Algorithm
Let us consider a system of $n$ linear equations in $n$ unknowns $x_1, x_2, \ldots, x_n$. It can be written in the form

$$
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
  a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\
  a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn}
\end{bmatrix} \begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  \vdots \\
  x_n
\end{bmatrix} = \begin{bmatrix}
  b_1 \\
  b_2 \\
  b_3 \\
  \vdots \\
  b_n
\end{bmatrix}
$$

The matrices in this equation are denoted by $A$, $x$, and $b$. Thus, our system is simply

$$Ax = b \quad (1)$$

**Easy-to-Solve Systems**

We begin by looking for special types of systems that can be *easily* solved. For example, suppose that the $n \times n$ matrix $A$ has a **diagonal structure**. This means that all the nonzero elements of $A$ are on the main diagonal, and System (1) is

$$
\begin{bmatrix}
  a_{11} & 0 & 0 & \cdots & 0 \\
  0 & a_{22} & 0 & \cdots & 0 \\
  0 & 0 & a_{33} & \cdots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & 0 & \cdots & a_{nn}
\end{bmatrix} \begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  \vdots \\
  x_n
\end{bmatrix} = \begin{bmatrix}
  b_1 \\
  b_2 \\
  b_3 \\
  \vdots \\
  b_n
\end{bmatrix}
$$
In this case, our system collapses to \( n \) simple equations, and the solution is

\[
x = \begin{bmatrix}
    b_1/a_{11} \\
    b_2/a_{22} \\
    b_3/a_{33} \\
    \vdots \\
    b_n/a_{nn}
\end{bmatrix}
\]

If \( a_{ii} = 0 \) for some index \( i \), and if \( b_i = 0 \) also, then \( x_i \) can be any real number. If \( a_{ii} = 0 \) and \( b_i \neq 0 \), no solution of the system exists.

Continuing our search for \textit{easy} solutions of System (1), we assume a \textbf{lower triangular structure} for \( A \). This means that all the nonzero elements of \( A \) are situated on or below the main diagonal, and System (1) is

\[
\begin{bmatrix}
    a_{11} & 0 & 0 & \cdots & 0 \\
    a_{21} & a_{22} & 0 & \cdots & 0 \\
    a_{31} & a_{32} & a_{33} & \cdots & 0 \\
    \vdots & \vdots & \vdots & \ddots & \vdots \\
    a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
    x_1 \\
    x_2 \\
    x_3 \\
    \vdots \\
    x_n
\end{bmatrix}
= 
\begin{bmatrix}
    b_1 \\
    b_2 \\
    b_3 \\
    \vdots \\
    b_n
\end{bmatrix}
\]

To solve this, assume that \( a_{ii} \neq 0 \) for all \( i \); then obtain \( x_1 \) from the first equation. With the known value of \( x_1 \) substituted in the second equation, solve the second equation for \( x_2 \). We proceed in the same way, obtaining \( x_1, x_2, \ldots, x_n \), one at a time and in this order. A formal algorithm for the solution in this case is called \textbf{forward substitution}:

\[
\begin{align*}
\text{input} & \quad n, (a_{ij}), (b_i) \\
\text{for} \ i = 1 \ \text{to} \ n \ \text{do} & \\
\quad x_i & \leftarrow \left( b_i - \sum_{j=1}^{i-1} a_{ij}x_j \right)/a_{ii} \\
\text{end do} & \\
\text{output} & \quad (x_i)
\end{align*}
\]
As is customary, any sum of the type \( \sum_{i=\alpha}^{\beta} x_i \) in which \( \beta < \alpha \) is interpreted to be 0.

The same ideas can be exploited to solve a system having an **upper triangular structure**. Such a matrix system has the form

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\
0 & a_{22} & a_{23} & \cdots & a_{2n} \\
0 & 0 & a_{33} & \cdots & a_{3n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n
\end{bmatrix}
=
\begin{bmatrix}
b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n
\end{bmatrix}
\]

Again, it must be assumed that \( a_{ii} \neq 0 \) for \( 1 \leq i \leq n \). The formal algorithm to solve for \( x \) is as follows and is called **back substitution**:

\[
\text{input } n, (a_{ij}), (b_i) \\
\text{for } i = n \text{ to } 1 \text{ step } -1 \text{ do} \\
\quad x_i \leftarrow \left( b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right) / a_{ii} \\
\text{end do} \\
\text{output } (x_i)
\]

Q9. To implement the back substitution.
**LU-Factorizations**

Suppose that $A$ can be factored into the product of a lower triangular matrix $L$ and an upper triangular matrix $U$: $A = LU$. Then in order to solve the system of equations $Ax = b$, it is enough to solve this problem in two stages:

$$Lz = b \quad \text{solve for } z$$

$$Ux = z \quad \text{solve for } x$$

Our previous analysis indicates that solving these two triangular systems is simple.

We shall show how the factorization $A = LU$ can be carried out, provided that in certain steps of the computation zero divisors are not encountered. Not every matrix has such a factorization, and this difficulty will be investigated presently.

We begin with an $n \times n$ matrix $A$ and search for matrices

$$L = \begin{bmatrix}
\ell_{11} & 0 & 0 & \cdots & 0 \\
\ell_{21} & \ell_{22} & 0 & \cdots & 0 \\
\ell_{31} & \ell_{32} & \ell_{33} & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\ell_{n1} & \ell_{n2} & \ell_{n3} & \cdots & \ell_{nn}
\end{bmatrix}$$

$$U = \begin{bmatrix}
u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\
0 & u_{22} & u_{23} & \cdots & u_{2n} \\
0 & 0 & u_{33} & \cdots & u_{3n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & u_{nn}
\end{bmatrix}$$

such that

$$A = LU$$

(8)
Q10. To implement the general LU-factorization.

The algorithm for the general LU-factorization is as follows:

input \( n, (a_{ij}) \)

for \( k = 1 \) to \( n \) do

Specify a nonzero value for either \( \ell_{kk} \) or \( u_{kk} \) and compute the other from

\[
\ell_{kk} u_{kk} = a_{kk} - \sum_{s=1}^{k-1} \ell_{ks} u_{sk}.
\]

for \( j = k + 1 \) to \( n \) do

\[
u_{kj} \leftarrow \left( a_{kj} - \sum_{s=1}^{k-1} \ell_{ks} u_{sj} \right) / \ell_{kk}
\]

end do

for \( i = k + 1 \) to \( n \) do

\[
\ell_{ik} \leftarrow \left( a_{ik} - \sum_{s=1}^{k-1} \ell_{is} u_{sk} \right) / u_{kk}
\]

end do

end do

output \((\ell_{ij}), (u_{ij})\)
Q11. To implement the Cholesky factorization.

**Theorem 2**

If $A$ is a real, symmetric, and positive definite matrix, then it has a unique factorization, $A = LL^T$, in which $L$ is lower triangular with a positive diagonal.

The algorithm for the Cholesky factorization will then be as follows:

- **Input**: $n$, $(a_{ij})$
- **Algorithm**:
  
  for $k = 1$ to $n$
  
  $\ell_{kk} \leftarrow \left( a_{kk} - \sum_{s=1}^{k-1} \ell_{ks}^2 \right)^{1/2}$
  
  for $i = k + 1$ to $n$
  
  $\ell_{ik} \leftarrow \left( a_{ik} - \sum_{s=1}^{k-1} \ell_{is} \ell_{ks} \right) / \ell_{kk}$
  
  end do

  end do

- **Output**: $(\ell_{ij})$
Richardson Method

As an illustration of these concepts, we consider the Richardson method, in which
Q is chosen to be the identity matrix. Equation (3) in this case reads as follows:

\[ x^{(k)} = (I - A)x^{(k-1)} + b = x^{(k-1)} + r^{(k-1)} \]  \hspace{1cm} (12)

where \( r^{(k-1)} \) is the residual vector, defined by \( r^{(k-1)} = b - Ax^{(k-1)} \). According to
Theorem 1, the Richardson iteration will produce a solution to \( Ax = b \) (in the limit)
if \( \| I - A \| < 1 \) for some subordinate matrix norm. (See Problems 2 and 3 for two
classes of matrices having the required property.)

An algorithm to carry out the Richardson iteration is as follows:

**input** \( n, (a_{ij}), (b_i), (x_i), M \)

for \( k = 1 \) to \( M \) do

for \( i = 1 \) to \( n \) do

\[ r_i \leftarrow b_i - \sum_{j=1}^{n} a_{ij}x_j \]

end do

for \( i = 1 \) to \( n \) do

\[ x_i \leftarrow x_i + r_i \]

end do

end do

**output** \( k, (x_i), (r_i) \)
Compute 100 iterates on the following problem, using the Richardson method starting with $x = (0, 0, 0)^T$:

\[
\begin{bmatrix}
1 & \frac{1}{2} & \frac{1}{3} \\
\frac{1}{3} & 1 & \frac{1}{2} \\
\frac{1}{2} & \frac{1}{3} & 1
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix} =
\begin{bmatrix}
\frac{11}{18} \\
\frac{11}{18} \\
\frac{1}{18}
\end{bmatrix}
\]

**Solution** A computer program based on the above algorithm was written. A few of the iterates produced by this routine are shown here:

\[
x^{(0)} = (0.00000, 0.00000, 0.00000)^T
\]

\[
x^{(1)} = (0.61111, 0.61111, 0.61111)^T
\]

\[
\vdots
\]

\[
x^{(10)} = (0.27950, 0.27950, 0.27950)^T
\]

\[
\vdots
\]

\[
x^{(40)} = (0.33311, 0.33311, 0.33311)^T
\]

\[
\vdots
\]

\[
x^{(80)} = (0.33333, 0.33333, 0.33333)^T
\]
**Jacobi Method**

Another illustration of our basic theory is provided by the Jacobi iteration, in which \( Q \) is the diagonal matrix whose diagonal entries are the same as those in the matrix \( A = (a_{ij}) \). In this case, the generic element of \( Q^{-1}A \) is \( a_{ij}/a_{ii} \). The diagonal elements of this matrix are all 1, and hence,

\[
\| I - Q^{-1}A \| = \max_{1 \leq i \leq n} \sum_{\substack{j=1 \atop j \neq i}}^{n} |a_{ij}/a_{ii}|
\]  

(13)

**Theorem 2** If \( A \) is diagonally dominant, then the sequence produced by the Jacobi iteration converges to the solution of \( Ax = b \) for any starting vector.

**Proof** Diagonal dominance means that

\[
|a_{ii}| > \sum_{\substack{j=1 \atop j \neq i}}^{n} |a_{ij}| \quad (1 \leq i \leq n)
\]

From Equation (13), we then conclude that

\[
\| I - Q^{-1}A \| < 1
\]

By Theorem 1, the Jacobi iteration converges. 

An algorithm for the Jacobi method follows:
Q13. To implement the Jacobi method.

An algorithm for the Jacobi method follows:

input \( n, (a_{ij}), (b_i), (x_i), M \)
for \( k = 1 \text{ to } M \) do

\[
\begin{align*}
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad u_i &\leftarrow \left( b_i - \sum_{j=1}^{n} a_{ij}x_j \right) / a_{ii} \\
\text{end do} \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad x_i &\leftarrow u_i \\
\text{end do} \\
\text{output } k, (x_i) \\
\text{end do}
\end{align*}
\]

This algorithm and others in this section can be made more efficient by performing all divisions before the iteration begins. Thus, we could start the computation with these operations:

\[
\begin{align*}
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad d &\leftarrow 1 / a_{ii} \\
\quad b_i &\leftarrow db_i \\
\text{for } j = 1 \text{ to } n \text{ do} \\
\quad a_{ij} &\leftarrow da_{ij} \\
\text{end do} \\
\text{end do}
\end{align*}
\]
Gauss-Seidel Method

Let us examine Gauss-Seidel iteration in more detail. It is defined by letting $Q$ be the lower triangular part of $A$, including the diagonal.

**THEOREM 6**  If $A$ is diagonally dominant, then the Gauss-Seidel method converges for any starting vector.

An algorithm for the **Gauss-Seidel iteration** follows:

```
input $n$, $(a_{ij})$, $(b_i)$, $(x_i)$, $M$
for $k = 1$ to $M$ do
  for $i = 1$ to $n$ do
    $x_i \leftarrow \left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}x_j \right) / a_{ii}$
  end do
output $k$, $(x_i)$
end do
```
Example 3  Consider the system

\[
\begin{bmatrix}
2 & -1 & 0 \\
1 & 6 & -2 \\
4 & -3 & 8
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix} =
\begin{bmatrix}
2 \\
-4 \\
5
\end{bmatrix}
\]

Apply Gauss-Seidel iteration starting with \( x^{(0)} = (0, 0, 0)^T \).

Solution  After the scaling referred to previously, \( D^{-1}Ax = D^{-1}b \) where \( D = \text{diag}(A) \), the system becomes

\[
\begin{bmatrix}
1 & -\frac{1}{2} & 0 \\
\frac{1}{6} & 1 & -\frac{1}{3} \\
\frac{1}{2} & -\frac{3}{8} & 1
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix} =
\begin{bmatrix}
1 \\
-\frac{2}{3} \\
\frac{5}{8}
\end{bmatrix}
\]

We refer to this system as \( Ax = b \). In the Gauss-Seidel method, \( Q \) is taken to be the lower triangular part of \( A \), including the diagonal. The formula defining the iteration is

\[
Qx^{(k)} = (Q - A)x^{(k-1)} + b
\]

or

\[
\begin{bmatrix}
1 & 0 & 0 \\
\frac{1}{6} & 1 & 0 \\
\frac{1}{2} & -\frac{3}{8} & 1
\end{bmatrix}
\begin{bmatrix}
x_1^{(k)} \\
x_2^{(k)} \\
x_3^{(k)}
\end{bmatrix} =
\begin{bmatrix}
0 & \frac{1}{2} & 0 \\
0 & 0 & \frac{1}{3} \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x_1^{(k-1)} \\
x_2^{(k-1)} \\
x_3^{(k-1)}
\end{bmatrix} +
\begin{bmatrix}
1 \\
-\frac{2}{3} \\
\frac{5}{8}
\end{bmatrix}
\]

From this we obtain \( x^{(k)} \) by solving a lower triangular system. The pertinent formulas in this example are

\[
x_1^{(k)} = \frac{1}{2}x_2^{(k-1)} + 1
\]

\[
x_2^{(k)} = -\frac{1}{6}x_1^{(k)} + \frac{1}{3}x_3^{(k-1)} - \frac{2}{3}
\]

\[
x_3^{(k)} = -\frac{1}{2}x_1^{(k)} + \frac{3}{8}x_2^{(k)} + \frac{5}{8}
\]
The computations yield the following iterates, of which $x^{(13)}$ is correct:

$$x^{(1)} = (1.000000, -0.833333, -0.187500)^T$$

$$x^{(5)} = (0.622836, -0.760042, 0.028566)^T$$

$$x^{(10)} = (0.620001, -0.760003, 0.029998)^T$$

$$x^{(13)} = (0.620000, -0.760000, 0.030000)^T$$
CHAPTER FIVE
Selected Topics in Numerical Linear Algebra

5.1 Matrix Eigenvalue Problem: Power Method
5.2 Schur's and Gershgorin's Theorems
*5.3 Orthogonal Factorizations and Least-Squares Problems
*5.4 Singular-Value Decomposition and Pseudoinverses
*5.5 The QR-Algorithm of Francis for the Eigenvalue Problem
Power Method

The next numerical method that we shall discuss is the **power method**. This procedure is designed to compute the dominant eigenvalue and an eigenvector corresponding to the dominant eigenvalue. For the theory to proceed smoothly, it is necessary to assume that $A$ has the following two properties:

(i) There is a single eigenvalue of maximum modulus.
(ii) There is a linearly independent set of $n$ eigenvectors.

According to the first assumption, the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ can be labeled so that

$$ |\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n| $$

According to the second assumption, there is a basis $\{u^{(1)}, u^{(2)}, \ldots, u^{(n)}\}$ for $\mathbb{C}^n$ such that

$$ Au^{(j)} = \lambda_j u^{(j)} \quad (1 \leq j \leq n) $$

(1)
Let $x^{(0)}$ be any element of $\mathbb{C}^n$ such that when $x^{(0)}$ is expressed as a linear combination of the basis elements $u^{(1)}, u^{(2)}, \ldots, u^{(n)}$, the coefficient of $u^{(1)}$ is not 0. Thus,

$$x^{(0)} = a_1 u^{(1)} + a_2 u^{(2)} + \cdots + a_n u^{(n)} \quad (a_1 \neq 0) \quad (2)$$

We form then

$$x^{(1)} = A x^{(0)}, \quad x^{(2)} = A x^{(1)}, \quad \cdots \quad x^{(k)} = A x^{(k-1)}$$

so that

$$x^{(k)} = A^k x^{(0)} \quad (3)$$

In the analysis to follow, there is no loss of generality in absorbing all the coefficients $a_j$ in the vectors $u^{(j)}$ that they multiply. Hence, we may write Equation (2) as

$$x^{(0)} = u^{(1)} + u^{(2)} + \cdots + u^{(n)}$$

By this equation and (3), we have

$$x^{(k)} = A^k u^{(1)} + A^k u^{(2)} + \cdots + A^k u^{(n)}$$

Using Equation (1), we arrive at

$$x^{(k)} = \lambda_1^k u^{(1)} + \lambda_2^k u^{(2)} + \cdots + \lambda_n^k u^{(n)}$$
This last equation can be rewritten in the form

\[ x^{(k)} = \lambda_1^k \left[ u^{(1)} + \left( \frac{\lambda_2}{\lambda_1} \right)^k u^{(2)} + \cdots + \left( \frac{\lambda_n}{\lambda_1} \right)^k u^{(n)} \right] \]

Since \(|\lambda_1| > |\lambda_j|\) for \(2 \leq j \leq n\), we see that the coefficients \(\left( \frac{\lambda_j}{\lambda_1} \right)^k\) tend to 0 and the vector within the brackets converges to \(u^{(1)}\) as \(k \to \infty\).

To simplify the notation, we write \(x^{(k)}\) in the form

\[ x^{(k)} = \lambda_1^k \left[ u^{(1)} + \varepsilon^{(k)} \right] \]

where \(\varepsilon^{(k)} \to 0\) as \(k \to \infty\). In order to be able to take ratios, let \(\varphi\) be any linear functional on \(\mathbb{C}^n\) for which \(\varphi(u^{(1)}) \neq 0\). Recall that a linear functional \(\varphi\) satisfies \(\varphi(\alpha x + \beta y) = \alpha \varphi(x) + \beta \varphi(y)\), for scalars \(\alpha\) and \(\beta\) and vectors \(x\) and \(y\). (For example, \(\varphi\) could simply evaluate the \(j\)th component of any given vector.) Then

\[ \varphi(x^{(k)}) = \lambda_1^k [\varphi(u^{(1)}) + \varphi(\varepsilon^{(k)})] \] (4)
Consequently, the following ratios converge to \( \lambda_1 \) as \( k \to \infty \):

\[
r_k = \frac{\varphi(x^{(k+1)})}{\varphi(x^{(k)})} = \lambda_1 \left[ \frac{\varphi(u^{(1)}) + \varphi(e^{(k+1)})}{\varphi(u^{(1)}) + \varphi(e^{(k)})} \right] \to \lambda_1
\]

This constitutes the **power method** for computing \( \lambda_1 \). Since the direction of the vector \( x^{(k)} \) aligns more and more with \( u^{(1)} \) as \( k \to \infty \), the method can also give us the eigenvector \( u^{(1)} \). Many variations and refinements of the power method are found in the literature.

**Algorithm**

The algorithm for the power method as just described is as follows:

```plaintext
input n, A, x, M
output 0, x
for k = 1 to M do
    y ← Ax
    r ← \( \varphi(y)/\varphi(x) \)
    x ← y
    output k, x, r
end do
```

Here \( \varphi \) is some linear functional.
In a practical realization of the power method, it is advisable to introduce a normalization of the vectors $x^{(k)}$, since otherwise they may converge to 0 or become unbounded. Thus, we can modify the iteration as follows:

```
input n, A, x, M
output 0, x
for k = 1 to M do
    y ← A x
    r ← \varphi(y)/\varphi(x)
    x ← y/\|y\|
    output k, x, r
end do
```

The norm used here can be any convenient one: the $\ell_\infty$-norm $\|x\|_\infty = \max_{1 \leq j \leq n} |x_j|$, for example. The ratios $r$ are the same as in the unnormalized version of the algorithm. (Problem 2 asks for a proof of this assertion.)
Use the power method on a matrix and an initial vector as follows:

\[ A = \begin{bmatrix} 6 & 5 & -5 \\ 2 & 6 & -2 \\ 2 & 5 & -1 \end{bmatrix} \quad x = (-1, 1, 1)^T \]

**Solution**

The normalized version of the algorithm was programmed, and the linear functional \( \varphi \) was taken to be \( \varphi(x) = x_2 \). The normalized vectors \( x^{(k)} \) and the ratios \( r_k \) are shown here for a few values of \( k \):

\[
\begin{align*}
  k = 0 & \quad x^{(0)} = (-1.00000, 1.00000, 1.00000) \\
  k = 1 & \quad x^{(1)} = (-1.00000, 0.33333, 0.33333) \quad r_0 = 2.0 \\
  k = 2 & \quad x^{(2)} = (-1.00000, -0.11111, -0.11111) \quad r_1 = -2.0 \\
  k = 3 & \quad x^{(3)} = (-1.00000, -0.40741, -0.40741) \quad r_2 = 22.0 \\
  k = 4 & \quad x^{(4)} = (-1.00000, -0.60494, -0.60494) \quad r_3 = 8.9091 \\
  \vdots & \quad \vdots \\
  k = 6 & \quad x^{(6)} = (-1.00000, -0.82442, -0.82442) \quad r_5 = 6.71508 \\
  \vdots & \quad \vdots \\
  k = 28 & \quad x^{(28)} = (-1.00000, -0.99998, -0.99998) \quad r_{27} = 6.00007
\end{align*}
\]

The leading eigenvalue of \( A \) is 6, and an eigenvector is \((1, 1, 1)^T\).\[\Box\]
Gram-Schmidt Process

The classic Gram-Schmidt process can be used to obtain orthonormal systems in any inner-product space. To describe it, we suppose that a linearly independent sequence of vectors is given in an inner-product space: \([x_1, x_2, \ldots]\). (This sequence can be finite or infinite.) We generate an orthonormal sequence \([u_1, u_2, \ldots]\) by the formula

\[
    u_k = \left( \left\| x_k - \sum_{i<k} (x_k, u_i)u_i \right\|_2 \right)^{-1} \left( x_k - \sum_{i<k} (x_k, u_i)u_i \right) \quad (k \geq 1) \tag{2}
\]

**Theorem 1**

*The Gram-Schmidt sequence \([u_1, u_2, \ldots]\) has the property that \([u_1, u_2, \ldots, u_k]\) is an orthonormal base for the linear span of \([x_1, x_2, \ldots, x_k]\) for \(k \geq 1\).*

**Proof**

We proceed by induction on \(k\). For \(k = 1\), we have from Equation (2) \(u_1 = (\|x_1\|_2)^{-1}x_1\). Hence, the set \([u_1]\) is orthonormal, and its linear span is identical with the linear span of \([x_1]\). Notice that \(\|x_1\|_2 \neq 0\) because the set \([x_1, x_2, \ldots]\) is linearly independent.

Suppose, as an induction hypothesis, that \([u_1, u_2, \ldots, u_{k-1}]\) is an orthonormal base for \(\text{span}\{x_1, x_2, \ldots, x_{k-1}\}\). Let

\[
    v = x_k - \sum_{i<k} (x_k, u_i)u_i \tag{3}
\]

Then \(v\) is orthogonal to \(u_1, u_2, \ldots, u_{k-1}\) because for \(j < k\),

\[
    \langle v, u_j \rangle = \langle x_k, u_j \rangle - \sum_{i<k} \langle x_k, u_i \rangle \langle u_i, u_j \rangle \]

\[
    = \langle x_k, u_j \rangle - \sum_{i<k} \langle x_k, u_i \rangle \delta_{ij} = \langle x_k, u_j \rangle - \langle x_k, u_j \rangle = 0
\]

If \(v = 0\), we see from Equation (3) that \(x_k \in \text{span}\{u_1, u_2, \ldots, u_{k-1}\}\). Using the induction hypothesis, we conclude that \(x_k \in \text{span}\{x_1, x_2, \ldots, x_{k-1}\}\), contradicting the assumed linear independence of \([x_1, x_2, \ldots, x_k]\). Consequently, \(v \neq 0\), and \(u_k\) is well defined by the expression \((\|v\|_2)^{-1}v\). Since \(u_k\) is of norm one, the set \([u_1, u_2, \ldots, u_k]\) is orthonormal. The induction hypothesis together with Equation (3) shows that \(v\) (and \(u_k\)) is in \(\text{span}\{x_1, x_2, \ldots, x_k\}\). Therefore, we see that \(\text{span}\{u_1, u_2, \ldots, u_k\} \subseteq \text{span}\{x_1, x_2, \ldots, x_k\}\). Since both \([u_1, u_2, \ldots, u_k]\) and \([x_1, x_2, \ldots, x_k]\) are orthonormal, it follows that \(\text{span}\{u_1, u_2, \ldots, u_k\} = \text{span}\{x_1, x_2, \ldots, x_k\}\).
Q16. To implement Gram-Schmidt algorithm.

\[
\begin{align*}
&\text{for } j = 1 \text{ to } n \text{ do} \\
&\quad \text{for } i = 1 \text{ to } j - 1 \text{ do} \\
&\quad\quad t_{ij} \leftarrow \langle A_j, B_i \rangle \\
&\quad \text{end do} \\
&\quad C_j \leftarrow A_j - \sum_{i<j} t_{ij} B_i \\
&\quad t_{jj} \leftarrow \|C_j\|_2 \\
&\quad B_j \leftarrow t_{jj}^{-1} C_j \\
&\text{end do}
\end{align*}
\]
Q17. To implement Modified Gram-Schmidt algorithm.

Modified Gram-Schmidt Algorithm

Experience has shown (Rice [1966]) that a certain rearrangement of the Gram-Schmidt process generally has superior numerical properties. The modified Gram-Schmidt algorithm goes as follows:

for $k = 1$ to $n$ do
  $A_k \leftarrow \left( \|A_k\|_2 \right)^{-1} A_k$
  for $j = k + 1$ to $n$ do
    $A_j \leftarrow A_j - \langle A_j, A_k \rangle A_k$
  end do
end do

To avoid the square roots involved in computing $\|x\|_2$, the modified Gram-Schmidt algorithm is often given in the following form, which yields a slightly different factorization:

for $k = 1$ to $n$ do
  $d_k \leftarrow \|A_k\|_2^2$
  $t_{kk} \leftarrow 1$
  for $j = k + 1$ to $n$ do
    $t_{kj} \leftarrow d_k^{-1} \langle A_j, A_k \rangle$
    $A_j \leftarrow A_j - t_{kj} A_k$
  end do
end do

Program both the Gram-Schmidt algorithm and the modified Gram-Schmidt algorithm. Then test the two to see which is better. The first test could involve a $20 \times 10$ matrix whose elements are random numbers uniformly distributed in the interval $[0,1]$. The second test could involve a $20 \times 10$ matrix whose elements are generated by some elementary function such as

$$a_{ij} = \left( \frac{2i - 21}{19} \right)^{j-1} \quad (1 \leq i \leq 20, \ 1 \leq j \leq 10)$$

In each case, generate from $A$ a matrix $B$ whose columns should be orthonormal.
The QR-factorization of a matrix and to observe the process in QR-algorithm.

The QR-algorithm of Francis [1961] is an iterative procedure designed to reveal the eigenvalues of $A$ by producing $T$ in Equation (1). As may be inferred from its name, the algorithm uses the QR-factorization. Here, all matrices will be $n \times n$.

In Section 5.3, an algorithm was developed for producing a factorization

$$A = QR$$

(2)

where $Q$ is unitary and $R$ is upper triangular. Here, we need a slight refinement of this. Namely, we wish $R$ to have a nonnegative diagonal. This is easily arranged. In fact, if Equation (2) is given and $R$ does not have a nonnegative diagonal, we can define a certain diagonal unitary matrix $D$ and replace Equation (2) by

$$A = (QD)(D^*R) = \hat{Q}\hat{R}$$

(3)

The definition of $D = \text{diag}(d_{ii})$ should be $d_{ii} = r_{ii}/|r_{ii}|$ if $r_{ii} \neq 0$ and $d_{ii} = 1$ if $r_{ii} = 0$. It is easily verified that $D$ is unitary and that the matrix $\hat{R} = D^*R$ has a nonnegative diagonal.

Q18. Given a matrix and to observe the process in QR-algorithm.

The QR-algorithm, in its basic form, is as follows:

$$A_1 \leftarrow A$$

for $k = 1$ to $M$ do

QR-factorization: $A_k = Q_k R_k$, where $Q_k$ is unitary and $R_k$ is upper triangular with nonnegative diagonal

$$A_{k+1} \leftarrow R_k Q_k$$

end do
CHAPTER SEVEN
Numerical Differentiation and Integration

7.1  Numerical Differentiation and Richardson Extrapolation
7.2  Numerical Integration Based on Interpolation
7.3  Gaussian Quadrature
7.4  Romberg Integration
7.5  Adaptive Quadrature
*7.6  Sard’s Theory of Approximating Functionals
*7.7  Bernoulli Polynomials and the Euler-Maclaurin Formula
Numerical Differentiation

We illustrate these matters by examining a formula for numerical differentiation that emerges directly from the limit definition of \( f'(x) \):

\[
f'(x) \approx \frac{1}{h} \left[ f(x + h) - f(x) \right]
\]  

(1)

For a linear function, \( f(x) = ax + b \), the approximate Formula (1) is exact; that is, it yields the correct value of \( f'(x) \) for any nonzero value of \( h \). The formula may be exact in other cases too, but then only fortuitously. Let us therefore attempt to assess the error involved in this formula for numerical differentiation. The starting point is Taylor’s Theorem in this form:

\[
f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f''(\xi)
\]  

(2)

Here \( \xi \) is a point in the open interval between \( x \) and \( x + h \). For the validity of Equation (2), \( f \) and \( f' \) should be continuous on the closed interval between \( x \) and \( x + h \), and \( f'' \) should exist on the corresponding open interval. A rearrangement of Equation (2) yields

\[
f'(x) = \frac{1}{h} \left[ f(x + h) - f(x) \right] - \frac{h}{2} f''(\xi)
\]  

(3)

Equation (3) is more useful than Equation (1) because now on a large class of functions as described above, an error term is available along with the basic numerical formula. Notice that the error term in Equation (3) has two parts: a power of \( h \) and a factor involving some higher-order derivative of \( f \). The latter gives us an indication of the class of functions to which the error estimate is applicable. The \( h \)-term in the error makes the entire expression converge to 0 as \( h \) approaches 0. The rapidity of this convergence will depend on the power of \( h \). These remarks apply to many error estimates in numerical analysis: There will usually be a power of \( h \) and a factor telling us to what smoothness class the function must belong so that the estimate is valid.
Q19. Given $f(x) = \tan^{-1} x$ and to compute the $f'(x)$ at $x = \sqrt{2}$ by the way as following.

A glance at Equation (3) shows that in order to compute $f'(x)$ accurately, the step size $h$ must be small. Therefore, let us perform an experiment in which $h$ converges to 0 through a sequence of values, and the corresponding approximations to $f'(x)$ are computed. We select $f(x) = \tan^{-1} x$ and use the point $x = \sqrt{2}$. The result should be $f'(x) = (x^2 + 1)^{-1}$ at $\sqrt{2}$, which is $1/3$. Here is an algorithm for this task:

$$f(x) := \tan^{-1} x$$

input $s \leftarrow \sqrt{2}$; $h \leftarrow 1$; $M \leftarrow 26$

$F_1 \leftarrow f(a)$

for $k = 0$ to $M$ do

$F_2 \leftarrow f(s + h)$

d $\leftarrow F_2 - F_1$

r $\leftarrow d/h$

end do

output $k, h, F_2, F_1, d, r$

$h \leftarrow h/2$

Some output from a 32-bit computer is shown here:

<table>
<thead>
<tr>
<th>$k$</th>
<th>$h$</th>
<th>$F_2$</th>
<th>$F_1$</th>
<th>$d$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>$0.62 \times 10^{-1}$</td>
<td>0.97555 095</td>
<td>0.95531 660</td>
<td>0.02023 435</td>
<td>0.32374 954</td>
</tr>
<tr>
<td>12</td>
<td>$0.24 \times 10^{-3}$</td>
<td>0.95539 796</td>
<td>0.95531 660</td>
<td>0.00008 136</td>
<td>0.33325 195</td>
</tr>
<tr>
<td>20</td>
<td>$0.95 \times 10^{-6}$</td>
<td>0.95531 690</td>
<td>0.95531 660</td>
<td>0.00000 030</td>
<td>0.31250 000</td>
</tr>
<tr>
<td>24</td>
<td>$0.60 \times 10^{-7}$</td>
<td>0.95531 666</td>
<td>0.95531 660</td>
<td>0.00000 006</td>
<td>1.00000 000</td>
</tr>
<tr>
<td>26</td>
<td>$0.15 \times 10^{-7}$</td>
<td>0.95531 660</td>
<td>0.95531 660</td>
<td>0.00000 000</td>
<td>0.00000 000</td>
</tr>
</tbody>
</table>
Numerical differentiation formulas find their most important application in the numerical solution of differential equations. A common stratagem there is to replace derivatives by approximations such as the one given in Equation (1). The precision of such numerical differentiation formulas is often judged simply by the power of \( h \) present in the error term. The higher the power of \( h \) the better because \( h \) is always a small number. In this assessment, Formula (2) fares poorly, as the error is \( O(h) \). A superior formula is

\[
f'(x) \approx \frac{1}{2h} [f(x + h) - f(x - h)]
\]  

This is derived from two cases of Taylor's Theorem—namely,

\[
f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{3!} f'''(\xi_1)
\]

\[
f(x - h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{3!} f'''(\xi_2)
\]

Subtracting one of these equations from the other and rearranging, we obtain

\[
f'(x) = \frac{1}{2h} [f(x + h) - f(x - h)] - \frac{h^2}{12} [f'''(\xi_1) + f'''(\xi_2)]
\]
This is a more favorable result because of the $h^2$ term in the error. Notice, however, the presence of $f^{iii}$ in the error. This error term is applicable if $f^{iii}$ exists.

The error term in Equation (7) can be simplified if we make the slight additional assumption that the function $f^{iii}$ is continuous on $[x - h, x + h]$. Let $M$ and $m$ denote the greatest and least values of $f^{iii}$ on the interval $[x - h, x + h]$. Then $f^{iii}(\xi_1)$, $f^{iii}(\xi_2)$, and $c = \frac{1}{2} [f^{iii}(\xi_1) + f^{iii}(\xi_2)]$ all lie in the interval $[m, M]$. Since $f^{iii}$ is continuous, it assumes the value $c$ at some point $\xi$ in $[x - h, x + h]$. Hence,

$$f^{iii}(\xi) = \frac{1}{2} [f^{iii}(\xi_1) + f^{iii}(\xi_2)]$$

When this expression is substituted in Equation (7), the result is

$$f'(x) = \frac{1}{2h} \left[ f(x + h) - f(x - h) \right] - \frac{h^2}{6} f^{iii}(\xi)$$

An important formula for second derivatives is obtained by extending Equations (5) and (6) by one more term and then adding the equations. After rearrangement and applying the method used previously, we have

$$f''(x) = \frac{1}{h^2} \left[ f(x + h) - 2f(x) + f(x - h) \right] - \frac{h^2}{12} f^{(4)}(\xi)$$

for some $\xi \in (x - h, x + h)$. This formula is often used in the numerical solution of second-order differential equations.
Q20. Given $f(x) = \tan^{-1} x$ and to compute the $f'(x)$ at $x = \sqrt{2}$ by the central difference method.

Example 2

Use a computer to approximate $f'(x)$, where $f(x) = \tan^{-1} x$ and $x = \sqrt{2}$. Use Equation (8) with step size $h$ approaching 0. Recall that the correct value is $1/3$.

Solution

A suitable algorithm follows:

$$f(x) := \tan^{-1} x$$

input $s \leftarrow \sqrt{2}; \ h \leftarrow 1; \ M \leftarrow 26$

for $k = 0$ to $M$ do

$F_2 \leftarrow f(s + h)$

$F_1 \leftarrow f(s - h)$

$d \leftarrow F_2 - F_1$

$r \leftarrow d/(2h)$

output $k, h, F_2, F_1, d, r$

$h \leftarrow h/2$

end do

Some of the output from a 32-bit machine is shown here:

<table>
<thead>
<tr>
<th>$k$</th>
<th>$h$</th>
<th>$F_2$</th>
<th>$F_1$</th>
<th>$d$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.25</td>
<td>1.02972674</td>
<td>0.86112982</td>
<td>0.16859692</td>
<td>0.33719385</td>
</tr>
<tr>
<td>10</td>
<td>$0.9765 \times 10^{-3}$</td>
<td>0.95564199</td>
<td>0.95499092</td>
<td>0.00065106</td>
<td>0.33334351</td>
</tr>
<tr>
<td>18</td>
<td>$0.3815 \times 10^{-5}$</td>
<td>0.95531786</td>
<td>0.95531535</td>
<td>0.00000250</td>
<td>0.32812500</td>
</tr>
<tr>
<td>26</td>
<td>$0.1490 \times 10^{-7}$</td>
<td>0.95531660</td>
<td>0.95531660</td>
<td>0.00000000</td>
<td>0.00000000</td>
</tr>
</tbody>
</table>
Q21. \( f(x) = \tan^{-1}(x) \) and to compute the \( f'(x) \) at \( x = \sqrt{2} \) by the incorporating Richardson extrapolation.

Recompute the derivative in Example 2, incorporating Richardson extrapolation.

A suitable algorithm is as follows:

\[ f(x) := \tan^{-1}(x) \]

**input** \( s \leftarrow \sqrt{2}; \ h \leftarrow 1; \ M \leftarrow 30 \)

**for** \( k = 0 \) to \( M \) **do**

\[ d_k \leftarrow \frac{[f(s + h) - f(s - h)]}{2h} \]

\[ h \leftarrow h/2 \]

**end do**

**for** \( k = 1 \) to \( M \) **do**

\[ r_k \leftarrow d_k + (d_k - d_{k-1})/3 \]

**end do**

**output** \( [k, d_k, r_k : 0 \leq k \leq M] \)

A few lines of output are shown here.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( d_k )</th>
<th>( r_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.33719385</td>
<td>0.33333480</td>
</tr>
<tr>
<td>4</td>
<td>0.33357477</td>
<td>0.33333364</td>
</tr>
<tr>
<td>8</td>
<td>0.33332825</td>
<td>0.33332571</td>
</tr>
<tr>
<td>16</td>
<td>0.33203125</td>
<td>0.33138022</td>
</tr>
<tr>
<td>26</td>
<td>0.00000000</td>
<td>0.00000000</td>
</tr>
</tbody>
</table>
The formulas given for $D(n, 0)$ and $D(n, k)$ allow us to construct a triangular array:

\[
\begin{array}{cccc}
D(0, 0) & D(1, 0) & D(1, 1) & D(2, 2) \\
D(2, 0) & D(2, 1) & D(2, 2) & \\
 \vdots & \vdots & \vdots & \ddots \\
D(M, 0) & D(M, 1) & D(M, 2) & \cdots & D(M, M)
\end{array}
\]

An algorithm to generate this triangular array is given next. In it, we refer to a function $\varphi$ that must be made available to the code separately (that is, by a subprogram or procedure).

input $h, M$
for $n = 0$ to $M$ do
\[D(n, 0) \leftarrow \varphi(h/2^n)\]
end do
for $k = 1$ to $M$ do
for $n = k$ to $M$ do
\[D(n, k) \leftarrow D(n, k - 1) + [D(n, k - 1) - D(n - 1, k - 1)]/(4^k - 1)\]
end do
end do
output $D(n, k)$ \hspace{3mm} $(0 \leq n \leq M, 0 \leq k \leq n)$
Q22. \( f(x) = \tan^{-1} x \) and to compute the \( f'(x) \) at \( x = \sqrt{2} \) by the complete Richardson extrapolation.

Recompute the derivative in Example 5 using the complete Richardson extrapolation algorithm just described.

Some output from this algorithm using Example 2 is shown here:

<table>
<thead>
<tr>
<th>( n )</th>
<th>( D(n, 0) )</th>
<th>( D(n, 1) )</th>
<th>( D(n, 2) )</th>
<th>( D(n, 3) )</th>
<th>( D(n, 4) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.3926991</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.3487710</td>
<td>0.3341283</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.3371938</td>
<td>0.3333348</td>
<td>0.3332819</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.3342981</td>
<td>0.3333329</td>
<td>0.3333328</td>
<td>0.3333336</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.3335748</td>
<td>0.3333336</td>
<td>0.3333337</td>
<td>0.3333337</td>
<td>0.3333337</td>
</tr>
</tbody>
</table>

This algorithm obtains approximately the same precision as in Example 5. Eventually it too will produce meaningless results because of subtractive cancellation.
Romberg Algorithm

This formula is used in the Romberg algorithm. Letting $R(n, 0)$ denote the trapezoid estimate with $2^n$ subintervals, we have

\[
\begin{aligned}
R(0, 0) &= \frac{1}{2} (b - a) [f(a) + f(b)] \\
R(n, 0) &= \frac{1}{2} R(n - 1, 0) + h_n \sum_{i=1}^{2^{n-1}} f(a + (2i - 1)h_n)
\end{aligned}
\]  

(4)

The estimates $R(0, 0)$, $R(1, 0)$, $R(2, 0), \ldots$, $R(M, 0)$ are computed for a modest value of $M$, and there are no duplicate function evaluations. In the remainder of the Romberg algorithm, additional quantities $R(n, m)$ are to be computed. All of these can be interpreted as estimates of the integral $I$. Further evaluations of the integrand $f$ are not necessary after the element $R(M, 0)$ has been computed. The subsequent columns of the $R$-array for $n \geq 1$ and $m \geq 1$ are constructed from the formula

\[
R(n, m) = R(n, m - 1) + \frac{1}{4^m - 1} [R(n, m - 1) - R(n - 1, m - 1)]
\]  

(5)

This calculation is very simple. It is used to provide a final array of the form

\[
\begin{array}{ccccccc}
R(0, 0) & & & & & & \\
R(1, 0) & R(1, 1) & & & & & \\
R(2, 0) & R(2, 1) & R(2, 2) & & & & \\
R(3, 0) & R(3, 1) & R(3, 2) & R(3, 3) & & & \\
R(4, 0) & R(4, 1) & R(4, 2) & R(4, 3) & R(4, 4) & & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\
R(M, 0) & R(M, 1) & R(M, 2) & R(M, 3) & R(M, 4) & \cdots & R(M, M)
\end{array}
\]
Here is the pseudocode for the Romberg algorithm computed row-wise:

```
input a, b, M
h ← b - a
R(0, 0) ← \frac{1}{2}(b - a)[f(a) + f(b)]
for n = 1 to M do
    h ← h/2
    R(n, 0) ← \frac{1}{2}R(n - 1, 0) + h \sum_{i=1}^{2^{n-1}} f(a + (2i - 1)h)
    for m = 1 to n do
        R(n, m) ← R(n, m - 1) + [R(n, m - 1) - R(n - 1, m - 1)]/(4^m - 1)
    end do
end do
output R(n, m) (0 ≤ n ≤ M, 0 ≤ m ≤ n)
```

Usually, only a moderate value of M is needed because the number of function values required is $2^M + 1$. A more sophisticated algorithm might include a procedure for stopping the calculations automatically when a certain prescribed error criterion is met.

Q23.

Write a subprogram to carry out the Romberg algorithm for a function $f$ defined on an arbitrary interval $[a, b]$. The user will specify the number of rows to be computed in the array and will want to see the entire array when it has been computed. Write a main program and test your Romberg subprogram on these three examples:

(a) $\int_{0}^{1} \frac{\sin x}{x} \, dx$
CHAPTER EIGHT
Numerical Solution of Ordinary Differential Equations

8.1 The Existence and Uniqueness of Solutions
8.2 Taylor-Series Method
8.3 Runge-Kutta Methods
8.4 Multistep Methods
*8.5 Local and Global Errors: Stability
8.6 Systems and Higher-Order Ordinary Differential Equations
*8.7 Boundary-Value Problems
8.8 Boundary-Value Problems: Shooting Methods
8.9 Boundary-Value Problems: Finite-Difference Methods
*8.10 Boundary-Value Problems: Collocation
*8.11 Linear Differential Equations
8.12 Stiff Equations
Taylor-Series Method

In the numerical solution of differential equations, we rarely expect to obtain the solution directly as a formula giving $x(t)$ as a function of $t$. Instead, we usually construct a table of function values of the form

\[
\begin{array}{c|c|c|c|c|c}
    t_0 & t_1 & t_2 & \cdots & t_m \\
    x_0 & x_1 & x_2 & \cdots & x_m \\
\end{array}
\]  

(1)

Here, $x_i$ is the computed approximate value of $x(t_i)$, our notation for the exact solution at $t_i$. From a table such as (1), a spline function or other approximating function can be constructed. However, most numerical methods for solving ordinary differential equations produce such a table first.

We consider again the initial-value problem

\[
\begin{align*}
    x' &= f(t, x) \\
    x(t_0) &= x_0 \\
\end{align*}
\]

(2)

where $f$ is a prescribed function of two variables, and $(t_0, x_0)$ is a single given point through which the solution curve should pass. A solution of (2) is a function $x(t)$ such that $dx(t)/dt = f(t, x(t))$ for all $t$ in some neighborhood of $t_0$, and $x(t_0) = x_0$.

Illustration

For the Taylor-series method, it is necessary to assume that various partial derivatives of $f$ exist. To illustrate the method, we take a concrete example:

\[
\begin{align*}
    x' &= \cos t - \sin x + t^2 \\
    x(-1) &= 3
\end{align*}
\]

(3)
At the heart of the procedure is the Taylor series for $x$, which we write as

\[ x(t + h) = x(t) + hx'(t) + \frac{h^2}{2!} x''(t) + \frac{h^3}{3!} x'''(t) + \frac{h^4}{4!} x^{(4)}(t) + \cdots \]  

(4)

The derivatives appearing here can be obtained from the differential equation in (3). They are

\[ x'' = -\sin t - x' \cos x + 2t \]
\[ x''' = -\cos t - x'' \cos x + (x')^2 \sin x + 2 \]
\[ x^{(4)} = \sin t - x''' \cos x + 3x' x'' \sin x + (x')^3 \cos x \]

At this point, our patience wears thin and we decide to use only terms up to and including $h^4$ in Formula (4). The terms that we have not included start with a term in $h^5$, and they constitute collectively the truncation error inherent in our procedure. The resulting numerical method is said to be of order 4. (The order of the Taylor-series method is $n$ if terms up to and including $h^n x^{(n)}(t)/n!$ are used.) Notice that in differentiating terms such as $\sin x$ with respect to $t$, we must think of it as $d[\sin(x(t))]/dt$ and employ the chain rule of differentiation. This, of course, accounts for the complexity of the formulas for $x'', x''', \ldots$. We could perform various substitutions to obtain formulas for $x'', x''', \ldots$ containing no derivatives of $x$ on the right-hand side. It is not necessary to do this if the formulas are used in the order listed. They are recursive in nature.

Here is an algorithm to solve the initial-value problem (3), starting at $t = -1$ and progressing in steps of $h = 0.01$. We desire a solution in the $t$-interval $[-1, 1]$, and thus we must take 200 steps.
input \( M \leftarrow 200; \ h \leftarrow 0.01; \ t \leftarrow -1.0; \ x \leftarrow 3.0 \)
output \( 0, t, x \)
for \( k = 1 \) to \( M \) do
\[
\begin{align*}
x' & \leftarrow \cos t - \sin x + t^2 \\
x'' & \leftarrow -\sin t - x' \cos x + 2t \\
x''' & \leftarrow -\cos t - x'' \cos x + (x')^2 \sin x + 2 \\
x^{(4)} & \leftarrow \sin t + ((x')^3 - x''') \cos x + 3x'x'' \sin x \\
x & \leftarrow x + h(x' + \frac{h}{2}(x'' + \frac{h}{3}(x''' + \frac{h}{4}(x^{(4)})))) \\
t & \leftarrow t + h
\end{align*}
\]
output \( k, t, x \)
end do

When the preceding algorithm was programmed and run, the solution at \( t = 1 \) was \( x_{200} = 6.42194 \). Here is a sample of the output from that computer program:

<table>
<thead>
<tr>
<th>( k )</th>
<th>( t )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1.00000</td>
<td>3.00000</td>
</tr>
<tr>
<td>1</td>
<td>-0.99000</td>
<td>3.01400</td>
</tr>
<tr>
<td>2</td>
<td>-0.98000</td>
<td>3.02803</td>
</tr>
<tr>
<td>3</td>
<td>-0.97000</td>
<td>3.04209</td>
</tr>
<tr>
<td>4</td>
<td>-0.96000</td>
<td>3.05617</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>196</td>
<td>0.96000</td>
<td>6.36566</td>
</tr>
<tr>
<td>197</td>
<td>0.97000</td>
<td>6.37977</td>
</tr>
<tr>
<td>198</td>
<td>0.98000</td>
<td>6.39386</td>
</tr>
<tr>
<td>199</td>
<td>0.99000</td>
<td>6.40791</td>
</tr>
<tr>
<td>200</td>
<td>1.00000</td>
<td>6.42194</td>
</tr>
</tbody>
</table>
Runge-Kutta Methods

The Taylor-series method of the preceding section has the drawback of requiring some analysis prior to programming it. Thus, if we wish to use the fourth-order Taylor-series method on the general problem

\[
\begin{align*}
  x' &= f(t, x) \\
  x(t_0) &= x_0
\end{align*}
\]

we have to determine formulas for \(x''\), \(x'''\), and \(x^{(4)}\) by successive differentiation in (1). Then these functions will have to be programmed.

The Runge-Kutta methods avoid this difficulty, although they do imitate the Taylor-series method by means of clever combinations of values of \(f(t, x)\). We illustrate by deriving a second-order Runge-Kutta procedure.

Fourth-Order Runge-Kutta Method

The higher-order Runge-Kutta formulas are very tedious to derive, and we shall not do so. The formulas are rather elegant, however, and are easily programmed once they have been derived. Here are the formulas for the classical fourth-order Runge-Kutta method:

\[
x(t + h) = x(t) + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4)
\]

where

\[
\begin{align*}
  F_1 &= hf(t, x) \\
  F_2 &= hf(t + \frac{1}{2}h, x + \frac{1}{2}F_1) \\
  F_3 &= hf(t + \frac{1}{2}h, x + \frac{1}{2}F_2) \\
  F_4 &= hf(t + h, x + F_3)
\end{align*}
\]

This is called a fourth-order method because it reproduces the terms in the Taylor series up to and including the one involving \(h^4\). The error is therefore \(O(h^5)\). Exact expressions for the \(h^5\) error term are available.
Example 1. Give an algorithm incorporating the Runge-Kutta method of order 4 for solving the following initial-value problem:

\[
\begin{align*}
  x' &= t^{-2}(tx - x^2) \\
  x(1) &= 2
\end{align*}
\] (9)

on the interval [1, 3], using steps of \( h = 1/128 \).

Solution. Since this is a numerical experiment to gauge the effectiveness of the procedure, a problem has been chosen with a known analytic solution. The solution of (9) is given by \( x(t) = (\frac{1}{2} + \ln t)^{-1}t \). The values of the error are printed by the computer program.

\[
\begin{align*}
  \text{input } & M \leftarrow 256; \quad t \leftarrow 1.0; \quad x \leftarrow 2.0; \quad h \leftarrow 0.0078125 \\
  \text{define } & f(t, x) = (tx - x^2)/t^2 \\
  \text{define } & u(t) = t/(\frac{1}{2} + \ln t) \\
  e & \leftarrow |u(t) - x|
\end{align*}
\]

\[
\text{output } 0, t, x, e
\]

\[
\text{for } k = 1 \text{ to } M \text{ do}
\]

\[
\begin{align*}
  F_1 & \leftarrow hf(t, x) \\
  F_2 & \leftarrow hf(t + \frac{1}{2}h, x + \frac{1}{2}F_1) \\
  F_3 & \leftarrow hf(t + \frac{1}{2}h, x + \frac{1}{2}F_2) \\
  F_4 & \leftarrow hf(t + h, x + F_3) \\
  x & \leftarrow x + (F_1 + 2F_2 + 2F_3 + F_4)/6 \\
  t & \leftarrow t + h \\
  e & \leftarrow |u(t) - x|
\end{align*}
\]

\[
\text{output } k, t, x, e
\]

end do
Some of the output from a computer program based on this algorithm follows:

<table>
<thead>
<tr>
<th>$k$</th>
<th>$t$</th>
<th>$x$</th>
<th>$e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.00000</td>
<td>2.00000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.00781</td>
<td>1.98473</td>
<td>$1.19 \times 10^{-07}$</td>
</tr>
<tr>
<td>2</td>
<td>1.01563</td>
<td>1.97016</td>
<td>0.</td>
</tr>
<tr>
<td>3</td>
<td>1.02344</td>
<td>1.95623</td>
<td>0.</td>
</tr>
<tr>
<td>4</td>
<td>1.03125</td>
<td>1.94293</td>
<td>$1.19 \times 10^{-07}$</td>
</tr>
<tr>
<td>5</td>
<td>1.03906</td>
<td>1.93020</td>
<td>$1.19 \times 10^{-07}$</td>
</tr>
<tr>
<td>6</td>
<td>1.04688</td>
<td>1.91802</td>
<td>0.</td>
</tr>
<tr>
<td>7</td>
<td>1.05469</td>
<td>1.90637</td>
<td>$1.19 \times 10^{-07}$</td>
</tr>
<tr>
<td>8</td>
<td>1.06250</td>
<td>1.89521</td>
<td>$1.19 \times 10^{-07}$</td>
</tr>
<tr>
<td>9</td>
<td>1.07031</td>
<td>1.88452</td>
<td>0.</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>249</td>
<td>2.94531</td>
<td>1.86387</td>
<td>$7.15 \times 10^{-07}$</td>
</tr>
<tr>
<td>250</td>
<td>2.95313</td>
<td>1.86569</td>
<td>$5.96 \times 10^{-07}$</td>
</tr>
<tr>
<td>251</td>
<td>2.96094</td>
<td>1.86750</td>
<td>$7.15 \times 10^{-07}$</td>
</tr>
<tr>
<td>252</td>
<td>2.96875</td>
<td>1.86932</td>
<td>$5.96 \times 10^{-07}$</td>
</tr>
<tr>
<td>253</td>
<td>2.97656</td>
<td>1.87115</td>
<td>$4.77 \times 10^{-07}$</td>
</tr>
<tr>
<td>254</td>
<td>2.98438</td>
<td>1.87297</td>
<td>$5.96 \times 10^{-07}$</td>
</tr>
<tr>
<td>255</td>
<td>2.99219</td>
<td>1.87480</td>
<td>$5.96 \times 10^{-07}$</td>
</tr>
<tr>
<td>256</td>
<td>3.00000</td>
<td>1.87663</td>
<td>$5.96 \times 10^{-07}$</td>
</tr>
</tbody>
</table>
Boundary-Value Problems: Shooting Methods

The two-point boundary-value problem being considered is

\[\begin{align*}
  x'' &= f(t, x, x') \\
  x(a) &= \alpha \quad x(b) = \beta
\end{align*}\]
Q26. 呈現動態系統中的離散混沌動態系統之數學模型

Q27. 呈現動態系統中的離散混沌動態系統之數學模型

Q28. 呈現動態系統中的連續混沌動態系統之數學模型
   Ref.s:
   ● http://en.wikipedia.org/wiki/Lorenz_attractor

Q29. 呈現動態系統中的連續混沌動態系統之數學模型
   Ref.s:
   ● http://en.wikipedia.org/wiki/Lorenz_attractor

Q30. 呈現動態系統中的碎形之數學模型
   http://en.wikipedia.org/wiki/Fractal

Q31. 呈現動態系統中的碎形之數學模型
   http://en.wikipedia.org/wiki/Fractal