Chapter 1
Error Analysis

Several sources of errors are important for numerical data processing:

Experimental uncertainty: *Input data from an experiment have a limited precision. Instead of the vector of exact values* $\mathbf{x}$ the calculation uses $\mathbf{x}+\Delta\mathbf{x}$, with an uncertainty $\Delta\mathbf{x}$. *This can lead to large uncertainties of the calculated results if an unstable algorithm is used or if the unavoidable error inherent to the problem is large.*

Rounding errors: *The arithmetic unit of a computer uses only a subset of the real numbers, the so called machine numbers* $A \subset \Re$. *The input data as well as the results of elementary operations have to be represented by machine numbers whereby rounding errors can be generated. This kind of numerical error can be avoided in principle by using arbitrary precision arithmetics*[1] *or symbolic algebra programs. But this is unpractical in many cases due to the increase in computing time and memory requirements.*

Truncation errors: *Results from more complex operations like square roots or trigonometric functions can have even larger errors since series expansions have to be truncated and iterations can accumulate the errors of the individual steps.*

1.1 Machine Numbers and Rounding Errors

Floating point numbers are internally stored as the product of sign, mantissa and a power of 2. According to the IEEE754 standard [1] single, double and quadruple precision numbers are stored as 32, 64 or 128 bits (Table 1.1).

The sign bit s is 0 for positive and 1 for negative numbers. The exponent b is biased by adding E which is half of its maximum possible value (Table 1.2).[2] The value of a number is given by

[1] *For instance the open source GNU MP bignum library.*

[2] In the following the usual hexadecimal notation is used which represents a group of 4 bits by one of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

P.O.J. Scherer, *Computational Physics*, Graduate Texts in Physics,
DOI 10.1007/978-3-319-61088-7_1

Table 1.1 Binary floating-point formats

Format	Sign	Exponent	Hidden bit	Fraction	Precision ε_M
Float	s	$b_0 \cdots b_7$	1	$a_0 \cdots a_{22}$	$2^{-24} = 5.96\text{E}^{-8}$
Double	s	$b_0 \cdots b_{10}$	1	$a_0 \cdots a_{51}$	$2^{-53} = 1.11\text{E}^{-16}$
Quadruple	s	$b_0 \cdots b_{14}$	1	$a_0 \cdots a_{111}$	$2^{-113} = 9.63\text{E}^{-35}$

Table 1.2 Exponent bias E

Decimal value	Binary value	Hexadecimal value	Data type
127_{10}	1111111_2	\$ 3F	Single
1023_{10}	1111111111_2	\$ 3FF	Double
16383_{10}	11111111111111_2	\$3FFF	Quadruple

Table 1.3 Special double precision numbers

Hexadecimal value	Symbolic value
\$ 000 0000000000000	+0
\$ 080 00000000000000	−0
\$ 7FF 0000000000000	+inf
\$ FFF 0000000000000	-inf
\$ 7FF 0000000000001 $\cdots$ \$ 7FF FFFFFFFFFFFF	NAN
\$ 001 0000000000000	Min_Normal
\$ 7FE FFFFFFFFFFFF	Max_Normal
\$ 000 0000000000001	Min_Subnormal
\$ 000 FFFFFFFFFFFF	Max_Subnormal

$$x = (-)^s \times a \times 2^{b-E}. \tag{1.1}$$

The mantissa a is normalized such that its first bit is 1 and its value is between 1 and 2

$$1.000_2 \cdots 0 \leq a \leq 1.111 \cdots 1_2 < 10.0_2 = 2_{10}. \tag{1.2}$$

Since the first bit of a normalized floating point number always is 1, it is not necessary to store it explicitly (hidden bit or J-bit). However, since not all numbers can be normalized, only the range of exponents from \$001 $\cdots$ \$7FE is used for normalized numbers. An exponent of \$000 signals that the number is not normalized (zero is an important example, there exist even two zero numbers with different sign) whereas the exponent \$7FF is reserved for infinite or undefined results (Table 1.3). The range of normalized double precision numbers is between

$$\text{Min\_Normal} = 2.2250738585072014 \times 10^{-308}$$

and

$$\text{Max\_Normal} = 1.7976931348623157E \times 10^{308}.$$

Example
Consider the following bit pattern which represents a double precision number:

$$\$4059000000000000.$$

The exponent is $100\,0000\,0101_2 - 011\,1111\,1111_2 = 110_2$ and the mantissa including the J-bit is $1\,1001\,0000\,0000\cdots_2$. Hence the decimal value is

$$1.5625 \times 2^6 = 100_{10}.$$

Input numbers which are not machine numbers have to be rounded to the nearest machine number. This is formally described by a mapping $\Re \to A$

$$x \to rd(x)$$

with the property[3]

$$|x - rd(x)| \leq |x - g| \text{ for all } g \in A. \tag{1.3}$$

For the special case that x is exactly in the middle between two successive machine numbers, a tie-breaking rule is necessary. The simplest rules are to round up always (*round-half-up*) or always down *(round-half-down)*. However, these are not symmetric and produce a bias in the average round-off error. The IEEE-754 standard [1] recommends the *round-to-nearest-even* method, i.e. the least significant bit of the rounded number should always be zero. Alternatives are *round-to-nearest-odd,* stochastic rounding and alternating rounding.

The cases of *exponent overflow* and *exponent underflow* need special attention:

Whenever the exponent b has the maximum possible value $b = b_{\max}$ and $a = 1.11\cdots11$ has to be rounded to $a' = 10.00\cdots0$, the rounded number is not a machine number and the result is $\pm\inf$.

Numbers in the range $2^{b_{\min}} > |x| \geq 2^{b_{\min}-t}$ can be represented with loss of accuracy by denormalized machine numbers. Their mantissa cannot be normalized since it is $a < 1$ and the exponent has the smallest possible value $b = b_{\min}$. Even smaller numbers with $|x| < 2^{-t+b_{\min}}$ have to be rounded to ± 0.

[3] Sometimes rounding is replaced by a simpler truncation operation which, however leads to significantly larger rounding errors.

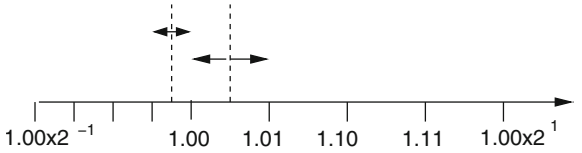

Fig. 1.1 (Round to nearest) Normalized machine numbers with $t = 3$ binary digits are shown. Rounding to the nearest machine number produces a round-off error which is bounded by half the spacing of the machine numbers

The maximum rounding error for normalized numbers with t binary digits

$$a' = s \times 2^{b-E} \times 1.a_1 a_2 \cdots a_{t-1} \tag{1.4}$$

is given by (Fig. 1.1)

$$|a - a'| \leq 2^{b-E} \times 2^{-t} \tag{1.5}$$

and the relative error is bounded by

$$\left| \frac{rd(x) - x}{x} \right| \leq \frac{2^{-t} \times 2^b}{|a| \times 2^b} \leq 2^{-t}. \tag{1.6}$$

The error bound determines the relative machine precision[4]

$$\varepsilon_M = 2^{-t} \tag{1.7}$$

and the rounding operation can be described by

$$rd(x) = x(1 + \varepsilon) \text{ with } |\varepsilon| \leq \varepsilon_M. \tag{1.8}$$

The round-off error takes its maximum value if the mantissa is close to 1. Consider a number

$$x = 1 + \varepsilon.$$

If $\varepsilon < \varepsilon_M$ then $\mathrm{rd}(x) = 1$ whereas for $\varepsilon > \varepsilon_M$ rounding gives $\mathrm{rd}(x) = 1 + 2^{1-t}$ (Fig. 1.2). Hence ε_M is given by the largest number ε for which $\mathrm{rd}(1.0 + \varepsilon) = 1.0$ and is therefore also called *unit roundoff*.

[4] Also known as machine epsilon.

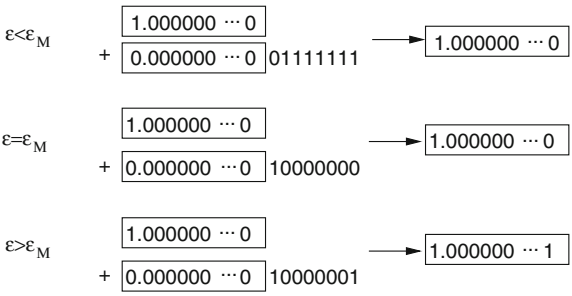

Fig. 1.2 (Unit round off)

1.2 Numerical Errors of Elementary Floating Point Operations

Even for two machine numbers $x, y \in A$ the results of addition, subtraction, multiplication or division are not necessarily machine numbers. We have to expect some additional round-off errors from all these elementary operations [2]. We assume that the results of elementary operations are approximated by machine numbers as precisely as possible. The IEEE754 standard [1] requires that the exact operations $x + y, x - y, x \times y, x \div y$ are approximated by floating point operations $A \to A$ with the property:

$$\begin{aligned} fl_+(x, y) &= rd(x + y) \\ fl_-(x, y) &= rd(x - y) \\ fl_*(x, y) &= rd(x \times y) \\ fl_\div(x, y) &= rd(x \div y). \end{aligned} \tag{1.9}$$

1.2.1 Numerical Extinction

For an addition or subtraction one summand has to be denormalized to line up the exponents (for simplicity we consider only the case $x > 0, y > 0$)

$$x + y = a_x 2^{b_x - E} + a_y 2^{b_y - E} = (a_x + a_y 2^{b_y - b_x}) 2^{b_x - E}. \tag{1.10}$$

If the two numbers differ much in their magnitude, numerical extinction can happen. Consider the following case:

$$y < 2^{b_x - E} \times 2^{-t} \tag{1.11}$$

$$a_y 2^{b_y - b_x} < 2^{-t}.$$

The mantissa of the exact sum is

$$a_x + a_y 2^{b_y - b_x} = 1.\alpha_2 \cdots \alpha_{t-1} 01 \beta_2 \cdots \beta_{t-1}. \tag{1.12}$$

Rounding to the nearest machine number gives

$$rd(x + y) = 2^{b_x} \times (1.\alpha_2 \cdots \alpha_{t-1}) = x \tag{1.13}$$

since

$$\begin{aligned} &|0.01\beta_2 \cdots \beta_{t-1} - 0| \leq |0.011 \cdots 1| = 0.1 - 0.00 \cdots 01 \\ &|0.01\beta_2 \cdots \beta_{t-1} - 1| \geq |0.01 - 1| = 0.11. \end{aligned} \tag{1.14}$$

Consider now the case

$$y < x \times 2^{-t-1} = a_x \times 2^{b_x - E - t - 1} < 2^{b_x - E - t}. \tag{1.15}$$

For normalized numbers the mantissa is in the interval

$$1 \leq |a_x| < 2 \tag{1.16}$$

hence we have

$$rd(x + y) = x \text{ if } \frac{y}{x} < 2^{-t-1} = \frac{\varepsilon_M}{2}. \tag{1.17}$$

Especially for $x = 1$ we have

$$rd(1 + y) = 1 \text{ if } y < 2^{-t} = 0.00 \cdots 0_{t-1} 1_t 000 \cdots \tag{1.18}$$

2^{-t} could be rounded to 0 or to 2^{1-t} since the distance is the same $|2^{-t} - 0| = |2^{-t} - 2^{1-t}| = 2^{-t}$.

The smallest machine number with $fl_+(1, \varepsilon) > 1$ is either $\varepsilon = 0.00 \cdots 1_t 0 \cdots = 2^{-t}$ or $\varepsilon = 0.00 \cdots 1_t 0 \cdots 01_{2t-1} = 2^{-t}(1 + 2^{1-t})$. Hence the machine precision ε_M can be determined by looking for the smallest (positive) machine number ε for which $fl_+(1, \varepsilon) > 1$.

1.2.2 Addition

Consider the sum of two floating point numbers

$$y = x_1 + x_2. \tag{1.19}$$

First the input data have to be approximated by machine numbers:

$$x_1 \to rd(x_1) = x_1(1+\varepsilon_1)$$
$$x_2 \to rd(x_2) = x_2(1+\varepsilon_2) \tag{1.20}$$

The addition of the two summands may produce another error α since the result has to be rounded. The numerical result is

$$\tilde{y} = fl_+(rd(x_1), rd(x_2)) = (x_1(1+\varepsilon_1) + x_2(1+\varepsilon_2))(1+\alpha). \tag{1.21}$$

Neglecting higher orders of the error terms we have in first order

$$\tilde{y} = x_1 + x_2 + x_1\varepsilon_1 + x_2\varepsilon_2 + (x_1 + x_2)\alpha \tag{1.22}$$

and the relative error of the numerical sum is

$$\frac{\tilde{y} - y}{y} = \frac{x_1}{x_1 + x_2}\varepsilon_1 + \frac{x_2}{x_1 + x_2}\varepsilon_2 + \alpha. \tag{1.23}$$

If $x_1 \approx -x_2$ then numerical extinction can produce large relative errors and uncertainties of the input data can be strongly enhanced.

1.2.3 Multiplication

Consider the multiplication of two floating point numbers

$$y = x_1 \times x_2. \tag{1.24}$$

The numerical result is

$$\tilde{y} = fl_*(rd(x_1), rd(x_2)) = x_1(1+\varepsilon_1)x_2(1+\varepsilon_2)(1+\mu) \approx x_1x_2(1+\varepsilon_1+\varepsilon_2+\mu) \tag{1.25}$$

with the relative error

$$\frac{\tilde{y} - y}{y} = 1 + \varepsilon_1 + \varepsilon_2 + \mu. \tag{1.26}$$

The relative errors of the input data and of the multiplication just add up to the total relative error. There is no enhancement. Similarly for a division

$$y = \frac{x_1}{x_2} \tag{1.27}$$

the relative error is

$$\frac{\tilde{y} - y}{y} = 1 + \varepsilon_1 - \varepsilon_2 + \mu. \tag{1.28}$$

1.3 Error Propagation

Consider an algorithm consisting of a sequence of elementary operations. From the set of input data which is denoted by the vector

$$\mathbf{x} = (x_1 \cdots x_n) \tag{1.29}$$

a set of output data is calculated

$$\mathbf{y} = (y_1 \cdots y_m). \tag{1.30}$$

Formally this can be denoted by a vector function

$$\mathbf{y} = \varphi(\mathbf{x}) \tag{1.31}$$

which can be written as a product of r simpler functions representing the elementary operations

$$\varphi = \varphi^{(r)} \times \varphi^{(r-1)} \cdots \varphi^{(1)}. \tag{1.32}$$

Starting with $\mathbf{x}$ intermediate results $\mathbf{x}_i = (x_{i1}, \cdots x_{in_i})$ are calculated until the output data $\mathbf{y}$ result from the last step:

$$\begin{aligned} \mathbf{x}_1 &= \varphi^{(1)}(\mathbf{x}) \\ \mathbf{x}_2 &= \varphi^{(2)}(\mathbf{x}_1) \\ &\vdots \\ \mathbf{x}_{r-1} &= \varphi^{(r-1)}(\mathbf{x}_{r-2}) \\ \mathbf{y} &= \varphi^{(r)}(\mathbf{x}_{r-1}). \end{aligned} \tag{1.33}$$

In the following we analyze the influence of numerical errors onto the final results. We treat all errors as small quantities and neglect higher orders. Due to round-off errors and possible experimental uncertainties the input data are not exactly given by $\mathbf{x}$ but by

$$\mathbf{x} + \Delta\mathbf{x}. \tag{1.34}$$

The first step of the algorithm produces the result

$$\tilde{\mathbf{x}_1} = rd(\varphi^{(1)}(\mathbf{x} + \Delta\mathbf{x})). \tag{1.35}$$

Taylor series expansion gives in first order

$$\tilde{\mathbf{x}_1} = \left(\varphi^{(1)}(\mathbf{x}) + D\varphi^{(1)}\Delta\mathbf{x}\right)(1 + E_1) + \cdots \tag{1.36}$$

with the partial derivatives

$$D\varphi^{(1)} = \left(\frac{\partial x_{1i}}{\partial x_j}\right) = \begin{pmatrix} \frac{\partial x_{11}}{\partial x_1} & \cdots & \frac{\partial x_{11}}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_{1n_1}}{\partial x_1} & \cdots & \frac{\partial x_{1n_1}}{\partial x_n} \end{pmatrix} \tag{1.37}$$

and the round-off errors of the first step

$$E_1 = \begin{pmatrix} \varepsilon_1^{(1)} & & \\ & \ddots & \\ & & \varepsilon_{n_1}^{(1)} \end{pmatrix}. \tag{1.38}$$

The error of the first intermediate result is

$$\Delta\mathbf{x}_1 = \tilde{\mathbf{x}_1} - \mathbf{x}_1 = D\varphi^{(1)}\Delta\mathbf{x} + \varphi^{(1)}(\mathbf{x})E_1. \tag{1.39}$$

The second intermediate result is

$$\begin{aligned} \tilde{\mathbf{x}_2} &= \left(\varphi^{(2)}(\tilde{\mathbf{x}_1})\right)(1 + E_2) = \varphi^{(2)}(\mathbf{x}_1 + \Delta\mathbf{x}_1)(1 + E_2) \\ &= \mathbf{x}_2(1 + E_2) + D\varphi^{(2)}D\varphi^{(1)}\Delta\mathbf{x} + D\varphi^{(2)}\mathbf{x}_1E_1 \end{aligned} \tag{1.40}$$

with the error

$$\Delta\mathbf{x}_2 = \mathbf{x}_2E_2 + D\varphi^{(2)}D\varphi^{(1)}\Delta\mathbf{x} + D\varphi^{(2)}\mathbf{x}_1E_1. \tag{1.41}$$

Finally the error of the result is

$$\Delta\mathbf{y} = \mathbf{y}E_r + D\varphi^{(r)}\cdots D\varphi^{(1)}\Delta\mathbf{x} + D\varphi^{(r)}\cdots D\varphi^{(2)}\mathbf{x}_1E_1 + \cdots D\varphi^{(r)}\mathbf{x}_{r-1}E_{r-1}. \tag{1.42}$$

The product of the matrices $D\varphi^{(r)}\cdots D\varphi^{(1)}$ is the matrix which contains the derivatives of the output data with respect to the input data (chain rule)

$$D\varphi = D\varphi^{(r)} \cdots D\varphi^{(1)} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}. \tag{1.43}$$

The first two contributions to the total error do not depend on the way in which the algorithm is divided into elementary steps in contrary to the remaining summands. Hence the inevitable error which is inherent to the problem can be estimated as [2]

$$\Delta^{(\text{in})} y_i = \varepsilon_M |y_i| + \sum_{j=1}^{n} \left| \frac{\partial y_i}{\partial x_j} \right| |\Delta x_j| \tag{1.44}$$

or in case the error of the input data is dominated by the round-off errors $|\Delta x_j| \leq \varepsilon_M |x_j|$

$$\Delta^{(\text{in})} y_i = \varepsilon_M |y_i| + \varepsilon_M \sum_{j=1}^{n} \left| \frac{\partial y_i}{\partial x_j} \right| |x_j|. \tag{1.45}$$

Additional errors which are smaller than this inevitable error can be regarded as harmless. If all errors are harmless, the algorithm can be considered well behaved.

1.4 Stability of Iterative Algorithms

Often iterative algorithms are used which generate successive values starting from an initial value $\mathbf{x}_0$ according to an iteration method

$$\mathbf{x}_{j+1} = f(\mathbf{x}_j), \tag{1.46}$$

for instance to solve a large system of equations or to approximate a time evolution $\mathbf{x}_j \approx \mathbf{x}(j\Delta t)$. Consider first a linear iteration equation which can be written in matrix form as

$$\mathbf{x}_{j+1} = A\mathbf{x}_j. \tag{1.47}$$

If the matrix A is the same for all steps we have simply

$$\mathbf{x}_j = A^j \mathbf{x}_0. \tag{1.48}$$

Consider the unavoidable error originating from errors $\Delta\mathbf{x}$ of the start values:

$$\mathbf{x}_j = A^j(\mathbf{x}_0 + \Delta\mathbf{x}) = A^j\mathbf{x}_0 + A^j\Delta\mathbf{x}. \tag{1.49}$$

The initial errors $\Delta\mathbf{x}$ can be enhanced exponentially if A has at least one eigenvalue[5] λ with $|\lambda| > 1$. On the other hand the algorithm is conditionally stable if for all eigenvalues $|\lambda| \leq 1$ holds. For a more general nonlinear iteration

$$\mathbf{x}_{j+1} = \varphi(\mathbf{x}_j) \tag{1.50}$$

the error propagates according to

$$\begin{aligned}
\mathbf{x}_1 &= \varphi(\mathbf{x}_0) + D\varphi\Delta\mathbf{x} \\
\mathbf{x}_2 &= \varphi(\mathbf{x}_1) = \varphi(\varphi(\mathbf{x}_0)) + (D\varphi)^2\Delta x \\
&\vdots \\
\mathbf{x}_j &= \varphi(\varphi\cdots\varphi(\mathbf{x}_0)) + (D\varphi)^j\Delta x.
\end{aligned} \tag{1.51}$$

The algorithm is conditionally stable if all eigenvalues of the derivative matrix $D\varphi$ have absolute values $|\lambda| \leq 1$.

1.5 Example: Rotation

Consider a simple rotation in the complex plane. The equation of motion

$$\dot{z} = i\omega z \tag{1.52}$$

obviously has the exact solution

$$z(t) = z_0 \mathrm{e}^{i\omega t}. \tag{1.53}$$

As a simple algorithm for numerical integration we use a time grid

$$t_j = j\Delta t \quad j = 0, 1, 2\cdots \tag{1.54}$$

$$z_j = z(t_j) \tag{1.55}$$

and iterate the function values

$$z_{j+1} = z_j + \dot{z}(t_j) = (1 + i\omega\Delta t)z_j. \tag{1.56}$$

Since

$$|1 + i\omega\Delta t| = \sqrt{1 + \omega^2\Delta t^2} > 1 \tag{1.57}$$

[5]The eigenvalues of A are solutions of the eigenvalue equation $A\mathbf{x} = \lambda\mathbf{x}$ (Chap. 10).

uncertainties in the initial condition will grow exponentially and the algorithm is not stable. A stable method is obtained by taking the derivative in the middle of the time interval (p. 296)

$$\dot{z}\left(t+\frac{\Delta t}{2}\right)=i\omega z\left(t+\frac{\Delta t}{2}\right)$$

and making the approximation (p. 297)

$$z\left(t+\frac{\Delta t}{2}\right)\approx\frac{z(t)+z(t+\Delta t)}{2}.$$

This gives the implicit equation

$$z_{j+1}=z_j+i\omega\Delta t\frac{z_{j+1}+z_j}{2} \tag{1.58}$$

which can be solved by

$$z_{j+1}=\frac{1+\frac{i\omega\Delta t}{2}}{1-\frac{i\omega\Delta t}{2}}z_j. \tag{1.59}$$

Now we have

$$\left|\frac{1+\frac{i\omega\Delta t}{2}}{1-\frac{i\omega\Delta t}{2}}\right|=\frac{\sqrt{1+\frac{\omega^2\Delta t^2}{4}}}{\sqrt{1+\frac{\omega^2\Delta t^2}{4}}}=1 \tag{1.60}$$

and the calculated orbit is stable.

1.6 Truncation Error

The algorithm in the last example is stable but of course not perfect. Each step produces an error due to the finite time step. The exact solution

$$z(t+\Delta t)=z(t)\mathrm{e}^{i\omega\Delta t}=z(t)\left(1+i\omega\Delta t-\frac{\omega^2\Delta t^2}{2}+\frac{-i\omega^3\Delta t^3}{6}\cdots\right) \tag{1.61}$$

is approximated by

$$z(t+\Delta t)\approx z(t)\frac{1+\frac{i\omega\Delta t}{2}}{1-\frac{i\omega\Delta t}{2}}$$

$$= z(t)\left(1 + \frac{i\omega\Delta t}{2}\right)\left(1 + \frac{i\omega\Delta t}{2} - \frac{\omega^2\Delta t^2}{4} - \frac{i\omega^3\Delta t^3}{8} + \cdots\right) \quad (1.62)$$

$$= z(t)\left(1 + i\omega\Delta t - \frac{\omega^2\Delta t^2}{2} + \frac{-i\omega^3\Delta t^3}{4}\cdots\right) \quad (1.63)$$

which deviates from the exact solution by a term of the order $O(\Delta t^3)$, hence the *local error* order of this algorithm is $O(\Delta t^3)$ which is indicated by writing

$$z(t + \Delta t) = z(t)\frac{1 + \frac{i\omega\Delta t}{2}}{1 - \frac{i\omega\Delta t}{2}} + O(\Delta t^3). \quad (1.64)$$

Integration up to a total time $T = N\Delta t$ accumulates a *global error* of the order $N\Delta t^3 = T\Delta t^2$.

Problems

Problem 1.1 Machine Precision

In this computer experiment we determine the machine precision ε_M. Starting with a value of 1.0, x is divided repeatedly by 2 until numerical addition of 1 and $x = 2^{-M}$ gives 1. Compare single and double precision calculations.

Problem 1.2 Maximum and Minimum Integers

Integers are used as counters or to encode elements of a finite set like characters or colors. There are different integer formats available which store signed or unsigned integers of different length (Table 1.4). There is no infinite integer and addition of 1 to the maximum integer gives the minimum integer.

In this computer experiment we determine the smallest and largest integer numbers. Beginning with $I = 1$ we add repeatedly 1 until the condition $I + 1 > I$ becomes invalid or subtract repeatedly 1 until $I - 1 < I$ becomes invalid. For the 64 bit long integer format this takes too long. Here we multiply alternatively I by 2 until $I - 1 < I$ becomes invalid. For the character format the corresponding ordinal number is shown which is obtained by casting the character to an integer.

Table 1.4 Maximum and minimum integers

Java format	Bit length	Minimum	Maximum
Byte	8	−128	127
Short	16	−32768	32767
Integer	32	−2147483647	2147483648
Long	64	−9223372036854775808	9223372036854775807
Char	16	0	65535

Problem 1.3 Truncation Error

This computer experiment approximates the cosine function by a truncated Taylor series

$$\cos(x) \approx \text{mycos}(x, n_{\max}) = \sum_{n=0}^{n_{\max}} (-)^n \frac{x^{2n}}{(2n)!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \cdots \quad (1.65)$$

in the interval $-\pi/2 < x < \pi/2$. The function $\text{mycos}(x, n_{\max})$ is numerically compared to the intrinsic cosine function.