

## 1

**Basic Concepts**

Examples provided in this chapter can be found in the directory `Voll_Chapter1` within the enclosed CD-ROM.



## 1.1

**Introduction**

The advent of computers has led to a revolution in numerical methods for solving scientific and engineering problems for different reasons:

- 1) With the use of the first computers, large and complex problems were solved significantly faster than by manual procedures. Typical algorithms for manual computing became obsolete for large-sized problems, whereas other algorithms, previously discarded, became more acceptable for numerical solution by computer.

For instance, the Cramer's method for linear algebraic systems requires long computational times when applied to a large number of unknowns (see Chapter 4); therefore, today, it can be considered obsolete.

Conversely, the minimization of functions, the integration of differential equations, and so on, represent operations that have been the subject of numerous new algorithms although, previously, they were practically unthinkable for their complexity.

Together with computers, specific programming languages were introduced to solve numerical problems; two of the most important were Fortran and Pascal. These languages adopt a *procedural* type of programming philosophy. The idea, on which this new philosophy is based, was fundamental to the writing of general codes to solve different types of problems. It consists of the implementation of a general algorithm, which allows one to solve a family of similar problems, in an adequate `SUBROUTINE`. This new approach to numerical methods radically changed the earlier manual computing, as the procedural programming allowed the writing of general codes to solve problems involving different physical and chemical phenomena. The procedural character of the algorithms was realized to be very useful for improvement, revision, implementation, and extension to different applications of a code.

A notable example is given by nonlinear algebraic equations systems: according to the manual computing approach, the success of the solution involved first the rearrangement of the system into a more manageable form, which required an *ad hoc* technique for each problem. In contrast, using procedural programming, all available algorithms can be implemented into one or more general SUBROUTINES, so as to have a ready-to-use collection (library). Coding the nonlinear system into a specific user subroutine, the scientist or the engineer can call, repeatedly, any of the algorithms included in the library. By such a computing technique, it is possible to solve a wide range of similar problems by means of a common solver.

The use of generalized SUBROUTINES for solving numerical problems led to an important collateral effect.



In solving a problem, there is a clear separation between the problem formulation and the algorithm implementation, the second operation being carried out first, independent of the problem formulation.

Accordingly, the problem formulation is not influenced by the method adopted for the solution.

In manual calculations, these two steps were often mixed, to the detriment of their complete understanding and to the clear explanation of the problem formulation.

- 2) Contrary to classical analysis and to numerical computing based on manual calculations, the *round-off errors* play a fundamental role in computer-handled numerical analysis. In manual computing, the weight of *round-off errors* was smoothed by human ability to attain the required accuracy of the problem solution.

When a problem is solved numerically on a computer, *round-off error* may become so important that even modifications to traditional thinking of classical analysis appear necessary.

Two consequences due to *round-off error* are shown below.



**First consequence:** some aspects of classical analysis prove inappropriate in numerical analysis, when a problem is solved by a computer.

For example, in classical analysis there are no differences in writing a linear system in either the form

$$\mathbf{Ax} = \mathbf{b}$$

or

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

under the assumption that the matrix  $\mathbf{A}$  is nonsingular. On the other hand, in numerical analysis, evaluating the vector  $\mathbf{x}$  by solving the system  $\mathbf{Ax} = \mathbf{b}$  is different from evaluating the product  $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ . This aspect is explained in Chapter 4.

Similarly, if one evaluates a derivative according to its definition

$$y'(x) = \lim_{h \rightarrow 0} \frac{y(x+h) - y(x)}{h}$$

and the value of  $h$  is progressively decreased, the derivative should have more and more inaccurate values (see Chapter 9).

Again, in an important theorem, Galois demonstrated the impossibility of finding the roots of a polynomial with a degree higher than 4 through a limited number of calculations. Nevertheless, although any numerical algorithm consists of a finite series of operations, algorithms exist that allow one to find roots for polynomials of whatever type. Consequently, the meaning of the *problem solution* has to be redefined.

**Second consequence:** a feature that is valid for a specific value of a variable in classical analysis is *not* “almost valid” when the numerical value of the same variable approaches this “classical” value.



For example, for the solution of linear systems in classical analysis, a null matrix determinant allows one to discriminate between systems with zero or infinite solutions and others with a unique solution.

In numerical analysis, on the contrary, a very small matrix determinant, that is,  $10^{-300}$ , does not give any useful information about the system.

## 1.2

### Modeling Physical Phenomena

Suppose a physical phenomenon is modeled with a computer. We deliberately define neither the physical phenomenon details nor the meaning of the physical phenomenon modeling.

In scientific, engineering, and technological studies, models can refer directly to either chemical and physical phenomena, such as the scattering of a particle, the progress of a virus, or the hydrogen unbrittlement of steel, or to manufacturing products, such as a pacemaker or a steam generator, and complex industrial systems, such as a refrigeration unit or a railway network. In the former models, elementary processes, such as momentum conservation, heat and mass transfer, changes of state, are directly involved and, by conservative and constitutive equations, they are translated in mathematical terms. Often, such models must capture variations of quantities in more than one spatial direction and along time. For other models, balance equations are used again, but typically at a larger scale and often with a lumped spatial representation; when complex systems are considered, modeling may involve optimization problems, statistical treatment, and dynamic analysis.

Mathematical modeling activity today covers almost all the scientific and engineering systems, including natural and economic sciences, from geometrical microscale to macroscale. With the widespread introduction of computers, especially of high-performance machines and clusters, modeling has become an alternative, competitive technique to experimental activity for studying real systems, such as process and electrical plants and chemical–physical phenomena, for improving technology, and for simulating special events such as accidents and failures. Nevertheless, mathematical models cannot wholly take the place of experiments; these remain

indispensable for several reasons, in particular for validating the same models and for all those applications where a high level of safety and reliability is required.



Actually, a mathematical description of a physical phenomenon is essential for its modeling.

Therefore, before implementation on a computer, a primary, independent modeling activity is necessary: the *mathematical modeling of the physical phenomenon*.



The mathematical model is to be considered as a *tool* specifically designed for evaluating some variables.

For instance, the phenomenon description may require the solution of a linear or nonlinear algebraic system, the integration of a differential equations system, the minimization of an objective function, and so on.

However, once the mathematical model has been defined, it cannot be directly implemented on a computer. It is important to distinguish the following three steps:

- *appropriate numerical methods* to solve the mathematical model have to be proposed;
- the numerical methods have to be converted to *appropriate algorithms*;



An *algorithm* is a procedure involving basic and unambiguous operations for solving a specific problem.

- finally, the algorithms have to be implemented in a computer program.



The aim of this book, and other books by the same authors, is to lead the reader to the *threefold selection* of

- 1) the best methods for solving an assigned problem;
- 2) the best development of the algorithms based on the selected methods;
- 3) the best algorithm implementation to build a computer program.

It could seem curious that the same numerical method might be codified into more than one algorithm. Let us consider the following simple example to understand how it is possible even with the most elementary procedures.

Given  $c$ ,  $d$ ,  $e$ , and  $x$ , the function to be evaluated is

$$y = c + dx + ex^2$$

$y$  can be obtained either through the algorithm

$$\begin{aligned} y &= ex \\ y &= (y + d)x \\ y &= y + c \end{aligned}$$

or through the alternative algorithm

$$\begin{aligned} y &= c + dx \\ y &= y + (ex)x \end{aligned}$$

In the former case, two multiplications and two additions are needed; in the latter case, three multiplications and two additions.

### Example 1.1 Electrical network

Consider an electrical network containing several resistances and one source of electromotive force, as shown in Figure 1.1.

This circuit can be mathematically described by applying Kirchhoff's laws and Ohm's law. If  $x_1, x_2, x_3,$  and  $x_4$  are the loop currents, the following system is obtained:

$$\begin{cases} 27x_1 - 3x_2 - 7x_3 = 500 \\ -3x_1 + 15x_2 - 4x_3 - 2x_4 = 0 \\ -7x_1 - 4x_2 + 19x_3 - 8x_4 = 0 \\ -2x_2 - 8x_3 + 20x_4 = 0 \end{cases}$$

The mathematical model of the electrical network corresponds to a linear algebraic system. The Gauss method represents the best approach to solve a linear system with a square matrix.

In the *BzzMath* library, the objects of the `BzzFactorizedGauss` class allow solution of this type of linear system.



The program is

```
#include "BzzMath.hpp"
void main(void)
{
    BzzFactorizedGauss A(4,4, // Define object A
        27.,-3.,-7.,0., // Initialize A
        -3.,15.,-4.,-2.,
        -7.,-4.,19.,-8.,
        0.,-2.,-8.,20.);
    // Define and initialize b
    BzzVector b(4,500.,0.,0.,0.);
    BzzVector x; // Define x
    Solve(A,b,&x); // Solve the system
    x.BzzPrint("Solution"); // Print the solution
}
```

This book deals with the solution of problems involving linear systems.

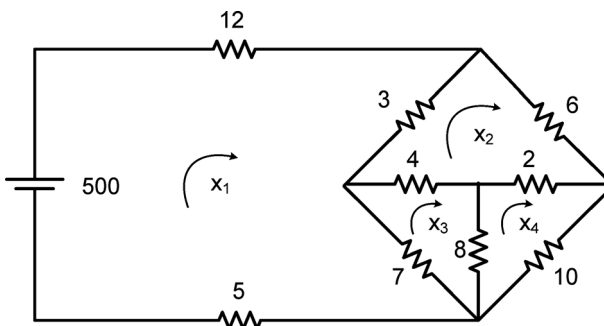


Figure 1.1 Electrical network.

**Example 1.2 Best model selection**

Suppose a dependent variable is to be correlated to an independent variable through the experimental data provided in Table 1.1.

As a general consideration, to develop adequate models for correlating the two variables, it is very important that one knows the physical phenomena on which the experimental data are based.

Assuming the following three models can be formulated:

$$y = b_1 + b_2 \left( \frac{b_3 x}{1. + b_3 x} \right)$$

$$y = b_1 + b_2 (1. - \exp(-b_3 x))$$

$$y = b_1 + b_2 \cdot \exp\left(-\frac{b_3}{x}\right) + b_4 x$$

the task is to find which is the best model.

From a mathematical point of view, the model parameters have to be evaluated by minimizing a function depending on the analyzed problem. Such an optimization problem presents some features that require specific algorithms.



In **BzzMath** library, the `BzzNonLinearRegression` class allows the solution of these problems.

**Table 1.1** Experimental data.

#	x	y
1	0.01	12.4699
2	.5	19.4172
3	1.	22.1502
4	2.	36.6730
5	3.	38.0526
6	4.	52.4134
7	6.	63.5087
8	9.	85.4125
9	12.	94.6759
10	16.	100.756
11	18.	109.302
12	20.	108.119
13	40.	114.922
14	70.	120.980
15	100.	122.013

The program is

```

#include "BzzMath.hpp"
// Models definition
void Model (int model, int ex, BzzVector &b,
            BzzVector &x, BzzVector &y)
{
    switch (model)
    {
        case 1:
            y[1] = b[1] + b[2] * (b[3] * x[1] /
                (1. + b[3] * x[1]));
            break;
        case 2:
            y[1] = b[1] + b[2] * (1. - exp(-b[3] * x[1]));
            break;
        case 3:
            y[1] = b[1] + b[2] * exp(-b[3] / x[1]) +
                b[4] * x[1];
            break;
    }
}

void main (void)
{
    int numModels = 3; // Three models
    int numX = 1; // One independent variable
    int numY = 1; // One dependent variable
    int numExperiments = 15;
    BzzMatrix X (numExperiments, numX, // Experimental x
                0.01, .5, 1., 2., 3., 4., 6., 9., 12., 16.,
                18., 20., 40., 70., 100.);
    BzzMatrix Y (numExperiments, numY, // Experimental y
                12.4699, 19.4172, 22.1502, 36.6730, 38.0526,
                52.4134, 63.5087, 85.4125, 94.6759, 100.756,
                109.302, 108.119, 114.922, 120.980, 122.013);
    BzzNonLinearRegression nonLinReg (numModels, X, Y,
    Model);
    BzzVector s2 (1, 7.5); // Error variance
    int df = 10; // degree of freedom
    nonLinReg.SetVariance (df, s2);
    BzzVector b1 (3, 10., 100., 1.);
    nonLinReg.InitializeModel (1, b1);
    BzzVector b2 (3, 10., 100., 1.);
    nonLinReg.InitializeModel (2, b2);
    BzzVector b3 (4, 10, 100., 1., 1.);
    nonLinReg.InitializeModel (3, b3);
}

```

```

        nonLinReg.LeastSquareAnalysis();
    }

```

Regression problems with linear and nonlinear models and, generally speaking, applied statistical problems are discussed in Buzzi-Ferraris and Manenti (2010).

### Example 1.3 Flash separator

This is a typical process unit operation used to separate chemical species with different volatility by an expansion (flash). The flow rate  $F$ , with molar composition  $z_i$ , at thermodynamic equilibrium conditions after expansion, is fed to a flash separator working at constant pressure  $P$  and temperature  $T$  (Figure 1.2).

The following relations govern the unit operation:

- overall material balance:  $F = V + L$
- $i$ th component balance:  $Fz_i = Vy_i + Lx_i$
- equilibrium relations:  $y_i = K_i x_i = 0$
- stoichiometry constraint:  $\sum_{k=1}^M (y_k - x_k) = 0$
- energy balance:  $VH_V + LH_L + Q = FH_F$

With  $M$  components,  $2M + 3$  equations are obtained.

Given the pressure  $P$ , the molar flow rate  $F$ , the duty  $Q$ , the enthalpy  $H_F$ , and the feed composition  $z_i$ , unknowns  $x_i$ ,  $y_i$ ,  $V$ ,  $L$ , and  $T$  can be evaluated. Alternatively, the problem may have different unknowns: given the pressure  $P$ , the molar flow rate  $F$ , the vapor flow rate  $V$ , the enthalpy  $H_F$ , and the feed composition  $z_i$ , unknowns  $x_i$ ,  $y_i$ ,  $L$ ,  $T$ , and the duty  $Q$  can be evaluated.

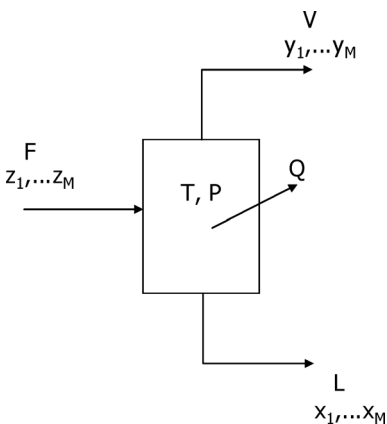


Figure 1.2 Flash separator scheme.

Moreover, the following functions should be assigned:

- equilibrium constants:  $k_i = k_i(T, P, \mathbf{x}, \mathbf{y})$
- vapor enthalpy:  $H_V = H_V(T, P, \mathbf{y})$
- liquid enthalpy:  $H_L = H_L(T, P, \mathbf{x})$

When steady-state conditions are considered, the mathematical model is a nonlinear algebraic system.

Such mathematical systems occur frequently in scientific and engineering modeling. Several algorithms are available to solve nonlinear systems: some of them perform particularly well, but they may diverge in some circumstances, whereas others may be more robust but perform less well.

In **BzzMath** library, the `BzzNonLinearSystem` class allows solution of this kind of problem.



A concrete simplified example could be the following one (Henley and Rosen, 1969). Consider a flash separator with an inlet process flow rate characterized by a molar composition of 30% toluene and 70% 1-butanol. The partial pressures of the two components can be expressed as follows:

- toluene:  $\log_{10} P_1 = 6.95508 - \frac{1345.087}{219.516 + T}$
- 1-butanol:  $\log_{10} P_2 = 8.19659 - \frac{1781.719}{217.675 + T}$  where  $P_i$  and  $T$  are the partial pressure in mmHg and the temperature in °C, respectively.

For the sake of conciseness, let us assume the mixture to be ideal. Unknowns are  $x_1$ ,  $x_2$ ,  $y_1$ ,  $y_2$ ,  $L$ ,  $T$ , and  $Q$ ; however, since the last variable  $Q$  appears only in the energy balance equation, it is possible to remove both the equation and the variable, and, therefore, a reduced system is to be solved. The following values are assigned:  $V = 25.$ ;  $P = 760.$ ;  $F = 100.$ ;  $z_1 = .3$ ;  $z_2 = .7$ .

The program is

```
#include "BzzMath.hpp"
void Flash(BzzVector &v, BzzVector &f)
{
    double x1 = v[1];
    double x2 = v[2];
    double y1 = v[3];
    double y2 = v[4];
    double T = v[5];
    double L = v[6];
    double V = 25., F = 100., P = 760., z1 = .3, z2 = .7;
    double k1 = pow(10., 6.95508 - 1345.087 /
        (219.516 + T)) / P;
    double k2 = pow(10., 8.19659 - 1781.719 /
        (217.675 + T)) / P;
    f[1] = F * z1 - V * y1 - L * x1;
    f[2] = F * z2 - V * y2 - L * x2;
```

```

    f[3] = y1 - k1 * x1;
    f[4] = y2 - k2 * x2;
    f[5] = y1 + y2 - x1 - x2;
    f[6] = F - V - L;
}
void main(void)
{
    BzzPrint("\n\nFlash");
    BzzVector v0(6, .3, .7, .3, .7, 100., 50.);
    BzzNonLinearSystem nls(v0, Flash);
    nls();
    nls.BzzPrint("Solution");
}

```

A more detailed discussion on nonlinear algebraic systems and optimization problems is given in Buzzi-Ferraris and Manenti (2011a).

When the flash separator works under unsteady conditions, a differential-algebraic system arises due to the time derivative of some dependent variables. In particular, mass and energy balance equations become of differential type, whereas stoichiometric and equilibrium relations preserve their algebraic form. Similarly, the solution can be obtained by selecting a proper numerical method, coding it into an algorithm, and finally implementing it in a calculation program.

Problems on differential and differential-algebraic systems with initial conditions are discussed in Buzzi-Ferraris and Manenti (2011b).

#### Example 1.4 Differential equation systems

The following differential system is to be integrated:

$$\begin{cases} y_1' = \frac{\lambda_1 + \lambda_2}{2} y_1 + \frac{\lambda_1 - \lambda_2}{2} y_2 \\ y_2' = \frac{\lambda_1 - \lambda_2}{2} y_1 + \frac{\lambda_1 + \lambda_2}{2} y_2 \end{cases}$$

with  $y_1(0) = 10.$  and  $y_2(0) = 1.$  as initial conditions and  $\lambda_1 = -1.$  and  $\lambda_2 = -10\,000.$  as constant coefficients.

Although the system can be easily solved by means of both numerical and analytical approaches, it represents an interesting case of the importance of knowledge about the physics of the problem for an efficient mathematical formulation and relevant implementation into an algorithm.

The above system is a typical, stiff numerical problem, as it models phenomena characterized by different large temporal or spatial scales. If a clever algorithm is not used to find its solution, a very large number of iteration steps could be necessary, increasing excessively the computational time.



In the **BzzMath** library, the `BzzOdeStiff` class is available for efficiently solving stiff problems.

The solving program is

```
#include "BzzMath.hpp"
void Ode(BzzVector &y, double t, BzzVector &f)
{
    double l1 = -1.;
    double l2 = -10000.;
    double a1 = .5 * (l1 + l2);
    double a2 = .5 * (l1 - l2);
    f[1] = a1 * y[1] + a2 * y[2];
    f[2] = a2 * y[1] + a1 * y[2];
}
void main(void)
{
    BzzPrint("\n\nOde");
    BzzVector y0(2, 10., 1.), y;
    double t0 = 0., tOut = 100.;
    BzzOdeStiff o(y0, t0, Ode);
    y = o(tOut);
    o.BzzPrint("Results");
}
```

If the classic Runge–Kutta method is used instead, a very large number of iterations are needed.

Problems on differential and differential-algebraic systems are discussed in Buzzi-Ferraris and Manenti (2011b).

### 1.3 Number Representation on the Computer

The basic element of the computer memory, where everything is stored and calculations are executed, is the *bit*.

A *bit* can assume only two values: *true* or *false*. It is false when it is equal to zero and true otherwise. Eight successive bits constitute another important element: the *byte*.

To store a number, a set of bits or bytes is required, depending on the number type and the processor characteristics.

In the following, only current Intel 32-bit processors are taken into account.

A programming language such as C++ allows the storage of different types of numbers, from those stored on a single bit to longer ones with different features.



Concerning numerical analysis, the most interesting number type is the *floating point* to represent real numbers. Conversely, for some non-numerical operations (i.e., selection of a vector element), integer numbers are appropriate.



Only floating points and integer numbers are considered in the following.

Contrary to other programming languages (e.g., Basic or Fortran77), C++ requires each adopted identifier to be defined in the program before it is used.



This feature is not much appreciated by scientists and engineers used to programming in Basic or Fortran; however, it is significantly useful since it avoids possible errors that are difficult to find during the debugging phase.

For instance, identifiers `variableInt` for integer numbers and `variableFloat` and `variableDouble` for floating point numbers must be defined as

```
int variableInt = 6;
float variableFloat = 3.2F;
double variableDouble = 4.567;
```

There are no problems in storing an integer number: it can be stored on a certain number of bytes.



For the selected processors, an integer number requires 4 bytes (32 bits). Therefore, it is possible to store and use a positive integer number ranging from 0 to  $2^{32}-1$  and a signed integer number ranging from  $-2^{31}$  to  $2^{31}-1$ .

The keyword `unsigned` is used in the number definition to denote a positive number. In the case of integer numbers,

```
unsigned int variableIntUnsigned = 6;
```

On the other hand, it is not necessary to use the keyword `signed`, since a number has a sign by default.

The representation of a real number is completely different.

In this case, the scientific notation has to be adopted. One bit is used for the sign, some bits are dedicated to the significant digits, and remaining bits to the exponent.

For example, the number  $-0.002347$  is stored in its scientific form  $-0.2347 \cdot 10^{-2}$ .

All the languages allow the storing of this kind of numbers, called floating points, at least in two different ways: in single or in double precision.



Only the most common and widespread representation is considered here: the *normalized floating point*.

Even though numbers are not stored on the computer according to base 10, it is, however, advisable to adopt this base to make clearer the explanation below.

A real number  $x$  can be represented as follows:

$$x = \pm \left( \frac{d_1}{10} + \frac{d_2}{10^2} + \dots + \frac{d_t}{10^t} \right) 10^e \quad (1.1)$$

where integers  $d_1, d_2, \dots, d_t$  and  $e$  satisfy the relations

$$1 \leq d_1 \leq 9 \quad (1.2)$$

$$0 \leq d_i \leq 9 \quad (i = 2, \dots, t) \quad (1.3)$$

$$L \leq e \leq U \quad (1.4)$$

Relation (1.2) is required to obtain the *normalized* form.

When  $x = 0$ ,

$$d_1 = d_2 = \dots = d_t = 0, \quad e = L \quad (1.5)$$

The number  $\frac{d_1}{10} + \frac{d_2}{10^2} + \dots + \frac{d_t}{10^t}$  is called the *mantissa* (or *fraction*) of the number  $x$  represented in the floating point form. The length  $t$  of the mantissa is called the *precision*.



It is not possible to represent all the real numbers in the floating point notation for two reasons:

- they can have an infinite number of digits;
- they are infinite and the floating point notation allows representation of a finite number of them.

As a consequence, there is not a biunique correspondence between real and floating point numbers. If  $x$  is a real number and the following relation is verified:

$$\left(\frac{d_1}{10} + \frac{d_2}{10^2} + \dots + \frac{d_t}{10^t}\right)10^e < |x| < \left(\frac{d_1}{10} + \frac{d_2}{10^2} + \dots + \frac{d_t+1}{10^t}\right)10^e \quad (1.6)$$

then one has to choose between the two floating point numbers where  $x$  is placed in between. This can be done by one of two methods.

- 1) **Round-off:** the closest number is selected.
- 2) **Chopping:** the number with the smallest mantissa is selected.



Consider the number .123456789 with a precision  $t = 8$ . Values .12345679 and .12345678 are given by round-off and chopping, respectively.

Only the first method is considered here: the round-off operation is applied for each number that cannot be represented as a floating point.



An expression such as  $fl(x)$  is generally used to indicate the operation by which a real number  $x$  is approximated through the round-off.

By the aforementioned compilers, the single precision requires 4 bytes and allows representation of numbers in the range  $\pm 3 \cdot 10^{-38}$  to  $\pm 3 \cdot 10^{38}$ ; double



precision requires 8 bytes and allows a larger range of representation,  $\pm 10^{-308}$  to  $\pm 10^{308}$ .

During the calculations, some numbers may overcome the floating point boundaries.



When the absolute value of the number is lower than the minimum bound, the error is called *underflow*; conversely, when it exceeds the maximum bound, the error is called *overflow*.

In the case of underflow, a zero is generally assigned to the number.

In the case of overflow, a run-time error was detected by former compilers, and consequently computation was stopped. Recent compilers detect the number and assign to it a special symbol, without stopping the computing procedure. Usually, no overflow and underflow messages are visualized.

This shortcoming can be dangerous and the user should be warned when at least an overflow problem is detected.



It is possible (and recommended) to exclude or modify this compiler default condition, to have indication of possible overflows.

It is useful to evaluate the accuracy of calculations, as it may change with different processors. A good estimation of the accuracy is achieved by variable *macheps*.



*Macheps* is the smallest positive number such that

$$fl(1. + macheps) > 1. \quad (1.7)$$

With 32-bit compilers, the *macheps* is about  $1.210 \cdot 10^{-7}$  in single precision and  $2.210 \cdot 10^{-16}$  in double precision.

Starting from the previous property, it is possible to define an algorithm to evaluate the *macheps* of a specific machine with an assigned programming language.

### Algorithm 1.1 Macheps evaluation

```
macheps = 1.
for i=1, 2, ...
    macheps = macheps/2.
    eps = macheps + 1.
    if (eps = 1.)
        macheps = 2.*macheps
    quit
```

C++ allows execution of any type of operation before starting the program. In **BzzMath** library, this feature is exploited to carry out some simple operations suitable for the user. For example, both the *macheps* in single and double precisions are evaluated through Algorithm 1.1. They are stored in two global constants, `MACH_EPS_FLOAT` and `MACH_EPS_DOUBLE`, and then they can be used in any part of the program.

## 1.4

### Elementary Operations

A generic real number is approximated by its floating point representation when processed by a compiler; analogously, basic operations among numbers are approximated.

The current numerical handling adopted by all the compilers in executing whatever elementary operation (sum, subtraction, multiplication, and division) on two floating point numbers  $x$  and  $y$  is briefly described below.

Let  $\omega$  be a generic, exact, basic operation and  $\omega^*$  the corresponding floating point operation; for a computation performed by a compiler, the following relation is valid:

$$x \omega^* y = fl(x \omega y) \quad (1.8)$$

where  $fl(x)$  represents the round-off operation. The accuracy of the numerical approximation introduced by rounding-off can be evaluated by

$$x \omega^* y = (x \omega y)(1 + \varepsilon), \quad \text{where } \varepsilon \leq macheps \quad (1.9)$$

For example, consider a calculation with seven significant digits where one has to add the following numbers:  $x = 0.1234567$  and  $y = 0.7654321 \cdot 10^{-7}$  as they are stored on the computer memory. The sum  $z = x + y$  is executed by the computer with the number of significant digits higher than 7, which allowed storage of the results, and can be practically considered as an exact sum for the two selected numbers:  $z = 0.12345677654321$ . Since this number cannot be stored with all the digits, it has to be approximated according to  $z = x +^* y = 0.1234568$ .

The computer operates a round-off in storing numbers and in performing calculations. The round-off introduces some relevant, numerical consequences that have only recently been deeply investigated. Two of them have already been underlined.



## 1.5

### Error Sources

When a mathematical model is solved by a numerical method, results are inexact because of different errors and approximations introduced during computing.

#### 1.5.1

##### Model Error

The first error may be due to the intrinsic structure of the mathematical model adopted for describing the system under study. Selecting the most appropriate model is a crucial and difficult task, as it requires a deep understanding of both involved chemical and physical phenomena and numerical techniques to solve it. Typically, scientist and engineers have to face a “trade-off” problem between the implementation of a detailed and time-consuming model and an approximated and

computationally light one. In the first case, the necessary computing time or hardware resources may not practically fit those of the user; and in the second case the results obtained via a simplified model may have an unacceptable accuracy and reliability.

### 1.5.2

#### Approximation Error

The second error may arise as the mathematical model is converted from its analytical to numerical form. During this step, a necessary approximation, that is, errors, is introduced.

For instance, a finite-difference method is often used for solving systems of differential equations. In this case, replacing the derivatives by truncated Taylor expansions, the change of variables is converted from continuous to discrete, and therefore information is lost.

### 1.5.3

#### Round-Off Error

The third error arises when the selected numerical method is converted into an algorithm and implemented in a computer program; as already explained, some calculation errors are introduced because of the round-off errors.

According to the algorithm, a specific sequence of elementary operations is required to achieve a numerical solution. By changing the algorithm, the operations sequence may be modified and, consequently, the error may be introduced and propagated in a different way. Several studies on algorithm errors due to the round-off operation have been carried out (Givens, 1954; Lanczos, 1956; Wilkinson, 1963), the main purpose being to answer the following question.



“What is the gap due to round-off errors between the numerical and the exact solution?”

This kind of analysis is called *forward analysis*.



Classical analysis cannot be used anymore because of the round-off errors. In fact, some features of the elementary operations are no longer valid.

Consider the following three numbers:  $a = 12345678.$ ,  $b = -12344321.$ , and  $c = .12345678.$

By using eight significant digits,  $d = a + b = 1357.0000$  and, hence,  $d = d + c = 1357.1235$ . Conversely,  $d = a + c = 12345678.$  and, consequently,  $d = d + b = 1357.0000$ .

According to the sequence of elementary operations, the result is different. The associative property is no longer valid for the sum.



The *backward analysis* of the error is completely different. In this case, the question is: “How does the solved problem differ from the original one, under the assumption that the exact solution has been reached?”

It is important to have a deep understanding of the difference between these two points of view.

Let  $x \omega^* y = fl(x \omega y) = (x \omega y)(1 + \varepsilon)$  be the error generated by an elementary operation  $\omega$  between two numbers  $x$  and  $y$ . For the sake of simplicity, suppose both  $x$  and  $y$  are exactly stored.

- 1) In the *forward analysis*, the difference due to round-off errors between the floating point result and the exact operation is equal to  $(x \omega y)\varepsilon$ .
- 2) In the *backward analysis*, one can think to exactly solve a slightly different problem. For example, the exact solution can be found when the variable  $y$  undergoes a small modification:

$$x \omega^* y = fl(x \omega y) = x \omega [y(1 + \varepsilon)]$$

In the former case,  $x$  and  $y$  are exact and the operation generates an error. In the latter case, the elementary operation is exact and the problem is slightly different. How much different? One or both the variables  $x$  and  $y$  have been modified by a quantity that can be evaluated when the solution is achieved.

In the analysis of the error, the elementary operations can now be used as in classical algebra since their properties are still valid.

Let us consider another very important case: the estimation of the error related to the use of a particular algorithm to solve the linear system

$$\mathbf{Ax} = \mathbf{b}$$

The forward analysis searches for the difference between the exact solution and the one obtained through a specific algorithm. After several years, no one has demonstrated whether a specific algorithm was valid.

The backward analysis allows one to find the new matrix  $\mathbf{B} = \mathbf{A} + \mathbf{E}$  that exactly solves the system

$$\mathbf{Bx}' = \mathbf{b}$$

where the (inexact) solution  $\mathbf{x}'$  is achieved by using a specific algorithm. If the matrix  $\mathbf{E}$  has very small coefficients compared to the matrix  $\mathbf{A}$ , the algorithm can be considered reasonably accurate.

This new approach in the error analysis led to significant results:

- 1) the analysis is useful in many practical problems (even though its utilization is not very easy);
- 2) the results obtained through the backward analysis are often important both practically and theoretically;
- 3) the solution accuracy of a numerical problem has been redefined.

If it is possible to demonstrate that an algorithm exactly solves a problem slightly different from the original problem, the algorithm can be considered accurate; conversely, when the new problem is considerably different, the algorithm is inaccurate.



Since this modification can be seen as a variation of the physical problem, it is easy to check its importance.

Lanczos (1956) first discussed this last point in the explanation of the  $\tau$  method.



In the  $\tau$  method, the original problem is slightly modified to obtain a new problem that is exactly solvable without any difficulty.

As an example, an approximate polynomial solution in the range  $0 \leq x \leq 1$  of the following differential equation is required:

$$y' + y = 0, \quad y(0) = 1$$

No polynomials fully satisfy the equation. Nevertheless, if the equation is transformed into

$$y' + y = \tau(128x^4 - 256x^3 + 160x^2 - 32x + 1)$$

the polynomial approximation is possible. The polynomial between brackets is the four-degree Chebyshev polynomial, in the range  $0 \leq x \leq 1$  (see Buzzi-Ferraris and Manenti, 2010a).

All the Chebyshev polynomials have the characteristic of being smaller than unity in the aforementioned range.

Let the polynomial function

$$P_4(x) = a + bx + cx^2 + dx^3 + ex^4$$

fully satisfy the converted differential equation. Parameters  $a$ ,  $b$ ,  $c$ ,  $d$ ,  $e$ , and  $\tau$  can be obtained by introducing the polynomial function into the differential equation at five selected points and accounting for the initial condition.

The following polynomial is obtained:

$$P_4(x) = 1 - \frac{4960}{4961}x + \frac{2464}{4961}x^2 + \frac{768}{4961}x^3 + \frac{128}{4961}x^4$$

with  $\tau = 1./4961$ . Since the value of the Chebyshev polynomial is ever smaller than unity, the difference between the converted differential equation and the original one corresponds to  $\tau$ .

#### 1.5.4

#### Local and Propagation Errors

Two extreme situations turn out to be significantly important for computational errors analysis.

- 1) The problem data are not affected by any error and it is possible to estimate the error due to the algorithm approximation.



The error related to the algorithm approximation is called the *local error*. It is estimated by supposing that both the data and the operations are exact.

- 2) The data are slightly perturbed and the perturbation effects on the solution can be studied.

The error in the solution obtained by perturbing data and evaluated under the condition that no round-off errors affect the calculations is called the *propagation error*.



For example, integrating the differential equation

$$y' = f(y, t)$$

by the Euler's forward method:

$$y_{n+1} = y_n + hf(y_n, t_n) + R$$

the value of  $y_{n+1}$  is affected by two causes of error (see Buzzi-Ferraris and Manenti, 2011b):

- 1) The differential equation is approximated through a finite-difference expression; this error is given by the residual  $R$  and it is the local error, which can be estimated under the assumption that  $y_n$  is exact and calculations are carried out without any round-off error.

It is possible (and important) to evaluate the magnitude of this error (whatever algorithm is selected) to adapt the integration step to the required accuracy.

- 2) The value of  $y_n$  is inexact because of some previous calculations; this causes an error propagation in the calculation of  $y_{n+1}$ , which is related to the features of the function  $f$ , to the selected algorithm, and to the integration step  $h$ .

It is therefore important to check if the selected algorithm avoids a progressive error increment.

## 1.6 Error Propagation

For the sake of simplicity, let us consider the relation

$$y = \varphi(x) \quad (1.10)$$

If the variable  $x$  is slightly perturbed or, in other words, if  $|x - \hat{x}|$  is appreciably small,

$$\hat{y} = y + \varphi'(x)(\hat{x} - x) \quad (1.11)$$

and

$$\varepsilon_y = \frac{\hat{y} - y}{y} = \left| \frac{\varphi'(x)x}{\varphi(x)} \right| \frac{\hat{x} - x}{x} = \kappa(x)\varepsilon_x \quad (1.12)$$

The number  $\kappa(x) = \left| \frac{\varphi'(x)x}{\varphi(x)} \right|$  correlates the relative error in  $y$  to the relative error in  $x$  and is called the *condition number*.





The *condition number* is the amplification factor of the relative error.

It is shown in the following paragraphs how it is possible to extend this concept to vectors and matrices.

Consider the four basic operations: addition, subtraction, multiplication, and division. As an example, take an operation between the exact number  $a$ , which can also be stored in the computer memory without any round-off error, and the number  $x$  affected by the relative error  $\varepsilon_x$ .

**Multiplication:**

$$y = a \cdot x, \quad \varepsilon_y = \left| \frac{ax}{ax} \right| \varepsilon_x = \varepsilon_x \quad (1.13)$$

Multiplication is a safe operation, since it never amplifies the relative error.

**Division:**

$$y = a/x, \quad \varepsilon_y = \left| \frac{ax^2}{ax^2} \right| \varepsilon_x = \varepsilon_x \quad (1.14)$$

Even division is a safe operation.

**Addition:**

$$y = a + x, \quad \varepsilon_y = \left| \frac{a}{a+x} \right| \varepsilon_x \quad (1.15)$$

Addition is an operation that dramatically amplifies the relative error when the numbers are quite close, but with opposite signs. Concerning the propagation error, the sum is a safe operation only when numbers have the same sign.

**Subtraction:**

$$y = a - x, \quad \varepsilon_y = \left| \frac{a}{a-x} \right| \varepsilon_x \quad (1.16)$$

Subtraction is an operation that dramatically amplifies the relative error when the numbers are quite close and with the same signs. Concerning the propagation error, the subtraction is a safe operation only when numbers have opposite signs.

For example, consider the two numbers  $a = 0.12345678$  and  $x = 0.12345612$ . Since  $y = a - x = 0.00000066$ , this means that when  $x$  has an error on the eighth significant digit,  $y$  inevitably has an error on the second significant digit.



The **first enemy** in numerical calculations on the computer is subtraction between two numbers of the same order and with the same sign.

It is worth noting that it is not due to the local error of the operation, but to the amplification of a preexisting error.

If the data are exact and they are even stored without any round-off error, the sum and the subtraction are both safe operations concerning the propagation error.



This is another example where one should modify a feature of classical analysis.



If it is necessary to subtract two quite close numbers within an algorithm, one must modify the evaluation procedure, when possible.

Suppose the subtraction is performed between a function  $g(x)$  and a number  $z$ . In some situations, the function value and the number might be quite close and with the same sign. For this, three devices are proposed.

### 1.6.1

#### First Device

The first useful device to avoid the dangerous operation is to exploit the following property:

$$(z-g(x))(z+g(x)) = z^2 - (g(x))^2$$

As an example, in classical analysis, the roots of the quadratic equation  $ax^2 + bx + c = 0$  are  $x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$  and  $x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$ . If the term  $4ac$  is small, but nonzero, one of the two roots is badly evaluated.

If  $b > 0$ , the root  $x_1$  may be ill-conditioned and the following formulae should be used:

$$\begin{aligned} x_1 &= \frac{(-b + \sqrt{b^2 - 4ac})(b + \sqrt{b^2 - 4ac})}{2a(b + \sqrt{b^2 - 4ac})} \\ &= \frac{(-b^2 + b^2 - 4ac)}{2a(b + \sqrt{b^2 - 4ac})} = \frac{-2c}{b + \sqrt{b^2 - 4ac}} = \frac{c}{ax_2} \\ x_2 &= \frac{-b - \sqrt{b^2 - 4ac}}{2a} \end{aligned}$$

On the other hand, when  $b < 0$ , the root  $x_2$  may be ill-conditioned and the following formulae should be used:

$$\begin{aligned} x_1 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \\ x_2 &= \frac{(-b - \sqrt{b^2 - 4ac})(-b + \sqrt{b^2 - 4ac})}{2a(-b + \sqrt{b^2 - 4ac})} \\ &= \frac{(-b^2 - b^2 + 4ac)}{2a(-b + \sqrt{b^2 - 4ac})} = \frac{2c}{-b + \sqrt{b^2 - 4ac}} = \frac{c}{ax_1} \end{aligned}$$

If it is required to implement such a procedure in a computer program, it is necessary to take into consideration all the situations that can create some problems during the solution. It is not enough to use those equations to make the problem less sensitive to round-off errors.

For example, if  $a = 0$ , there is a single solution; on the other hand, when  $a$  is small and  $c$  is large, an overflow condition may occur.



The object-oriented programming allows creation of an object that takes care of all the possibilities and makes the calculations as accurate as possible.

Finding the root of a quadratic function has been implemented in an appropriate class.



In the **BzzMath** library, the class is called `BzzQuadraticEquation`.

This class has two constructors: the default one and another that requires three double:  $a$ ,  $b$ , and  $c$ . If the object is initialized with the default constructor, it is necessary to provide these three values later in the program. Some examples to initialize an object of the `BzzQuadraticEquation` class are

```
double a = 10.; b = 3., c = 1.;
BzzQuadraticEquation q1;
q1(a, b, c);
BzzQuadraticEquation q2(a, b, c);
```

The available functions are `QuadraticSolutionType`, `BzzPrint`, `GetX1`, `GetX2`, `GetReal`, `GetImaginary`, and `GetStationaryPoint`.

The function `QuadraticSolutionType` returns an `int` indicating the solution type obtained by the object. The possible alternatives are

- 1) No solutions
- 2) Identity  $0 = 0$
- 3) One real solution
- 4) One real solution with negative overflow
- 5) One real solution with positive overflow
- 6) Two real solutions
- 7) Two real solutions with the first one with negative overflow
- 8) Two real solutions with the second one with positive overflow
- 9) Two real solutions with both overflows
- 10) Two complex and conjugate solutions

The functions `GetX1` and `GetX2` return 0 either when the solution does not exist or when it leads to the overflow condition; they return 1, otherwise.

The functions `GetReal` and `GetImaginary` return 0 when the solutions are both real; they return 1, otherwise.

The function `GetStationaryPoint` returns 0 either when the stationary point does not exist or when it is an overflow; it returns 1, otherwise.

### Example 1.5

Let  $a = 1$ ,  $b = 3$ , and  $c = 5$ . Find the solutions of the quadratic function and check their nature.

The program is

```
#include "BzzMath.hpp"
void main(void)
```

```

{
int quadraticSolutionType;
double a = 1.;
double b = 3.;
double c = .5;
BzzQuadraticEquation q(a,b,c);
quadraticSolutionType =
    q.QuadraticSolutionType();
q.BzzPrint("\nResult %d", quadraticSolutionType);
double x1,x2,stationaryPoint;
if(q.GetX1(&x1))
    BzzPrint("\nx1 = %e", x1);
if(q.GetX2(&x2))
    BzzPrint("\nx2 = %e", x2);
if(q.GetReal(&x1))
    BzzPrint("\nReal Part = %e", x1);
if(q.GetImaginary(&x2))
    BzzPrint("\nImaginary Part = %e", x2);
// 1 OK; 0 No stationary point or overflow;
if(q.GetStationaryPoint(&stationaryPoint))
    BzzPrint("\nStationaryPoint=%e",
        stationaryPoint);
}

```

Additional examples on the use of the BzzQuadraticEquation class can be found in the directory

```

BzzMath/Examples/BzzMathBasic/Utility/
BzzQuadraticEquation

```

either on the enclosed CD-ROM or at the web-site

[www.chem.polimi.it/homes/gbuzzi](http://www.chem.polimi.it/homes/gbuzzi)



## 1.6.2

### Second Device

The second useful device can be used when the difference between a function  $g(x)$  and a number with the same sign and order is to be evaluated. It is based on a Taylor series expansion of the function  $g(x)$ .

As an example, suppose  $y = \frac{1-\cos(x)}{x}$  is evaluated when  $x$  is small. If  $x \rightarrow 0$ , then  $\cos(x) \rightarrow 1$ . Once again, subtraction between two quite close numbers occurs. Expanding the cosine  $\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^6)$  and inserting this expression into the function, the following expression is obtained:  $y = \frac{x}{2} - \frac{x^3}{24} + O(x^5)$ . With such a manipulation, in the case of  $x \rightarrow 0$ , the operation is accurate.

## 1.6.3

**Third Device**

This device is useful when a summation of a Taylor series consisting of terms with alternate signs is to be computed. It is usually possible to convert the problem and obtain only the terms with the same sign.

For example, the calculation of the series

$$e^{-x} = 1 - x + \frac{x^2}{2} - \frac{x^3}{6} + O(x^4)$$

is very inaccurate for large values of  $x$ . To avoid this problem, it is enough to evaluate the following series:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + O(x^4)$$

and, therefore, the result can be inverted.

## 1.7

**Decision-Making for an Optimal Program**

The goal of research and engineering activity is to describe, at best, a physical phenomenon and code it in a program that assures a proper calculation efficiency and reliability. In this regard, we have to make some decisions at different levels. Before discussing decisions, it is important to introduce a major concept in developing numerical algorithms and programs: the *stability*. This term assumes different meanings with reference to different steps involved in computer programming:

- 1) stability of the physical phenomenon to be analyzed;
- 2) stability of the mathematical formulation to be adopted for describing the physical phenomenon;
- 3) stability of the algorithm to be used for numerically solving the mathematical formulation;
- 4) stability of the program written for solving the problem.

It is suitable to assign a specific name to each meaning, to avoid any ambiguity in a detailed discussion.

The following paragraphs deal with the decision-making at each level to convert a physical phenomenon into a calculation program.

## 1.7.1


**Check of the Physical Phenomenon**

It is important to underline that, in order to numerically describe a real phenomenon, the phenomenon should not involve explosive or chaotic behavior, or, in other words, it should be characterized by stable conditions. For example, it is well known that it is

not possible to mathematically describe the oscillations of a *double pendulum* when the same oscillations are significantly wide.

When a physical phenomenon is impossible to treat numerically, some specific features of this phenomenon should be evaluated, for example, the initial conditions of an explosion. The boundary between these two situations is not clearly defined; nevertheless, recent progress in numerical techniques and computational resources is pushing modeling capabilities toward more and more complex problems.

However, the discrimination is important: those problems that can be mathematically formulated shall be called *well-posed* and those that cannot be mathematically modeled shall be called *ill-posed*.


To clarify this concept and avoid any misunderstanding with other kinds of instability problems, when we refer to the real physical problem, we will speak about *well-posed* and *ill-posed* problems. 


In the following, the physical phenomenon is assumed to be a well-posed problem.

### 1.7.2

#### Mathematical Formulation

Given a well-posed physical problem, there may be several ways of mathematically modeling it. Some of these formulations may be sensitive to small perturbations, whereas others may not.

In the former case, the mathematical formulation is *well-conditioned*, whereas in the latter case the formulation is *ill-conditioned*. 

It is important to observe that the problem formulation is *well-* or *ill-conditioned* independent of the numerical algorithm. If the physical phenomenon is *ill-posed* or its formulation is *ill-conditioned*, no numerical methods can solve it. 

Consider the problem

$$y' = 9y - 10e^{-x}, \quad y(0) = 1$$

Its analytical solution is

$$y = e^{-x}$$

Slightly modifying the initial condition  $y(0) = 1.0001$ , the following analytical solution is obtained:

$$y = e^{-x} + 0.0001 \cdot e^{9x}$$

The solutions of the previous problems in correspondence with  $x = 2$  are 0.1353353 and 6566.132, respectively.

As a result, this model formulation is *ill-conditioned*. 

As discussed in Buzzi-Ferraris and Manenti (2011b), an adequate analysis of the previous equation before the integration allows identifying *a priori* its ill-conditioning.



It is not necessary to dispose of any numerical algorithms to quantify the conditioning nature of a model formulation: only classical analysis is needed.

Another example is the following polynomial (Wilkinson 1963):

$$P_{20}(x) = (x-1)(x-2)(x-3) \cdots (x-20)$$



Such a polynomial is well-conditioned, since small variations in its parameters produce small changes in the results.

If this polynomial is analytically transformed (without any calculation error) into its standard form (Buzzi-Ferraris and Manenti, 2010),

$$P_{20}(x) = x^{20} - 210x^{19} + \cdots + 20!$$

If we add  $10^{-7}$  to the exact value 210, the roots of the new polynomial are completely different from the original ones and some of them become complex numbers.



This formulation is significantly ill-conditioned.

Again, consider the system of the flash separator in Example 1.3. If the system is solved in the unknowns  $T$ ,  $V$ ,  $L$ ,  $x_i$ , and  $y_i$ , the remaining variables being defined, the problem is strictly related to the real physical phenomena occurring in the unit. Conversely, if the same system is solved in the variables  $T$ ,  $F$ ,  $L$ ,  $x_i$ , and  $z_i$ , an equivalent nonlinear algebraic equations system is obtained from a mathematical standpoint, but the solution could be a more problematic issue as the system loses its correspondence to the physics of the flash separation and, at worst, it could have no solution.



An object-oriented programming obliges the programmer to change the point of view from the procedural programming based on SUBROUTINES.

It is worth remarking that, in simulating a physical phenomenon, the object represents the same phenomenon. Therefore, in object-oriented programming, the attention is not focused on the algorithm that should solve a class of mathematical problems as in procedural programming. Now, the algorithm has to solve a specific problem, rather than a class of problems.

It is not easy for people used to programming in the procedural way to understand this difference. If the object is developed by looking at a physical phenomenon, it is easier to avoid the error of solving a nonlinear system that could not directly correspond to the physical phenomenon.

One can object that this kind of programming is similar to the manual procedure, where calculations were oriented to solve a specific problem. This is partially true, but with two important differences.

- In object-oriented programming, the importance of generic algorithms is not underestimated. The object describing the mathematical problem includes an object where these algorithms are inserted.

- Each physical phenomenon requires a specific object, but the spread of these objects is now feasible thanks to the following circumstances:
  - The modern editors are more efficient.
  - The calculation speed has been dramatically improved and it is possible to write and validate programs in less time.
  - C++ has a feature that makes it very efficient to extend or modify existing codes: the inheritance. A certain amount of experience is necessary to understand and use this feature in depth. It will be shown later in some practical cases.

One of the inheritance features is the following.

Suppose a class that solves a specific family of problems is available, but, unfortunately, this class is not completely satisfactory for a specific problem. Moreover, suppose it cannot be modified because other people use it or because no source files are available. In such a context, it is possible to develop a new class by deriving it from the original one. It requires one code line. The new class is similar to the previous one and it is sufficient to add some new functions or overlap existing functions to obtain a fully satisfactory class.



To summarize, the modeling of a well-posed physical phenomenon must be done by well-conditioned mathematical formulation, to be selected from all the possible alternatives.

According to the problem, it is necessary to define some criteria to check that the mathematical formulation is well-conditioned, independent of the algorithm adopted to solve the problem.



A secondary requirement deals with the calculation efficiency and the memory allocation, when there are different well-conditioned alternatives.

For example, the same problem could lead either to a linear or to a nonlinear system. If the linear system is well-conditioned, it is preferable.

*Remember:* the priority is that the mathematical formulation is well-conditioned.



### 1.7.3

#### Selection of the Best Algorithm

In developing the first scientific numerical programs, the significant digits available in single precision appeared adequate for solving with an acceptable accuracy all the computing problems. For this reason, Fortran adopted the single precision by default. This choice was mainly dictated by the necessity to limit the memory allocation: in the 1960s,  $30 \times 30$  was a large matrix! On the other hand, single precision speeded up computational times as well.

Then, in the past, the algorithm selection was mainly related to the calculation speed and to the memory allocation.

It is interesting to realize the evolution of the criteria for the selection of algorithms for solving a specific problem and the reasons that led to this evolution.



It is paradigmatic to analyze criteria to select algorithms for integrating a differential equations system.

Since the first numerical computations, the scientific community realized that for differential equations systems it was indispensable to use algorithms characterized by a very small local error.

The first, simplest, device was to replace the single precision with the double one that the most recent programming languages adopt by default. Mathematicians searched for algorithms with the minimum local error; for example, at the end of the 1970s both Milne and Hamming algorithms were appreciated for their small local error (Hamming, 1962). Nevertheless, the importance of the round-off error, intrinsic in computer calculations, was discovered by applying numerical programs to complex and large problems, for which the rounding-off can cause algorithm instability due to the propagation error. Consequently, algorithms that were found to be advantageous without round-off errors became unsuccessful because of their instability.

The consequences of round-off error propagation were introduced in Section 1.6.



An algorithm is *stable* if it is able to control the increase in round-off error when the problem is well-posed and its formulation is well-conditioned.



Conversely, the algorithm is *unstable* when it is incapable of controlling the amplification of round-off error, although the problem is well-posed and its formulation is well-conditioned.

Programmers quickly realized that round-off errors induced other side effects besides error propagation and that it was not sufficient to adopt the double precision to avoid them. On the other hand, each mathematical problem may present specific numerical difficulties related to the same problem, which only a few algorithms may overcome.

For example, in a minimization problem, the presence of very narrow valleys makes the problem hard for many algorithms; as another example, for integrating a stiff differential equations system, basic algorithms require a large number of integration steps.



An algorithm is *robust* not only if it is stable, but also if it is able to avoid, or minimize, all the consequences deriving from round-off errors and to overcome problematic situations for the specific problem under study. Conversely, the algorithm is *nonrobust*.

Here, it is recalled that the concept of *stability* can take different meanings, according to the context.



A physical problem may be *well-* or *ill-posed*.

A mathematical formulation may be *well-* or *ill-conditioned*.

An algorithm may be *stable* or *unstable*, *robust* or *nonrobust*.

Two icons are introduced to highlight these concepts and discriminate their different meaning.



This icon is used when the conditioning of the mathematical formulation of a problem is discussed.

This icon is used when the robustness and stability of a specific algorithm is discussed.



Let us see what other consequences arise from the round-off error. Floating point operations have three other error sources. Whereas the previous one was due to the propagation of existing errors, these are caused by the local error.

The **second enemy** in computer numerical calculations is addition or subtraction of numbers with different orders of magnitude.



Many authors underestimate this kind of error since addition and subtraction have the same local error as multiplication and division:

$$x \omega y = fl(x \omega y) = (x \omega y)(1 + \varepsilon) \quad (1.17)$$

On the other hand, the round-off has a very different effect on them. It is important to underline that if two numbers have different orders of magnitude and one executes either a multiplication or a division, both the numbers transfer in the result all their pieces of information. Conversely, when one adds or subtracts the same numbers, only a portion of the total amount of information of the smallest number is transferred into the result. In some cases, the loss of information may become very dangerous.

### Example 1.6 Sum and product

Consider the following problem: add and multiply two pairs of numbers, the first pair with the same values and the second one with different orders of magnitude. A possible program may be

```
#include "BzzMath.hpp"
void main(void)
{
    double x1 = 1234567890123456.;
    double y1 = 1234567890123456.;
    double sum1 = x1 + y1;
    double prod1 = x1 * y1;
    double x2 = 1234567890123456.e-8;
    double y2 = 1234567890123456.;
    double sum2 = x2 + y2;
    double prod2 = x2 * y2;
    BzzPrint("\n%20.14e\n%20.14e", sum1, sum2);
    BzzPrint("\n%20.14e\n%20.14e", prod1, prod2);
}
```

It results in

```
Sum1 = 2.46913578024691e+015
Sum2 = 1.23456790246914e+015
Prod1 = 1.52415787532388e+030
Prod2 = 1.52415787532388e+022
```

Whereas the result in `Sum1`, `Prod1`, and `Prod2` contains all the information of both the numbers involved in the operation, the result obtained in `Sum2` contains all the information of  $x_2$ , but only half information about  $x_1$ , while the other portion is lost in the operation.

In the extreme case, where the ratio of the smallest number to the largest number is smaller than the macheps, all the pieces of information of the smallest number are lost. For example, consider the summations  $1 + 10^{-20}$  and  $1 + 10^{20}$  in double precision: the result is 1. in the former case and  $10^{20}$  in the latter one.



It is important to understand when such operations are safe or, conversely, when it is preferable to avoid them, if possible.

Note that, often, numerical problems are not caused by the local error, but rather by effects of the local error on the error propagation. In these cases, the error propagation should be analyzed, analogously to the analysis carried out when either a number cannot be exactly stored or when a round-off error affects an operation. By doing so, it is advantageous to discard algorithms involving subtraction of quite similar numbers in favor of other algorithms.

For example, consider the following linear system:

$$10x_1 + 10^{20}x_2 = 10^{20}$$

$$x_1 - .9x_2 = .1$$



As will be shown in Chapter 4, this system is well-conditioned.

Getting  $x_1$  from the former equation:

$$x_1 = 10^{19} - 10^{19}x_2$$

and replacing it in the latter one:

$$x_2 = \frac{.1 - 10^{19}}{-.9 - 10^{19}} = 1$$

hence

$$x_1 = 10^{19} - 10^{19} = 0$$

On the other hand, getting  $x_2$  from the former equation:

$$x_2 = 1 - 10^{-19}x_1$$

and replacing it in the latter one:

$$x_1 = \frac{.1 + .9}{1. + .9 \cdot 10^{-19}} = 1$$


hence

$$x_2 = 1 - 10^{-19} = 1$$


The second solution is good, whereas the first one is very bad.

Actually, the right-hand vector and the residuals obtained by replacing the first solution in the original system are different.

In the first case, the inaccuracy due to the local error in the evaluation of  $x_2$  dramatically affects the evaluation of  $x_1$ , since the algorithm leads to the subtraction of two similar numbers.

This algorithm is unstable. 


In the second case, the inaccuracy due to the local error in evaluating  $x_1$  has no effect on the evaluation of  $x_2$ , since it does not require any subtraction between quite similar numbers.

This algorithm is stable. 

Conversely, consider the following system:

$$10^{-10}x_1 + 10^{-10}x_2 = 1.$$

$$x_1 - x_2 = 1./3.$$

As will be discussed in Chapter 4, this system is ill-conditioned. 

Getting  $x_1$  from the former equation:

$$x_1 = 10^{10} - x_2$$

and replacing it in the latter one:


$$x_2 = \frac{1./3. - 10^{10}}{-1. - 1.} = 4.99999999983333e + 009$$

hence

$$x_1 = 5.00000000016667e + 009$$

The achieved solution satisfies the former equation, but not the second one, which has only the first six significant digits correct. In such a case, the failure is not related to the error propagation, since the difference between two quite similar numbers is never executed, rather it is due to the specific error analyzed here: the addition or the subtraction of numbers with different orders of magnitude. Replacing  $10^{-10}$  by  $10^{-20}$  in the first equation, the result should be completely wrong since an addition of two numbers with a ratio smaller than the machine should be executed.

Whereas a stable algorithm could have been obtained in the previous case because the system is well-conditioned, this is not possible anymore because the system is ill-conditioned.

Due to its ill-conditioning, there are no stable algorithms to solve this system. 

When calculations were carried out in single precision, it was necessary to be careful in the addition of numerous terms.

For example, let us evaluate

$$S = \sum_{i=1}^{10^7} 1 + 10^9$$

by using `float` numbers.

If the term  $10^9$  is selected as the first term of the sum, the result is still  $10^9$ . It represents the most classical situation, where this problem is independent of the error propagation: actually, the error in the result does not arise for a difference of quite similar numbers.

When `float` numbers are used, it is sufficient to make the sum in `double` precision to avoid this problem. Conversely, it is theoretically appropriate to sort them before evaluating the sum, when `double` numbers are used. Usually, one can avoid the reordering of the numbers, which may require a computational time considerably larger than that required for the same addition, by operating in `double` precision. In this way, the risk of having a significant error due to the successive addition of different numbers is very rare. Actually, a series of numbers of order of  $10^{13}$  should be summed to note an error in the third significant digit.



Often, the physical problem we are solving provides the order of magnitude of terms involved in a summation. It is suitable to start from the smallest one by adding more and more large terms at a time.



The **third enemy** in computer numerical calculations is the possibility of having an overflow.



The **fourth enemy** in computer numerical calculations is the possibility of having an underflow.

It is even necessary to prevent both these possibilities.

Often, the strategy is very simple. For instance, if the function `exp` is used, it is possible to limit the value of its argument to avoid any overflow possibility.

An important example is given by the Euclidean norm of a vector (Chapter 3). For a two-dimensional case, it is defined as follows:

$$\sqrt{a_1^2 + a_2^2} \quad (1.18)$$

If  $a_1 = 10^{200}$  and/or  $a_2 = 10^{200}$ , by raising such a number to the power of 2, an overflow would occur with unforeseeable results.

Analogously, if  $a_1 = 10^{-200}$  and/or  $a_2 = 10^{-200}$ , by raising the numbers to the same power, an underflow would occur with an incorrect result again.

Both these potential difficulties can be eliminated through the following procedure. By indicating with  $m$  the maximum absolute value between  $a_1$  and  $a_2$ , relation (1.18) becomes

$$m \sqrt{\left(\frac{a_1}{m}\right)^2 + \left(\frac{a_2}{m}\right)^2} \quad (1.19)$$



Calculations in floating point should be carried out by taking care of *four* aforementioned error sources, out of which two are related to addition and subtraction operations.

As different algorithms exist for the numerical solution of the same problem, it is strongly recommended to select the optimal one. Selection must take into account the following properties, listed below in descending order of weight.

- 1) Stability and robustness
- 2) Accuracy
- 3) Efficiency
- 4) Memory allocation



In general, this order can be considered valid for today's numerical computing; such properties were sorted differently in the past.

To show the conceptual difference between the algorithm stability and the mathematical formulation conditioning, the following examples are reported.

The difference (Conte and De Boor, 1980)

$$\sqrt{1+x} - \sqrt{x}$$

is a well-conditioned formulation for large values of  $x$ . Actually, the condition number is

$$\kappa(x) = \frac{\phi'(x)x}{\phi(x)} = \frac{x}{2(\sqrt{x+1}\sqrt{x})}$$

where  $\kappa = 0.5$  when  $x$  is significantly large.

This formulation of the problem is well-conditioned.



Nevertheless, the algorithm adopted to evaluate such a difference is unstable. For example, with eight significant digits and  $x = 10000000$ , one obtains

$$\sqrt{10000001} - \sqrt{10000000} = 3162.278 - 3162.278 = 0$$

Conversely, by adopting the aforementioned device 1,

$$y = \frac{(\sqrt{x+1} - \sqrt{x})(\sqrt{x+1} + \sqrt{x})}{\sqrt{x+1} + \sqrt{x}} = \frac{1}{\sqrt{x+1} + \sqrt{x}}$$

and with  $x = 10000000$ :

$$y = \frac{1}{3162.78 + 3162.278} = 0.0001581139$$

The last algorithm is stable, not the first.



Another example deals with the differential equation

$$y' = -1000y, \quad y(0) = 1$$

with the following analytical solution:

$$y = e^{-1000x}$$



This differential equation is very well-conditioned, as explained in Buzzi-Ferraris and Manenti (2011b).

After integration by the forward Euler's method,

$$y_{n+1} = y_n + h(-1000y_n)$$

with an integration step  $h = 0.01$ , the solution diverges.



The forward Euler's method is unstable for this problem with the selected integration step.

The differential equation (Rice, 1993)

$$y' = 0.5y + \frac{4}{2x^2} = f(x, y)$$

$$y(1) = \sqrt{e} - 1$$

is well-conditioned and its analytical solution is

$$y = e^{1-\frac{x}{2}} - \frac{1}{x^2}$$



This differential equation is very well-conditioned.

To know, for example, the exact values of  $y$  in  $x = 1$ ,  $x = 1 + h$ ,  $x = 1 + 2h$ , and  $x = 1 + 3h$ , it is possible to adopt the following integration expression (five-point central differences):

$$y(x + 4h) = y(x) - 8y(x + h) - 12f(x + 2h, y(x + 2h)) + 8y(x + 3h)$$

This formula has a very small local error but is unstable in solving this problem for any integration step. Contrary to the general thinking, the solution worsens by decreasing the integration step. Actually, in this case, the algorithm instability is related to the number of integration steps. After 5 steps, 5 significant digits are lost; after 20 steps, 18 digits are lost.



This algorithm is unstable.

Again, consider the expression  $y = \frac{1 - \cos(x)}{x}$ .




This expression is well-conditioned, when  $x$  approaches zero.


In particular,  $\kappa \rightarrow 1$  for  $x \rightarrow 0$ ,

$$\kappa(x) = \frac{|\phi'(x)x|}{|\phi(x)|} = \left| \frac{x^2 \left( \frac{\sin(x)}{x} - \frac{1 - \cos(x)}{x^2} \right)}{1 - \cos(x)} \right|$$


being the condition number. Hence, the problem is well-conditioned.

Nevertheless, dividing by  $x$  the difference  $1 - \cos(x)$  for  $x \rightarrow 0$ , the algorithm is unstable. 

The algorithm stability is fundamental. As already seen, some very promising methods prove to be unstable and, therefore, unreliable.


It is necessary to define some criteria to check the algorithm stability for each kind of problem. 

Another requirement is related to the algorithm accuracy.

It is also necessary to define some criteria to check the algorithm accuracy. 

The computational time needed for solving a problem is correlated to the algorithm, the machine processor, the programming language, the compiler, the operating system, and the optimization options adopted for compiling the code.


Until a few years ago, the number of required elementary operations was deemed a good criterion to assess the algorithm efficiency. When the problem solution needed a large number of linear algebra operations, *flop* was used as a unit of measure.

The following set of elementary operations: 

$$s_i = s_i + a_{ik}b_k \quad (1.20)$$

is called a *flop*.

This set of operations includes multiplication, addition, assignment, and searching for the required coefficients.

As an example, the Gauss algorithm to solve a dense linear system requires  $O(n^3/3)$  flops. 

For fixing the number of flops, only the higher order is usually considered. However, nowadays, this criterion has lost part of its significance for different reasons (see Appendix A):

- A well-defined hierarchy of the elementary operations, in terms of computational time, characterized calculations on all computers. Division was much slower than multiplication; multiplication was slower than addition and subtraction. Today, the computational time for elementary operations depends strongly on the machine processor and the optimization level adopted in the compiler.
- Computing time changes even for the same elementary operation, according to the numerical values involved. For example, multiplications by 2. are faster than multiplications by 0.0009472.
- Several modern processors automatically parallelize some operations when the members of the operations are in adjacent positions. Consequently, today the loading sequence of matrix coefficients may be much more important than the number of elementary operations.
- Some special codes (the so-called BLAS) optimize linear algebraic operations according to processor characteristics. Therefore, an algorithm can strongly change its performance by means of these built-in routines.

- Whereas, a few years ago, calculations were rigorously of a sequential type and, therefore, it was advantageous to reduce the total amount of operations, at present this is no longer valid when an algorithm is structured to carry out calculations according to a parallelization technique simultaneously on more than one single processor.



A consistent way to compare the efficiency of two programs is to run a calculation and check relevant computational times. However, it is important to note that the comparison is valid only for the specific calculation and for the specific processor by which it is carried out.

The above description should give the impression that comparing performances of the different algorithms is an awkward task.



Comparing computational times is gradually losing significance in the selection of the best algorithm.

It is worth remarking that the most important properties to be considered are stability and accuracy.

When algorithm selection was based on the computational time, a small improvement in performance was sufficient to prefer the best performing algorithm.

Today, algorithm selection must not be based on the difference in performance. With the increasing power of processors and the possibility of developing more complex programs, the wisest decision should be to introduce both performing and stable algorithms in the program and to switch between them according to the calculation necessity.



It is simple to check the memory for the data allocation considering the dimension of the required matrices and vectors.

Whereas such a problem was historically important, today better performing machines make it less relevant.

In the following, it is shown how to allocate matrices in a compact form when they are significantly sparse.

## 1.8

### Selection of Programming Languages: Why C++?

A good calculation program should include several algorithms selected for their stability and/or efficiency. It is not enough to select good algorithms to obtain an optimal program; in particular, the following properties must be satisfied in the development of the program.



- 1) The algorithm stability has to be preserved.
- 2) The program has to be efficient.
- 3) The program has to minimize the memory allocation.
- 4) The program has to be robust.

5) The program has to be easy to use.

To clarify the first statement, consider the linear system:

$$\begin{aligned} 10x_1 + 10^{20}x_2 &= 10^{20} \\ x_1 - .9x_2 &= .1 \end{aligned}$$

If it is solved through the Gauss method (Chapter 4), it gives the numerical result  $x_1 = x_2 = 1$ . Nevertheless, if the algorithm was not implemented in a proper way, the result might be  $x_1 = 0$  and  $x_2 = 1$ , leading to an instability that should have been avoided.

The *program robustness* involves two concepts:

- More than one good algorithm may be available for real problems and the program should be able to select the best algorithm at the appropriate time.
- The ability to limit human errors, usually due to an inexperienced user who does not know any feature of the implemented algorithms as discussed in the following.

Algorithms are the fundamental bases for *procedural programming*. By implementing algorithms in `SUBROUTINES` or in *functions* and then collecting `SUBROUTINES` and *functions* in procedures, these can be reused every time a similar problem is to be solved.

Algorithms play a fundamental role even in *object-oriented programming*, but they are invisible to the user, as they are encapsulated in the *object*.

As they are independent of the adopted programming language, the efficiency of a C++ code is at least equivalent to that developed in another procedural language such as Fortran, C, and Pascal. Nevertheless, by means of object-oriented programming, C++ allows the writing of more robust codes, is easier to use, and, sometimes, performs better.

Even though object-oriented programming allows the development of more complex structures, the difference in using a function by procedural and by object-oriented languages is discussed below.

The function considered as an example is very simple: the product between two dense matrices

$$C = AB$$

Suppose the selected algorithm is the function implemented in the `BLAS` numerical library:

```
dgemm(N, N, &m, &n, &k, &one, a, &k, b, &n, &zero, c, &m);
```

In a Fortran or C program, the user who has to provide all the variables of the argument for executing this simple operation should directly call it. In C++, the same function can be encapsulated in an object of a class called, for example, `Matrix`. If matrices `A`, `B`, and `C` are objects of the `Matrix` class, the required code might be

```
C = A * B;
```

or

```
Product(A, B, &C);
```

The use of this function is clearly easier in C++. Additional considerations enhance the functionality of object-oriented programming. Suppose a new function, more efficient than the previous `dgemm`, is proposed. Probably, this new function has a new interface too. In object-oriented programming, the expert managing library has the task to replace the old function by the new one, by modifying the interface within the `Matrix` class.



The user interface should be unchanged in C++, whereas, almost certainly, the user should rewrite a portion of its code in procedural programming.

C++ code is also more robust. First, as the user is not forced to pass all the variables to the function `dgemm`, this function being encapsulated within the object, it is the same object that takes care to properly initialize the function. Second, the user should not worry about the dimensions of the matrix **C**: the object automatically resizes the matrix. In addition, the operation is carried out only if the matrices **A** and **B** are congruent: as they are objects, **A** and **B** can automatically and mutually check their dimensions. Finally, some trivial errors, such as those listed below, can be avoided:

```
A = A * B;
```

or

```
Product (A, B, &B) ;
```

In the first case, the operator `*` realizes that the user is employing the matrix **A** for storing the result and, therefore, it automatically creates an auxiliary object, where it can execute the operation, returning the result to the original matrix **A** only at the end. The same approach is adopted for the matrix **B** by using the previous function `Product`.

Even for this simple case, it is possible to improve the performances of the function when the matrices **A** and **B** can be either dense or sparse, but one does not know *a priori* their real degree of sparsity. If **A** and **B** are objects, they know how much and which coefficients are nonzero at the run-time. If one or both the matrices are significantly sparse, the function `Product` or the operator `*` should no longer use the previous function `dgemm`, which is efficient only for dense matrices, rather another function that exploits the matrix sparsity should be used.